

Backup Logical Backup



Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Alexey Beresnev

Photo: Oleg Bartunov (Phu Monastery and Bhrikuti Peak, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

What is Logical Backup

Backup and Restore of Individual Tables

Backup and Restore of Databases

Backup and Restore of the Whole Cluster

SQL commands that create objects and fill them with data

- + can back up a separate object or database
- + can be restored to a different version or architecture (binary compatibility is not required)
- modest operation speed
- no point-in-time recovery

A logical backup contains a set of SQL commands that can be used to restore a cluster, database, or specific tables.

During the recovery process, the necessary objects are created from the logical backup and filled with data.

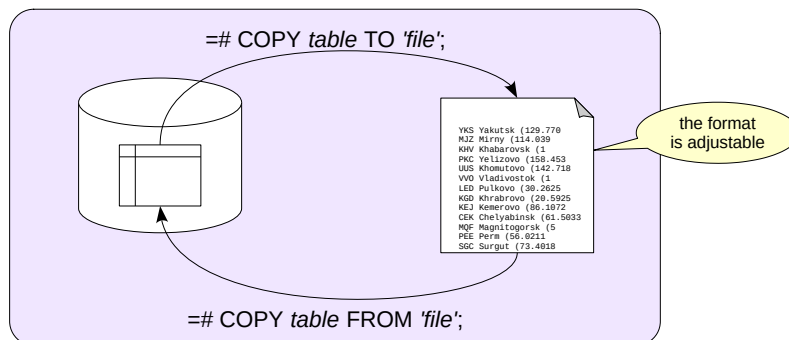
These commands can be run on a different server version or on a different platform or architecture (binary compatibility is not required).

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.

However, for a large database, these commands may take a while to execute. In addition to that, a logical backup can only restore your database to the point in time when the backup process was initiated.

<https://postgrespro.com/docs/postgresql/16/backup-dump>

The SQL COPY command



file in the server file system owned by the PostgreSQL instance owner
you can select specific columns or use any query you need
new rows are added to already existing ones during recovery

4

The COPY TO command writes the contents of a table or a query result to a file or an output stream. Rows of views or partitioned tables can be copied using a query.

COPY TO requires the `pg_write_server_file` role.

The COPY FROM command adds rows from a file or standard input stream to the table. It is significantly faster than similar INSERT commands. You can monitor the process in the `pg_stat_progress_copy` view.

COPY FROM requires the `pg_read_server_files` role (or read input stream `pg_execute_server_program`).

Note: the COPY FROM command ignores the defined rules, although integrity constraints and triggers are respected. If a table column should receive values generated by a sequence when inserted, but COPY fills that column explicitly, the sequence will not change its state (counter).

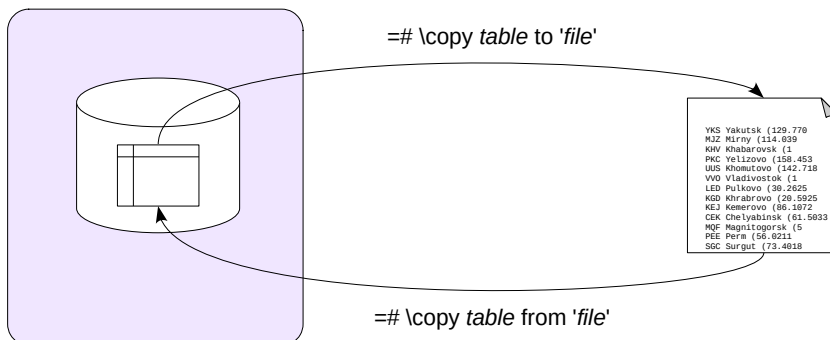
Both variants of the COPY command allow you to select the columns of the table

and specify options such as:

- **FORMAT** – data format (text, CSV or binary);
- **DELIMITER** – field delimiter;
- **NULL** – NULL text representation;
- **ENCODING** – encoding;
- **HEADER** – header describing the columns.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

The psql \copy Command



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 the same as with COPY

There is a \copy meta command in psql – a variant of the COPY command with the same syntax – that you can run on the client side.

The file must be available to the OS user running psql. The file is accessed on the client, and only the content is sent to the server.

<https://postgrespro.com/docs/postgresql/16/app-psql#APP-PSQL-META-COMMANDS-COPY>

The COPY command

Create a database and a table.

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

CREATE DATABASE

```
α=> \c backup_logical
```

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

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

CREATE TABLE

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

INSERT 0 3

```
α=> SELECT * FROM t;
```

id	s
1	Hello World!
2	
3	

(3 rows)

Print the table contents:

```
α=> COPY t TO stdout;
```

1	Hello World!
2	
3	\N

You can see how empty rows are different from the NULL values.

The output format can be configured quite flexibly. You can change the separator, the NULL display, etc. For example:

```
α=> COPY t TO stdout WITH (NULL '<NULL>', DELIMITER ',');
```

```
1,Hello World!
2,
3,<NULL>
```

Note that the inline separator character has been escaped (the escape character is also configurable).

Instead of a table, you can specify an arbitrary query.

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

1	Hello World!
2	

This way, you can save query results, presentation data, etc.

The command supports output in CSV format, which is supported almost universally.

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

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

Input from file or console works in a similar manner.

```
α=> TRUNCATE TABLE t;
```

TRUNCATE TABLE

However, for input from console, the end of file character (a backslash and a period) is required.

In order for the data to load, you must specify the same parameters during input as were specified during output.

You can also add:

```
α=> COPY t FROM stdin WHERE id != 2;
1      Привет, мир!
2
3      \N
\.
```

COPY 2

Here are the table contents (with NULL output configured in psql for clarity):

```
α=> \pset null '\\N'
```

Null display is "\\N".

```
α=> SELECT * FROM t;
```

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

Load all the rows:

```
α=> TRUNCATE TABLE t;
```

TRUNCATE TABLE

```
α=> COPY t FROM stdin;
```

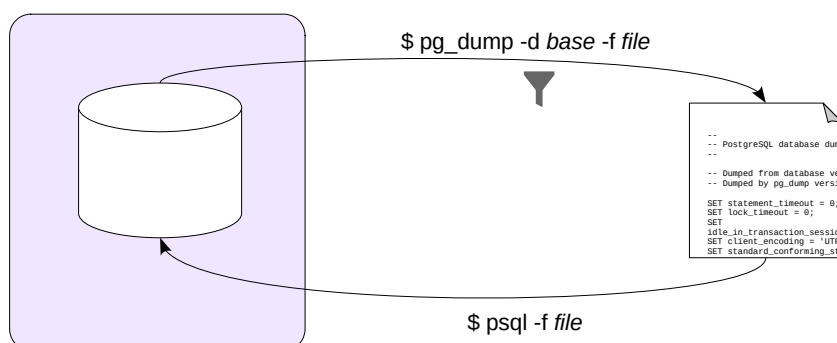
```
1      Привет, мир!
2
3      \N
\.
```

COPY 3

```
α=> SELECT * FROM t;
```

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

Database Backup



format: SQL commands

you can specify the database objects to be dumped

the new database must be created from the template0 template

roles and tablespaces must be created in advance

you should usually run ANALYZE after recovery

7

The `pg_dump` utility creates a full-scale database backup.

If you do not specify a file name (`-f`, `--file`), the utility will display the result to the standard output stream.

The produced output is a script to be run in `psql`; it contains the commands that will create the objects and fill them with data.

Additional keys can limit the set of objects: select specified tables or all objects in specified schemas, upload only object definitions or only data, etc.

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 is a good idea to run the `ANALYZE` command: it will collect statistics that the optimizer requires for query planning.

<https://postgrespro.com/docs/postgresql/16/app-pgdump>

pg_dumpall Utility

pg_dump, ran without additional parameters, issues SQL commands that create all objects in the current database:

To get commands to recover to a system with a different set of roles, use the options:

- -O, --no-owner — do not generate commands to set the owner of objects;
- -x, --no-acl — do not generate commands to set privileges

```
student$ pg_dump -d backup_logical --no-acl

--
-- PostgreSQL database dump
--

\restrict gJnV8i7WLYQP14BMyhefNauB3FcoS7S01BBdaZb0zLEf6c8cdx6v10As0g7GdGX

-- Dumped from database version 16.14 (Ubuntu 16.14-1.pgdg24.04+1)
-- Dumped by pg_dump version 16.14 (Ubuntu 16.14-1.pgdg24.04+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    Привет, мир!
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
--
```

```
\unrestrict gJnV8i7WLYQP14BMyhefNauB3FcoS7S01BBdaZb0zLEf6c8cdx6vL0As0g7GdGX
```

Here, `pg_dump` has created the table `t` and populated it using the `COPY` command from before. The `\--column-inserts` flag enables the use of `INSERT` commands, but the process will take significantly longer.

Logical backups in plain format may include malicious metacommands, for example, `\gexec`. To avoid such issues when running the script in `psql`, `pg_dump` prepends it with the `\restrict` metacommand with the key. This command switches on the restricted mode, making `psql` ignore all metacommands except for `\unrestrict` with the same key.

The keys are handy for dumping and uploading data in segments:

- `-s, --schema-only` — back up only object definitions without data
- `-a, --data-only` — back up only data without creating objects

For recovering to a system already containing some data (or, on the other hand, to a completely clean system):

- `-c, --clean` — generate `DROP` commands for created objects
- `-C, --create` — generate commands for creating a database and connecting to it

Important: changes to the `'template1'` database are dumped too. Therefore, you should recover to a database created from `'template0'`. The `\--create` flag takes it into account automatically:

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

CREATE DATABASE backup_logical WITH TEMPLATE = template0 ENCODING = 'UTF8'
LOCALE_PROVIDER = libc LOCALE = 'en_US.UTF-8';
```

Some flags specify the objects to be dumped:

- `-n, --schema` — template for schema names
- `-t, --table` — template for table names

Or, conversely, not to be, while dumping everything else:

- `-N, --exclude-schema` — template for schema names
- `-T, --exclude-table` — template for table names

For example, recover the table `'t'` to another database on another server.

```
student$ psql -p 5433
| => CREATE DATABASE backup_logical_1;
| CREATE DATABASE

student$ pg_dump --table=t -d backup_logical | psql -p 5433 -d backup_logical_1

SET
SET
SET
SET
SET
set_config
-----

(1 row)

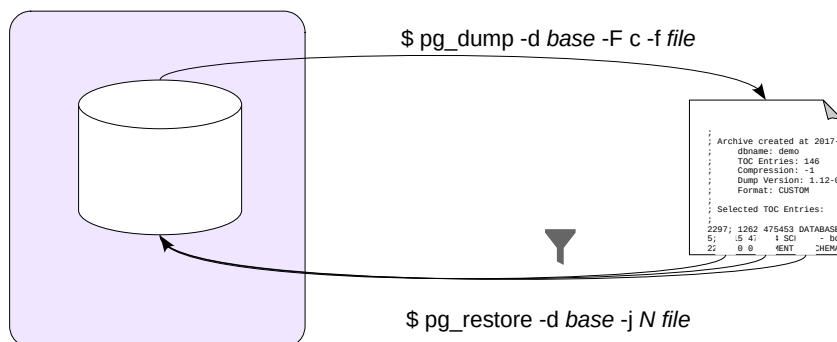
SET
SET
SET
SET
SET
SET
CREATE TABLE
```

```
ALTER TABLE
ALTER TABLE
COPY 3
  setval
-----
      3
(1 row)
```

```
ALTER TABLE
```

```
|  β=> \c backup_logical_1
|
| You are now connected to database "backup_logical_1" as user "student".
|
|  β=> SELECT * FROM t;
|
|  id |      s
|-----+-----
|   1 | Привет, мир!
|   2 |
|   3 |
| (3 rows)
```

The Custom Format



an internal format with a table of contents

database objects to be restored can be selected at the time of recovery

recovery can be performed in parallel mode

9

The `pg_dump` utility supports several backup formats. By default, the plain format is used; it provides pure `psql` commands.

The custom (`-F c`, `--format = custom`) format creates a backup in a special format containing not only objects, but also a table of contents. 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 the custom format output 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 standard output stream. If you do specify a database name, the utility will connect to that database and execute the commands directly, without using `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`. `pg_restore` actually supports many of the same options as `pg_dump`.

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).

Uploading can be performed in parallel by specifying the number of threads with the `-j` switch – this is another advantage of the custom format.

<https://postgrespro.com/docs/postgresql/16/app-pgdump>

pg_dump: the 'custom' format

The 'plain' format has a serious limitation: you have to specify all the objects you will need in your backup before making the dump. The 'custom' format allows you to make a full dump first, then specify which objects to recover from it later.

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

This is done with the pg_restore tool. Time to recover 't' again.

```
| β=> DROP TABLE t;
|
| DROP TABLE
```

```
student$ pg_restore --table=t -p 5433 -d backup_logical_1 --no-acl
/home/student/tmp/backup_logical.custom
```

pg_restore will recognize the dump format automatically, there is no need to specify it.

pg_restore also understands all the flags that pg_dump does, and more:

- -I, --index — load specific indexes
- -P, --function — load certain functions
- -T, --trigger — load specific triggers.

Let's check how the table recovered:

```
| β=> SELECT * FROM t;
|
|  id |      s
|----+-----
|  1 | Привет, мир!
|  2 |
|  3 |
| (3 rows)
```

Another example: restore the entire original backup_logical database to another server.

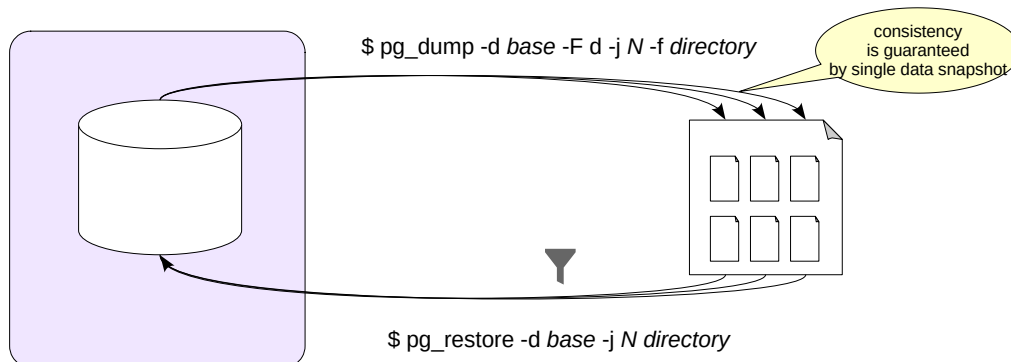
```
student$ pg_restore --create -p 5433 -d postgres --no-acl /home/student/tmp/backup_logical.custom
```

Here, we have specified the 'postgres' database. If another database was specified, pg_restore would have created the necessary database and immediately switched to it.

Check:

```
| β=> \c backup_logical
|
| You are now connected to database "backup_logical" as user "student".
|
| β=> SELECT * FROM t;
|
|  id |      s
|----+-----
|  1 | Привет, мир!
|  2 |
|  3 |
| (3 rows)
```

The Directory Format



the directory contains a TOC file and a separate file for each database object
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

11

Another backup format is Directory. Using this format, `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/16/functions-admin#FUNCTIONS-SNAPSHOT-SYNCHRONIZATION>

Recovery, as with the custom format, supports parallel streaming.

Otherwise, working with the directory format offers the same capabilities as the formats discussed earlier: it supports the same options and approaches.

pg_dump: the 'directory' format

The 'directory' format, notably, supports dumping in parallel.

```
student$ pg_dump --format=directory \
--jobs=2 \
-d backup_logical \
-f /home/student/tmp/backup_logical.directory
```

Consistency is guaranteed: all processes use the same snapshot.

Take a look inside the directory:

```
student$ ls -l /home/student/tmp/backup_logical.directory

total 12
-rw-rw-r-- 1 student student 60 Sep 16 23:52 3440.dat.gz
-rw-rw-r-- 1 student student 4985 Sep 16 23:52 toc.dat
```

There is a table of contents file, and a file for each dumped object (we have only one):

```
student$ zcat /home/student/tmp/backup_logical.directory/3440.dat.gz

1      Привет, мир!
2
3      \N
\.
```

To restore from a backup, first disconnect from the backup_logical database:

```
| β=> \q
```

The '--clean' key generates a command to delete the existing database:

```
student$ pg_restore --clean \
--create \
--jobs=2 \
-p 5433 \
-d postgres /home/student/tmp/backup_logical.directory \
--no-acl

student$ psql --pset null='\N' -p 5433 -d backup_logical -c 'SELECT * FROM t'

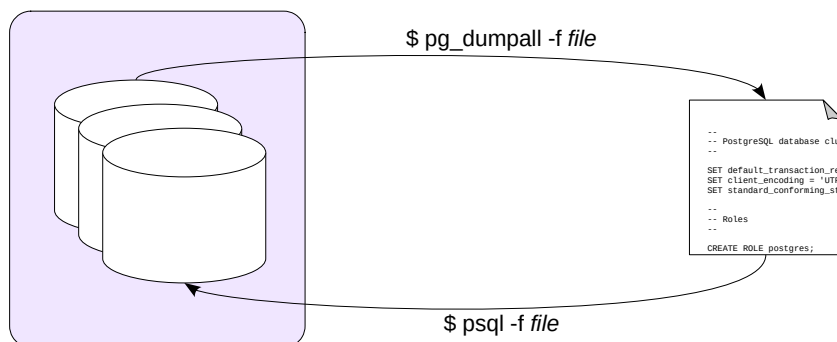
 id |      s
-----+-----
  1 | Привет, мир!
  2 |
  3 | \N
(3 rows)
```

Format Comparison

	plain	custom	directory	tar
utility for of recovery	psql	pg_restore		
compression	gzip, lz4, or zstd			
partial restore		possible	possible	possible
parallel backup			possible	
parallel restore		possible	possible	

This slide compares different features provided by different backup formats. There is also a fourth format – tar. It was not discussed here, since it does not introduce anything new and does not provide advantages over 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.

Cluster Backup



format: SQL commands

dumps the whole cluster, including roles and tablespaces

the user must have access to all objects in the database cluster

parallel backups are not supported

14

`pg_dumpall` creates a backup of the entire cluster, including roles and tablespaces.

Since `pg_dumpall` requires access to all objects of all databases, it should be ran by the superuser. `pg_dumpall` connects to each database in the cluster one by one and backups them using `pg_dump`. In addition, it also stores data related to the cluster as a whole.

To start this process, `pg_dumpall` has to establish a connection with a database.

By default, `postgres` or `template1` is selected, but you can specify another one.

The result of `pg_dumpall` is a script for `psql`. Other formats are not supported. This means that `pg_dumpall` does not support parallel execution, which can be a problem for larger clusters. In this case, you can use the `--globals-only` key to backup only roles and table spaces, and unload the databases themselves separately using parallel `pg_dump`.

<https://postgrespro.com/docs/postgresql/16/app-pg-dumpall>

The pg_dumpall Utility

The pg_dump utility is good for dumping a single database, but it never dumps shared database cluster objects such as roles and tablespaces. To create a full cluster backup, you need to use pg_dumpall.

The pg_dumpall, pg_dump, and pg_restore utilities do not require any separate privileges, but the role performing them must have privileges to read (create) all affected objects. For example, pg_dump can be used by the database owner. But since it is necessary to have access to all databases to copy the cluster, we run pg_dumpall as the superuser.

```
student$ pg_dumpall --clean -U postgres -f /home/student/tmp/alpha.sql
```

The backup includes:

- commands that the pg_dump backs up for each database
- commands for shared cluster objects (roles and tablespaces)

Commands for shared objects can be obtained separately:

```
student$ pg_dumpall --clean --globals-only -U postgres
```

```
--
-- PostgreSQL database cluster dump
--

\restrict e0B6e88kQ1UH20c5WvNCp9aKNhv1EmAjMAub1ie8JhGQUF6v4A2nX04FH1ZQzZa

SET default_transaction_read_only = off;

SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;


--
-- Drop roles
--

DROP ROLE postgres;
DROP ROLE student;


--
-- Roles
--

CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN REPLICATION
BYPASSRLS PASSWORD
'SCRAM-SHA-256$4096:AsXBHJhp+lTo5cMw8rTwVA==$w9/d0pa/3sFrh+sLgBYnALVw+k7RETzghidz4i0F850=:
A0khfLkZ1ZZrQUD39l/Uiro+tu/0FoR0E0k9UNyzqx0=';
CREATE ROLE student;
ALTER ROLE student WITH NOSUPERUSER INHERIT CREATEROLE CREATEDB LOGIN REPLICATION
NOBYPASSRLS PASSWORD
'SCRAM-SHA-256$4096:79AZA/AciQ7/18/dXmAZZQ==$iy9Rop0HolUKGAK/JK2K/pUdhj2GmsZLTgbWc8vH6sc=:
ZiyLRbFp1NxhgDuzBGithNum7+5zdbGG8XeAZJ6MR7A=';


--
-- User Configurations
--


--
-- Role memberships
--

GRANT pg_checkpoint TO student WITH INHERIT TRUE GRANTED BY postgres;
GRANT pg_create_subscription TO student WITH INHERIT TRUE GRANTED BY postgres;
GRANT pg_monitor TO student WITH INHERIT TRUE GRANTED BY postgres;
GRANT pg_read_server_files TO student WITH INHERIT TRUE GRANTED BY postgres;
GRANT pg_write_server_files TO student WITH INHERIT TRUE GRANTED BY postgres;


--
-- Role privileges on configuration parameters
--
```

```
\unrestrict e0B6e88kQ1UH20c5WvNCp9aKNhv1EmAjMAub1ie8JhGQUF6v4A2nX04FH1ZQzZa
```

Recovery is done via `psql`, there is no other way around it.

[illegible]

```
SET
SET
SET
SET
  set_config
-----
```

(1 row)

```
SET
SET
SET
SET
UPDATE 1
DROP DATABASE
CREATE DATABASE
ALTER DATABASE
You are now connected to database "template1" as user "postgres".
SET
SET
SET
SET
SET
  set_config
-----
```

(1 row)

```
SET
SET
SET
SET
COMMENT
ALTER DATABASE
You are now connected to database "template1" as user "postgres".
SET
SET
SET
SET
SET
  set_config
-----
```

(1 row)

```
SET
SET
SET
SET
REVOKE
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
  set_config
-----
```

(1 row)

```
SET
SET
SET
SET
CREATE DATABASE
ALTER DATABASE
You are now connected to database "backup_logical" as user "postgres".
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
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
SET
SET
SET
SET
  set_config
-----
```

(1 row)

```
SET
SET
SET
SET
DROP DATABASE
CREATE DATABASE
ALTER DATABASE
You are now connected to database "postgres" as user "postgres".
SET
SET
SET
SET
SET
  set_config
-----
```

(1 row)

```
SET
SET
SET
SET
COMMENT
SET
SET
SET
SET
SET
  set_config
-----
```

(1 row)

```
SET
SET
SET
SET
CREATE DATABASE
```

```

ALTER DATABASE
You are now connected to database "student" as user "postgres".
SET
SET
SET
SET
SET
  set_config
-----

(1 row)

SET
SET
SET
SET
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT

```

During recovery, errors related to irremovable objects or objects already existing may come up. These are normal and do not affect recovery in this case.

```
student$ psql -p 5433
```

```

β=> SELECT datname FROM pg_database;

    datname
-----
template0
backup_logical_1
template1
backup_logical
postgres
student
(6 rows)

β=> \c backup_logical
You are now connected to database "backup_logical" as user "student".

β=> SELECT * FROM t;

 id |      s
----+-----
  1 | Привет, мир!
  2 |
  3 |
(3 rows)

```

The impact of row level security

If some tables have row level security on (more on that in the Row Level Security topic in the DBS course), you could end up with incomplete data and not even notice. To avoid that, before running COPY, set `row\_security` to `off`. This way, when a row level security policy triggers, the process will throw an explicit error.

A simple example. Set up a policy that prohibits displaying empty rows, then enable it for the table owner.

```
α=> CREATE POLICY t_s_not_null ON t USING (s IS NOT NULL);
```

```
CREATE POLICY
```

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

```
ALTER TABLE
```

```
α=> ALTER TABLE t FORCE ROW LEVEL SECURITY;
```

```
ALTER TABLE
```

Now COPY returns just two rows:

```
α=> COPY t TO stdout;
```

```
1      Привет, мир!
2
```

But when the parameter is set to off, we get an error:

```
α=> SET row_security = off;
```

```
SET
```

```
α=> COPY t TO stdout;
```

```
ERROR:  query would be affected by row-level security policy for table "t"
HINT:   To disable the policy for the table's owner, use ALTER TABLE NO FORCE ROW LEVEL
SECURITY.
```

Both pg_dump and pg_dumpall automatically set `row\_security = off`, no additional actions needed:

```
student$ pg_dump -d backup_logical -t t > /dev/null
```

```
pg_dump: error: query failed: ERROR:  query would be affected by row-level security
policy for table "t"
HINT:   To disable the policy for the table's owner, use ALTER TABLE NO FORCE ROW LEVEL
SECURITY.
pg_dump: detail: Query was: COPY public.t (id, s) TO stdout;
```

The tool attempts to COPY the rows and returns the error: unable to copy all rows from t.

If you want to copy just the rows that are available, you can use `--enable-row-security`:

```
student$ pg_dump -d backup_logical -t t --enable-row-security | grep -A3 '^COPY'
```

```
COPY public.t (id, s) FROM stdin;
1      Привет, мир!
2
\.
```

To bypass row level security, the user must have the BYPASSRLS attribute:

```
α=> \c - postgres
```

You are now connected to database "backup_logical" as user "postgres".

```
α=> ALTER USER student BYPASSRLS;
```

```
ALTER ROLE
```

Now, all queries by `student` ignore RLS policies:

```
student$ pg_dump -d backup_logical -t t --data-only
```

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

\restrict wtjlko2okrK5ZUoPNI4415UxcGzgIzm5dFN3ump9Pbcy0rMHALPxQvSN500tX8d

-- Dumped from database version 16.14 (Ubuntu 16.14-1.pgdg24.04+1)
-- Dumped by pg_dump version 16.14 (Ubuntu 16.14-1.pgdg24.04+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;

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

COPY public.t (id, s) FROM stdin;
1      Привет, мир!
2
3      \N
\.
```

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

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

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

```
\unrestrict wtj1ko2okrK5ZUoPNI4415UxcGzgIzm5dFN3ump9Pbcy0rMHALPxQvSN500tX8d
```

You can make a logical backup of a whole cluster, a specific database or a database object

Good for

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

Not so good for

- crash recovery with minimal data loss

1. Create two databases with tables and views, use `pg_dumpall` to dump global cluster objects only, then create database backups using `pg_dump` in parallel mode.
Restore the entire cluster on a different server using the backups you created.
2. Create a table with two columns, one of which is the primary key containing a number sequence. Simulate a case when during uploading with the `COPY` command, new sequence values are not generated, leading to a unique constraint violation error when attempting to insert new rows.

2. Declare a column as `GENERATED ALWAYS AS IDENTITY`.

After completing the practice tasks, delete the created databases and logical backup files.

1. Logical Backup

Let's create some databases, a couple of tables, and some views.

```
α=> CREATE DATABASE backup_logical1;
```

```
CREATE DATABASE
```

```
α=> \c backup_logical1
```

You are now connected to database "backup_logical1" as user "student".

```
α=> CREATE TABLE t1(n integer);
```

```
CREATE TABLE
```

```
α=> INSERT INTO t1 VALUES (1), (2), (3);
```

```
INSERT 0 3
```

```
α=> CREATE VIEW v1 AS SELECT * FROM t1;
```

```
CREATE VIEW
```

```
α=> CREATE DATABASE backup_logical2;
```

```
CREATE DATABASE
```

```
α=> \c backup_logical2
```

You are now connected to database "backup_logical2" as user "student".

```
α=> CREATE TABLE t2(n integer);
```

```
CREATE TABLE
```

```
α=> INSERT INTO t2 VALUES (1), (2), (3);
```

```
INSERT 0 3
```

```
α=> CREATE VIEW v2 AS SELECT * FROM t2;
```

```
CREATE VIEW
```

Back up the global objects.

```
student$ pg_dumpall --clean --globals-only -U postgres -f /home/student/tmp/alpha_globals.sql
```

Parallel logical database backups:

```
student$ pg_dump --jobs=2 \  
--format=directory \  
-d backup_logical1 \  
-f /home/student/tmp/backup_logical1.directory
```

```
student$ pg_dump --jobs=2 \  
--format=directory \  
-d backup_logical2 \  
-f /home/student/tmp/backup_logical2.directory
```

Now restore the cluster. Restore global objects first:

```
student$ psql -p 5433 -U postgres -f /home/student/tmp/alpha_globals.sql 2> /dev/null
```

```
SET  
SET  
SET  
ALTER ROLE  
ALTER ROLE  
GRANT ROLE  
GRANT ROLE  
GRANT ROLE  
GRANT ROLE  
GRANT ROLE  
GRANT  
GRANT  
GRANT  
GRANT  
GRANT  
GRANT  
GRANT
```

GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT
GRANT

Then restore the two databases in parallel:

```
student$ pg_restore -p 5433 \  
-U postgres \  
-d postgres \  
--create \  
--jobs=2 \  
/home/student/tmp/backup_logical1.directory
```

```
student$ pg_restore -p 5433 \  
-U postgres \  
-d postgres \  
--create \  
--jobs=2 \  
/home/student/tmp/backup_logical2.directory
```

Check:

```
student$ psql -p 5433
```

```
| β=> \c backup_logical1  
| You are now connected to database "backup_logical1" as user "student".
```

```
| β=> \d  
|  
|          List of relations  
| Schema | Name | Type  | Owner  
|-----+-----+-----+-----  
| public | t1   | table | student  
| public | v1   | view  | student  
| (2 rows)
```

```
| β=> \c backup_logical2  
| You are now connected to database "backup_logical2" as user "student".
```

```
| β=> \d  
|  
|          List of relations  
| Schema | Name | Type  | Owner  
|-----+-----+-----+-----  
| public | t2   | table | student  
| public | v2   | view  | student  
| (2 rows)
```

Drop the databases we created.

```
α=> \c postgres
```

You are now connected to database "postgres" as user "student".

```
α=> DROP DATABASE backup_logical1;
```

DROP DATABASE

```
α=> DROP DATABASE backup_logical2;
```

DROP DATABASE

2. COPY with a Sequence-Based Table

A table with two columns:

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

CREATE DATABASE

```
α=> \c backup_logical
```

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

```
α=> CREATE TABLE t(
      id int PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
      txt text
    );
```

CREATE TABLE

A table and a sequence have been created:

```
α=> \d
```

```

              List of relations
Schema |   Name   | Type  | Owner
-----+-----+-----+-----
public | t         | table | student
public | t_id_seq | seq   | student
(2 rows)
```

Load rows without specifying values for the first column.

```
α=> COPY t(txt) FROM STDIN;
```

Один

Два

Три

\.

COPY 3

The table contents are as expected:

```
α=> SELECT * FROM t;
```

```

 id | txt
----+----
  1 | Один
  2 | Два
  3 | Три
(3 rows)
```

The sequence counter is correct:

```
α=> SELECT currval('t_id_seq');
```

```

 currval
-----
        3
(1 row)
```

Now insert another row:

```
α=> INSERT INTO t(txt) VALUES('Four');
```

INSERT 0 1

When using INSERT, you cannot specify a value for a column declared as GENERATED ALWAYS. It must be generated by the sequence:

```
α=> INSERT INTO t(id,txt) VALUES(5,'Five');
```

```

ERROR:  cannot insert a non-DEFAULT value into column "id"
DETAIL:  Column "id" is an identity column defined as GENERATED ALWAYS.
HINT:   Use OVERRIDING SYSTEM VALUE to override.
```

However, COPY allows this:

```
α=> COPY t FROM STDIN;
```

```

5      Пять
6      Шесть
7      Семь
\.
```

COPY 3

At first glance, everything looks fine...

```
α=> SELECT * FROM t;
```

```

id | txt
---+-----
1 | Один
2 | Два
3 | Три
4 | Four
5 | Пять
6 | Шесть
7 | Семь
(7 rows)

```

But it is not:

```
α=> INSERT INTO t(txt) VALUES('Eight');
```

```

ERROR:  duplicate key value violates unique constraint "t_pkey"
DETAIL:  Key (id)=(5) already exists.

```

The primary key violation occurred because COPY does not use the sequence:

```
α=> SELECT currval('t_id_seq');
```

```

currval
-----
5
(1 row)

```

Let's fix this by updating the sequence state.

```
α=> SELECT setval('t_id_seq', max(id)) FROM t;
```

```

setval
-----
7
(1 row)

```

Now the insert will succeed:

```
α=> INSERT INTO t(txt) VALUES('Eight');
```

```
INSERT 0 1
```

```
α=> SELECT * FROM t;
```

```

id | txt
---+-----
1 | Один
2 | Два
3 | Три
4 | Four
5 | Пять
6 | Шесть
7 | Семь
8 | Eight
(8 rows)

```

Cleanup

Drop the database.

```
α=> \c postgres
```

You are now connected to database "postgres" as user "student".

```
α=> DROP DATABASE backup_logical;
```

```
DROP DATABASE
```

1. Create a table with multiple rows. Select such \copy parameters so that when writing to a file, the correct data and NULL values are mixed up due to unsuccessful code selection.
2. Using \copy, create a text file that denotes the missing values as N/A. Load rows from this file into a table where missing values are NULL.
3. Create a data table several MB in size and get the sizes of logical backups in the custom format with different compression methods. Also measure the time required for each compression method during the backup process.

1. If you mistakenly choose a NULL code that matches actual data values in the table columns, the resulting backup will prevent proper data restoration.

3. A way to generate the data:

```
SELECT (random()::numeric*(g.i))^(g.i) AS n
```

```
FROM generate_series(1,2_500) AS g(i);
```

You can measure the time using this command: `/usr/bin/time -f% e pg_dump...`

After completing the practice tasks, delete the created databases and temporary files.

1. Breaking COPY

Create a database.

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

CREATE DATABASE

```
α=> \c backup_logical
```

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

For this example, set the display of undefined values that match some data:

```
α=> CREATE TABLE anticopy(s text);
```

CREATE TABLE

```
α=> INSERT INTO anticopy(s) VALUES ('N'), (NULL);
```

INSERT 0 2

Set the null value representation to N:

```
α=> \copy anticopy to '/home/student/tmp/anticopy.dump' with (null 'N');
```

COPY 2

The output of two different values now looks identical. Properly recovering data from such a copy is impossible.

```
student$ cat ${HOME}/tmp/anticopy.dump
```

```
N
N
```

Be careful when changing the default format!

2. Loading Data with Missing Values

Create a file where missing values are marked as N/A.

```
α=> \copy (VALUES (0,'N/A'), (1,'One'), (2,'N/A'), (3,'Three')) to /home/student/tmp/backup_logical.txt
```

COPY 4

```
α=> \! cat /home/student/tmp/backup_logical.txt
```

```
0      N/A
1      One
2      N/A
3      Three
```

Create a table to load the data into.

```
α=> CREATE TABLE t (k int, v text);
```

CREATE TABLE

Load the data from the file into the table. The N/A value should be loaded as NULL.

```
α=> \copy t from /home/student/tmp/backup_logical.txt null 'N/A'
```

COPY 4

```
α=> SELECT *, v IS NULL "IsNull" FROM t;
```

k	v	IsNull
0		t
1	One	f
2		t
3	Three	f

(4 rows)

3. Compression in pg_dump

Create a large enough table.

```

α=> CREATE TABLE mb AS
      SELECT (random()::numeric*(g.i))^(g.i) AS n
      FROM generate_series(1,2_500) AS g(i);

SELECT 2500

α=> SELECT pg_size_pretty(pg_table_size('mb'));

pg_size_pretty
-----
5128 kB
(1 row)

```

Measure the time it takes to dump data without compression (and warm up the cache in the process).

```

student$ /usr/bin/time -f %e pg_dump \
--format=custom --dbname=backup_logical \
--compress=none --file=/home/student/tmp/backup_logical.none

0.45

```

How long will the dump take with gzip compression (used by default)?

```

student$ /usr/bin/time -f %e pg_dump \
--format=custom --dbname=backup_logical \
--compress=gzip --file=/home/student/tmp/backup_logical.gzip

2.01

```

Now try lz4.

```

student$ /usr/bin/time -f %e pg_dump \
--format=custom --dbname=backup_logical \
--compress=lz4 --file=/home/student/tmp/backup_logical.lz4

0.33

```

Now zstd.

```

student$ /usr/bin/time -f %e pg_dump \
--format=custom --dbname=backup_logical \
--compress=zstd --file=/home/student/tmp/backup_logical.zstd

0.64

```

Compare the file sizes:

```

student$ ls -lhS ~/tmp/backup_logical.*

-rw-rw-r-- 1 student student 8.3M Sep 17 00:06 /home/student/tmp/backup_logical.none
-rw-rw-r-- 1 student student 7.9M Sep 17 00:06 /home/student/tmp/backup_logical.lz4
-rw-rw-r-- 1 student student 4.1M Sep 17 00:06 /home/student/tmp/backup_logical.zstd
-rw-rw-r-- 1 student student 3.9M Sep 17 00:06 /home/student/tmp/backup_logical.gzip
-rw-rw-r-- 1 student student  26 Sep 17 00:06 /home/student/tmp/backup_logical.txt

```

Conclusions:

- gzip has the best compression ratio, but takes several times longer to run
- lz4 is the fastest, but offers very little compression benefit
- zstd strikes a good balance: its compression ratio is only slightly behind gzip, while its speed is comparable to lz4

Cleanup

Delete the databases and temporary files we created.

```

student$ rm ~/tmp/backup_logical.*

student$ rm ~/tmp/anticopy.dump

```

```

α=> \c postgres

```

You are now connected to database "postgres" as user "student".

```

α=> DROP DATABASE backup_logical;

DROP DATABASE

```