POSTGRES

P. Luzanov, E. Rogov, I. Levshin
Translated by L. Mantrova

THE FIRST EXPERIENCE

15

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. Originally developed in the academic environment, it has managed to bring together a wide developer community through its long history. Nowadays, PostgreSQL offers everything that most customers need, and it 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 the result of rethinking all the previous work and the desire to overcome the limitations of its rigid type system.

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

4 i

POSTGRES is sometimes considered to be a so-called post-relational database system. The relational model had always been criticized for its restrictions, even though they were 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 databases had to support custom data types and such features as inheritance or creating and managing complex objects.

The first version of this database system appeared in 1989. It was being improved and enhanced for several years, but in 1993, when version 4.2 was released, the project was shut down. However, despite its official cancellation, UC Berkeley alumni Andrew Yu and Jolly Chen revived the project and resumed its development in 1994, taking advantage of its liberal BSD license and open source. 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. This name may be quite hard to articulate, but nevertheless, we should pronounce it as “Post-Gres-Q-L” or simply “postgres,” but not as “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 seven people at the moment.

Apart from developers who contribute to the project from time to time, there is a group of main developers who have made significant contributions 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. The current list of developers is published on the PostgreSQL's official website: postgresql.org/community/contributors.

The contribution of Russian developers into PostgreSQL is compelling. 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.

In 2015, Oleg Bartunov, a professional astronomer and research scientist at Sternberg Astronomical Institute of Lomonosov Moscow State University, teamed up with Teodor Sigaev and Alexander Korotkov to start the Postgres Professional company, which is now the main talent foundry in Russia when it comes to database system development.

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 also 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.

Major versions 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 15, which was released in October 2022.

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 14.5 contains bug fixes for the 14.4 release, while version 15.2 provides fixes for PostgreSQL 15.1.

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 can be released as fast as 24 hours.

Apart from the community support, 24x7 commercial support for PostgreSQL is also provided by a number of companies in different countries, including Postgres Professional (www.postgrespro.com).

Current State

PostgreSQL is one of the most popular databases. Based on the solid foundation of academic development, over several decades 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 15, 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 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 standardizes JSON support. Much of this functionality is already implemented in PostgreSQL 15.

In general, PostgreSQL provides a high rate of conformance to the SQL standard, supporting 170 out of 177 mandatory features and many optional ones.

Transaction Support

PostgreSQL provides full support for ACID properties and efficient transaction isolation based on the multi-version concurrency control (MVCC). This method allows us 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 serializable isolation level, which is the strictest one. 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 rich set of database objects that allow you to efficiently 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 that supports a variety of languages, extended with efficient index access methods.
- Semi-structured data typical of NoSQL: hstore (storage of key–value pairs), xml, json (represented as text or in a more robust 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 CPU architecture. Its performance grows almost linearly as the number of cores increases.

PostgreSQL can parallelize queries and some commands (such as index creation and vacuuming). In this mode, reads and joins are performed by several concurrent processes. JIT-compilation of queries can speed up operations thanks to better use of hardware resources. Each PostgreSQL version adds new parallelization features.

Horizontal scaling can rely on both physical and logical replication. It allows you to build PostgreSQL-based clusters to achieve high performance, fault tolerance, and geo-distribution. Some examples of such systems are Citus (Citus-data), Postgres-BDR (2ndQuadrant), Multimaster (Postgres Professional), Patroni (Zalando).

Query Planner

PostgreSQL relies on 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 even the most complex queries. It can use all access paths and join methods 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 in some cases they can prove to be more efficient and compact.
- 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 plane and can serve to implement fast k-nearest neighbor (k-NN) search, or indexing overlapping intervals.
- SP-GiST,
  a generalized non-balanced tree based on dividing the search space into non-intersecting nested partitions. For example, quad-trees for spatial data and radix trees for text strings.
- GIN,
  a generalized inverted index used for compound multi-element values. It is mainly applied in full-text search to

find documents that contain the words used in the search query. Another example is search for elements in data arrays.

- RUM,
  an enhancement of the GIN method for full-text search. Available as an extension, this index type can speed up phrase search and return the results in the order of relevance without any additional computations.
- BRIN,
  a compact structure that provides a trade-off between the index size and search efficiency. Such index is useful for huge clustered tables.
- Bloom,
  an index based on the Bloom filter. Having a compact representation, this index can quickly filter out non-matching tuples, but the remaining ones have to be re-checked.

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. It is also possible to create partial indexes for specific sets of rows. Covering indexes can speed up queries, since all the required data is retrieved from the index itself, avoiding heap access.

The planner can use a bitmap scan, which allows combining 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 on Windows systems.

Its portable open-source C code allows building 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 various features, such as:

- 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 the PostgreSQL core. For example, the following complex systems are built as extensions:

- CitusDB,
  which implements massively parallel query execution and data distribution between different PostgreSQL instances (sharding).
- PostGIS,
  one of the most popular and powerful geoinformation data processing systems.

- TimescaleDB,
  which provides support for time-series data, including special partitioning and sharding.

The standard PostgreSQL 15 distribution 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 unrestricted use of PostgreSQL; you may also modify PostgreSQL code without any limitations and integrate it 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 15

If you are familiar with the previous versions of PostgreSQL, this chapter can give you a sense of what has changed over the past year. It mentions only some of the updates; for the full list of changes, see the Release Notes: postgrespro.com/docs/postgresql/15/release-15.

SQL Commands

After an unsuccessful attempt made as early as PostgreSQL 11, the **MERGE command is finally implemented**. This command is defined by the SQL standard. It is more flexible and sometimes also more efficient than the previously available `INSERT ... ON CONFLICT` command.

Should **NULL values be considered distinct** in integrity constraints? There used to be only one correct answer to this question (negative), but now the `NULLS [NOT] DISTINCT` clause allows choosing the desired behavior.

You can now provide **a list of columns in the `ON DELETE SET NULL` clause** for composite foreign keys: instead of resetting values in all the columns, simply specify the exact fields to be affected when the parent entry is deleted.

16 ii

The COPY command can now take the **table header in the first data row** as input, as well as include it in the output.

Functions

The **`unnest` and `range_agg` functions have been added for multidimensional data types** introduced in version 14.

New regular expression functions have been added to facilitate migration: `regexp_like`, `regexp_count`, `regexp_instr`, and `regexp_substr`.

Functions `pg_size_pretty` and `pg_size_bytes` have been improved **to support petabyte units**. Prefixes for larger units have been reserved for future enhancements.

Partitioning

If a trigger on a partitioned table gets renamed, **triggers on table partitions are renamed** automatically.

The **`CLUSTER` command is now supported** for partitioned tables.

Planning is now performed faster if only a few partitions are relevant to the query.

Write-Ahead Log

LZ4 and ZStd compression algorithms for full page images have been added. As compared to the standard PGLZ algorithm, LZ4 usually consumes less resources but is just as efficient, while ZStd is more CPU-intensive but shows better compression results. The LZ4 algorithm is also supported by the `pg_receivewal` utility.

WAL records for recovery can now be prefetched, which may speed up server restart after a failure, backup restore, and application of WAL records during replication.

Continuous archiving now allows using **custom modules** instead of an OS command. Developers of backup solutions can now rely on this feature.

A new feature that is not directly available to end users is the ability to create **custom resource managers**. This functionality is important for developers of both index and table access methods. Incidentally, **table access methods can now be altered** using the `ALTER TABLE ... SET METHOD` command (but sadly, there is no alternative available yet).

Added **the `pg_walinspect` extension**, which enables viewing WAL records via an SQL query (in addition to the already available `pg_waldump` utility).

Logical Replication

Logical replication has been significantly improved. There is hope that such long-awaited features as replication of sequences and DDL commands will be soon implemented as well.

When creating a publication, you can now **filter rows and columns** to be replicated, as well as **replicate all tables that belong to a particular schema** (`FOR ALL TABLES IN SCHEMA`).

Added **support for prepared transactions** to logical replication.

Subscriber processes are now executed with **the privileges of the subscription owner**; superuser rights are not required anymore.

Subscriptions can now be stopped on conflict. Previously, the WAL receiver process was restarted every second; now **the conflicting transaction can be skipped** (`ALTER SUBSCRIPTION ... SKIP`).

Backup

The good-old `pg_basebackup` utility was refactored.

You can now **choose the backup location**: it can be created either on the client or on the server. Following the principle of extensibility, you can create custom backup targets; for example, the new `basebackup_to_shell` extension passes the backup to an OS command.

Data compression on the server side can now be performed by gzip, LZ4, and ZStd algorithms, which can be useful if the network throughput is low.

Security

The `CREATE` privilege has been revoked from the `public` schema; this privilege used to be inherited by all users from the `public` pseudorole. The `public` schema is now owned by a **new pseudorole called `pg_database_owner`**, which implicitly includes the owner of the current database.

PostgreSQL now provides more opportunities for delegating system administration tasks to non-privileged users. **The right to perform the `CHECKPOINT` command** is granted to the new role called `pg_checkpointer`. **The access to statistics on backend memory contexts** (`pg_backend_memory_contexts` and `pg_shmem_allocations` catalog views) is granted to the `pg_read_all_stats` role. You can also **control access to configuration parameters** (`GRANT SET` and `GRANT ALTER SYSTEM`).

It is now possible to create **views with the caller's privileges**: the user must have the right to access these views' objects in this case.

Regular users **can no longer manage their own role membership** (that is, use the `ADMIN OPTION`) by default to add or remove members of its own role).

Monitoring

Cumulative statistics is now stored in shared memory; a separate statistics collector process is not used anymore, and there is no need to mount tmpfs for the files that accumulate statistics.

20 ii

New **wait events for archive commands** have appeared: ArchiveCommand, ArchiveCleanupCommand, RestoreCommand, RecoveryEndCommand.

Server log can now be written in the JSON format; it is much better suited for programmatic parsing than a plain-text output.

The `pg_log_backend_memory_contexts` function now logs **memory contexts of auxiliary processes** in addition to backend memory contexts.

Postgres_fdw

The `postgres_fdw` foreign data wrapper, which is the cornerstone of the future built-in sharding, can now **pass CASE expressions** to the foreign server and **set the *application_name* parameter** to an arbitrary value.

Transactions started on foreign servers **can now be committed in parallel**, which may boost performance. Unfortunately, we are still far from having genuine distributed transactions.

Vacuuming and Freezing

Vacuuming and freezing have been significantly refactored. Now `VACUUM` can partially process the pages that cannot be locked instead of skipping them altogether. As a result, **the `relfrozenxid` threshold can also be advanced by routine vacuuming**, so aggressive vacuuming can be performed less frequently.

The **`VACUUM VERBOSE` command now displays more useful information**, which was previously included into the server log only.

Optimizations

Sorting now utilizes memory more efficiently, which increases the chances to avoid disk access; **sorting of a single column is now performed faster**.

Encoding validation of UTF-8 text strings has been sped up by processing several bytes at a time.

Data transfer from workers to the leader process in parallelized queries has been accelerated. The *parallel_tuple_cost* value has not been changed yet.

Parallel aggregation has been already available for a long time, and now the **`SELECT DISTINCT` command can also be executed in parallel**.

New support functions added for the planner improve **cardinality estimation for conditions with the `starts_with` function or the `^@` operator**.

The default ***hash_mem_multiplier* value has been increased to 2.0** (it used to be 1.0). It means that hash tables can now use twice as much memory as specified in the *work_mem* parameter: it's good for queries, but you have to keep an eye on memory consumption.

You can **manage the size of the working table of a recursive query** using the *recursive_worktable_factor* parameter.

Miscellaneous

The **ICU locale provider**, which was added as early as version 10, can now be used as the default one for databases and clusters; collation versions are recorded in the system catalog.

Python 2 is not supported anymore, both `plpython2u` and `plpythonu` languages were removed.

Documentation

The documentation was extended with a new **chapter on hash indexes**: postgrespro.com/docs/postgresql/15/hash-index.

III Installation and Quick Start

What is required to get started with PostgreSQL? In this chapter, we'll learn how to install PostgreSQL and manage the corresponding service, and in the next one we'll continue our ramp-up by creating a simple database and trying out some basic SQL queries.

We are going to use a regular (often called "vanilla") distribution of PostgreSQL 15. Depending on your operating system, the procedure of installing and setting up PostgreSQL will differ:

- If you are using Windows, read on.
- To set up PostgreSQL on Linux-based Debian or Ubuntu systems, go to p. 28.

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

You can also use Postgres Pro Standard 15: it is fully compatible with vanilla PostgreSQL, includes some additional features developed by Postgres Professional, and is free when used for trial or educational purposes. Check out installation instructions at postgrespro.com/products/download in this case.

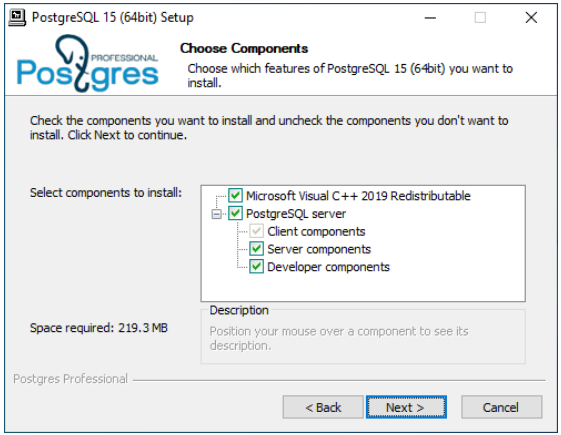
Windows

Installation

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

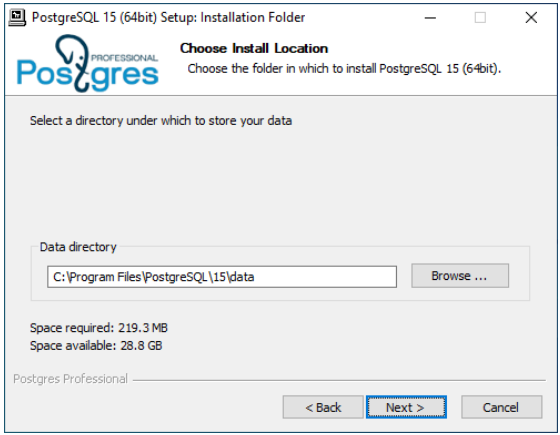
The installer provides a conventional wizard interface: you can simply keep clicking the "Next" button if you are fine with the default options. Let's go over the main steps.

Choose components (keep the current selection if you are uncertain what to choose):



Then you have to specify PostgreSQL installation directory. By default, PostgreSQL server is installed into `C:\Program Files\PostgreSQL\15`.

You can also specify the location of the data directory.

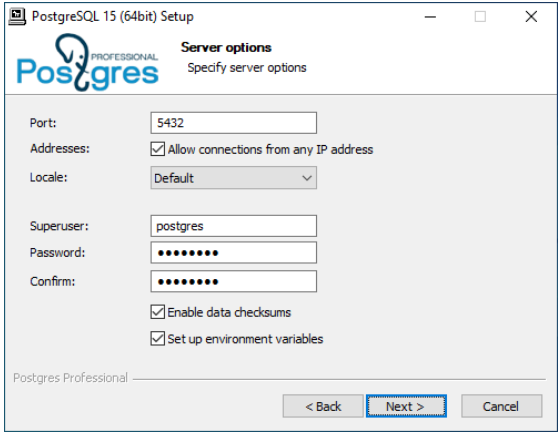


This directory will hold all the information stored in your database system, so make sure you have enough disk space if you plan to keep a lot of data.

If you are going 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. You should also select the “Set up environment variables” checkbox to connect to the PostgreSQL server on behalf of the current OS user.

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

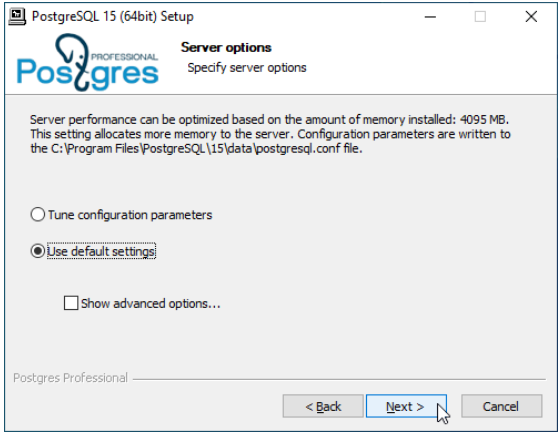


If you are planning to install PostgreSQL for training purposes only, you can select the "Use the default settings" option for the database system to take up less RAM.

Managing the Service and the Main Files

When PostgreSQL is installed, the "postgresql-15" 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 (you can typically find it at `C:\Program Files\PostgreSQL\15\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. You do not have to modify them to get started with PostgreSQL, but you’ll definitely need them in real work. Take a look into these files, they are fully documented:

- `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.

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 10 "Buster," 11 "Bullseye," and 12 "Bookworm." The currently supported Ubuntu versions are 18.04 "Bionic," 20.04 "Focal," 22.04 "Jammy," and 22.10 "Kinetic".

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://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 process non-English data, the LC_CTYPE and LC_COLLATE variables may have to be configured. For example, it makes sense to set these variables to "fr_FR.UTF8" for the French language, even though the "en_US.UTF8" may do too:

```
$ 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-15
```

It was the final step; 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 on your system. All the server processes work on behalf of this user, and all the 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-15-main.log`.

All information stored in the database is located in the `/var/lib/postgresql/15/main/` directory. If you are going to keep a lot of data, make sure that you have enough disk space.

Server settings are defined by several configuration files. There's no need to edit all these files to get started, but it's worth checking them out since you'll definitely need them in the future:

- `/etc/postgresql/15/main/postgresql.conf` is the main configuration file that contains server parameters.
- `/etc/postgresql/15/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. Yet it is really worth mastering.

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 point in getting lost on an unknown system.

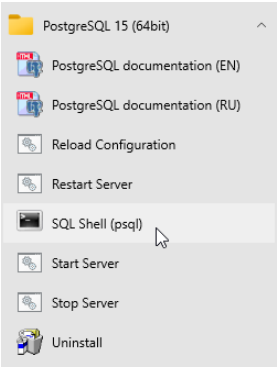
Secondly, `psql` is really convenient for everyday DBA tasks, writing small queries, and automating processes. For example, you can use it to periodically deploy application code updates on your database server. The `psql` client provides its own commands that can help you find your way around database 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 simi-

lar 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=#`. Here "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.

Now let's try out some commands. Enter only the part printed in bold; the prompt and the system response are provided here solely for your convenience.

Databases

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

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

Don't forget to finish each command with a semicolon: PostgreSQL expects you to continue typing until you enter this symbol (so you can split the command between 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 run in `psql` (so if you are using pgAdmin or some other 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 structure of the table is defined by its **columns**; the data itself is stored in table **rows**. The data is not ordered, so rows are not necessarily stored in the same order they were added into the table.

For each column, a **data type** is defined. All the values in the corresponding row fields must belong to this type. You can use multiple built-in data types provided by PostgreSQL (postgrespro.com/doc/datatype) 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 be NULL. It can be interpreted as "the value is unknown" or "the value is not set."

Let's create a table of university courses:

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

CREATE TABLE
```

Note that the `psql` command prompt has changed: it is a hint that the command continues on the new line. For

convenience, we will not repeat the prompt on each line in the examples that follow.

The above command creates the `courses` table with three columns: `c_no` specifies the course number represented as a text string, `title` provides the course title, and `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 will not 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 the 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. The postgrespro.com/doc/ddl-constraints page lists all the available constraints.

You can find the exact syntax of the `CREATE TABLE` command in the 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.

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
```

The exams table contains all the scores 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 an 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)
);
```

Besides, the REFERENCES clause here defines two referential integrity checks called **foreign keys**. Such keys show that the values of one table **reference** rows of another table.

After any action performed on the database, PostgreSQL checks that all the `s_id` identifiers in the `exams` table correspond to real students (that is, entries in the `students` table), while `c_no` course numbers 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. The AS clause allows you to rename the column if required:

```
test=# SELECT title AS course_title, hours
FROM courses;
```

```
 course_title | hours
--------------+-------
 Databases    |    30
 Networks     |    60
(2 rows)
```

The asterisk * displays all the columns:

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

In production applications, it is recommended to explicitly specify only those columns that are really needed: then the query is performed more efficiently, and the result does not depend on new columns that may appear. But in interactive queries you can simply use an asterisk.

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 the documentation: postgrespro.com/doc/sql-select#SQL-DISTINCT.

In general, you can use any expressions after the SELECT operator. And if you omit the FROM clause, the query will return 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 specified 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 operators =, <> (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 result will contain only those rows for which the filtering condition is true; if the condition is false or **undefined**, the row is excluded.

Remember:

- The result of comparing something to NULL is undefined.
- The result of logical operations on NULLs is usually undefined (exceptions: `true OR NULL = true`, `false AND NULL = false`).
- To check whether the value is undefined, the following operators are used: `IS NULL` (`IS NOT NULL`) and `IS DISTINCT FROM` (`IS NOT DISTINCT FROM`).

The `coalesce` expression is often used to replace `NULL` values with something else, such as an empty string for text types or zero for numeric types.

For more details, see the documentation: postgrespro.com/doc/functions-comparison.

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 of several tables, specifying their 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 match the courses to the corresponding exams to get all the scores for all the courses:

```
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 the 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, these queries are completely equivalent, so you can use any approach that seems more natural.

In this example, the result does not include any rows of the table specified on the left side of the join clause if they have no pair in the right table: although the condition is applied to the subjects, the students that did not take the exam in this subject are also excluded. To include all the students into the result, we have to 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 of 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 of the right table are NULL in this case.

The `WHERE` conditions are applied to the result of the join operation. Thus, if you move the subject restriction from the join condition to the `WHERE` clause, Nina will be excluded from the result because the corresponding `exams.c_no` is NULL:

```
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 efficient mechanisms to perform it. Do not join data at the application level, let the database server do the job it is sure to do better.

For more details, see the documentation: postgrespro.com/doc/sql-select#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 result in the example above). Thus, you can expand scalar subqueries by replacing them with a join, but it must be an outer join.

Scalar subqueries can also be used in filtering conditions. Let's display all the exams taken by the students enrolled after 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 the 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, which returns the opposite result. For example, the following query returns the list of students who did not get any excellent scores:

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

  name  | start_year
--------+------------
 Victor |       2014
(1 rows)
```

Note that this query result can also include those students who have not received any scores at all.

Another option is to use the `EXISTS` predicate, which checks whether 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
--------+------------
 Victor |       2014
(1 rows)
```

For more details, see the documentation: postgrespro.com/doc/functions-subquery.

In the examples above, we appended table names to column names to avoid ambiguity, but it is not always sufficient. For example, the same table can be used in the query twice, or the FROM clause can contain a nameless subquery instead of a table name. In such cases, you can specify an arbitrary name after the query, which is called an alias. Regular tables can also be assigned an alias.

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. You should choose an alias that is short but comprehensive.

The same query can also be written without subqueries:

```
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 already know, table data is not sorted. To return the rows in a particular order, we can use the ORDER BY clause with the list of sorting expressions. After each expression (sorting key), you can specify the sort direction: 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 the score, in ascending order. For the same scores, the rows are further sorted by the student ID card number, also in ascending order. If the first two keys are the same, rows are additionally sorted by the course number, in descending order.

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

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

Grouping

When grouping is used, the query returns a single row with the value calculated based on several rows of data stored

in the original tables. Grouping typically involves **aggregate functions**. For example, this is how we can 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 if you provide the GROUP BY clause with grouping keys:

```
test=# SELECT c_no, count(*),
count(DISTINCT s_id), avg(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.

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 the 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)
```

For more details, see the documentation: postgrespro.ru/doc/sql-select#SQL-GROUPBY.

Changing and Deleting Data

To modify data in a table, you should use the UPDATE operator, which provides new field values for rows defined by the WHERE clause (like for the SELECT operator).

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

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

```
UPDATE 1
```

For more details, see the documentation: postgrespro.com/doc/sql-update.

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

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

```
DELETE 1
```

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). For this purpose, let's create the `groups` table:

```
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: 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 the tables available in the database:

```
test=# \d

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

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

Note the following subtle point. We cannot create a group without a monitor, but neither can a student become the monitor of the group unless they are already a member of this group—it would make our data logically incorrect and inconsistent. Taken separately, these two operations make no sense: they must be performed simultaneously. A group of operations constituting an indivisible logical unit of work is called a **transaction**.

So let's start our transaction:

```
test=# BEGIN;

BEGIN
```

Next, we need to add a new group, together with its monitor. Naturally, we cannot remember all the students' ID, so we'll use the following 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
```

The asterisk in the prompt reminds us that the transaction is not yet completed.

Now let's open a new terminal window and launch another `psql` process: this session will be running in parallel with the first one. To avoid confusion, we will indent the commands of the second session.

Will the second session see the changes made in the first session?

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

No, 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 was started.

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

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

```
test=*# COMMIT;

COMMIT
```

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

```
test=# SELECT * FROM groups;
  g_no | monitor
-------+---------
 A-101 |    1451
(1 row)

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

It is guaranteed that several important 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 results in 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 a transaction is committed, all integrity constraints must hold true, otherwise the transaction has to be aborted. 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 efficient implementation of isolation: several sessions can perform reads and writes in parallel, without blocking each other. Blocking occurs only if two different processes try changing the same row simultaneously.

And finally, **durability** is guaranteed: the committed data is never lost even in case of a failure (if the database is set up correctly and is regularly backed up, of course).

These properties are extremely important; it is impossible to imagine a relational database management system without them.

To learn more about transactions, see postgrespro.com/doc/tutorial-transactions (Even more details are available here: postgrespro.com/doc/mvcc).

Useful psql Commands

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

Conclusion

We have managed to cover only 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 end the session, enter:

```
test=# \q
```

V Demo Database

About the Demo Database

Overview

To get to grips with more complex queries, we need to create a more substantial database (with not just three but eight tables) and fill it up with some reasonable data.

We have selected airline flights as the subject area. Our database contains statistics on all the flights of our not-yet-existing airline company that were performed within a particular timeframe. These scenarios must be familiar to anyone who has ever traveled by plane, but we'll explain everything anyway.

The database schema is shown on p. 61. 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** (mapped to the `bookings` table). Each booking can include several passengers, with a separate **ticket** issued for each passenger (`tickets`). We do not have any reliable unique ID for a passenger as a person (who might have flown with our airline multiple times), so the passenger does not constitute a

separate entity. We will assume that all the passengers are unique.

Each ticket always contains one or more **flight segments** (`ticket_flights`). Several flight segments can be included into a single ticket either if there are no direct flights between the points of departure and destination (so a multi-leg flight is required), or if it is a round-trip ticket.

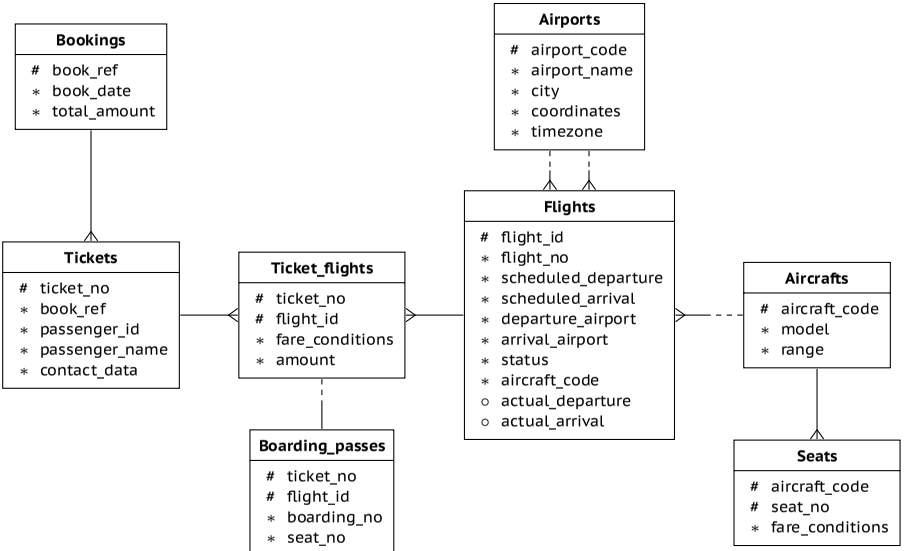
It is assumed that all tickets in a single booking have the same flight segments, even though there is no such constraint in the schema.

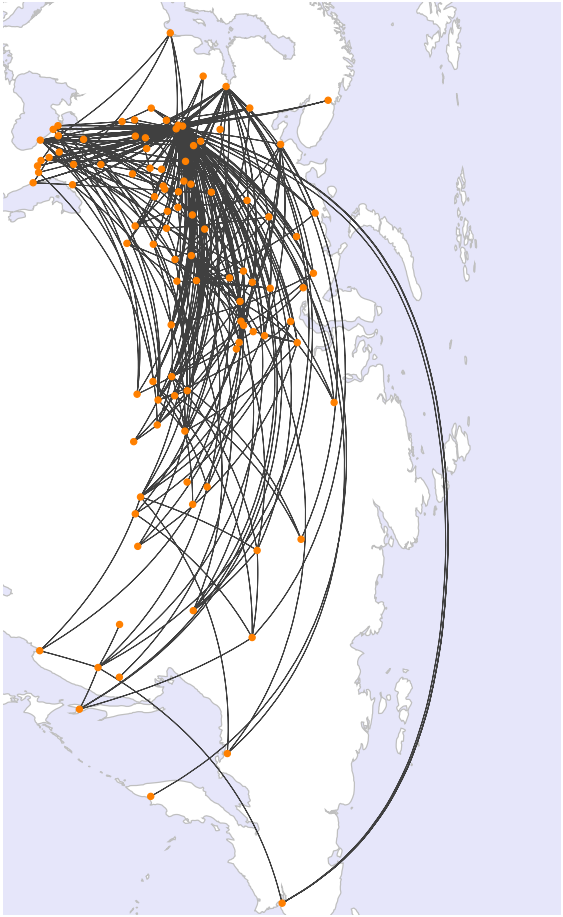
Each **flight** (`flights`) goes from one **airport** (`airports`) to another. Flights with the same flight number have the same points of departure and destination but different departure dates.

At flight check-in, each passenger is issued a **boarding pass** (`boarding_passes`), where the seat number is specified. Passengers can check in for the flight only if they have a ticket for this flight. The flight/seat combination must be unique to avoid issuing several 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 include any checks on whether the seat specified in the boarding pass is actually available in the particular aircraft.

In the sections that follow, we'll describe each of the tables, as well as additional views and functions. You can always 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



Bookings

To fly with our airline, passengers book the required tickets in advance (`book_date`, which must be within 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 number of the passenger's identity document (`passenger_id`), as well as the passenger's first and last names (`passenger_name`) and contact information (`contact_data`).

Note that neither the ID nor the name of the passenger 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 the 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

A unique ID can be either natural (if it is related to real-life objects) or surrogate (if it is generated by the system, typically as an increasing sequence of numbers).

The natural composite key of the `flights` table consists of the flight number (`flight_no`) and the date of the departure (`scheduled_departure`). To make foreign keys that refer to this table a bit shorter, a surrogate key `flight_id` is used as the primary key.

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

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

Each flight has a scheduled date and time of departure and arrival (`scheduled_departure` and `scheduled_arrival`). The `actual_departure` and `actual_arrival` times may differ from the scheduled ones: the difference is usually insignificant, 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 can be booked. This value is set one month before the planned departure date; at this point, the information about the flight is entered into 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`.

There is no separate entity for the `city`; a city name is simply an airport attribute, which is required to identify all the airports of the same city. The table also includes `coordinates` (longitude and latitude) and the `timezone`.

Boarding Passes

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

Boarding pass numbers (`boarding_no`) are assigned sequentially, 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

To identify an aircraft model, a three-digit `aircraft_code` is employed. 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 has a number (`seat_no`) and an assigned travel class (`fare_conditions`): Economy, Comfort, or Business.

Flights View

There is a `flights_v` view built over the `flights` table. Views can be queried in the same way as tables, but they do not store any data: they simply perform a particular query. The following psql command displays the definition of the view and its query:

```
postgres=# \d+ flights_v
```

This view adds the following 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 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 was taken.

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

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).

A small database is quite suitable for learning how to write queries, but if you would like to deal with query optimization specifics, choose the large database: then you'll be able to see how queries work on large volumes of data.

The files contain a logical backup of the `demo` database created with the `pg_dump` utility. Note that if you already have some database named `demo`, it will be dropped and restored from this backup. The user who runs the script becomes the owner of this database.

To install the demo database on a Linux system, switch to the `postgres` user and download the corresponding file. For example, to download 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\15` directory.

The pgAdmin application (described on p. 109) does not allow us 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

The installation is complete. Now launch `psql` and connect to the demo database:

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

The `bookings` schema stores all the entities that we need. 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, we must specify the schema for the `bookings.now` function, as we have to differentiate it from the standard `now` function:

```
demo=# SELECT bookings.now();

          now
------------------------
2017-08-15 18:00:00+03
(1 row)
```

The following query returns cities and airports:

```
demo=# SELECT airport_code, city
FROM airports LIMIT 4;

 airport_code |      city
--------------+---------------
 YKS          | Yakutsk
 MJZ          | Mirnyj
 KHV          | Khabarovsk
 PKC          | Petropavlovsk
(4 rows)
```

The contents 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. You can change this setting as follows:

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

ALTER DATABASE
```

The language has been changed at the database level; now we 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 4;

 airport_code |           city
--------------+-------------------------
 YKS          | Якутск
 MJZ          | Мирный
 KHV          | Хабаровск
 PKC          | Петропавловск-Камчатский
(4 rows)
```

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

If you want to learn more about managing schemas, see the documentation: postgrespro.com/doc/ddl-schemas.

For more details on setting configuration parameters, see postgrespro.com/doc/config-setting.

Simple Queries

Let's use this schema to discuss several problems, starting from the simplest questions and getting to more complex ones. Most of the problems are followed by a solution, but it is better to first try to come up with your own query without looking at the provided key if you want to learn SQL.

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

Solution. "The day before yesterday" is counted from the `bookings.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. One of the options is to use the `NOT EXISTS` expression 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
       );
```

Another approach relies on the operation of set subtraction. Different solutions give the same result but may sometimes differ in performance, so you have to take it into account if it matters.

```
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;
```

74 v

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

Solution. The query needs to include only those flights that have already departed.

```
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 passengers can be included into a single booking?

Solution. Let's count the number of passengers in each booking and then find 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.

```
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);
```

As you can see, the time cushion between flights can reach up to several days: round-trip tickets and one-way tickets are treated in the same way, and the time of the stay in the point of destination is treated just like 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.

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

Solution. The total number of passengers is calculated using a window function.

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

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

Solution. Let's take a look at how to count first names. The query for counting last names will differ only by the `p` subquery.

As this complex query shows, you should avoid using a single text field for different values if you are going to use them separately; in scientific terms, it is called the first normal form.

```
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;
```

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 to the last point of destination. Display airports of departure and destination for each ticket, ignoring connections, and specify whether it's a round-trip ticket or not.

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.

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.

```
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;
```

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

Problem. Find 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? What will the flight time be?

Solution. Here you virtually have to find the shortest path in the graph. We will use a recursive query to complete this task.

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.

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 through 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.

```
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;
```

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

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

82 v

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 the possible airport pairs, not just a single pair: each airport must be connected to all the 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 our demo database satisfies this condition.

```
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;
```

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

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

Solution. To avoid infinite looping, we use the CYCLE clause introduced in PostgreSQL 14.

```
WITH RECURSIVE p(
  last_arrival,
  destination,
  flights,
  flight_time,
  min_time
) AS (
  SELECT a_from.airport_code,
         a_to.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.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  p.flight_time + r.duration <
         coalesce(p.min_time, INTERVAL '1 year')
) CYCLE last_arrival SET is_cycle USING hops
```

```
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;
```

Note that the found route may be suboptimal with regards to the number of connections.

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 remote points, you must take into account the non-trivial shape of the Earth. This task is best performed by the PostGIS extension, which can approximate the Earth surface as a geoid.

But a simple spherical model will also do for our purpose. Let's use the `earthdistance` and then convert the result from 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 the 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 the PostgreSQL installation. But since the `postgres` user is a superuser, it should not be used to connect to the database from an application. It is better to create a new user and make it the owner of a separate database; then its rights will be limited to this database.

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

To learn about users and privileges, see: postgrespro.com/doc/user-manag and postgrespro.com/doc/ddl-priv.

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 can work with the 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:

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

We have to 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 the 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 (clearly, an IP address is not required for local connections).

90 vi

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 a local server.

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.

To allow the `app` user to access the `appdb` database from any address upon providing a valid password, add the following line to the end of the `pg_hba.conf` file:

```
    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 details about authentication settings, see postgrespro.com/doc/client-authentication.html

Pinging the Server

To access PostgreSQL from an application, you have to find 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. The library provides application developers with a convenient way to access low-level features of the protocol.

Below we provide simple code snippets in 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; there is nothing else, not even error handling functionality. Don't take these code snippets as verbatim examples to follow.

If you are using a Windows system, to ensure the correct display of extended character sets, you may need to switch to a TrueType font (such as "Lucida Console" or "Consolas") in the Command Prompt window 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 php-cli php-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 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;
use open ':std', ':utf8';
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 the 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 modern versions of Debian and Ubuntu, Python 3 is preinstalled, so you only need to add the corresponding driver:

```
$ sudo apt-get install python3-psycopg2
```

You can download Python for Windows from the python.org website. The psycopg library is available at initd.org/psycopg (choose the version that corresponds 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:

```
$ python3 test.py

Hello, world!
```

Java

In Java, databases are accessed via the JDBC interface. We are going to install Java SE 11; additionally, we will need a package with the JDBC driver:

```
$ sudo apt-get install openjdk-11-jdk
$ sudo apt-get install libpostgresql-jdbc-java
```

You can download JDK for Windows from 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 durability. 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 contents 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.

The 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 a 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 back up the contents of a single database only. To make a backup of the whole cluster, including all the databases, users, and tablespaces, you should use a 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 the documentation: postgrespro.com/doc/backup.

Built-in PostgreSQL features enable you to implement almost everything you need, but you have to complete multi-step workflows that lack automation. That's why many companies create their own backup tools. Postgres Professional also has such a tool called **pg_probackup,** which is distributed for free. This tool enables you 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.

Its full documentation is available at postgrespro.com/doc/app-pgprobackup.

What's next?

Now you are ready to develop your application. With regards to the database, the application will always consist of two parts: server and client. The server part comprises everything that relates to the database system: tables, indexes, views, triggers, stored functions and procedures. 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 thick 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 move the logic out of the database and implement it all on the client. It often happens when developers are unfamiliar with all the capabilities provided by a relational database system and prefer to rely on what they know well, that is, on the 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 relying on the database system to ensure 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 you 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, but it requires an extra data transfer over the network. Furthermore, such algorithms as scans, joins, sorting, and aggregation provided by database systems are guaranteed to perform better than the application code, as 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 us to start PostgreSQL on virtually any hardware, even on the weakest one. 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 cover only some of the basic settings that must be considered for a 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. 145).

Changing Configuration Parameters

To change a configuration parameter, you have to 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 reload the server configuration:

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

Now check the current setting using the `SHOW` command. If the parameter value has not changed, take a look into the server log: you might have made a mistake when editing the file.

Instead of changing the file in a text editor, you can set the parameter value using an SQL command (it also requires the server configuration to be reloaded):

```
postgres=# ALTER SYSTEM SET work_mem='128MB';
```

Such settings get into the `postgresql.auto.conf` file and take priority over the values specified in the main file. The advantage of such method is that the new parameter values get validated at once.

The Most Important Parameters

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

The **shared_buffers** parameter defines the size of shared buffers, which are used to keep frequently used data in RAM to avoid extra disk access. A reasonable starting value is 25 % of all the 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** parameter defines the amount of memory allocated for service processes. Higher values can speed up indexing and vacuuming. 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, random access is assumed to be four times slower than sequential one (which works well for regular HDDs). For disk arrays and SSDs you should lower the value of the **random_page_cost** parameter (but never change the **seq_page_cost** value, which is 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.

In most cases, you can start with the following configuration:

- 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).

It's equally important to properly configure the processes related to buffer cache and WAL maintenance, but the 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).

To learn tips and tricks for configuring various parameters, you can take the DBA2 course (p. 149).

Connection Settings

We have already covered this topic in the "PostgreSQL for Applications" chapter on p. 87, so here we can simply recall 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 benefits, but it will also lead to garbage accumulation in data and bloating of tables and indexes. Sooner or later your database system is sure to 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 tangible 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 (if you happen to have one).

PostgreSQL and 1C Solutions

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

As any other applications, 1C products will work much faster if PostgreSQL is configured appropriately. Besides, there are specific server 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 patched version of PostgreSQL. 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 50 users, 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 `'*'`.

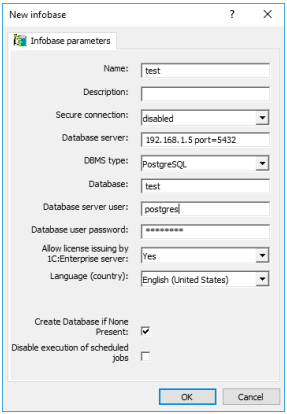
Add the following line at the start of the `pg_hba.conf` configuration file, specifying the actual address and subnet mask instead of the "IP-address-of-the-1C-server" placeholder:

```
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 1C connections.

1C establishes a connection as a superuser, usually `postgres`. Set a password for this role:

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



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 (do not create this database using PostgreSQL means). Provide the name and password of a superuser role that will be used to establish connections.

These recommendations should help you to quickly get started, even though they cannot 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 enables you 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 its redesigned interface, pgAdmin 4 got a cool reception at first, but its development continues, and the product is constantly getting improved.

Installation

To launch pgAdmin 4 on Windows, use the installer available at 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. 28) 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 a dozen of languages. To switch to another 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.

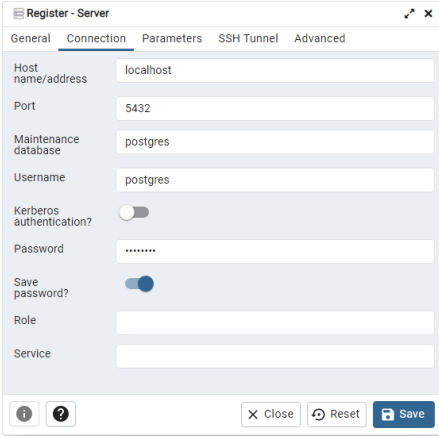
In the **Connection** tab, enter host name or address, port number, username, and password.

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';
```

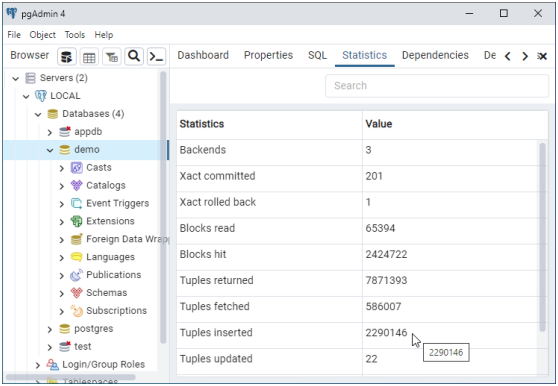
When you click the **Save** button, the application checks that the server with the specified parameters is available, and registers a new connection.



Browser

The left pane displays the Browser tree. As you expand its objects, you can get to the server, which we have called LOCAL. At the next level, you can see all its databases:

- `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.



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

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

The right pane includes several tabs that display reference information:

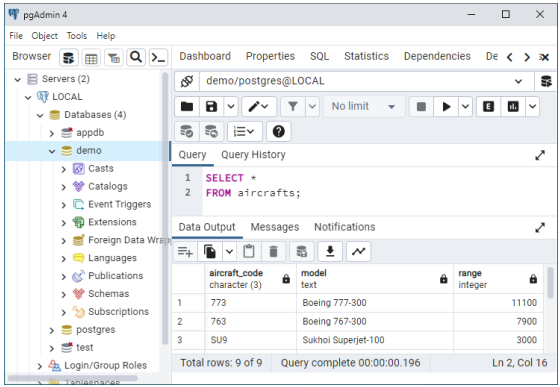
- **Dashboard** provides system activity charts.
- **Properties** displays the properties of the object selected in the Browser (data types for columns, etc.)
- **SQL** shows the SQL command used to create the selected object.

- **Statistics** lists information used by the query optimizer to build query plans; it 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.



To type in the next query, you do not have to delete the previous one: 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 pgAdmin features either on the product website pgadmin.org or in the built-in pgAdmin help system.

IX Additional Features

Full-Text Search

Searching for documents written in natural languages and sorting the results 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. However, despite all the strength of the SQL language, its capabilities are not always enough for effective data handling. It has become especially obvious recently, when avalanches of Big Data, usually poorly structured and hard to parse, started filling data storages.

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 quite decent quality. In this case, an index, which is the most important search tool that can accelerate it, 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 access policies, and many more.

Shortcomings of document-oriented database management systems usually 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 power of SQL accumulated over years.

However, traditional SQL database systems do have built-in full-text search capabilities. The `LIKE` operator is included into the standard SQL syntax, but its flexibility is obviously insufficient. Therefore, 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 the full-text search itself, it's important to understand that they are far from being standardized: each database system uses its own approach and syntax. Russian users of PostgreSQL have some 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 go into low-level details, if required. Here we'll only provide some simple examples.

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

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

CREATE TABLE
```

Now 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      | We start our acquaintance with the
           fascinating world of databases
-[ 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
```

Now let's find some information in our database with the help of traditional SQL means (using the LIKE operator):

```
test=# SELECT ch_no AS no, ch_title, txt
FROM course_chapters
WHERE txt LIKE '%fascination%' \gx
```

It's easy to guess the result: 0 rows. The `LIKE` operator sees no connection between "fascination" and the words "fascinating" and "fascinated" present in the text.

The query

```
test=# SELECT ch_no AS no, ch_title, 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 ]-----------------------------------------
no       | II
ch_title | First Steps
txt      | Getting more fascinated with the world of
           databases
```

PostgreSQL provides the `ILIKE` operator, which allows us not to worry about letter cases; otherwise, you would also have to take uppercase and lowercase letters into account. True, there are also regular expressions (search patterns), and setting them up is a truly engaging task, little short of art, but sometimes you just want a tool that can simply do the job. So let's add one more column to the `course_chapters` table; it will have a special type called `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':10 'fascin':7
            'start':2 'world':9
-[ RECORD 2 ]-------------------------------------
txtvector | 'databas':8 'fascin':3 'get':1 'world':6
-[ RECORD 3 ]-------------------------------------
txtvector | 'adventur':5 'intrigu':9 'journey':6
            'network':12 'start':3 'world':10
```

As we can see, the rows have changed:

- Words are reduced to their unchangeable parts (lexemes).
- Numbers have appeared. They indicate the word position in the text.
- There are no prepositions included (and neither there would be any conjunctions or other parts of the sentence that are unimportant for search; they are the so-called stop words).

The search will be even more efficient if it includes chapter titles, which are also given more weight in respect to the rest of the text (it can be done using the `setweight` function). Let's modify the table:

```
test=# UPDATE course_chapters
  SET txtvector =
      setweight(to_tsvector('english',ch_title),'B')
      || ' ' ||
      setweight(to_tsvector('english',txt),'D');
UPDATE 3
test=# SELECT txtvector FROM course_chapters \gx
-[ RECORD 1 ]-------------------------------------
txtvector | 'acquaint':5 'databas':1B,11 '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 weights when writing queries, which will give us more flexibility.

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 query `'fascinated & database'` and its other grammatical variants will return the same result. Here we have used the comparison operator `@@`, which plays the same role in full-text search as the `LIKE` operator does in regular search. The syntax of the `@@` operator does not allow natural language expressions with spaces, so words in the query are connected by the "and" logical operator.

The `english` argument indicates the configuration used by PostgreSQL. The configuration defines pluggable dictionaries and the parser, 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; it allows search to ignore word endings in queries. You can also plug in other dictionaries, for example:

- regular dictionaries like `ispell`, `myspell`, or `hunspell`, which can better handle word morphology
- dictionaries of synonyms
- thesaurus
- `unaccent`, which can remove diacritics from letters

Thanks to assigned weights, the displayed search results are ranked:

```
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 weights. 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>,
  MaxWords=50, MinWords=5'
)
FROM course_chapters
WHERE to_tsvector('english', txt) @@
      to_tsquery('english', 'world');

-[ RECORD 1 ]----------------------------------------
ts_headline | with the fascinating <b>world</b> of
              databases
-[ 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 the 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: lack of strict consistency support and a much simpler data structure (at first, it was simply a storage of key–value

pairs) provided a remarkable search speedup. Actively using parallel computations, they could process unprecedented volumes of information and were easy to scale.

Once the initial shock had passed, it became clear that for most real tasks such a simple structure was insufficient. 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 typical of NoSQL.

Since changing the database schema in a relational database incurs high costs, a new JSON data type came in handy. Having a hierarchical structure similar to XML, it was first targeted at JavaScript development (hence JS in the title), including AJAX application development. JSON flexibility allowed application developers to add diverse data with unpredictable structure without having to redesign the database schema.

Suppose we need to enter personal data into the `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 at your discretion" and "other" fields. If we followed the traditional approach, the information that does not fit the current structure would require adding multiple new tables and columns with lots of empty fields. As more and more data is being added, the whole database may have to be refactored.

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 sometimes 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 in place. For convenience, we will join the `student_details` and `students` tables using the `WHERE` clause, as 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 the students' merits. Let's 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 have seen 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 fix 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';
```

The new 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;
```

We can see that Victor is a fan of Fender:

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

The `json` type has a younger sibling: `jsonb`. The letter "b" implies the binary (rather than text) format of data storage and structure, which can often result in faster search. Nowadays, `jsonb` is used much more frequently 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"}
```

Apart from a different notation, we can see that 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 than `json`. 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.

130 ix

But what's more important, `jsonb` allows you to create indexes that support the `@>` operator, its inverse `<@`, and many other ones (the GIN index is typically the most efficient). The `json` type does not support indexes, so for high-load applications it is usually recommended to use `jsonb`.

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

However, the `jsonb` functionality was insufficient, 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 its 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.

- `$.a.b.c` replaces the `'a'->'b'->'c'` syntax that had to be used in PostgreSQL 11 or lower.
- The `$` symbol represents the current context element, that is, the JSON fragment that has to be parsed.
- `@` represents the current context element in filter expressions. All the paths available in the `$` expression are searched.
- `*` is a wildcard symbol. In expressions with `$` and `@` it denotes any value of the path taking the hierarchy into account.
- `**` is a wildcard that can denote any part of the path in expressions with `$` or `@`, without taking the hierarchy into account. It comes in handy if you don't know the exact nesting level of the elements.
- The `?` operator is used to create a filter similar to `WHERE`. For example: `$.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 the JSON branch that begins with the "hobbies" key, checking whether the corresponding value equals "cosplay." But if we replace "cosplay" with "guitarist," the query won't return anything because "guitarist" is used in our table as a key, not as a value of the nested element.

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 defined a filter expression for each entry within the "hobbies.guitarist.band" branch. 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. It is an extremely useful feature! Suppose we are not sure 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 whether we have this word in our table at all. Previously, it would have required 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 the documentation (postgrespro.com/docs/postgresql/12/datatype-json#DATATYPE-JSONPATH) and in the article

134 ix "JSONPath in PostgreSQL: committing patches and selecting apartments" (habr.com/en/company/postgrespro/blog/500440/).

Integration with External Systems

Real-life applications are not isolated, and they often have to send data to each other. Such interactions can be implemented at the application level, for example, using web services or file exchange, or you can rely on the database functionality for this purpose.

PostgreSQL supports the ISO/IEC 9075-9 standard (SQL/MED, Management of External Data), which defines access to 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, as 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 itself. 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. To get the full list, take a look at pgxn.org/tag/fdw.

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

- Oracle — github.com/laurenz/oracle_fdw;
- MySQL — github.com/EnterpriseDB/mysql_fdw;
- SQL Server — github.com/tds-fdw/tds_fdw.

Follow the instructions on these web pages to build and install these extensions, and this process should run smoothly.

If all goes 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 | 1.2
```

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 of 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 following queries on the 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)
```

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 we can also 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.

PostgreSQL

Create an extension and a wrapper:

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

```
CREATE EXTENSION
```

We are going to connect to another database of the same server instance, so we only have to provide the `dbname` parameter when creating a foreign server. Other parameters (such as `host`, `port`, etc.) can be omitted.

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

```
CREATE SERVER
```

There is no need to specify the password if you create a user mapping within a single 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 the documentation: postgrespro.com/doc/postgres-fdw.

Foreign data wrappers are also worth mentioning as the community considers them to be 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 miscellaneous enhancements. You can also use foreign tables as partitions, which virtually turns partitioning into sharding.

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

- Consistency is not guaranteed: external data is accessed in separate local transactions rather than 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; you can download it at github.com/postgrespro/shardman.

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

X Education and Certification

Documentation

Reading the documentation is indispensable for professional use of PostgreSQL. It describes all the database features and provides an exhaustive reference that should always be readily available. Here 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. The PostgreSQL documentation is available at www.postgresql.org/docs or postgrespro.com/docs.

We at Postgres Professional have translated into Russian the whole PostgreSQL documentation set, including the latest version. It is available on our website: postgrespro.ru/docs.

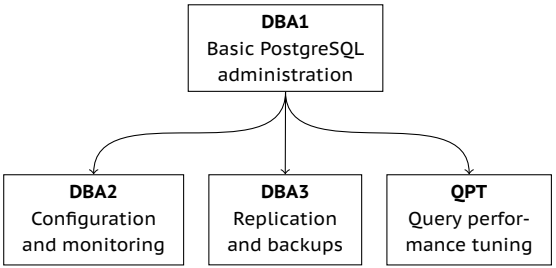
While working on this translation, we also compiled an English-Russian glossary, published at postgrespro.com/education/glossary. We recommend consulting this glossary when translating English articles into Russian to use consistent terminology familiar to a wide audience.

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

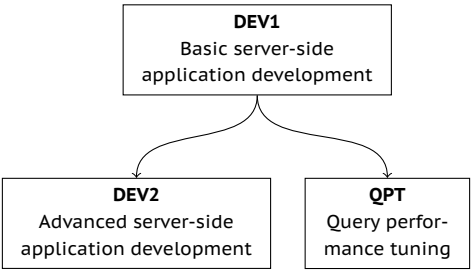
Training Courses

We develop training courses for those who start using PostgreSQL or would like to improve their professional skills.

Courses for database administrators:



Courses for application developers:



The documentation contains all the conceivable details about PostgreSQL, but the information is often scattered

across different chapters, so you may have to carefully read it several times before you gain full understanding.

Training courses are intended to complement the documentation rather than replace it. They consist of separate modules that gradually explain a particular topic, focusing on important practical information. Courses can broaden your outlook, structure the previously gained bits of knowledge, and help you find your way around the documentation, should you need to quickly get some particular details.

Each course topic includes theory and practice. In most cases, theory includes both slides and a live demo on a real system. Students get all the slides with extensive comments, outputs 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/community/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 will receive a certificate of completion. Authorized training centers are listed here: 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 structure and physical data layout.
Basic administration tasks.
User and access management.
Backup, recovery, and replication.

Topics:

Basic toolkit

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

Architecture

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

Data management

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

Administration tasks

12. Monitoring

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 postgrespro.ru/education/courses/DBA1. Their English version for a shortened two-day introductory course is available at postgrespro.com/community/courses/2dINTRO.

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 row 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 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 cluster technologies.

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

Cluster Technologies

8. Overview

Course materials in Russian are available for self-study at 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. A general overview of PostgreSQL
3. Isolation and MVCC
4. Buffer cache and WAL

Data organization

5. Logical structure
6. Physical structure

Bookstore application

7. Application schema and interface

SQL

8. Functions

9. Procedures
10. Composite types

PL/pgSQL

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

Access control

19. Access control overview

Backup

20. Logical backup

Course materials in English are available for self-study at postgrespro.com/community/courses/DEV1.

DEV2. Advanced server-side application development

Duration: 4 days

Background knowledge required:

General understanding of PostgreSQL architecture.
Strong SQL and PL/pgSQL skills.
Basic knowledge of Unix OS.

Knowledge and skills gained:

Understanding server internals.

Using all PostgreSQL capabilities in application logic implementations.
Extending database functionality to address specific tasks.

Topics:

Architecture

1. Isolation
2. Server internals
3. Vacuum
4. Write-ahead logging
5. Locks

Bookstore

6. Bookstore application 2.0

Extensibility

7. Connection pooling
8. Data types for large values
9. User-defined data types
10. Operator classes
11. Semi-structured data
12. Background processes
13. Asynchronous processing
14. Creating extensions
15. Programming languages
16. Aggregate and window functions
17. Full-text search
18. Physical replication
19. Logical replication
20. Foreign data

Course materials in Russian are available for self-study at postgrespro.ru/education/courses/DEV2.

QPT. Query Performance Tuning

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 DBAs) 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.

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 postgrespro.ru/education/courses/QPT.

Professional Certification

The certification program, which was launched in 2019, is useful for both database professionals and their employers. If you have a certificate, you are likely to get additional points when hunting for a job or negotiating your salary. Besides, it's a good opportunity to get an impartial evaluation of your knowledge.

For employers, certification program facilitates recruiting, enables verification of 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 we plan to launch certification programs for PostgreSQL application developers too.

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.

Master level additionally confirms practical skills required 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.

To pass a test, you should have:

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

While taking the test, you can refer to our course materials and the PostgreSQL documentation, but usage of 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. So in several years you may want to take a test for a more recent PostgreSQL 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 the courses can be used in educational institutions for free. Lecturers can use textbooks, slides, lecture videos, and other educational materials published on our website: postgrespro.ru/education/university.

Postgres Professional has contributed to several courses read in such universities as 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 are a university representative and would like to add database courses to 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, students will gradually get to more complex constructs, learn about transactions and query optimization.

This course is based on the following textbook (published in Russian):

Morgunov, E. *PostgreSQL. SQL Basics*. St. Petersburg : BHV-Petersburg, 2018.

ISBN 978-5-9775-4022-3

Contents:

A soft copy of this book in Russian is available on our website: 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.

Evgeny lives in Krasnoyarsk. Before joining the University in 2000, he had been working as a programmer for more than 10 years; among other things, he had been developing a banking application system. He got to learn 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. Evgeny is a member of the International Society for Engineering Pedagogy (IGIP). He has been using PostgreSQL in teaching for more than 20 years.

Database Technology Fundamentals

A modern academic course that combines in-depth theory with relevant practical skills of database design and deployment.

Novikov, B., Gorshkova, E., and Grafeeva N. *Database Technology Fundamentals*. 2nd ed. Moscow : DMK Press, 2020.

ISBN 978-5-97060-841-8

The first part contains the key information about database management systems: relational data model, the SQL language, transaction processing.

The second part dives into the underlying database technology and its development trends. Some topics covered in the first part are discussed again at a deeper level.

Contents:

Part I. From Theory to Practice

Database system architecture
Storage structures and the main algorithms
Query execution and optimization
Transaction management
Database reliability
Advanced SQL features
Database functions and procedures
PostgreSQL extensibility
Full-text search
Data security
Database administration
Replication
Parallel and distributed database systems

A soft copy of this book in Russian is available on our website: postgrespro.ru/education/books/dbtech.

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 deploy-

ing 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.

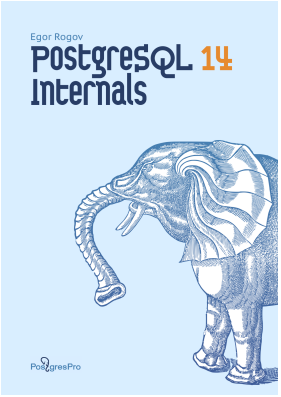
Books

PostgreSQL Internals

This book is for those who will not settle for a black-box approach when working with a database. Targeted at readers who have some experience with PostgreSQL, this book will also be useful for those who are familiar with another database system but switch over to PostgreSQL and would like to understand how they differ.

Rogov E. *PostgreSQL 14 Internals*. Moscow : DMK Press, 2022

ISBN 978-5-6045970-4-0 (in English)
ISBN 978-5-93700-122-1 (in Russian)



You will not find any ready-made recipes in this book. But the provided explanations of the inner mechanics will enable you to critically evaluate other people's experience and come to your own conclusions. The author goes into details of PostgreSQL internals and shows how to run experiments and verify information that inevitably gets deprecated.

Egor Rogov has been working in the education department in Postgres Professional since 2015; he develops and teaches training courses, publishes blog posts, writes and edits books.

Contents:

Part III. Locks

Relation-level locks • Row-level locks • Miscellaneous locks • Locks on memory structures

Part IV. Query execution

Query execution stages • Statistics • Table access methods • Index access methods • Index scans • Nested loop • Hashing • Sorting and Merging

Part V. Index types

Hash • B-tree • GiST • SP-GiST • GIN • BRIN

A soft copy of this book is available on our website: postgrespro.com/community/books/internals.

XI The Hacker's Guide to the Galaxy

News and Discussions

Anyone can follow PostgreSQL news, learn about the features planned for the next release, and stay up-to-date with the current events.

Plenty of interesting and useful content is published in various related blogs. For example, the planet.postgresql.org website aggregates all the English-language articles in one place. 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/. There are also dedicated YouTube channels, such as youtube.com/RuPostgres and youtube.com/PostgresTV.

There is also a Wiki project (wiki.postgresql.org), where you can find FAQ, training materials, articles about system setup and optimization, migration specifics from different database systems, and much more.

More than 9 000 Russian-speaking PostgreSQL users are subscribed to the "pgsql—PostgreSQL" channel in Telegram (t.me/

pgsql); more than 4 000 people are members of the Facebook group "PostgreSQL in Russia" (facebook.com/groups/postgresql).

You can also ask your questions on stackoverflow.com. Do not forget to add the "postgresql" tag.

As for Postgres Professional news, they are published in its corporate blog at postgrespro.com/blog.

Mailing Lists

To get all the news firsthand, without waiting for someone to write a blog post, you can subscribe to mailing lists. Respecting the tradition, PostgreSQL developers discuss all questions exclusively by email.

You can find all the mailing lists at postgresql.org/list. Some of them are:

- pgsql-hackers (typically called simply "hackers"), the main list for everything related to development
- pgsql-general used to discuss general questions
- pgsql-bugs for bug reports
- pgsql-docs for documentation
- pgsql-translators for translation-related discussions
- pgsql-announce to get new release announcements

and many more.

Having signed up for any of these lists, you will start receiving regular emails and will be able to participate in discussions if you like. Another option is to browse through the email

archive at postgresql.org/list or on our company's website (postgrespro.com/list).

Commitfest

Another way to keep up with the news without spending too much time is to check the commitfest.postgresql.org page. Here the community opens the so-called commitfests for developers to submit their patches. For example, commitfest 01.03.2022–31.03.2022 was open for version 15, while commitfest 01.07.2022–31.07.2022 was related to the next version already. It allows the community 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, and then they are either accepted, or moved to the next commitfest, or rejected (if you are completely out of luck).

This way, 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)

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

Besides, several Russian cities host conferences on broader topics, including databases in general and PostgreSQL in particular. We will name only a few:

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 Ottawa, Canada (pgcon.org)

PGConf Europe (pgconf.eu)

The list of upcoming events can be found at postgresql.org/about/events.

In addition to conferences, there are less official regular meetups, including online ones.

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 database development expertise in Russia, the company currently employs about 150 developers, architects, and engineers.

The Postgres Professional company delivers several versions of Postgres Pro database system 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 Database System

Postgres Pro is a Russian commercial database system 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 the PostgreSQL features and additional extensions and core patches, including those that are not yet accepted by the community. As a result, its users can get access to useful functionality and improve performance without having to wait for the next PostgreSQL version to be released.

Postgres Pro Enterprise is a considerably reworked version of the database system; offering better stability and increased performance, it can address 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.

Services

Fault-Tolerant Solutions for Postgres

Designing and implementing high-load, high-performance, and fault-tolerant 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 both 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 database administrators, system architects, and application developers covering Postgres specifics and efficient use of 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
Edited by Peter Lagutkin
Cover design by Alexander Gruzdev

9th edition, revised and updated

postgrespro.com/community/books/introbook

ISBN 978-5-6045970-3-3