Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Logical and physical backups

Backup and restore of separate tables

Backup and restore of separate databases

Backup and restore of the whole cluster

Logical Backup

SQL commands to create objects and fill them with data

+ backup of a separate object or a database
+ recovery on a different architecture or PostgreSQL version (binary compatibility is not required)
+ ease of use
− modest operation speed
− no point-in-time recovery

A logical backup is a set of SQL commands that can restore the database cluster (or a separate database/table) from scratch: it creates all the required objects and fills them with data.

These commands can be run on a different server version (if it provides compatibility at the command level) or on a different platform/architecture (binary compatibility is not required).

In particular, a logical backup can be used for long-term storage: you can restore it even after upgrading the server to a higher version.

The process of creating a logical backup is relatively easy. It is usually enough to run a single command or launch a single utility.

But for large databases, the execution of these commands can take a very long time. Using a logical backup, you can restore your database system only to its state exactly at the time the process of taking a backup was started.

https://postgrespro.com/docs/postgresql/12/backup-dump

Physical Backup

A copy of the database cluster's file system

+ faster than logical backup
+ statistics are restored
− recovery is only possible on a compatible system, with the same PostgreSQL major version installed
− partial backup is impossible, the whole cluster is copied

WAL archive

+ point-in-time recovery is available

A physical backup implies creating a copy of all files related to the database cluster, i.e., creating its full binary copy.

It is faster to copy files than dump SQL commands; besides, unlike restoring a logical backup, starting a server using a physical copy is a matter of several minutes. Another advantage is that you do not have to recollect statistical data: it is also restored from a physical copy.

But this approach has its own shortcomings. A physical backup can be used to restore the system only on a compatible platform (that has the same OS, architecture, etc.) with the same PostgreSQL major version installed. Besides, it is impossible to create a physical copy of a separate database: you can only back up the whole database cluster.

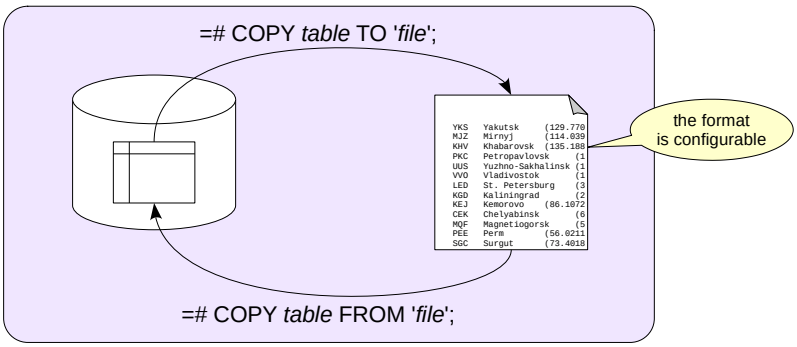
Physical backups are usually used together with WAL archives. It enables system recovery not only at the time of backup creation, but also at an arbitrary point in time.

https://postgrespro.com/docs/postgresql/12/backup-file

https://postgrespro.com/docs/postgresql/12/continuous-archiving

Creating physical backups for any important production systems is a common practice. It is the responsibility of a DBA to take such backups.

Making a Table Copy in SQL



the file is located in the server file system and can be accessed by the owner of the PostgreSQL instance

you can specify the columns to copy (or use an arbitrary query)

new rows are added to already existing ones during recovery

5

If you only need to save the contents of a single table, you can use the `COPY` command.

The `COPY TO` flavor of this command enables you to save the table (or some of its columns, or even the result of an arbitrary query) into a file, display it in the terminal, or provide it as input to an application. You can also specify some additional parameters, such as the format (plain text, CSV, or binary), delimiter characters, text representation of NULL values, etc.

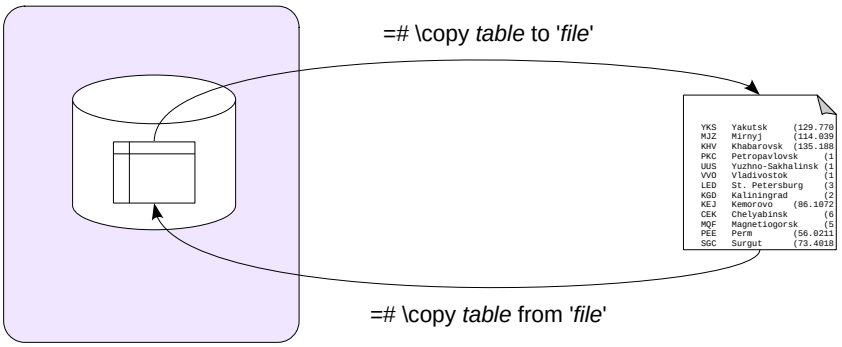
The `COPY FROM` flavor does the opposite: it reads data from a file or from the terminal and inserts the retrieved rows into the table. The table is not cleared in this case: new rows are simply appended to the already existing ones.

The `COPY` command is much faster than the analogous `INSERT` commands: the client does not have to access the server multiple times, and the server does not have to repeatedly analyze the received commands.

Here is a subtle point: the `COPY FROM` command ignores the defined rules, although integrity constraints and triggers are respected.

https://postgrespro.com/docs/postgresql/12/sql-copy

Making a Table Copy in psql



the file is located in the client file system and can be accessed by the OS user who has started psql

the data is transferred between the client and the server

the syntax and the supported features are analogous to those provided by `COPY`

6

The psql utility provides a client version of the `COPY` command with a similar syntax.

The file name provided in the `COPY` command corresponds to the file on the database server. The user on whose behalf PostgreSQL is started (usually `postgres`) must have access to this file.

The client implementation of this command refers to the file located on the client, which allows keeping a local copy of data even if there is no access to the server file system. The table contents is automatically sent between the client and the server.

https://postgrespro.com/docs/postgresql/12/app-psql

The COPY Command

Let's create a database and a table.

```
=> CREATE DATABASE db1;
```

```
CREATE DATABASE
```

```
=> \c db1
```

```
You are now connected to database "db1" as user "student".
```

```
=> CREATE TABLE t(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    s text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO t(s) VALUES ('Hello, world!'), (''), (NULL);
```

```
INSERT 0 3
```

Here is the output of the COPY command (we'll display it in the terminal instead of saving it into a file):

```
=> COPY t TO stdout;
```

```
1       Hello, world!
2
3       \N
```

You can see the difference between empty rows and null values in the output.

The output format is quite flexible. You can change the delimiter symbol, null value representation, etc. For example:

```
=> COPY t TO stdout WITH (null '<NULL>', delimiter ',');
```

```
1,Hello\, world!
2,
3,<NULL>
```

Note that the delimiter inside the row has been escaped (the escape symbol is also configurable).

Instead of the table name, you can specify an arbitrary query.

```
=> COPY (SELECT * FROM t WHERE s IS NOT NULL) TO stdout;
```

```
1       Hello, world!
2
```

This way, you can save the result of a query, the contents of a view, etc.

This command can also return the output in the CSV format, which is supported by many programs.

```
=> COPY t TO stdout WITH (format csv);
```

```
1,"Hello, world!"
2,""
3,
```

Data input from a file or from the terminal works quite similar to data output.

The input from the terminal requires an end-of-file indicator: a backslash followed by a dot. It is not required for a regular file.

All parameters provided on input must match those specified on output.

```
=> TRUNCATE TABLE t;
```

```
TRUNCATE TABLE
```

```
=> COPY t FROM stdin;
1       Hello, world!
2
3       \N
\.
```

```
COPY 3
```

Here is what has been loaded into the table (psql is configured to display null values for visual clarity):

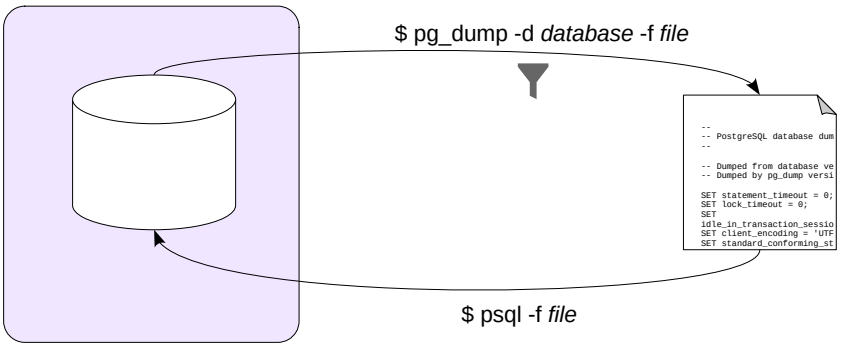
```
=> \pset null '\\N'

Null display is "\N".

=> SELECT * FROM t;

 id |       s
----+---------------
  1 | Hello, world!
  2 |
  3 | \N
(3 rows)
```

A Database Backup



format: SQL commands
you can specify the database objects to be dumped
the new database must be cloned from `template0`
roles and tablespaces must be created in advance
it makes sense to perform `ANALYZE` after recovery

To create a full-fledged backup of a database, use the pg_dump utility.

If you omit the file name (-f, --file), the utility's output will be displayed in the terminal. The produced output is a script to be run in psql; it contains the commands that will create the required objects and fill them with data.

You can use optional parameters to limit the set of backed up objects: for example, you can choose to back up only particular tables, objects in particular schemas, or use some other filters.

To restore the objects from the backup, it is enough to run the received script in psql.

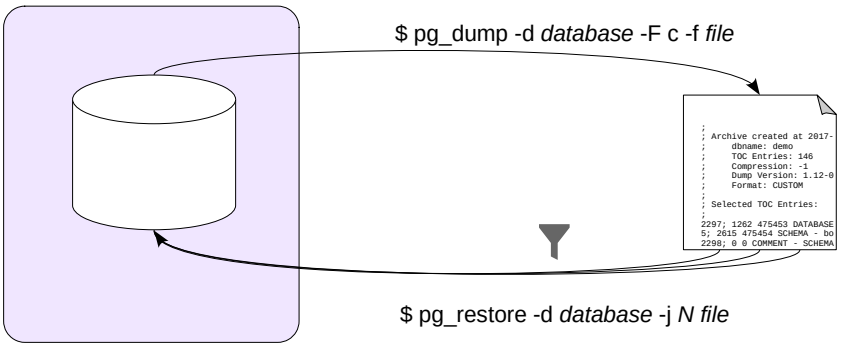
Note that the database to be restored should be cloned from `template0` since all the changes made in `template1` will also make it into the backup.

Besides, all the required roles and tablespaces must be set up in advance. Since these objects do not belong to any particular database, they won't be included into the dump.

Once the database is restored, it makes sense to run the `ANALYZE` command: it will collect statistics that the optimizer requires for query planning.

https://postgrespro.com/docs/postgresql/12/app-pgdump

The custom Format



an internal format with a table of contents (TOC)
database objects to be restored can be selected at the time of recovery
recovery can be performed in the parallel mode

The pg_dump utility allows you to specify the backup format. By default, the plain format is used; it provides pure psql commands.

The custom format (-F c, --format=custom) creates a backup in a special format that contains not only the backed up objects, but also a table of contents (TOC). Having a TOC allows you to choose the objects to be restored right at the time of recovery, not while making the dump.

By default, the output of the custom format is compressed.

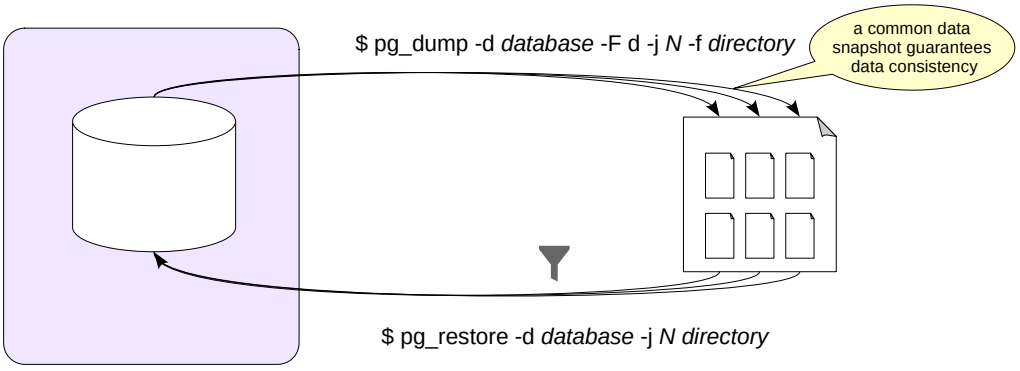
To restore the database, you need to run another utility: pg_restore.
It reads the file and converts it to psql commands. If you do not explicitly provide the database name (in the -d option), all commands will be output to the terminal. If the database is specified, pg_restore will connect to this database and execute the commands; you won't have to start psql.

To restore only some of the objects, you can use one of the following approaches. The first one is to filter the objects to be restored, just like it is done in pg_dump. In fact, pg_restore supports many pg_dump parameters.

The second option is to use the TOC to retrieve the list of objects included into the backup (via the --list option). Then you can edit this list manually: delete the objects you do not need and pass the modified list to pg_restore (via the --use-list option).

https://postgrespro.com/docs/postgresql/12/app-pgrestore.html

The directory Format



the directory contains a separate file for each database object and a TOC
database objects to be restored can be selected at the time of recovery
both dump and restore operations can be performed in the parallel mode

You can also create backups in the directory format. In this case, pg_dump produces a whole directory instead of a single file; it contains the backed up objects and the table of contents. By default, all files in the directory are compressed.

Its advantage over the custom format is that such a backup can be created concurrently using several processes (the number of processes is specified in option -j, --jobs).

Naturally, the backup will contain consistent data even though it has been created concurrently. Consistency is ensured by using a single data snapshot for all parallel processes.

https://postgrespro.com/docs/postgresql/12/functions-admin.html#FUNCTIONS-SNAPSHOT-SYNCHRONIZATION

Data recovery can also be performed in the parallel mode (it is also supported for the custom format).

Other capabilities are quite similar to those provided by the previously discussed formats: the directory format supports the same options and approaches.

Format Comparison

	plain	custom	directory	tar
recovery utility	psql	pg_restore		
compression		zlib		
partial restore		yes	yes	yes
parallel backup			yes	
parallel restore		yes	yes	

This slide compares different features provided by different backup formats.

Note that there is also one more format available: tar. We do not cover it here as it does not bring anything new and has no advantages as compared to other formats. In fact, this format is simply a tar-ed version of the directory format, but it does not support compression or parallel execution.

The pg_dump Utility

If started without optional parameters, pg_dump utility outputs SQL commands that recreate all objects of the database:

```
student$ pg_dump -d db1

--
-- PostgreSQL database dump
--

-- Dumped from database version 12.8 (Ubuntu 12.8-1.pgdg20.04+1)
-- Dumped by pg_dump version 12.8 (Ubuntu 12.8-1.pgdg20.04+1)

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

SET default_tablespace = '';

SET default_table_access_method = heap;

--
-- Name: t; Type: TABLE; Schema: public; Owner: student
--

CREATE TABLE public.t (
    id integer NOT NULL,
    s text
);


ALTER TABLE public.t OWNER TO student;

--
-- Name: t_id_seq; Type: SEQUENCE; Schema: public; Owner: student
--

ALTER TABLE public.t ALTER COLUMN id ADD GENERATED ALWAYS AS IDENTITY (
    SEQUENCE NAME public.t_id_seq
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1
);


--
-- Data for Name: t; Type: TABLE DATA; Schema: public; Owner: student
--

COPY public.t (id, s) FROM stdin;
1       Hello, world!
2
3       \N
.


--
-- Name: t_id_seq; Type: SEQUENCE SET; Schema: public; Owner: student
--

SELECT pg_catalog.setval('public.t_id_seq', 3, true);


--
-- Name: t t_pkey; Type: CONSTRAINT; Schema: public; Owner: student
--

ALTER TABLE ONLY public.t
    ADD CONSTRAINT t_pkey PRIMARY KEY (id);
```

```
--
-- PostgreSQL database dump complete
--
```

As you can see, pg_dump has created table t, enabled automatic ID generation, filled the table with data using the already discussed COPY command, and finally added the primary key constraint. The --column-inserts option allows using INSERT commands, but the restore operation will take much longer.

Let's take a look at some useful options.

These options can help you restore a copy on a system with a different set of roles:

- -O, --no-owner - do not generate commands that set object ownership;
- -x, --no-acl - do not generate commands for granting privileges.

These options are useful for partial dump/restore operations:

- -s, --schema-only - dump only object definitions without actual data;
- -a, --data-only - dump only data without creating any objects.

The first parameter of the following two is used to restore a backup on a system that already contains some data; the second one should be used on a clean system:

- -c, --clean - generate DROP commands for the created objects;
- -C, --create - generate commands to create the database and to connect to this database.

An important note: all changes made in the template1 database also make it into the dump. So it is better to restore the backup on a database cloned from template0. The --create option automatically takes it into account:

```
student$ pg_dump --create -d db1 | grep 'CREATE DATABASE'

CREATE DATABASE db1 WITH TEMPLATE = template0 ENCODING = 'UTF8' LC_COLLATE = 'en_US.UTF-8' LC_CTYPE = 'en_US.UTF-8';
```

There are several options for filtering objects that must be included into the backup:

- -n, --schema - a pattern for schema names;
- -t, --table - a pattern for table names.

And vice versa, the following options can be used to backup everything except the specified objects:

- -N, --exclude-schema - a pattern for schema names;
- -T, --exclude-table - a pattern for table names.

For example, let's restore table t in a different database.

```
=> CREATE DATABASE db2;

CREATE DATABASE

student$ pg_dump --table=t -d db1 | psql -d db2

SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
ALTER TABLE
COPY 3
 setval
--------
      3
(1 row)

ALTER TABLE
```

Now let's connect to database db2 and check the result:

```
=> \c db2
```

```
You are now connected to database "db2" as user "student".
```

```
=> SELECT * FROM t;
```

```
 id |       s
----+---------------
  1 | Hello, world!
  2 |
  3 | \N
(3 rows)
```

The pg_dump Utility: Custom Format

A major limitation of the plain format is that the objects to back up must be selected at the time of taking the dump. The custom format allows you to first create a full copy and then select the objects to restore.

```
student$ pg_dump --format=custom -d db1 -f /home/student/db1.custom
```

To restore objects from such a backup you can use the pg_restore utility. Let's restore table t once again.

```
=> DROP TABLE t;
```

```
DROP TABLE
```

You can omit the backup format parameter: the utility will recognize it anyway.

The pg_restore utility can take the same filtering options as pg_dump, and even more:

- -I, --index - restore the specified indexes;
- -P, --function - restore the specified functions;
- -T, --trigger - restore the specified triggers.

```
student$ pg_restore --table=t -d db2 /home/student/db1.custom
```

```
=> SELECT * FROM t;
```

```
 id |       s
----+---------------
  1 | Hello, world!
  2 |
  3 | \N
(3 rows)
```

Here is another example: let's restore the whole db1 database in its initial state.

```
=> DROP DATABASE db1;
```

```
DROP DATABASE
```

The -d option is used to specify any existing database; if the --create option is provided, pg_restore will automatically create the database specified in the archive and connect to it.

```
student$ pg_restore --create -d student /home/student/db1.custom
```

Let's check the result:

```
=> \c db1
```

```
You are now connected to database "db1" as user "student".
```

```
=> SELECT * FROM t;
```

```
 id |       s
----+---------------
  1 | Hello, world!
  2 |
  3 | \N
(3 rows)
```

A backup in the plain format can be modified in a text editor, if required. A backup in the custom format is stored in a binary format, but it provides wider capabilities for filtering objects than the options discussed above. The pg_restore utility can produce the list of all objects, i.e., the table of contents of the backup:

```
student$ pg_restore --list /home/student/db1.custom
```

```
;
; Archive created at 2021-10-19 17:04:51 MSK
;     dbname: db1
;     TOC Entries: 9
;     Compression: -1
;     Dump Version: 1.14-0
;     Format: CUSTOM
;     Integer: 4 bytes
```

```
;     Offset: 8 bytes
;     Dumped from database version: 12.8 (Ubuntu 12.8-1.pgdg20.04+1)
;     Dumped by pg_dump version: 12.8 (Ubuntu 12.8-1.pgdg20.04+1)
;
;
; Selected TOC Entries:
;
203; 1259 16944 TABLE public t student
202; 1259 16942 SEQUENCE public t_id_seq student
2962; 0 16944 TABLE DATA public t student
2969; 0 0 SEQUENCE SET public t_id_seq student
2834; 2606 16951 CONSTRAINT public t t_pkey student
```

You can save this list into a file, edit it, and use it to perform recovery via the --use-list option.

The pg_dump Utility: Directory Format

The directory format is worth noting because it allows you to perform the recovery in several parallel threads. Data consistency is guaranteed: all the threads will be using one and the same data snapshot.

```
student$ pg_dump --format=directory --jobs=2 -d db1 -f /home/student/db1.directory
```

Let's take a look inside the directory:

```
student$ ls -l /home/student/db1.directory
```

```
total 8
-rw-rw-r-- 1 student student   49 Oct 19 17:04 2961.dat.gz
-rw-rw-r-- 1 student student 1995 Oct 19 17:04 toc.dat
```

It contains the table of contents file and one file per each object to be restored (we have only one object here):

```
student$ zcat /home/student/db1.directory/2961.dat.gz
```

```
1       Hello, world!
2
3       \N
.
```

Let's restore the db1 database from a backup using two threads.

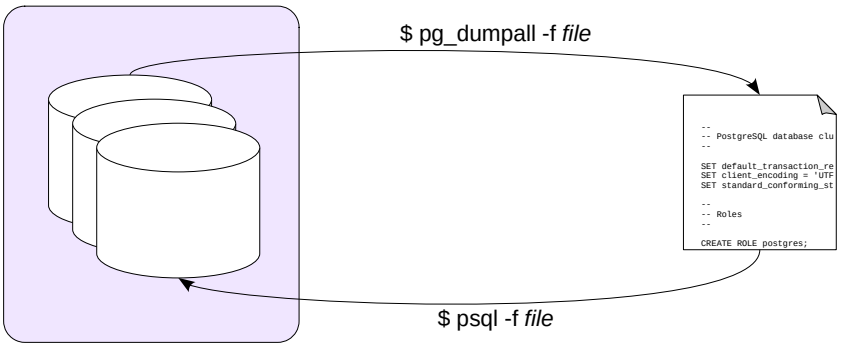
Here we have to add the --clean option to generate a command that will drop the database, because db1 already exists. You must first disconnect from db1:

```
=> \c db2
```

```
You are now connected to database "db2" as user "student".
```

```
student$ pg_restore --clean --create --jobs=2 -d student /home/student/db1.custom
```

A Database Cluster Backup



format: SQL commands
dumps the whole cluster, including roles and tablespaces
the user must have access to all objects of the database cluster
parallel backups are not supported

To back up the whole cluster, including roles and tablespaces, you can use the pg_dumpall utility.

Since pg_dumpall requires access to all objects of all databases, it make sense to run it on behalf of a superuser. The utility connects to each database and dumps their contents using pg_dump. Besides, it also saves cluster-wide data.

To start this process, pg_dumpall has to establish a connection with any available database. By default, either `postgres` or `template1` is selected, but you can also specify a different database.

The pg_dumpall utility produces a psql script. This is the only supported format. It means that pg_dumpall cannot perform parallel dumps, and it can turn out to be a problem for large volumes of data. In this case, you can use the --globals-only option to dump only roles and tablespaces, while all databases will be dumped separately using pg_dump in the parallel mode.

https://postgrespro.com/docs/postgresql/12/app-pg-dumpall

The pg_dumpall Utility

While the pg_dump utility is good for dumping a single database, it cannot dump common cluster objects, such as roles or tablespaces. To take a full backup of a cluster, you have to use pg_dumpall.

None of the utilities covered in this lecture require any specific privileges, but the role that runs them must have the right to read (or create) all the affected objects. For example, pg_dump can be run by the database owner. But to back up the whole cluster, the pg_dumpall utility must have access to all the databases, so you should use a role with superuser privileges.

```
student$ pg_dumpall --clean -f /home/student/main.sql
```

The cluster backup also includes such commands as:

```
student$ grep 'ROLE' main.sql
```

```
DROP ROLE buyer;
DROP ROLE employee;
DROP ROLE postgres;
DROP ROLE student;
CREATE ROLE buyer;
ALTER ROLE buyer WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS PASSWORD 'md546cbf9abe0323700ba0e419091271507';
CREATE ROLE employee;
ALTER ROLE employee WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS PASSWORD 'md59c0967753a201ecde21ef29efa514761';
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN REPLICATION BYPASSRLS PASSWORD 'md53175bce1d3201d16594cebf9d7eb3f9d';
CREATE ROLE student;
ALTER ROLE student WITH SUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS PASSWORD 'md550d9482e20934ce6df0bf28941f885bc';
```

Recovery is performed using psql; this is the only supported format. Here is the restore command (we are not going to run it):

```
student$ psql -f main.sql
```

During recovery, you might see some errors caused by already existing objects. It's OK in most cases: such errors usually do not interfere with the recovery process, although you should still analyze all messages to be on the safe side.

Summary

Logical backups can be taken for the whole cluster, a particular database, or separate database objects

Are good for

- small amounts of data
- long-term storage during which the server can been upgraded
- migration to a different platform

Are not so good for

- crash recovery with minimal data loss

Practice

1. Back up the `bookstore` database in the custom format.

   “Accidentally” empty the `authorship` table. Check that the application has stopped displaying book titles in “Bookstore,” “Books,” and “Catalog” tabs.

   Use the created backup to restore the lost data.

   Check that normal operation of the bookstore is restored.

Task 1. Use the --data-only option for the restore operation as an attempt to create a table will result in an error.

Task 1. Data Recovery

Create a backup:

```
student$ pg_dump --format=custom -d bookstore > /home/student/bookstore.custom
```

Delete some rows:

```
=> DELETE FROM authorship;
```

```
DELETE 10
```

Perform the recovery:

```
student$ pg_restore -t authorship --data-only -d bookstore /home/student/bookstore.custom
```

```
=> SELECT count(*) FROM authorship;
```

```
 count
-------
    10
(1 row)
```

Practice

1. Create a table with a policy that allows reading only some of the rows. Create an unprivileged role for Alice and grant her access to this table.

   Alice is responsible for creating table backups. Can she do it without superuser rights? Try it out.
2. The \copy command provided by psql enables you to pass the result as input to an arbitrary application. Use this capability to open the result of some query in the Calc spreadsheet of LibreOffice.

Task 1. While superuser roles bypass RLS policies, Alice won't be able to access some of the table rows, without even knowing that the received data is incomplete.

If the *row_security* parameter is set, Alice will be notified that some data has not been selected. Granting the BYPASSRLS privilege to this role will solve the problem.

Task 2. The command must save the result into a file and then start libreoffice with this file passed as a parameter. The file must be saved in the CSV format.

Naturally, this approach is platform-dependent and will require modifications, say, on Windows.

Task 1. Row-Level Security Policies

Create a table with a policy and a role.

```
=> CREATE DATABASE backup_logical;
```

```
CREATE DATABASE
```

```
=> \c backup_logical
```

```
You are now connected to database "backup_logical" as user "student".
```

```
=> CREATE TABLE t(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    s text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO t(s) VALUES ('foo'), ('bar'), ('baz');
```

```
INSERT 0 3
```

```
=> CREATE POLICY odd ON t USING (mod(id,2) = 1);
```

```
CREATE POLICY
```

```
=> ALTER TABLE t ENABLE ROW LEVEL SECURITY;
```

```
ALTER TABLE
```

```
=> CREATE ROLE alice LOGIN PASSWORD 'alicepass';
```

```
CREATE ROLE
```

```
=> GRANT SELECT ON t TO alice;
```

```
GRANT
```

Alice tries to access the table:

```
student$ psql "host=localhost user=alice dbname=backup_logical password=alicepass"
```

```
alice=> COPY t TO stdout; -- or SELECT * FROM t;
```

```
1       foo
3       baz
```

With the row_security parameter set to off, Alice will get an error if policies forbid her access to some of the rows:

```
alice=> SET row_security = off;
```

```
SET
```

```
alice=> COPY t TO stdout;
```

```
ERROR:  query would be affected by row-level security policy for table "t"
```

To bypass RLS policies as a non-superuser, Alice has to receive the BYPASSRLS attribute:

```
=> ALTER ROLE alice BYPASSRLS;
```

```
ALTER ROLE
```

```
alice=> COPY t TO stdout;
```

```
1       foo
2       bar
3       baz
```

Task 2. Opening Query Results in LibreOffice

Try this command:

```
\copy t TO PROGRAM 'cat > /home/student/t.csv; libreoffice /home/student/t.csv' WITH (format csv);
```

If you replace \copy with the COPY command available in SQL, the program will be started on the database server, which is certainly incorrect.