pgcopydb

Release 0.12

Dimitri Fontaine

Jun 28, 2023

DOCUMENTATION TABLE OF CONTENTS

The pgcopydb project is an Open Source Software project. The development happens at https://github.com/dimitri/pgcopydb and is public: everyone is welcome to participate by opening issues, pull requests, giving feedback, etc.

Remember that the first steps are to actually play with the `pgcopydb` command, then read the entire available documentation (after all, I took the time to write it), and then to address the community in a kind and polite way — the same way you would expect people to use when addressing you.

CHAPTER

ONE

INTRODUCTION TO PGCOPYDB

pgcopydb is a tool that automates copying a PostgreSQL database to another server. Main use case for pgcopydb is migration to a new Postgres system, either for new hardware, new architecture, or new Postgres major version.

The idea would be to run `pg_dump -jN | pg_restore -jN` between two running Postgres servers. To make a copy of a database to another server as quickly as possible, one would like to use the parallel options of `pg_dump` and still be able to stream the data to as many `pg_restore` jobs. Unfortunately, that approach can't be implemented by using pg_dump and pg_restore directly, see *Bypass intermediate files for the TABLE DATA*.

When using `pgcopydb` it is possible to achieve both concurrency and streaming with this simple command line:

```
$ export PGCOPYDB_SOURCE_PGURI="postgres://user@source.host.dev/dbname"
$ export PGCOPYDB_TARGET_PGURI="postgres://role@target.host.dev/dbname"

$ pgcopydb clone --table-jobs 4 --index-jobs 4
```

See the manual page for *pgcopydb clone* for detailed information about how the command is implemented, and many other supported options.

1.1 Feature Matrix

Here is a comparison of the features available when using pg_dump and pg_restore directly, and when using pgcopydb to handle the database copying.

Feature	pgcopydb	pg_dump ; pg_restore
Single-command operation	✓	✗
Snapshot consistency	✓	✓
Ability to resume partial run	✓	✗
Advanced filtering	✓	✓
Tables concurrency	✓	✓
Same-table concurrency	✓	✗
Index concurrency	✓	✓
Constraint index concurrency	✓	✗
Schema	✓	✓
Large Objects	✓	✓
Vacuum Analyze	✓	✗
Copy Freeze	✓	✗
Roles	✓	✗ (needs pg_dumpall)
Tablespaces	✗	✗ (needs pg_dumpall)
Follow changes	✓	✗

See documentation about pgcopydb *pgcopydb configuration* for its *Advanced filtering* capabilities.

1.2 pgcopydb uses pg_dump and pg_restore

The implementation of pgcopydb actually calls into the pg_dump and pg_restore binaries to handle a large part of the work, such as the pre-data and post-data sections. See pg_dump docs for more information about the three sections supported.

After using pg_dump to obtain the pre-data and the post-data parts, then pgcopydb restore the pre-data parts to the target Postgres instance using pg_restore.

Then pgcopydb uses SQL commands and the COPY streaming protocol to migrate the table contents, the large objects data, and to VACUUM ANALYZE tables as soon as the data is available on the target instance.

Then pgcopydb uses SQL commands to build the indexes on the target Postgres instance, as detailed in the design doc *For each table, build all indexes concurrently*. This allows to include *constraint indexes* such as Primary Keys in the list of indexes built at the same time.

1.3 Change Data Capture, or fork and follow

It is also possible with `pgcopydb` to implement Change Data Capture and replay data modifications happening on the source database to the target database. See the *pgcopydb follow* command and the `pgcopydb clone --follow` command line option at *pgcopydb clone* in the manual.

The simplest possible implementation of *online migration* with pgcopydb, where changes being made to the source Postgres instance database are replayed on the target system, looks like the following:

```
$ pgcopydb clone --follow &

# later when the application is ready to make the switch
$ pgcopydb stream sentinel set endpos --current

# later when the migration is finished, clean-up both source and target
$ pgcopydb stream cleanup
```

CHAPTER

TWO

DESIGN CONSIDERATIONS

The reason why pgcopydb has been developed is mostly to allow two aspects that are not possible to achieve directly with pg_dump and pg_restore, and that requires just enough fiddling around that not many scripts have been made available to automate around.

2.1 Bypass intermediate files for the TABLE DATA

First aspect is that for pg_dump and pg_restore to implement concurrency they need to write to an intermediate file first.

The docs for pg_dump say the following about the --jobs parameter:

> You can only use this option with the directory output format because this is the only output format where multiple processes can write their data at the same time.

The docs for pg_restore say the following about the --jobs parameter:

> Only the custom and directory archive formats are supported with this option. The input must be a regular file or directory (not, for example, a pipe or standard input).

So the first idea with pgcopydb is to provide the --jobs concurrency and bypass intermediate files (and directories) altogether, at least as far as the actual TABLE DATA set is concerned.

The trick to achieve that is that pgcopydb must be able to connect to the source database during the whole operation, when pg_restore may be used from an export on-disk, without having to still be able to connect to the source database. In the context of pgcopydb requiring access to the source database is fine. In the context of pg_restore, it would not be acceptable.

2.2 Notes about concurrency

The pgcopydb too implements many operations concurrently to one another, by ways of using the fork() system call. This means that pgcopydb creates sub-processes that each handle a part of the work.

The process tree then looks like the following:

- pgcopydb clone –follow –table-jobs 4 –index-jobs 4
  - pgcopydb clone worker
    - pgcopydb copy supervisor (--table-jobs 4)
      1. pgcopydb copy worker
      2. pgcopydb copy worker

3. pgcopydb copy worker
4. pgcopydb copy worker

* pgcopydb blob worker

1. pgcopydb index/constraints worker (`--index-jobs 4`)
2. pgcopydb index/constraints worker (`--index-jobs 4`)
3. pgcopydb index/constraints worker (`--index-jobs 4`)
4. pgcopydb index/constraints worker (`--index-jobs 4`)

1. pgcopydb vacuum analyze worker (`--table-jobs 4`)
2. pgcopydb vacuum analyze worker (`--table-jobs 4`)
3. pgcopydb vacuum analyze worker (`--table-jobs 4`)
4. pgcopydb vacuum analyze worker (`--table-jobs 4`)

* pgcopydb sequences reset worker

– pgcopydb follow worker

* pgcopydb stream receive
* pgcopydb stream transform
* pgcopydb stream catchup

We see that when using `pgcopydb clone --follow --table-jobs 4 --index-jobs 4` then pgcopydb creates 20 sub-processes, including one transient sub-process each time a JSON file is to be converted to a SQL file for replay.

The 20 total is counted from:

- 1 clone worker + 1 copy supervisor + 4 copy workers + 1 blob worker + 4 index workers + 4 vacuum workers + 1 sequence reset worker

  that's 1 + 1 + 4 + 1 + 4 + 4 + 1 = 16

- 1 follow worker + 1 stream receive + 1 stream transform + 1 stream catchup

  that's 1 + 1 + 1 + 1 = 4

- that's 16 + 4 = 20 total

Here is a description of the process tree:

- When starting with the TABLE DATA copying step, then pgcopydb creates as many sub-processes as specified by the `--table-jobs` command line option (or the environment variable `PGCOPYDB_TABLE_JOBS`).
- A single sub-process is created by pgcopydb to copy the Postgres Large Objects (BLOBs) found on the source database to the target database.
- To drive the index and constraint build on the target database, pgcopydb creates as many sub-processes as specified by the `--index-jobs` command line option (or the environment variable `PGCOPYDB_INDEX_JOBS`).

  It is possible with Postgres to create several indexes for the same table in parallel, for that, the client just needs to open a separate database connection for each index and run each CREATE INDEX command in its own connection, at the same time. In pgcopydb this is implemented by running one sub-process per index to create.

  The `--index-jobs` option is global for the whole operation, so that it's easier to setup to the count of available CPU cores on the target Postgres instance. Usually, a given CREATE INDEX command uses 100% of a single core.

- To drive the VACUUM ANALYZE workload on the target database pgcopydb creates as many sub-processes as specified by the `--table-jobs` command line option.
- To reset sequences in parallel to COPYing the table data, pgcopydb creates a single dedicated sub-process.
- When using the `--follow` option then another sub-process leader is created to handle the three Change Data Capture processes.
  - One process implements *pgcopydb stream receive* to fetch changes in the JSON format and pre-fetch them in JSON files.
  - As soon as JSON file is completed, the pgcopydb stream transform worker transforms the JSON file into SQL, as if by calling the command *pgcopydb stream transform*.
  - Another process implements *pgcopydb stream catchup* to apply SQL changes to the target Postgres instance. This process loops over querying the pgcopydb sentinel table until the apply mode has been enabled, and then loops over the SQL files and run the queries from them.

2.3 For each table, build all indexes concurrently

pgcopydb takes the extra step and makes sure to create all your indexes in parallel to one-another, going the extra mile when it comes to indexes that are associated with a constraint.

Postgres introduced the configuration parameter synchronize_seqscans in version 8.3, eons ago. It is on by default and allows the following behavior:

> This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload.

The other aspect that `pg_dump` and `pg_restore` are not very smart about is how they deal with the indexes that are used to support constraints, in particular unique constraints and primary keys.

Those indexes are exported using the `ALTER TABLE` command directly. This is fine because the command creates both the constraint and the underlying index, so the schema in the end is found as expected.

That said, those `ALTER TABLE ... ADD CONSTRAINT` commands require a level of locking that prevents any concurrency. As we can read on the docs for ALTER TABLE:

> Although most forms of ADD table_constraint require an ACCESS EXCLUSIVE lock, ADD FOREIGN KEY requires only a SHARE ROW EXCLUSIVE lock. Note that ADD FOREIGN KEY also acquires a SHARE ROW EXCLUSIVE lock on the referenced table, in addition to the lock on the table on which the constraint is declared.

The trick is then to first issue a `CREATE UNIQUE INDEX` statement and when the index has been built then issue a second command in the form of `ALTER TABLE ... ADD CONSTRAINT ... PRIMARY KEY USING INDEX ...`, as in the following example taken from the logs of actually running `pgcopydb`:

```
21:52:06 68898 INFO  COPY "demo"."tracking";
21:52:06 68899 INFO  COPY "demo"."client";
21:52:06 68899 INFO  Creating 2 indexes for table "demo"."client"
21:52:06 68906 INFO  CREATE UNIQUE INDEX client_pkey ON demo.client USING btree (client);
21:52:06 68907 INFO  CREATE UNIQUE INDEX client_pid_key ON demo.client USING btree (pid);
21:52:06 68898 INFO  Creating 1 indexes for table "demo"."tracking"
21:52:06 68908 INFO  CREATE UNIQUE INDEX tracking_pkey ON demo.tracking USING btree (client, ts);
21:52:06 68907 INFO  ALTER TABLE "demo"."client" ADD CONSTRAINT "client_pid_key" UNIQUE USING INDEX "client_
↪pid_key";
21:52:06 68906 INFO  ALTER TABLE "demo"."client" ADD CONSTRAINT "client_pkey" PRIMARY KEY USING INDEX "client_
↪pkey";
21:52:06 68908 INFO  ALTER TABLE "demo"."tracking" ADD CONSTRAINT "tracking_pkey" PRIMARY KEY USING INDEX
↪"tracking_pkey";
```

This trick is worth a lot of performance gains on its own, as has been discovered and experienced and appreciated by pgloader users already.

2.4 Same-table Concurrency

In some database schema design, it happens that most of the database size on-disk is to be found in a single giant table, or a short list of giant tables. When this happens, the concurrency model that is implemented with `--table-jobs` still allocates a single process to COPY all the data from the source table.

Same-table concurrency allows pgcopydb to use more than once process at the same time to process a single source table. The data is then logically partitionned (on the fly) and split between processes:

- To fetch the data from the source database, the COPY processes then use SELECT queries like in the following example:

```
COPY (SELECT * FROM source.table WHERE id BETWEEN      1 AND 123456)
COPY (SELECT * FROM source.table WHERE id BETWEEN 123457 AND 234567)
COPY (SELECT * FROM source.table WHERE id BETWEEN 234568 AND 345678)
...
```

  This is only possible when the source.table has at least one column of an integer type (`int2`, `int4`, and `int8` are supported) and with a UNIQUE or PRIMARY KEY constraint. We must make sure that any given row is selected only once overall to avoid introducing duplicates on the target database.

- To decide if a table COPY processing should be split, the command line option `split-tables-larger-than` is used, or the environment variable `PGCOPYDB_SPLIT_TABLES_LARGER_THAN`.

  The expected value is either a plain number of bytes, or a pretty-printed number of bytes such as `250 GB`.

  When using this option, then tables that have at least this amount of data and also a candidate key for the COPY partitioning are then distributed among a number of COPY processes.

  The number of COPY processes is computed by dividing the table size by the threshold set with the split option. For example, if the threshold is 250 GB then a 400 GB table is going to be distributed among 2 COPY processes.

  The command *pgcopydb list table-parts* may be used to list the COPY partitioning that pgcopydb computes given a source table and a threshold.

2.4.1 Significant differences when using same-table COPY concurrency

When same-table concurrency happens for a source table, some operations are not implemented as they would have been without same-table concurrency. Specifically:

- TRUNCATE and COPY FREEZE Postgres optimisation

  When using a single COPY process, it's then possible to TRUNCATE the target table in the same transaction as the COPY command, as in the following syntethic example:

```
BEGIN;
TRUNCATE table ONLY;
COPY table FROM stdin WITH (FREEZE);
COMMIT
```

  This technique allows Postgres to implement several optimisations, doing work during the COPY that would otherwise need to happen later when executing the first queries on the table.

  When using same-table concurrency then we have several transactions happening concurrently on the target system that are copying data from the source table. This means that we have to TRUNCATE separately and the FREEZE option can not be used.

- CREATE INDEX and VACUUM

Even when same-table COPY concurrency is enabled, creating the indexes on the target system only happens after the whole data set has been copied over. This means that only the when the last process is done with the COPYing then this process will take care of the the indexes and the *vacuum analyze* operation.

2.4.2 Same-table COPY concurrency performance limitations

Finally, it might be that same-table concurrency is not effective at all in some use cases. Here is a list of limitations to have in mind when selecting to use this feature:

- Network Bandwidth

  The most common performance bottleneck relevant to database migrations is the network bandwidth. When the bandwidth is saturated (used in full) then same-table concurrency will provide no performance benefits.

- Disks IOPS

  The second most command performance bottleneck relevant to database migrations is disks IOPS and, in the Cloud, burst capacity. When the disk bandwidth is used in full, then same-table concurrency will provide no performance benefits.

  Source database system uses read IOPS, target database system uses both read and write IOPS (copying the data writes to disk, creating the indexes both read table data from disk and then write index data to disk).

- On-disk data organisation

  When using a single COPY process, the target system may fill-in the Postgres table in a clustered way, using each disk page in full before opening the next one, in a sequential fashion.

  When using same-table COPY concurrency, then the target Postgres system needs to handle concurrent writes to the same table, resulting in a possibly less effective disk usage.

  How that may impact your application performance is to be tested.

- synchronize_seqscans

  Postgres implemented this option back in version 8.3. The option is now documented in the Version and Platform Compatibility section.

  The documentation reads:

  > This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload.

  The impact on performance when having concurrent COPY processes reading the same source table at the same time is to be assessed. At the moment there is no option in pgcopydb to *SET synchronize_seqscans TO off* when using same-table COPY concurrency.

  Use your usual Postgres configuration editing for testing.

CHAPTER

THREE

INSTALLING PGCOPYDB

Several distributions are available for pgcopydb.

3.1 debian packages

Binary packages for debian and derivatives (ubuntu) are available from apt.postgresql.org repository, install by following the linked documentation and then:

```
$ sudo apt-get install pgcopydb
```

3.2 RPM packages

The Postgres community repository for RPM packages is yum.postgresql.org and includes binary packages for pgcopydb. The way packages are built for RPM based systems means that the user needs to choose which version of Postgres pgcopydb was built with. In practice this doesn't have much importance, because `libpq` is meant to be compatible with many different Postgres server versions.

After following the instructions for installing the repository, in this example in a Docker image for Rocky Linux (`docker run --rm -it rockylinux:9`), then we get the following:

```
# dnf search pgcopydb
...
pgcopydb_11.x86_64 : Automate pg_dump | pg_restore between two running Postgres servers
pgcopydb_12.x86_64 : Automate pg_dump | pg_restore between two running Postgres servers
pgcopydb_13.x86_64 : Automate pg_dump | pg_restore between two running Postgres servers
pgcopydb_14.x86_64 : Automate pg_dump | pg_restore between two running Postgres servers
pgcopydb_15.x86_64 : Automate pg_dump | pg_restore between two running Postgres servers
```

3.3 Docker Images

Docker images are maintained for each tagged release at dockerhub, and also built from the CI/CD integration on GitHub at each commit to the *main* branch.

The DockerHub dimitri/pgcopydb repository is where the tagged releases are made available. The image uses the Postgres version currently in debian stable.

To use this docker image:

```
$ docker run --rm -it dimitri/pgcopydb:v0.12 pgcopydb --version
```

Or you can use the CI/CD integration that publishes packages from the main branch to the GitHub docker repository:

```
$ docker pull ghcr.io/dimitri/pgcopydb:latest
$ docker run --rm -it ghcr.io/dimitri/pgcopydb:latest pgcopydb --version
$ docker run --rm -it ghcr.io/dimitri/pgcopydb:latest pgcopydb --help
```

3.4 Build from sources

Building from source requires a list of build-dependencies that's comparable to that of Postgres itself. The pgcopydb source code is written in C and the build process uses a GNU Makefile.

See our main Dockerfile for a complete recipe to build pgcopydb when using a debian environment.

Then the build process is pretty simple, in its simplest form you can just use `make clean install`, if you want to be more fancy consider also:

```
$ make -s clean
$ make -s -j12 install
```

CHAPTER

FOUR

MANUAL PAGES

The `pgcopydb` command provides several sub-commands. Each of them have their own manual page.

4.1 pgcopydb

pgcopydb - copy an entire Postgres database from source to target

4.1.1 Synopsis

pgcopydb provides the following commands:

```
pgcopydb
  clone    Clone an entire database from source to target
  fork     Clone an entire database from source to target
  follow   Replay changes from the source database to the target database
  snapshot  Create and exports a snapshot on the source database
+ copy     Implement the data section of the database copy
+ dump     Dump database objects from a Postgres instance
+ restore  Restore database objects into a Postgres instance
+ list     List database objects from a Postgres instance
+ stream   Stream changes from the source database
  help     print help message
  version  print pgcopydb version
```

4.1.2 Description

The pgcopydb command implements a full migration of an entire Postgres database from a source instance to a target instance. Both the Postgres instances must be available for the entire duration of the command.

The pgcopydb command also implements a full Logical Decoding client for Postgres, allowing Change Data Capture to replay data changes (DML) happening on the source database after the base copy snapshot. The pgcopydb logical decoding client code is compatible with both test_decoding and wal2json output plugins, and defaults to using test_decoding.

4.1.3 pgcopydb help

The `pgcopydb help` command lists all the supported sub-commands:

```
$ pgcopydb help
 pgcopydb
   clone     Clone an entire database from source to target
   fork      Clone an entire database from source to target
   follow    Replay changes from the source database to the target database
   copy-db   Clone an entire database from source to target
   snapshot  Create and exports a snapshot on the source database
 + copy      Implement the data section of the database copy
 + dump      Dump database objects from a Postgres instance
 + restore   Restore database objects into a Postgres instance
 + list      List database objects from a Postgres instance
 + stream    Stream changes from the source database
   ping      Copy the roles from the source instance to the target instance
   help      print help message
   version   print pgcopydb version

 pgcopydb copy
   db           Copy an entire database from source to target
   roles        Copy the roles from the source instance to the target instance
   extensions   Copy the extensions from the source instance to the target instance
   schema       Copy the database schema from source to target
   data         Copy the data section from source to target
   table-data   Copy the data from all tables in database from source to target
   blobs        Copy the blob data from ther source database to the target
   sequences    Copy the current value from all sequences in database from source to target
   indexes      Create all the indexes found in the source database in the target
   constraints  Create all the constraints found in the source database in the target

 pgcopydb dump
   schema     Dump source database schema as custom files in work directory
   pre-data   Dump source database pre-data schema as custom files in work directory
   post-data  Dump source database post-data schema as custom files in work directory
   roles      Dump source database roles as custome file in work directory

 pgcopydb restore
   schema      Restore a database schema from custom files to target database
   pre-data    Restore a database pre-data schema from custom file to target database
   post-data   Restore a database post-data schema from custom file to target database
   roles       Restore database roles from SQL file to target database
   parse-list  Parse pg_restore --list output from custom file

 pgcopydb list
   databases    List databases
   extensions   List all the source extensions to copy
   collations   List all the source collations to copy
   tables       List all the source tables to copy data from
   table-parts  List a source table copy partitions
   sequences    List all the source sequences to copy data from
   indexes      List all the indexes to create again after copying the data
   depends      List all the dependencies to filter-out
   schema       List the schema to migrate, formatted in JSON
   progress     List the progress

 pgcopydb stream
   setup      Setup source and target systems for logical decoding
   cleanup    cleanup source and target systems for logical decoding
   prefetch   Stream JSON changes from the source database and transform them to SQL
   catchup    Apply prefetched changes from SQL files to the target database
   replay     Replay changes from the source to the target database, live
 + sentinel   Maintain a sentinel table on the source database
   receive    Stream changes from the source database
   transform  Transform changes from the source database into SQL commands
   apply      Apply changes from the source database into the target database

 pgcopydb stream sentinel
   create  Create the sentinel table on the source database
   drop    Drop the sentinel table on the source database
   get     Get the sentinel table values on the source database
 + set     Maintain a sentinel table on the source database

 pgcopydb stream sentinel set
   startpos  Set the sentinel start position LSN on the source database
   endpos    Set the sentinel end position LSN on the source database
   apply     Set the sentinel apply mode on the source database
   prefetch  Set the sentinel prefetch mode on the source database
```

4.1.4 pgcopydb version

The `pgcopydb version` command outputs the version string of the version of pgcopydb used, and can do that in the JSON format when using the `--json` option.

```
$ pgcopydb version
pgcopydb version 0.8
compiled with PostgreSQL 12.12 on x86_64-apple-darwin16.7.0, compiled by Apple LLVM version 8.1.0 (clang-802.0.
↪42), 64-bit
compatible with Postgres 10, 11, 12, 13, and 14
```

In JSON:

```
$ pgcopydb version --json
{
    "pgcopydb": "0.8",
    "pg_major": "12",
    "pg_version": "12.12",
    "pg_version_str": "PostgreSQL 12.12 on x86_64-apple-darwin16.7.0, compiled by Apple LLVM version 8.1.0␣
↪(clang-802.0.42), 64-bit",
    "pg_version_num": 120012
}
```

The details about the Postgres version applies to the version that's been used to build pgcopydb from sources, so that's the version of the client library `libpq` really.

4.1.5 pgcopydb ping

The `pgcopydb ping` command attempts to connect to both the source and the target Postgres databases, concurrently.

```
pgcopydb ping: Copy the roles from the source instance to the target instance
usage: pgcopydb ping  --source ... --target ...

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
```

An example output looks like the following:

```
$ pgcopydb ping
18:04:48 84679 INFO   Running pgcopydb version 0.10.31.g7e5fbb8.dirty from "/Users/dim/dev/PostgreSQL/pgcopydb/
↪src/bin/pgcopydb/pgcopydb"
18:04:48 84683 INFO   Successfully could connect to target database at "postgres://@:/plop?"
18:04:48 84682 INFO   Successfully could connect t source database at "postgres://@:/pagila?"
```

This command implements a retry policy (named *Decorrelated Jitter*) and can be used in automation to make sure that the databases are ready to accept connections.

4.2 pgcopydb clone

The main pgcopydb operation is the clone operation, and for historical and user friendlyness reasons three aliases are available that implement the same operation:

```
pgcopydb
  clone     Clone an entire database from source to target
  fork      Clone an entire database from source to target
  copy-db   Copy an entire database from source to target
```

4.2.1 pgcopydb clone

The command `pgcopydb clone` copies a database from the given source Postgres instance to the target Postgres instance.

```
pgcopydb clone: Clone an entire database from source to target
usage: pgcopydb clone  --source ... --target ... [ --table-jobs ... --index-jobs ... ]

  --source                  Postgres URI to the source database
  --target                  Postgres URI to the target database
  --dir                     Work directory to use
  --table-jobs              Number of concurrent COPY jobs to run
  --index-jobs              Number of concurrent CREATE INDEX jobs to run
  --split-tables-larger-than Same-table concurrency size threshold
  --drop-if-exists          On the target database, clean-up from a previous run first
  --roles                   Also copy roles found on source to target
  --no-role-passwords       Do not dump passwords for roles
  --no-owner                Do not set ownership of objects to match the original database
  --no-acl                  Prevent restoration of access privileges (grant/revoke commands).
  --no-comments             Do not output commands to restore comments
  --skip-large-objects      Skip copying large objects (blobs)
  --skip-extensions         Skip restoring extensions
  --skip-collations         Skip restoring collations
  --skip-vacuum             Skip running VACUUM ANALYZE
  --filters <filename>      Use the filters defined in <filename>
  --fail-fast               Abort early in case of error
  --restart                 Allow restarting when temp files exist already
  --resume                  Allow resuming operations after a failure
  --not-consistent          Allow taking a new snapshot on the source database
  --snapshot                Use snapshot obtained with pg_export_snapshot
  --follow                  Implement logical decoding to replay changes
  --plugin                  Output plugin to use (test_decoding, wal2json)
  --slot-name               Use this Postgres replication slot name
  --create-slot             Create the replication slot
  --origin                  Use this Postgres replication origin node name
  --endpos                  Stop replaying changes when reaching this LSN
```

4.2.2 pgcopydb fork

The command `pgcopydb fork` copies a database from the given source Postgres instance to the target Postgres instance. This command is an alias to the command `pgcopydb clone` seen above.

4.2.3 pgcopydb copy-db

The command `pgcopydb copy-db` copies a database from the given source Postgres instance to the target Postgres instance. This command is an alias to the command `pgcopydb clone` seen above, and available for backward compatibility only.

> **Warning:** The `pgcopydb copy-db` command is now deprecated and will get removed from pgcopydb when hitting version 1.0, please upgrade your scripts and integrations.

4.2.4 Description

The `pgcopydb clone` command implements both a base copy of a source database into a target database and also a full Logical Decoding client for the wal2json logical decoding plugin.

Base copy, or the clone operation

The `pgcopydb clone` command implements the following steps:

1. `pgcopydb` gets the list of ordinary and partitioned tables from a catalog query on the source database, and also the list of indexes, and the list of sequences with their current values.

   When filtering is used, the list of objects OIDs that are meant to be filtered out is built during this step.

2. `pgcopydb` calls into `pg_dump` to produce the `pre-data` section and the `post-data` sections of the dump using Postgres custom format.

3. The `pre-data` section of the dump is restored on the target database using the `pg_restore` command, creating all the Postgres objects from the source database into the target database.

   When filtering is used, the `pg_restore --use-list` feature is used to filter the list of objects to restore in this step.

4. Then as many as `--table-jobs` COPY sub-processes are started to share the workload and COPY the data from the source to the target database one table at a time, in a loop.

   A Postgres connection and a SQL query to the Postgres catalog table pg_class is used to get the list of tables with data to copy around, and the *reltuples* statistic is used to start with the tables with the greatest number of rows first, as an attempt to minimize the copy time.

5. An auxiliary process loops through all the Large Objects found on the source database and copies its data parts over to the target database, much like pg_dump itself would.

   This step is much like `pg_dump | pg_restore` for large objects data parts, except that there isn't a good way to do just that with the tooling.

6. As many as `--index-jobs` CREATE INDEX sub-processes are started to share the workload and build indexes. In order to make sure to start the CREATE INDEX commands only after the COPY operation has completed, a queue mechanism is used. As soon as a table data COPY has completed, all the indexes for the table are queued for processing by the CREATE INDEX sub-processes.

   The primary indexes are created as UNIQUE indexes at this stage.

7. Then the PRIMARY KEY constraints are created USING the just built indexes. This two-steps approach allows the primary key index itself to be created in parallel with other indexes on the same table, avoiding an EXCLUSIVE LOCK while creating the index.

8. As many as `-table-jobs` VACUUM ANALYZE sub-processes are started to share the workload. As soon as a table data COPY has completed, the table is queued for processing by the VACUUM ANALYZE sub-processes.

9. An auxilliary process is loops over the sequences on the source database and for each of them runs a separate query on the source to fetch the `last_value` and the `is_called` metadata the same way that pg_dump does.

   For each sequence, pgcopydb then calls `pg_catalog.setval()` on the target database with the information obtained on the source database.

10. The final stage consists now of running the `pg_restore` command for the `post-data` section script for the whole database, and that's where the foreign key constraints and other elements are created.

    The *post-data* script is filtered out using the `pg_restore --use-list` option so that indexes and primary key constraints already created in steps 6 and 7 are properly skipped now.

Postgres privileges, superuser, and dump and restore

Postgres has a notion of a superuser status that can be assigned to any role in the system, and the default role *postgres* has this status. From the Role Attributes documentation page we see that:

> superuser status:
>
> *A database superuser bypasses all permission checks, except the right to log in. This is a dangerous privilege and should not be used carelessly; it is best to do most of your work as a role that is not a superuser. To create a new database superuser, use CREATE ROLE name SUPERUSER. You must do this as a role that is already a superuser.*

Some Postgres objects can only be created by superusers, and some read and write operations are only allowed to superuser roles, such as the following non-exclusive list:

- Reading the pg_authid role password (even when encrypted) is restricted to roles with the superuser status. Reading this catalog table is done when calling `pg_dumpall --roles-only` so that the dump file can then be used to restore roles including their passwords.

  It is possible to implement a pgcopydb migration that skips the passwords entirely when using the option `--no-role-passwords`. In that case though authentication might fail until passwords have been setup again correctly.

- Most of the available Postgres extensions, at least when being written in C, are then only allowed to be created by roles with superuser status.

  When such an extension contains Extension Configuration Tables and has been created with a role having superuser status, then the same superuser status is needed again to pg_dump and pg_restore that extension and its current configuration.

When using pgcopydb it is possible to split your migration in privileged and non-privileged parts, like in the following examples:

```
$ coproc ( pgcopydb snapshot )

# first two commands would use a superuser role to connect
$ pgcopydb copy roles --source ... --target ...
$ pgcopydb copy extensions --source ... --target ...

# now it's possible to use a non-superuser role to connect
$ pgcopydb clone --skip-extensions --source ... --target ...

$ kill -TERM ${COPROC_PID}
$ wait ${COPROC_PID}
```

In such a script, the calls to *pgcopydb copy roles* and *pgcopydb copy extensions* would be done with connection strings that connects with a role having superuser status; and then the call to *pgcopydb clone* would be done with a non-privileged role, typically the role that owns the source and target databases.

> **Warning:** That said, there is currently a limitation in `pg_dump` that impacts pgcopydb. When an extension with configuration table has been installed as superuser, even the main `pgcopydb clone` operation has to be done with superuser status.
>
> That's because pg_dump filtering (here, there `--exclude-table` option) does not apply to extension members, and pg_dump does not provide a mechanism to exclude extensions.

Change Data Capture using Postgres Logical Decoding

When using the `--follow` option the steps from the *pgcopydb follow* command are also run concurrently to the main copy. The Change Data Capture is then automatically driven from a prefetch-only phase to the prefetch-and-catchup phase, which is enabled as soon as the base copy is done.

See the command *pgcopydb stream sentinel set endpos* to remote control the follow parts of the command even while the command is already running.

The command *pgcopydb stream cleanup* must be used to free resources created to support the change data capture process.

Important: Make sure to read the documentation for *pgcopydb follow* and the specifics about Logical Replication Restrictions as documented by Postgres.

Change Data Capture Example 1

A simple approach to applying changes after the initial base copy has been done follows:

```
$ pgcopydb clone --follow &

# later when the application is ready to make the switch
$ pgcopydb stream sentinel set endpos --current

# later when the migration is finished, clean-up both source and target
$ pgcopydb stream cleanup
```

Change Data Capture Example 2

In some cases, it might be necessary to have more control over some of the steps taken here. Given pgcopydb flexibility, it's possible to implement the following steps:

1. Grab a snapshot from the source database and hold an open Postgres connection for the duration of the base copy.

   In case of crash or other problems with the main operations, it's then possible to resume processing of the base copy and the applying of the changes with the same snapshot again.

   This step is also implemented when using `pgcopydb clone --follow`. That said, if the command was interrupted (or crashed), then the snapshot would be lost.

2. Setup the logical decoding within the snapshot obtained in the previous step, and the replication tracking on the target database.

   The following SQL objects are then created:

   - a replication slot on the source database,
   - a `pgcopydb.sentinel` table on the source database,
   - a replication origin on the target database.

   This step is also implemented when using `pgcopydb clone --follow`. There is no way to implement Change Data Capture with pgcopydb and skip creating those SQL objects.

3. Start the base copy of the source database, and prefetch logical decoding changes to ensure that we consume from the replication slot and allow the source database server to recycle its WAL files.

4. Remote control the apply process to stop consuming changes and applying them on the target database.

5. Re-sync the sequences to their now-current values.

   Sequences are not handled by Postgres logical decoding, so extra care needs to be implemented manually here.

   Important: The next version of pgcopydb will include that step in the `pgcopydb clone --snapshot` command automatically, after it stops consuming changes and before the process terminates.

6. Clean-up the specific resources created for supporting resumability of the whole process (replication slot on the source database, pgcopydb sentinel table on the source database, replication origin on the target database).
7. Stop holding a snaphot on the source database by stopping the `pgcopydb snapshot` process left running in the background.

If the command `pgcopydb clone --follow` fails it's then possible to start it again. It will automatically discover what was done successfully and what needs to be done again because it failed or was interrupted (table copy, index creation, resuming replication slot consuming, resuming applying changes at the right LSN position, etc).

Here is an example implement the previous steps:

```
$ pgcopydb snapshot &

$ pgcopydb stream setup

$ pgcopydb clone --follow &

# later when the application is ready to make the switch
$ pgcopydb stream sentinel set endpos --current

# when the follow process has terminated, re-sync the sequences
$ pgcopydb copy sequences

# later when the migration is finished, clean-up both source and target
$ pgcopydb stream cleanup

# now stop holding the snapshot transaction (adjust PID to your environment)
$ kill %1
```

4.2.5 Options

The following options are available to `pgcopydb clone`:

--source Connection string to the source Postgres instance. See the Postgres documentation for connection strings for the details. In short both the quoted form `"host=... dbname=..."` and the URI form `postgres://user@host:5432/dbname` are supported.

--target Connection string to the target Postgres instance.

--dir During its normal operations pgcopydb creates a lot of temporary files to track sub-processes progress. Temporary files are created in the directory location given by this option, or defaults to `${TMPDIR}/pgcopydb` when the environment variable is set, or then to `/tmp/pgcopydb`.

--table-jobs How many tables can be processed in parallel.

This limit only applies to the COPY operations, more sub-processes will be running at the same time that this limit while the CREATE INDEX operations are in progress, though then the processes are only waiting for the target Postgres instance to do all the work.

--index-jobs How many indexes can be built in parallel, globally. A good option is to set this option to the count of CPU cores that are available on the Postgres target system, minus some cores that are going to be used for handling the COPY operations.

--split-tables-larger-than Allow *Same-table Concurrency* when processing the source database. This environment variable value is expected to be a byte size, and bytes units B, kB, MB, GB, TB, PB, and EB are known.

--drop-if-exists When restoring the schema on the target Postgres instance, `pgcopydb` actually uses `pg_restore`. When this options is specified, then the following pg_restore options are also used: `--clean --if-exists`.

This option is useful when the same command is run several times in a row, either to fix a previous mistake or for instance when used in a continuous integration system.

This option causes `DROP TABLE` and `DROP INDEX` and other DROP commands to be used. Make sure you understand what you're doing here!

--roles The option `--roles` add a preliminary step that copies the roles found on the source instance to the target instance. As Postgres roles are global object, they do not exist only within the context of a specific database, so all the roles are copied over when using this option.

The `pg_dumpall --roles-only` is used to fetch the list of roles from the source database, and this command includes support for passwords. As a result, this operation requires the superuser privileges.

See also *pgcopydb copy roles*.

--no-role-passwords Do not dump passwords for roles. When restored, roles will have a null password, and password authentication will always fail until the password is set. Since password values aren't needed when this option is specified, the role information is read from the catalog view pg_roles instead of pg_authid. Therefore, this option also helps if access to pg_authid is restricted by some security policy.

--no-owner Do not output commands to set ownership of objects to match the original database. By default, `pg_restore` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail unless the initial connection to the database is made by a superuser (or the same user that owns all of the objects in the script). With `--no-owner`, any user name can be used for the initial connection, and this user will own all the created objects.

--skip-large-objects Skip copying large objects, also known as blobs, when copying the data from the source database to the target database.

--skip-extensions Skip copying extensions from the source database to the target database.

When used, schema that extensions depend-on are also skipped: it is expected that creating needed extensions on the target system is then the responsibility of another command (such as *pgcopydb copy extensions*), and schemas that extensions depend-on are part of that responsibility.

Because creating extensions require superuser, this allows a multi-steps approach where extensions are dealt with superuser privileges, and then the rest of the pgcopydb operations are done without superuser privileges.

--skip-collations Skip copying collations from the source database to the target database.

In some scenarios the list of collations provided by the Operating System on the

source and target system might be different, and a mapping then needs to be manually installed before calling pgcopydb.

Then this option allows pgcopydb to skip over collations and assume all the needed collations have been deployed on the target database already.

See also *pgcopydb list collations*.

--skip-vacuum Skip running VACUUM ANALYZE on the target database once a table has been copied, its indexes have been created, and constraints installed.

--filters <filename> This option allows to exclude table and indexes from the copy operations. See *Filtering* for details about the expected file format and the filtering options available.

--fail-fast Abort early in case of error by sending the TERM signal to all the processes in the pgcopydb process group.

--restart When running the pgcopydb command again, if the work directory already contains information from a previous run, then the command refuses to proceed and delete information that might be used for diagnostics and forensics.

In that case, the `--restart` option can be used to allow pgcopydb to delete traces from a previous run.

--resume When the pgcopydb command was terminated before completion, either by an interrupt signal (such as C-c or SIGTERM) or because it crashed, it is possible to resume the database migration.

When resuming activity from a previous run, table data that was fully copied over to the target server is not sent again. Table data that was interrupted during the COPY has to be started from scratch even when using `--resume`: the COPY command in Postgres is transactional and was rolled back.

Same reasonning applies to the CREATE INDEX commands and ALTER TABLE commands that pgcopydb issues, those commands are skipped on a `--resume` run only if known to have run through to completion on the previous one.

Finally, using `--resume` requires the use of `--not-consistent`.

--not-consistent In order to be consistent, pgcopydb exports a Postgres snapshot by calling the pg_export_snapshot() function on the source database server. The snapshot is then re-used in all the connections to the source database server by using the `SET TRANSACTION SNAPSHOT` command.

Per the Postgres documentation about `pg_export_snapshot`:

> Saves the transaction's current snapshot and returns a text string identifying the snapshot. This string must be passed (outside the database) to clients that want to import the snapshot. The snapshot is available for import only until the end of the transaction that exported it.

Now, when the pgcopydb process was interrupted (or crashed) on a previous run, it is possible to resume operations, but the snapshot that was exported does not exists anymore. The pgcopydb command can only resume operations with a new snapshot, and thus can not ensure consistency of the whole data set, because each run is now using their own snapshot.

--snapshot Instead of exporting its own snapshot by calling the PostgreSQL function `pg_export_snapshot()` it is possible for pgcopydb to re-use an already exported snapshot.

--follow

When the `--follow` option is used then pgcopydb implements Change Data Capture as detailed in the manual page for *pgcopydb follow* in parallel to the main copy database steps.

The replication slot is created using the same snapshot as the main database copy operation, and the changes to the source database are prefetched only during the initial copy, then prefetched and applied in a catchup process.

It is possible to give `pgcopydb clone --follow` a termination point (the LSN endpos) while the command is running with the command *pgcopydb stream sentinel set endpos*.

--plugin

Logical decoding output plugin to use. The default is test_decoding which ships with Postgres core itself, so is probably already available on your source server.

It is possible to use wal2json instead. The support for wal2json is mostly historical in pgcopydb, it should not make a user visible difference whether you use the default test_decoding or wal2json.

--slot-name

Logical decoding slot name to use. Defaults to `pgcopydb`. which is unfortunate when your use-case involves migrating more than one database from the source server.

--create-slot

Instruct pgcopydb to create the logical replication slot to use.

--endpos

Logical replication target LSN to use. Automatically stop replication and exit with normal exit status 0 when receiving reaches the specified LSN. If there's a record with LSN exactly equal to lsn, the record will be output.

The `--endpos` option is not aware of transaction boundaries and may truncate output partway through a transaction. Any partially output transaction will not be consumed and will be replayed again when the slot is next read from. Individual messages are never truncated.

See also documentation for pg_recvlogical.

--origin

Logical replication target system needs to track the transactions that have been applied already, so that in case we get disconnected or need to resume operations we can skip already replayed transaction.

Postgres uses a notion of an origin node name as documented in Replication Progress Tracking. This option allows to pick your own node name and defaults to "pgcopydb". Picking a different name is useful in some advanced scenarios like migrating several sources in the same target, where each source should have their own unique origin node name.

--verbose, --notice

Increase current verbosity. The default level of verbosity is INFO. In ascending order pgcopydb knows about the following verbosity levels: FATAL, ERROR, WARN, INFO, NOTICE, SQL, DEBUG, TRACE.

--debug

Set current verbosity to DEBUG level.

--trace

Set current verbosity to TRACE level.

--quiet

Set current verbosity to ERROR level.

4.2.6 Environment

PGCOPYDB_SOURCE_PGURI

Connection string to the source Postgres instance. When `--source` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_TARGET_PGURI

Connection string to the target Postgres instance. When `--target` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_TABLE_JOBS

Number of concurrent jobs allowed to run COPY operations in parallel. When `--table-jobs` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_INDEX_JOBS

Number of concurrent jobs allowed to run CREATE INDEX operations in parallel. When `--index-jobs` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_SPLIT_TABLES_LARGER_THAN

Allow *Same-table Concurrency* when processing the source database. This environment variable value is expected to be a byte size, and bytes units B, kB, MB, GB, TB, PB, and EB are known.

When `--split-tables-larger-than` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_DROP_IF_EXISTS

When true (or *yes*, or *on*, or 1, same input as a Postgres boolean) then pgcopydb uses the pg_restore options `--clean --if-exists` when creating the schema on the target Postgres instance.

When `--drop-if-exists` is ommitted from the command line then this environment variable is used.

PGCOPYDB_FAIL_FAST

When true (or *yes*, or *on*, or 1, same input as a Postgres boolean) then pgcopydb sends the TERM signal to all the processes in its process group as soon as one process terminates with a non-zero return code.

When `--fail-fast` is ommitted from the command line then this environment variable is used.

PGCOPYDB_SKIP_VACUUM

When true (or *yes*, or *on*, or 1, same input as a Postgres boolean) then pgcopydb skips the VACUUM ANALYZE jobs entirely, same as when using the `--skip-vacuum` option.

PGCOPYDB_SNAPSHOT

Postgres snapshot identifier to re-use, see also `--snapshot`.

TMPDIR

The pgcopydb command creates all its work files and directories in `${TMPDIR}/pgcopydb`, and defaults to `/tmp/pgcopydb`.

PGCOPYDB_LOG_TIME_FORMAT

The logs time format defaults to `%H:%M:%S` when pgcopydb is used on an interactive terminal, and to `%Y-%m-%d %H:%M:%S` otherwise. This environment variable can be set to any format string other than the defaults.

See documentation for strftime(3) for details about the format string. See documentation for isatty(3) for details about detecting if pgcopydb is run in an interactive terminal.

PGCOPYDB_LOG_JSON

When true (or *yes*, or *on*, or 1, same input as a Postgres boolean) then pgcopydb formats its logs using JSON.

```
{
  "timestamp": "2023-04-13 16:53:14",
  "pid": 87956,
  "error_level": 4,
  "error_severity": "INFO",
  "file_name": "main.c",
  "file_line_num": 165,
  "message": "Running pgcopydb version 0.11.19.g2290494.dirty from \"/Users/dim/dev/PostgreSQL/
↪pgcopydb/src/bin/pgcopydb/pgcopydb\""
}
```

PGCOPYDB_LOG_FILENAME

When set to a filename (in a directory that must exists already) then pgcopydb writes its logs output to that filename in addition to the logs on the standard error output stream.

If the file already exists, its content is overwritten. In other words the previous content would be lost when running the same command twice.

PGCOPYDB_LOG_JSON_FILE

When true (or *yes*, or *on*, or 1, same input as a Postgres boolean) then pgcopydb formats its logs using JSON when writing to PGCOPYDB_LOG_FILENAME.

XDG_DATA_HOME

The standard XDG Base Directory Specification defines several environment variables that allow controling where programs should store their files.

> *XDG_DATA_HOME defines the base directory relative to which user-specific data files should be stored. If $XDG_DATA_HOME is either not set or empty, a default equal to $HOME/.local/share should be used.*

When using Change Data Capture (through `--follow` option and Postgres logical decoding with wal2json) then pgcopydb pre-fetches changes in JSON files and transform them into SQL files to apply to the target database.

These files are stored at the following location, tried in this order:

1. when `--dir` is used, then pgcopydb uses the `cdc` subdirectory of the `--dir` location,
2. when `XDG_DATA_HOME` is set in the environment, then pgcopydb uses that location,
3. when neither of the previous settings have been used then pgcopydb defaults to using `${HOME}/.local/share`.

4.2.7 Examples

```
$ export PGCOPYDB_SOURCE_PGURI="port=54311 host=localhost dbname=pgloader"
$ export PGCOPYDB_TARGET_PGURI="port=54311 dbname=plop"
$ export PGCOPYDB_DROP_IF_EXISTS=on

$ pgcopydb clone --table-jobs 8 --index-jobs 12
13:09:08 81987 INFO  Running pgcopydb version 0.8.21.gacd2795.dirty from "/Applications/Postgres.app/Contents/
↪Versions/12/bin/pgcopydb"
13:09:08 81987 INFO  [SOURCE] Copying database from "postgres://@:/pagila?"
13:09:08 81987 INFO  [TARGET] Copying database into "postgres://@:/plop?"
13:09:08 81987 INFO  Using work dir "/var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//pgcopydb"
13:09:08 81987 INFO  Exported snapshot "00000003-00076012-1" from the source database
13:09:08 81991 INFO  STEP 1: dump the source database schema (pre/post data)
13:09:08 81991 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_dump -Fc --snapshot 00000003-
↪00076012-1 --section pre-data --file /var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//pgcopydb/schema/pre.
↪dump 'postgres://@:/pagila?'
```

(continues on next page)

(continued from previous page)

```
13:09:08 81991 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_dump -Fc --snapshot 00000003-
↪00076012-1 --section post-data --file /var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//pgcopydb/schema/post.
↪dump 'postgres://@:/pagila?'
13:09:08 81991 INFO  STEP 2: restore the pre-data section to the target database
13:09:09 81991 INFO  Listing ordinary tables in source database
13:09:09 81991 INFO  Fetched information for 21 tables, with an estimated total of 46 248 tuples and 3776 kB
13:09:09 81991 INFO  Fetching information for 13 sequences
13:09:09 81991 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_restore --dbname 'postgres://@:/
↪plop?' --single-transaction --clean --if-exists --use-list /var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//
↪pgcopydb/schema/pre.list /var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//pgcopydb/schema/pre.dump
13:09:09 81991 INFO  STEP 3: copy data from source to target in sub-processes
13:09:09 81991 INFO  STEP 4: create indexes and constraints in parallel
13:09:09 81991 INFO  STEP 5: vacuum analyze each table
13:09:09 81991 INFO  Now starting 8 processes
13:09:09 81991 INFO  Reset sequences values on the target database
13:09:09 82003 INFO  COPY "public"."rental"
13:09:09 82004 INFO  COPY "public"."film"
13:09:09 82009 INFO  COPY "public"."payment_p2020_04"
13:09:09 82002 INFO  Copying large objects
13:09:09 82007 INFO  COPY "public"."payment_p2020_03"
13:09:09 82010 INFO  COPY "public"."film_actor"
13:09:09 82005 INFO  COPY "public"."inventory"
13:09:09 82014 INFO  COPY "public"."payment_p2020_02"
13:09:09 82012 INFO  COPY "public"."customer"
13:09:09 82009 INFO  Creating 3 indexes for table "public"."payment_p2020_04"
13:09:09 82010 INFO  Creating 2 indexes for table "public"."film_actor"
13:09:09 82007 INFO  Creating 3 indexes for table "public"."payment_p2020_03"
13:09:09 82004 INFO  Creating 5 indexes for table "public"."film"
13:09:09 82005 INFO  Creating 2 indexes for table "public"."inventory"
13:09:09 82033 INFO  VACUUM ANALYZE "public"."payment_p2020_04";
13:09:09 82036 INFO  VACUUM ANALYZE "public"."film_actor";
13:09:09 82039 INFO  VACUUM ANALYZE "public"."payment_p2020_03";
13:09:09 82041 INFO  VACUUM ANALYZE "public"."film";
13:09:09 82043 INFO  VACUUM ANALYZE "public"."inventory";
...
...
...
13:09:09 81991 INFO  STEP 7: restore the post-data section to the target database
13:09:09 81991 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_restore --dbname 'postgres://@:/
↪plop?' --single-transaction --clean --if-exists --use-list /var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//
↪pgcopydb/schema/post.list /var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//pgcopydb/schema/post.dump

                                       Step   Connection    Duration   Concurrency
 ---------------------------------------------   ----------  ----------  ------------
                                Dump Schema       source       355ms             1
                             Prepare Schema       target       135ms             1
 COPY, INDEX, CONSTRAINTS, VACUUM (wall clock)         both       641ms        8 + 12
                           COPY (cumulative)         both       1s598             8
                  Large Objects (cumulative)         both        29ms             1
        CREATE INDEX, CONSTRAINTS (cumulative)       target       4s072            12
                              Finalize Schema       target       366ms             1
 ---------------------------------------------   ----------  ----------  ------------
                    Total Wall Clock Duration         both       1s499        8 + 12
 ---------------------------------------------   ----------  ----------  ------------
```

4.3 pgcopydb follow

The command `pgcopydb follow` replays the database changes registered at the source database with the logical decoding plugin of your choice, either the default test_decoding or wal2json, into the target database.

Important: While the `pgcopydb follow` is a full client for logical decoding, the general use case involves using `pgcopydb clone --follow` as documented in *Change Data Capture using Postgres Logical Decoding*.

When using Logical Decoding with pgcopydb or another tool, consider making sure you're familiar with the Logical Replication Restrictions that apply. In particular:

- DDL are not replicated.

When using DDL for partition scheme maintenance, such as when using the pg_partman extension, then consider creating a week or a month of partitions in advance, so that creating new partitions does not happen during the migration window.

- Sequence data is not replicated.

  When using `pgcopydb clone --follow` (starting with pgcopydb version 0.9) then the sequence data is synced at the end of the operation, after the cutover point implemented via the *pgcopydb stream sentinel set endpos*.

  Updating the sequences manually is also possible by running the command *pgcopydb copy sequences*.

- Large Objects are not replicated.

See the Postgres documentation page for Logical Replication Restrictions to read the exhaustive list of restrictions.

4.3.1 pgcopydb follow

```
pgcopydb follow: Replay changes from the source database to the target database
usage: pgcopydb follow  --source ... --target ...

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
  --filters <filename>  Use the filters defined in <filename>
  --restart             Allow restarting when temp files exist already
  --resume              Allow resuming operations after a failure
  --not-consistent      Allow taking a new snapshot on the source database
  --snapshot            Use snapshot obtained with pg_export_snapshot
  --plugin              Output plugin to use (test_decoding, wal2json)
  --slot-name           Use this Postgres replication slot name
  --create-slot         Create the replication slot
  --origin              Use this Postgres replication origin node name
  --endpos              Stop replaying changes when reaching this LSN
```

4.3.2 Description

This command runs three concurrent subprocesses in two possible modes of operation:

- The first mode of operation is named *prefetch and catchup* where the changes from the source database are stored in intermediate JSON and SQL files to be later replayed one file at a time in the catchup process.
- The second mode of operation is named *live replay* where the changes from the source database are streamed from the receiver process to the transform process using a Unix pipe, and then with the same mechanism from the transform process to the replay process.

Only one mode of operation may be active at any given time, and pgcopydb automatically switches from one mode to the other one, in a loop.

The follow command always starts using the *prefetch and catchup* mode, and as soon as the catchup process can't find the next SQL file to replay then it exits, triggering the switch to the *live replay* mode. Before entering the new mode, to make sure to replay all the changes that have been received, pgcopydb implements an extra catchup phase without concurrent activity.

Prefetch and Catchup

In the *prefetch and catchup* mode of operations, the three processes are implementing the following approach:

1. The first process pre-fetches the changes from the source database using the Postgres Logical Decoding protocol and save the JSON messages in local JSON files.
2. The second process transforms the JSON files into SQL. A Unix system V message queue is used to communicate LSN positions from the prefetch process to the transform process.
3. The third process catches-up with changes happening on the source database by applying the SQL files to the target database system.

   The Postgres API for Replication Progress Tracking is used in that process so that we can skip already applied transactions at restart or resume.

Live Replay

In the *live replay* mode of operations, the three processes are implementing the following approach:

1. The first process receives the changes from the source database using the Postgres Logical Decoding protocol and save the JSON messages in local JSON files.

   Additionnaly, the JSON changes are written to a Unix pipe shared with the transform process.
2. The second process transforms the JSON lines into SQL. A Unix pipe is used to stream the JSON lines from the receive process to the transform process.

   The transform process in that mode still writes the changes to SQL files, so that it's still possible to catchup with received changes if the apply process is interrupted.
3. The third process replays the changes happening on the source database by applying the SQL commands to the target database system. The SQL commands are read from the Unix pipe shared with the transform process.

   The Postgres API for Replication Progress Tracking is used in that process so that we can skip already applied transactions at restart or resume.

Remote control of the follow command

It is possible to start the `pgcopydb follow` command and then later, while it's still running, set the LSN for the end position with the same effect as using the command line option `--endpos`, or switch from prefetch mode only to prefetch and catchup mode. For that, see the commands *pgcopydb stream sentinel set endpos*, *pgcopydb stream sentinel set apply*, and *pgcopydb stream sentinel set prefetch*.

Note that in many case the `--endpos` LSN position is not known at the start of this command. Also before entering the *prefetch and apply* mode it is important to make sure that the initial base copy is finished.

Finally, it is also possible to setup the streaming replication options before using the `pgcopydb follow` command: see the *pgcopydb stream setup* and *pgcopydb stream cleanup* commands.

4.3.3 Replica Identity and lack of Primary Keys

Postgres Logical Decoding works with replaying changes using SQL statements, and for that exposes the concept of *Replica Identity* as described in the documentation for the ALTER TABLE … REPLICA IDENTITY command.

To quote Postgres docs:

> *This form changes the information which is written to the write-ahead log to identify rows which are updated or deleted. In most cases, the old value of each column is only logged if it differs from the new value; however, if the old value is stored externally, it is always logged regardless of whether it changed. This option has no effect except when logical replication is in use.*

To support Change Data Capture with Postgres Logical Decoding for tables that do not have a Primary Key, then it is necessary to use the `ALTER TABLE ... REPLICA IDENTITY` command for those tables.

In practice the two following options are to be considered:

- REPLICA IDENTITY USING INDEX index_name

  This form is prefered when a UNIQUE index exists for the table without a primary key. The index must be unique, not partial, not deferrable, and include only columns marked NOT NULL.

- REPLICA IDENTITY FULL

  When this is used on a table, then the WAL records contain the old values of all columns in the row.

4.3.4 Logical Decoding Pre-Fetching

When using `pgcopydb clone --follow` a logical replication slot is created on the source database before the initial COPY, using the same Postgres snapshot. This ensure data consistency.

Within the `pgcopydb clone --follow` approach, it is only possible to start applying the changes from the source database after the initial COPY has finished on the target database.

Also, from the Postgres documentation we read that Postgres replication slots provide an automated way to ensure that the primary does not remove WAL segments until they have been received by all standbys.

Accumulating WAL segments on the primary during the whole duration of the initial COPY involves capacity hazards, which translate into potential *File System is Full* errors on the WAL disk of the source database. It is crucial to avoid such a situation.

This is why pgcopydb implements CDC pre-fetching. In parallel to the initial COPY the command `pgcopydb clone --follow` pre-fetches the changes in local JSON and SQL files. Those files are placed in the XDG_DATA_HOME location, which could be a mount point for an infinite Blob Storage area.

The `pgcopydb follow` command is a convenience command that's available as a logical decoding client, and it shares the same implementation as the `pgcopydb clone --follow` command. As a result, the pre-fetching strategy is also relevant to the `pgcopydb follow` command.

4.3.5 The sentinel table, or the Remote Control

To track progress and allow resuming of operations, pgcopydb uses a sentinel table on the source database. The sentinel table consists of a single row with the following fields:

```
$ pgcopydb stream sentinel get
startpos   1/8D173AF8
endpos     0/0
apply      disabled
write_lsn  0/0
flush_lsn  0/0
replay_lsn 0/0
```

Note that you can use the command `pgcopydb stream sentinel get --json` to fetch a JSON formatted output, such as the following:

```
{
  "startpos": "1/8D173AF8",
  "endpos": "1/8D173AF8",
  "apply": false,
  "write_lsn": "0/0",
  "flush_lsn": "0/0",
  "replay_lsn": "0/0"
}
```

The first three fields (startpos, endpos, apply) are specific to pgcopydb, then the following three fields (write_lsn, flush_lsn, replay_lsn) follow the Postgres replication protocol as visible in the docs for the pg_stat_replication function.

- `startpos`

  The startpos field is the current LSN on the source database at the time when the Change Data Capture is setup in pgcopydb, such as when using the *pgcopydb stream setup* command.

  Note that both the `pgcopydb follow` and the `pgcopydb clone --follow` command implement the setup parts if the `pgcopydb stream setup` has not been used already.

- `endpos`

  The endpos field is last LSN position from the source database that pgcopydb replays. The command `pgcopydb follow` (or `pgcopydb clone --follow`) stops when reaching beyond this LSN position.

  The `endpos` can be set at the start of the process, which is useful for unit testing, or while the command is running, which is useful in production to define a cutover point.

  To define the `endpos` while the command is running, use *pgcopydb stream sentinel set endpos*.

- `apply`

  The apply field is a boolean (enabled/disabled) that control the catchup process. The pgcopydb catchup process replays the changes only when the apply boolean is set to true.

  The `pgcopydb clone --follow` command automatically enables the apply field of the sentinel table as soon as the initial COPY is done.

  To manually control the apply field, use the *pgcopydb stream sentinel set apply* command.

- `write_lsn`

  The Postgres documentation for `pg_stat_replication.write_lsn` is: Last write-ahead log location written to disk by this standby server.

  In the pgcopydb case, the sentinel field write_lsn is the position that has been written to disk (as JSON) by the streaming process.

- `flush_lsn`

The Postgres documentation for `pg_stat_replication.flush_lsn` is: Last write-ahead log location flushed to disk by this standby server

In the pgcopydb case, the sentinel field flush_lsn is the position that has been written and then fsync'ed to disk (as JSON) by the streaming process.

- `replay_lsn`

  The Postgres documentation for `pg_stat_replication.replay_lsn` is: Last write-ahead log location replayed into the database on this standby server

  In the pgcopydb case, the sentinel field replay_lsn is the position that has been applied to the target database, as kept track from the WAL.json and then the WAL.sql files, and using the Postgres API for Replication Progress Tracking.

  The replay_lsn is also shared by the pgcopydb streaming process that uses the Postgres logical replication protocol, so the pg_stat_replication entry associated with the replication slot used by pgcopydb can be used to monitor replication lag.

As the pgcopydb streaming processes maintain the sentinel table on the source database, it is also possible to use it to keep track of the logical replication progress.

4.3.6 Options

The following options are available to `pgcopydb follow`:

--source Connection string to the source Postgres instance. See the Postgres documentation for connection strings for the details. In short both the quoted form `"host=... dbname=..."` and the URI form `postgres://user@host:5432/dbname` are supported.

--target Connection string to the target Postgres instance.

--dir During its normal operations pgcopydb creates a lot of temporary files to track sub-processes progress. Temporary files are created in the directory location given by this option, or defaults to `${TMPDIR}/pgcopydb` when the environment variable is set, or then to `/tmp/pgcopydb`.

--restart When running the pgcopydb command again, if the work directory already contains information from a previous run, then the command refuses to proceed and delete information that might be used for diagnostics and forensics.

In that case, the `--restart` option can be used to allow pgcopydb to delete traces from a previous run.

--resume When the pgcopydb command was terminated before completion, either by an interrupt signal (such as C-c or SIGTERM) or because it crashed, it is possible to resume the database migration.

When resuming activity from a previous run, table data that was fully copied over to the target server is not sent again. Table data that was interrupted during the COPY has to be started from scratch even when using `--resume`: the COPY command in Postgres is transactional and was rolled back.

Same reasonning applies to the CREATE INDEX commands and ALTER TABLE commands that pgcopydb issues, those commands are skipped on a `--resume` run only if known to have run through to completion on the previous one.

Finally, using `--resume` requires the use of `--not-consistent`.

--not-consistent In order to be consistent, pgcopydb exports a Postgres snapshot by calling the pg_export_snapshot() function on the source database server. The snapshot is then re-used in all the connections to the source database server by using the `SET TRANSACTION SNAPSHOT` command.

Per the Postgres documentation about `pg_export_snapshot`:

> Saves the transaction's current snapshot and returns a text string identifying the snapshot. This string must be passed (outside the database) to clients that want to import the snapshot. The snapshot is available for import only until the end of the transaction that exported it.

Now, when the pgcopydb process was interrupted (or crashed) on a previous run, it is possible to resume operations, but the snapshot that was exported does not exists anymore. The pgcopydb command can only resume operations with a new snapshot, and thus can not ensure consistency of the whole data set, because each run is now using their own snapshot.

--snapshot Instead of exporting its own snapshot by calling the PostgreSQL function `pg_export_snapshot()` it is possible for pgcopydb to re-use an already exported snapshot.

--plugin Logical decoding output plugin to use. The default is test_decoding which ships with Postgres core itself, so is probably already available on your source server.

It is possible to use wal2json instead. The support for wal2json is mostly historical in pgcopydb, it should not make a user visible difference whether you use the default test_decoding or wal2json.

--slot-name Logical decoding slot name to use. Defaults to `pgcopydb`. which is unfortunate when your use-case involves migrating more than one database from the source server.

--create-slot Instruct pgcopydb to create the logical replication slot to use.

--endpos Logical decoding target LSN to use. Automatically stop replication and exit with normal exit status 0 when receiving reaches the specified LSN. If there's a record with LSN exactly equal to lsn, the record will be output.

The `--endpos` option is not aware of transaction boundaries and may truncate output partway through a transaction. Any partially output transaction will not be consumed and will be replayed again when the slot is next read from. Individual messages are never truncated.

See also documentation for pg_recvlogical.

--origin Logical replication target system needs to track the transactions that have been applied already, so that in case we get disconnected or need to resume operations we can skip already replayed transaction.

Postgres uses a notion of an origin node name as documented in Replication Progress Tracking. This option allows to pick your own node name and defaults to "pgcopydb". Picking a different name is useful in some advanced scenarios like migrating several sources in the same target, where each source should have their own unique origin node name.

--verbose Increase current verbosity. The default level of verbosity is INFO. In ascending order pgcopydb knows about the following verbosity levels: FATAL, ERROR, WARN, INFO, NOTICE, DEBUG, TRACE.

--debug Set current verbosity to DEBUG level.

--trace Set current verbosity to TRACE level.

--quiet Set current verbosity to ERROR level.

4.3.7 Environment

PGCOPYDB_SOURCE_PGURI

Connection string to the source Postgres instance. When `--source` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_TARGET_PGURI

Connection string to the target Postgres instance. When `--target` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_SNAPSHOT

Postgres snapshot identifier to re-use, see also `--snapshot`.

TMPDIR

The pgcopydb command creates all its work files and directories in `${TMPDIR}/pgcopydb`, and defaults to `/tmp/pgcopydb`.

XDG_DATA_HOME

The standard XDG Base Directory Specification defines several environment variables that allow controling where programs should store their files.

> *XDG_DATA_HOME defines the base directory relative to which user-specific data files should be stored. If $XDG_DATA_HOME is either not set or empty, a default equal to $HOME/.local/share should be used.*

When using Change Data Capture (through `--follow` option and Postgres logical decoding) then pgcopydb pre-fetches changes in JSON files and transform them into SQL files to apply to the target database.

These files are stored at the following location, tried in this order:

1. when `--dir` is used, then pgcopydb uses the `cdc` subdirectory of the `--dir` location,
2. when `XDG_DATA_HOME` is set in the environment, then pgcopydb uses that location,
3. when neither of the previous settings have been used then pgcopydb defaults to using `${HOME}/.local/share`.

4.4 pgcopydb snapshot

pgcopydb snapshot - Create and exports a snapshot on the source database

The command `pgcopydb snapshot` connects to the source database and executes a SQL query to export a snapshot. The obtained snapshot is both printed on stdout and also in a file where other pgcopydb commands might expect to find it.

```
pgcopydb snapshot: Create and exports a snapshot on the source database
usage: pgcopydb snapshot  --source ...

  --source          Postgres URI to the source database
  --dir             Work directory to use
  --follow          Implement logical decoding to replay changes
  --plugin          Output plugin to use (test_decoding, wal2json)
  --slot-name       Use this Postgres replication slot name
```

4.4.1 Options

The following options are available to `pgcopydb create` and `pgcopydb drop` subcommands:

--source Connection string to the source Postgres instance. See the Postgres documentation for connection strings for the details. In short both the quoted form `"host=... dbname=..."` and the URI form `postgres://user@host:5432/dbname` are supported.

--dir During its normal operations pgcopydb creates a lot of temporary files to track sub-processes progress. Temporary files are created in the directory location given by this option, or defaults to `${TMPDIR}/pgcopydb` when the environment variable is set, or then to `/tmp/pgcopydb`.

--follow When the `--follow` option is used then pgcopydb implements Change Data Capture as detailed in the manual page for *pgcopydb follow* in parallel to the main copy database steps.

The replication slot is created using the Postgres replication protocol command CREATE_REPLICATION_SLOT, which then exports the snapshot being used in that command.

--plugin Logical decoding output plugin to use. The default is test_decoding which ships with Postgres core itself, so is probably already available on your source server.

It is possible to use wal2json instead. The support for wal2json is mostly historical in pgcopydb, it should not make a user visible difference whether you use the default test_decoding or wal2json.

--slot-name Logical decoding slot name to use.

--verbose Increase current verbosity. The default level of verbosity is INFO. In ascending order pgcopydb knows about the following verbosity levels: FATAL, ERROR, WARN, INFO, NOTICE, DEBUG, TRACE.

--debug Set current verbosity to DEBUG level.

--trace Set current verbosity to TRACE level.

--quiet Set current verbosity to ERROR level.

4.4.2 Environment

PGCOPYDB_SOURCE_PGURI

Connection string to the source Postgres instance. When `--source` is ommitted from the command line, then this environment variable is used.

4.4.3 Examples

Create a snapshot on the source database in the background:

```
$ pgcopydb snapshot &
[1] 72938
17:31:52 72938 INFO  Running pgcopydb version 0.7.13.gcbf2d16.dirty from "/Users/dim/dev/PostgreSQL/pgcopydb/./
↪src/bin/pgcopydb/pgcopydb"
17:31:52 72938 INFO  Using work dir "/var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//pgcopydb"
17:31:52 72938 INFO  Removing the stale pid file "/var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//pgcopydb/
↪pgcopydb.aux.pid"
17:31:52 72938 INFO  Work directory "/var/folders/d7/zzxmgs9s16gdxxcm0hs0sssw0000gn/T//pgcopydb" already exists
```

(continues on next page)

(continued from previous page)

```
17:31:52 72938 INFO  Exported snapshot "00000003-000CB5FE-1" from the source database
00000003-000CB5FE-1
```

And when the process is done, stop maintaining the snapshot in the background:

```
$ kill %1
17:31:56 72938 INFO  Asked to terminate, aborting
[1]+  Done                    pgcopydb snapshot
```

4.5 pgcopydb copy

pgcopydb copy - Implement the data section of the database copy

This command prefixes the following sub-commands:

```
pgcopydb copy
  db           Copy an entire database from source to target
  roles        Copy the roles from the source instance to the target instance
  extensions   Copy the extensions from the source instance to the target instance
  schema       Copy the database schema from source to target
  data         Copy the data section from source to target
  table-data   Copy the data from all tables in database from source to target
  blobs        Copy the blob data from the source database to the target
  sequences    Copy the current value from all sequences in database from source to target
  indexes      Create all the indexes found in the source database in the target
  constraints  Create all the constraints found in the source database in the target
```

Those commands implement a part of the whole database copy operation as detailed in section *pgcopydb clone*. Only use those commands to debug a specific part, or because you know that you just want to implement that step.

Warning: Using the `pgcopydb clone` command is strongly advised.

This mode of operations is useful for debugging and advanced use cases only.

4.5.1 pgcopydb copy db

pgcopydb copy db - Copy an entire database from source to target

The command `pgcopydb copy db` is an alias for `pgcopydb clone`. See also *pgcopydb clone*.

```
pgcopydb copy db: Copy an entire database from source to target
usage: pgcopydb copy db  --source ... --target ... [ --table-jobs ... --index-jobs ... ]

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
  --table-jobs          Number of concurrent COPY jobs to run
  --index-jobs          Number of concurrent CREATE INDEX jobs to run
  --drop-if-exists      On the target database, clean-up from a previous run first
  --roles               Also copy roles found on source to target
  --no-owner            Do not set ownership of objects to match the original database
  --no-acl              Prevent restoration of access privileges (grant/revoke commands).
  --no-comments         Do not output commands to restore comments
  --skip-large-objects  Skip copying large objects (blobs)
  --filters <filename>  Use the filters defined in <filename>
  --restart             Allow restarting when temp files exist already
  --resume              Allow resuming operations after a failure
  --not-consistent      Allow taking a new snapshot on the source database
  --snapshot            Use snapshot obtained with pg_export_snapshot
```

4.5.2 pgcopydb copy roles

pgcopydb copy roles - Copy the roles from the source instance to the target instance

The command `pgcopydb copy roles` implements both *pgcopydb dump roles* and then *pgcopydb restore roles*.

```
pgcopydb copy roles: Copy the roles from the source instance to the target instance
usage: pgcopydb copy roles  --source ... --target ...

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
  --no-role-passwords   Do not dump passwords for roles
```

Note: In Postgres, roles are a global object. This means roles do not belong to any specific database, and as a result, even when the `pgcopydb` tool otherwise works only in the context of a specific database, this command is not limited to roles that are used within a single database.

When a role already exists on the target database, its restoring is entirely skipped, which includes skipping both the `CREATE ROLE` and the `ALTER ROLE` commands produced by `pg_dumpall --roles-only`.

The `pg_dumpall --roles-only` is used to fetch the list of roles from the source database, and this command includes support for passwords. As a result, this operation requires the superuser privileges.

4.5.3 pgcopydb copy extensions

pgcopydb copy extensions - Copy the extensions from the source instance to the target instance

The command `pgcopydb copy extensions` gets a list of the extensions installed on the source database, and for each of them run the SQL command CREATE EXTENSION IF NOT EXISTS.

```
pgcopydb copy extensions: Copy the extensions from the source instance to the target instance
usage: pgcopydb copy extensions  --source ... --target ...

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
```

When copying extensions, this command also takes care of copying any Extension Configuration Tables user-data to the target database.

4.5.4 pgcopydb copy schema

pgcopydb copy schema - Copy the database schema from source to target

The command `pgcopydb copy schema` implements the schema only section of the clone steps.

```
pgcopydb copy schema: Copy the database schema from source to target
usage: pgcopydb copy schema  --source ... --target ... [ --table-jobs ... --index-jobs ... ]

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
  --filters <filename>  Use the filters defined in <filename>
  --restart             Allow restarting when temp files exist already
  --resume              Allow resuming operations after a failure
  --not-consistent      Allow taking a new snapshot on the source database
  --snapshot            Use snapshot obtained with pg_export_snapshot
```

4.5.5 pgcopydb copy data

pgcopydb copy data - Copy the data section from source to target

The command `pgcopydb copy data` implements the data section of the clone steps.

```
pgcopydb copy data: Copy the data section from source to target
usage: pgcopydb copy data  --source ... --target ... [ --table-jobs ... --index-jobs ... ]

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
  --table-jobs          Number of concurrent COPY jobs to run
  --index-jobs          Number of concurrent CREATE INDEX jobs to run
  --drop-if-exists      On the target database, clean-up from a previous run first
  --no-owner            Do not set ownership of objects to match the original database
  --skip-large-objects  Skip copying large objects (blobs)
  --restart             Allow restarting when temp files exist already
  --resume              Allow resuming operations after a failure
  --not-consistent      Allow taking a new snapshot on the source database
  --snapshot            Use snapshot obtained with pg_export_snapshot
```

Note: The current command line has both the commands `pgcopydb copy table-data` and `pgcopydb copy data`, which are looking quite similar but implement different steps. Be careful for now. This will change later.

The `pgcopydb copy data` command implements the following steps:

```
$ pgcopydb copy table-data
$ pgcopydb copy blobs
$ pgcopydb copy indexes
$ pgcopydb copy constraints
$ pgcopydb copy sequences
$ vacuumdb -z
```

Those steps are actually done concurrently to one another when that's possible, in the same way as the main command `pgcopydb clone` would. The only difference is that the `pgcopydb clone` command also prepares and finishes the schema parts of the operations (pre-data, then post-data), which the `pgcopydb copy data` command ignores.

4.5.6 pgcopydb copy table-data

pgcopydb copy table-data - Copy the data from all tables in database from source to target

The command `pgcopydb copy table-data` fetches the list of tables from the source database and runs a COPY TO command on the source database and sends the result to the target database using a COPY FROM command directly, avoiding disks entirely.

```
pgcopydb copy table-data: Copy the data from all tables in database from source to target
usage: pgcopydb copy table-data  --source ... --target ... [ --table-jobs ... --index-jobs ... ]

  --source          Postgres URI to the source database
  --target          Postgres URI to the target database
  --dir             Work directory to use
  --table-jobs      Number of concurrent COPY jobs to run
  --restart         Allow restarting when temp files exist already
  --resume          Allow resuming operations after a failure
  --not-consistent  Allow taking a new snapshot on the source database
  --snapshot        Use snapshot obtained with pg_export_snapshot
```

4.5.7 pgcopydb copy blobs

pgcopydb copy blobs - Copy the blob data from ther source database to the target

The command `pgcopydb copy blobs` fetches list of large objects (aka blobs) from the source database and copies their data parts to the target database. By default the command assumes that the large objects metadata have already been taken care of, because of the behaviour of `pg_dump --section=pre-data`.

```
pgcopydb copy blobs: Copy the blob data from ther source database to the target
usage: pgcopydb copy blobs  --source ... --target ...

  --source          Postgres URI to the source database
  --target          Postgres URI to the target database
  --dir             Work directory to use
  --restart         Allow restarting when temp files exist already
  --resume          Allow resuming operations after a failure
  --not-consistent  Allow taking a new snapshot on the source database
  --snapshot        Use snapshot obtained with pg_export_snapshot
  --drop-if-exists  On the target database, drop and create large objects
```

4.5.8 pgcopydb copy sequences

pgcopydb copy sequences - Copy the current value from all sequences in database from source to target

The command `pgcopydb copy sequences` fetches the list of sequences from the source database, then for each sequence fetches the `last_value` and `is_called` properties the same way pg_dump would on the source database, and then for each sequence call `pg_catalog.setval()` on the target database.

```
pgcopydb copy sequences: Copy the current value from all sequences in database from source to target
usage: pgcopydb copy sequences  --source ... --target ... [ --table-jobs ... --index-jobs ... ]

  --source          Postgres URI to the source database
  --target          Postgres URI to the target database
  --dir             Work directory to use
  --restart         Allow restarting when temp files exist already
  --resume          Allow resuming operations after a failure
  --not-consistent  Allow taking a new snapshot on the source database
```

4.5.9 pgcopydb copy indexes

pgcopydb copy indexes - Create all the indexes found in the source database in the target

The command `pgcopydb copy indexes` fetches the list of indexes from the source database and runs each index CREATE INDEX statement on the target database. The statements for the index definitions are modified to include IF NOT EXISTS and allow for skipping indexes that already exist on the target database.

```
pgcopydb copy indexes: Create all the indexes found in the source database in the target
usage: pgcopydb copy indexes  --source ... --target ... [ --table-jobs ... --index-jobs ... ]

  --source          Postgres URI to the source database
  --target          Postgres URI to the target database
  --dir             Work directory to use
      --index-jobs      Number of concurrent CREATE INDEX jobs to run
  --restart         Allow restarting when temp files exist already
  --resume          Allow resuming operations after a failure
  --not-consistent  Allow taking a new snapshot on the source database
```

4.5.10 pgcopydb copy constraints

pgcopydb copy constraints - Create all the constraints found in the source database in the target

The command `pgcopydb copy constraints` fetches the list of indexes from the source database and runs each index ALTER TABLE … ADD CONSTRAINT … USING INDEX statement on the target database.

The indexes must already exist, and the command will fail if any constraint is found existing already on the target database.

```
pgcopydb copy indexes: Create all the indexes found in the source database in the target
usage: pgcopydb copy indexes  --source ... --target ... [ --table-jobs ... --index-jobs ... ]

  --source          Postgres URI to the source database
  --target          Postgres URI to the target database
  --dir             Work directory to use
  --restart         Allow restarting when temp files exist already
  --resume          Allow resuming operations after a failure
  --not-consistent  Allow taking a new snapshot on the source data
```

4.5.11 Description

These commands allow implementing a specific step of the pgcopydb operations at a time. It's useful mainly for debugging purposes, though some advanced and creative usage can be made from the commands.

The target schema is not created, so it needs to have been taken care of first. It is possible to use the commands *pgcopydb dump schema* and then *pgcopydb restore pre-data* to prepare your target database.

To implement the same operations as a `pgcopydb clone` command would, use the following recipe:

```
$ export PGCOPYDB_SOURCE_PGURI="postgres://user@source/dbname"
$ export PGCOPYDB_TARGET_PGURI="postgres://user@target/dbname"

$ pgcopydb dump schema
$ pgcopydb restore pre-data --resume --not-consistent
$ pgcopydb copy table-data --resume --not-consistent
$ pgcopydb copy sequences --resume --not-consistent
$ pgcopydb copy indexes --resume --not-consistent
$ pgcopydb copy constraints --resume --not-consistent
$ vacuumdb -z
$ pgcopydb restore post-data --resume --not-consistent
```

The main `pgcopydb clone` is still better at concurrency than doing those steps manually, as it will create the indexes for any given table as soon as the table-data section is finished, without having to wait until the last table-data has been copied over. Same applies to constraints, and then vacuum analyze.

4.5.12 Options

The following options are available to `pgcopydb copy` sub-commands:

--source Connection string to the source Postgres instance. See the Postgres documentation for connection strings for the details. In short both the quoted form `"host=... dbname=..."` and the URI form `postgres://user@host:5432/dbname` are supported.

--target Connection string to the target Postgres instance.

--dir During its normal operations pgcopydb creates a lot of temporary files to track sub-processes progress. Temporary files are created in the directory location given by this option, or defaults to `${TMPDIR}/pgcopydb` when the environment variable is set, or then to `/tmp/pgcopydb`.

--no-role-passwords Do not dump passwords for roles. When restored, roles will have a null password, and password authentication will always fail until the password is set. Since password values aren't needed when this option is specified, the role information is read from the catalog view pg_roles instead of pg_authid. Therefore, this option also helps if access to pg_authid is restricted by some security policy.

--table-jobs How many tables can be processed in parallel.

This limit only applies to the COPY operations, more sub-processes will be running at the same time that this limit while the CREATE INDEX operations are in progress, though then the processes are only waiting for the target Postgres instance to do all the work.

--index-jobs How many indexes can be built in parallel, globally. A good option is to set this option to the count of CPU cores that are available on the Postgres target system, minus some cores that are going to be used for handling the COPY operations.

--split-tables-larger-than Allow *Same-table Concurrency* when processing the source database. This environment variable value is expected to be a byte size, and bytes units B, kB, MB, GB, TB, PB, and EB are known.

--skip-large-objects Skip copying large objects, also known as blobs, when copying the data from the source database to the target database.

--restart When running the pgcopydb command again, if the work directory already contains information from a previous run, then the command refuses to proceed and delete information that might be used for diagnostics and forensics.

In that case, the `--restart` option can be used to allow pgcopydb to delete traces from a previous run.

--resume When the pgcopydb command was terminated before completion, either by an interrupt signal (such as C-c or SIGTERM) or because it crashed, it is possible to resume the database migration.

When resuming activity from a previous run, table data that was fully copied over to the target server is not sent again. Table data that was interrupted during the COPY has to be started from scratch even when using `--resume`: the COPY command in Postgres is transactional and was rolled back.

Same reasonning applies to the CREATE INDEX commands and ALTER TABLE commands that pgcopydb issues, those commands are skipped on a `--resume` run only if known to have run through to completion on the previous one.

Finally, using `--resume` requires the use of `--not-consistent`.

--not-consistent In order to be consistent, pgcopydb exports a Postgres snapshot by calling the pg_export_snapshot() function on the source database server. The snapshot is then re-used in all the connections to the source database server by using the `SET TRANSACTION SNAPSHOT` command.

Per the Postgres documentation about `pg_export_snapshot`:

> Saves the transaction's current snapshot and returns a text string identifying the snapshot. This string must be passed (outside the database) to clients that want to import the snapshot. The snapshot is available for import only until the end of the transaction that exported it.

Now, when the pgcopydb process was interrupted (or crashed) on a previous run, it is possible to resume operations, but the snapshot that was exported does not exists anymore. The pgcopydb command can only resume operations with a new snapshot,

and thus can not ensure consistency of the whole data set, because each run is now using their own snapshot.

--snapshot Instead of exporting its own snapshot by calling the PostgreSQL function `pg_export_snapshot()` it is possible for pgcopydb to re-use an already exported snapshot.

--verbose Increase current verbosity. The default level of verbosity is INFO. In ascending order pgcopydb knows about the following verbosity levels: FATAL, ERROR, WARN, INFO, NOTICE, DEBUG, TRACE.

--debug Set current verbosity to DEBUG level.

--trace Set current verbosity to TRACE level.

--quiet Set current verbosity to ERROR level.

4.5.13 Environment

PGCOPYDB_SOURCE_PGURI

Connection string to the source Postgres instance. When `--source` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_TARGET_PGURI

Connection string to the target Postgres instance. When `--target` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_TABLE_JOBS

Number of concurrent jobs allowed to run COPY operations in parallel. When `--table-jobs` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_INDEX_JOBS

Number of concurrent jobs allowed to run CREATE INDEX operations in parallel. When `--index-jobs` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_SPLIT_TABLES_LARGER_THAN

Allow *Same-table Concurrency* when processing the source database. This environment variable value is expected to be a byte size, and bytes units B, kB, MB, GB, TB, PB, and EB are known.

When `--split-tables-larger-than` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_DROP_IF_EXISTS

When true (or *yes*, or *on*, or 1, same input as a Postgres boolean) then pgcopydb uses the pg_restore options `--clean --if-exists` when creating the schema on the target Postgres instance.

PGCOPYDB_SNAPSHOT

Postgres snapshot identifier to re-use, see also `--snapshot`.

TMPDIR

The pgcopydb command creates all its work files and directories in `${TMPDIR}/pgcopydb`, and defaults to `/tmp/pgcopydb`.

4.5.14 Examples

Let's export the Postgres databases connection strings to make it easy to re-use them all along:

```
$ export PGCOPYDB_SOURCE_PGURI="port=54311 host=localhost dbname=pgloader"
$ export PGCOPYDB_TARGET_PGURI="port=54311 dbname=plop"
```

Now, first dump the schema:

```
$ pgcopydb dump schema
15:24:24 75511 INFO  Removing the stale pid file "/tmp/pgcopydb/pgcopydb.pid"
15:24:24 75511 WARN  Directory "/tmp/pgcopydb" already exists: removing it entirely
15:24:24 75511 INFO  Dumping database from "port=54311 host=localhost dbname=pgloader"
15:24:24 75511 INFO  Dumping database into directory "/tmp/pgcopydb"
15:24:24 75511 INFO  Using pg_dump for Postgres "12.9" at "/Applications/Postgres.app/Contents/Versions/12/bin/
↪pg_dump"
15:24:24 75511 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_dump -Fc --section pre-data --
↪file /tmp/pgcopydb/schema/pre.dump 'port=54311 host=localhost dbname=pgloader'
15:24:25 75511 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_dump -Fc --section post-data --
↪file /tmp/pgcopydb/schema/post.dump 'port=54311 host=localhost dbname=pgloader'
```

Now restore the pre-data schema on the target database, cleaning up the already existing objects if any, which allows running this test scenario again and again. It might not be what you want to do in your production target instance though!

```
PGCOPYDB_DROP_IF_EXISTS=on pgcopydb restore pre-data --no-owner
15:24:29 75591 INFO  Removing the stale pid file "/tmp/pgcopydb/pgcopydb.pid"
15:24:29 75591 INFO  Restoring database from "/tmp/pgcopydb"
15:24:29 75591 INFO  Restoring database into "port=54311 dbname=plop"
15:24:29 75591 INFO  Using pg_restore for Postgres "12.9" at "/Applications/Postgres.app/Contents/Versions/12/
↪bin/pg_restore"
15:24:29 75591 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_restore --dbname 'port=54311␣
↪dbname=plop' --clean --if-exists --no-owner /tmp/pgcopydb/schema/pre.dump
```

Then copy the data over:

```
$ pgcopydb copy table-data --resume --not-consistent
15:24:36 75688 INFO  [SOURCE] Copying database from "port=54311 host=localhost dbname=pgloader"
15:24:36 75688 INFO  [TARGET] Copying database into "port=54311 dbname=plop"
15:24:36 75688 INFO  Removing the stale pid file "/tmp/pgcopydb/pgcopydb.pid"
15:24:36 75688 INFO  STEP 3: copy data from source to target in sub-processes
15:24:36 75688 INFO  Listing ordinary tables in "port=54311 host=localhost dbname=pgloader"
15:24:36 75688 INFO  Fetched information for 56 tables
...
                                       Step   Connection    Duration   Concurrency
 ---------------------------------------------   ----------  ----------  ------------
                                Dump Schema       source          0ms             1
                             Prepare Schema       target          0ms             1
 COPY, INDEX, CONSTRAINTS, VACUUM (wall clock)         both          0ms         4 + 4
                           COPY (cumulative)         both       1s140             4
                     CREATE INDEX (cumulative)       target          0ms             4
                                Finalize Schema      target          0ms             1
 ---------------------------------------------   ----------  ----------  ------------
                     Total Wall Clock Duration         both       2s143         4 + 4
 ---------------------------------------------   ----------  ----------  ------------
```

And now create the indexes on the target database, using the index definitions from the source database:

```
$ pgcopydb copy indexes --resume --not-consistent
15:24:40 75918 INFO  [SOURCE] Copying database from "port=54311 host=localhost dbname=pgloader"
15:24:40 75918 INFO  [TARGET] Copying database into "port=54311 dbname=plop"
15:24:40 75918 INFO  Removing the stale pid file "/tmp/pgcopydb/pgcopydb.pid"
15:24:40 75918 INFO  STEP 4: create indexes in parallel
15:24:40 75918 INFO  Listing ordinary tables in "port=54311 host=localhost dbname=pgloader"
15:24:40 75918 INFO  Fetched information for 56 tables
15:24:40 75930 INFO  Creating 2 indexes for table "csv"."partial"
15:24:40 75922 INFO  Creating 1 index for table "csv"."track"
15:24:40 75931 INFO  Creating 1 index for table "err"."errors"
15:24:40 75928 INFO  Creating 1 index for table "csv"."blocks"
15:24:40 75925 INFO  Creating 1 index for table "public"."track_full"
15:24:40 76037 INFO  CREATE INDEX IF NOT EXISTS partial_b_idx ON csv.partial USING btree (b);
15:24:40 76036 INFO  CREATE UNIQUE INDEX IF NOT EXISTS track_pkey ON csv.track USING btree (trackid);
15:24:40 76035 INFO  CREATE UNIQUE INDEX IF NOT EXISTS partial_a_key ON csv.partial USING btree (a);
15:24:40 76038 INFO  CREATE UNIQUE INDEX IF NOT EXISTS errors_pkey ON err.errors USING btree (a);
```

(continues on next page)

(continued from previous page)

```
15:24:40 75987 INFO  Creating 1 index for table "public"."xzero"
15:24:40 75969 INFO  Creating 1 index for table "public"."csv_escape_mode"
15:24:40 75985 INFO  Creating 1 index for table "public"."udc"
15:24:40 75965 INFO  Creating 1 index for table "public"."allcols"
15:24:40 75981 INFO  Creating 1 index for table "public"."serial"
15:24:40 76039 INFO  CREATE INDEX IF NOT EXISTS blocks_ip4r_idx ON csv.blocks USING gist (iprange);
15:24:40 76040 INFO  CREATE UNIQUE INDEX IF NOT EXISTS track_full_pkey ON public.track_full USING btree␣
↪(trackid);
15:24:40 75975 INFO  Creating 1 index for table "public"."nullif"
15:24:40 76046 INFO  CREATE UNIQUE INDEX IF NOT EXISTS xzero_pkey ON public.xzero USING btree (a);
15:24:40 76048 INFO  CREATE UNIQUE INDEX IF NOT EXISTS udc_pkey ON public.udc USING btree (b);
15:24:40 76047 INFO  CREATE UNIQUE INDEX IF NOT EXISTS csv_escape_mode_pkey ON public.csv_escape_mode USING␣
↪btree (id);
15:24:40 76049 INFO  CREATE UNIQUE INDEX IF NOT EXISTS allcols_pkey ON public.allcols USING btree (a);
15:24:40 76052 INFO  CREATE UNIQUE INDEX IF NOT EXISTS nullif_pkey ON public."nullif" USING btree (id);
15:24:40 76050 INFO  CREATE UNIQUE INDEX IF NOT EXISTS serial_pkey ON public.serial USING btree (a);

                                       Step   Connection    Duration   Concurrency
 ---------------------------------------------  ----------  ----------  ------------
                                Dump Schema       source          0ms             1
                              Prepare Schema       target          0ms             1
 COPY, INDEX, CONSTRAINTS, VACUUM (wall clock)         both          0ms         4 + 4
                                COPY (cumulative)         both       619ms             4
                       CREATE INDEX (cumulative)       target       1s023             4
                              Finalize Schema       target          0ms             1
 ---------------------------------------------  ----------  ----------  ------------
                       Total Wall Clock Duration         both       400ms         4 + 4
 ---------------------------------------------  ----------  ----------  ------------
```

Now re-create the constraints (primary key, unique constraints) from the source database schema into the target database:

```
$ pgcopydb copy constraints --resume --not-consistent
15:24:43 76095 INFO  [SOURCE] Copying database from "port=54311 host=localhost dbname=pgloader"
15:24:43 76095 INFO  [TARGET] Copying database into "port=54311 dbname=plop"
15:24:43 76095 INFO  Removing the stale pid file "/tmp/pgcopydb/pgcopydb.pid"
15:24:43 76095 INFO  STEP 4: create constraints
15:24:43 76095 INFO  Listing ordinary tables in "port=54311 host=localhost dbname=pgloader"
15:24:43 76095 INFO  Fetched information for 56 tables
15:24:43 76099 INFO  ALTER TABLE "csv"."track" ADD CONSTRAINT "track_pkey" PRIMARY KEY USING INDEX "track_pkey
↪";
15:24:43 76107 INFO  ALTER TABLE "csv"."partial" ADD CONSTRAINT "partial_a_key" UNIQUE USING INDEX "partial_a_
↪key";
15:24:43 76102 INFO  ALTER TABLE "public"."track_full" ADD CONSTRAINT "track_full_pkey" PRIMARY KEY USING␣
↪INDEX "track_full_pkey";
15:24:43 76142 INFO  ALTER TABLE "public"."allcols" ADD CONSTRAINT "allcols_pkey" PRIMARY KEY USING INDEX
↪"allcols_pkey";
15:24:43 76157 INFO  ALTER TABLE "public"."serial" ADD CONSTRAINT "serial_pkey" PRIMARY KEY USING INDEX
↪"serial_pkey";
15:24:43 76161 INFO  ALTER TABLE "public"."xzero" ADD CONSTRAINT "xzero_pkey" PRIMARY KEY USING INDEX "xzero_
↪pkey";
15:24:43 76146 INFO  ALTER TABLE "public"."csv_escape_mode" ADD CONSTRAINT "csv_escape_mode_pkey" PRIMARY KEY␣
↪USING INDEX "csv_escape_mode_pkey";
15:24:43 76154 INFO  ALTER TABLE "public"."nullif" ADD CONSTRAINT "nullif_pkey" PRIMARY KEY USING INDEX
↪"nullif_pkey";
15:24:43 76159 INFO  ALTER TABLE "public"."udc" ADD CONSTRAINT "udc_pkey" PRIMARY KEY USING INDEX "udc_pkey";
15:24:43 76108 INFO  ALTER TABLE "err"."errors" ADD CONSTRAINT "errors_pkey" PRIMARY KEY USING INDEX "errors_
↪pkey";

                                       Step   Connection    Duration   Concurrency
 ---------------------------------------------  ----------  ----------  ------------
                                Dump Schema       source          0ms             1
                              Prepare Schema       target          0ms             1
 COPY, INDEX, CONSTRAINTS, VACUUM (wall clock)         both          0ms         4 + 4
                                COPY (cumulative)         both       605ms             4
                       CREATE INDEX (cumulative)       target       1s023             4
                              Finalize Schema       target          0ms             1
 ---------------------------------------------  ----------  ----------  ------------
                       Total Wall Clock Duration         both       415ms         4 + 4
 ---------------------------------------------  ----------  ----------  ------------
```

The next step is a VACUUM ANALYZE on each table that's been just filled-in with the data, and for that we can just use the vacuumdb command from Postgres:

```
$ vacuumdb --analyze --dbname "$PGCOPYDB_TARGET_PGURI" --jobs 4
vacuumdb: vacuuming database "plop"
```

Finally we can restore the post-data section of the schema:

```
$ pgcopydb restore post-data --resume --not-consistent
15:24:50 76328 INFO  Removing the stale pid file "/tmp/pgcopydb/pgcopydb.pid"
15:24:50 76328 INFO  Restoring database from "/tmp/pgcopydb"
15:24:50 76328 INFO  Restoring database into "port=54311 dbname=plop"
15:24:50 76328 INFO  Using pg_restore for Postgres "12.9" at "/Applications/Postgres.app/Contents/Versions/12/
↪bin/pg_restore"
15:24:50 76328 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_restore --dbname 'port=54311␣
↪dbname=plop' --use-list /tmp/pgcopydb/schema/post.list /tmp/pgcopydb/schema/post.dump
```

4.6 pgcopydb dump

pgcopydb dump - Dump database objects from a Postgres instance

This command prefixes the following sub-commands:

```
pgcopydb dump
  schema     Dump source database schema as custom files in target directory
  pre-data   Dump source database pre-data schema as custom files in target directory
  post-data  Dump source database post-data schema as custom files in target directory
  roles      Dump source database roles as custome file in work directory
```

4.6.1 pgcopydb dump schema

pgcopydb dump schema - Dump source database schema as custom files in target directory

The command `pgcopydb dump schema` uses pg_dump to export SQL schema definitions from the given source Postgres instance.

```
pgcopydb dump schema: Dump source database schema as custom files in target directory
usage: pgcopydb dump schema  --source <URI> --target <dir>

  --source          Postgres URI to the source database
  --target          Directory where to save the dump files
  --dir             Work directory to use
  --snapshot        Use snapshot obtained with pg_export_snapshot
```

4.6.2 pgcopydb dump pre-data

pgcopydb dump pre-data - Dump source database pre-data schema as custom files in target directory

The command `pgcopydb dump pre-data` uses pg_dump to export SQL schema *pre-data* definitions from the given source Postgres instance.

```
pgcopydb dump pre-data: Dump source database pre-data schema as custom files in target directory
usage: pgcopydb dump schema  --source <URI> --target <dir>

  --source          Postgres URI to the source database
  --target          Directory where to save the dump files
  --dir             Work directory to use
  --snapshot        Use snapshot obtained with pg_export_snapshot
```

4.6.3 pgcopydb dump post-data

pgcopydb dump post-data - Dump source database post-data schema as custom files in target directory

The command `pgcopydb dump post-data` uses pg_dump to export SQL schema *post-data* definitions from the given source Postgres instance.

```
pgcopydb dump post-data: Dump source database post-data schema as custom files in target directory
usage: pgcopydb dump schema  --source <URI> --target <dir>

  --source          Postgres URI to the source database
  --target          Directory where to save the dump files
  --dir             Work directory to use
  --snapshot        Use snapshot obtained with pg_export_snapshot
```

4.6.4 pgcopydb dump roles

pgcopydb dump roles - Dump source database roles as custome file in work directory

The command `pgcopydb dump roles` uses pg_dumpall –roles-only to export SQL definitions of the roles found on the source Postgres instance.

```
pgcopydb dump roles: Dump source database roles as custome file in work directory
usage: pgcopydb dump roles  --source <URI>

  --source            Postgres URI to the source database
  --target            Directory where to save the dump files
  --dir               Work directory to use
  --no-role-passwords Do not dump passwords for roles
```

The `pg_dumpall --roles-only` is used to fetch the list of roles from the source database, and this command includes support for passwords. As a result, this operation requires the superuser privileges.

It is possible to use the option `--no-role-passwords` to operate without superuser privileges. In that case though, the passwords are not part of the dump and authentication might fail until passwords have been setup properly.

4.6.5 Description

The `pgcopydb dump schema` command implements the first step of the full database migration and fetches the schema definitions from the source database.

When the command runs, it calls `pg_dump` to get first the pre-data schema output in a Postgres custom file, and then again to get the post-data schema output in another Postgres custom file.

The output files are written to the `schema` sub-directory of the `--target` directory.

The `pgcopydb dump pre-data` and `pgcopydb dump post-data` are limiting their action to respectively the pre-data and the post-data sections of the pg_dump.

4.6.6 Options

The following options are available to `pgcopydb dump schema`:

--source Connection string to the source Postgres instance. See the Postgres documentation for connection strings for the details. In short both the quoted form `"host=... dbname=..."` and the URI form `postgres://user@host:5432/dbname` are supported.

--target Connection string to the target Postgres instance.

--dir During its normal operations pgcopydb creates a lot of temporary files to track sub-processes progress. Temporary files are created in the directory location given by this option, or defaults to `${TMPDIR}/pgcopydb` when the environment variable is set, or then to `/tmp/pgcopydb`.

--no-role-passwords Do not dump passwords for roles. When restored, roles will have a null password, and password authentication will always fail until the password is set. Since password values aren't needed when this option is specified, the role information is read from the catalog view pg_roles instead of pg_authid. Therefore, this option also helps if access to pg_authid is restricted by some security policy.

--snapshot Instead of exporting its own snapshot by calling the PostgreSQL function `pg_export_snapshot()` it is possible for pgcopydb to re-use an already exported snapshot.

--verbose Increase current verbosity. The default level of verbosity is INFO. In ascending order pgcopydb knows about the following verbosity levels: FATAL, ERROR, WARN, INFO, NOTICE, DEBUG, TRACE.

--debug Set current verbosity to DEBUG level.

--trace Set current verbosity to TRACE level.

--quiet Set current verbosity to ERROR level.

4.6.7 Environment

PGCOPYDB_SOURCE_PGURI

Connection string to the source Postgres instance. When `--source` is ommitted from the command line, then this environment variable is used.

4.6.8 Examples

First, using `pgcopydb dump schema`

```
$ pgcopydb dump schema --source "port=5501 dbname=demo" --target /tmp/target
09:35:21 3926 INFO  Dumping database from "port=5501 dbname=demo"
09:35:21 3926 INFO  Dumping database into directory "/tmp/target"
09:35:21 3926 INFO  Found a stale pidfile at "/tmp/target/pgcopydb.pid"
09:35:21 3926 WARN  Removing the stale pid file "/tmp/target/pgcopydb.pid"
09:35:21 3926 INFO  Using pg_dump for Postgres "12.9" at "/Applications/Postgres.app/Contents/Versions/12/bin/
↪pg_dump"
09:35:21 3926 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_dump -Fc --section pre-data --file␣
↪/tmp/target/schema/pre.dump 'port=5501 dbname=demo'
09:35:22 3926 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_dump -Fc --section post-data --
↪file /tmp/target/schema/post.dump 'port=5501 dbname=demo'
```

Once the previous command is finished, the pg_dump output files can be found in `/tmp/target/schema` and are named `pre.dump` and `post.dump`. Other files and directories have been created.

```
$ find /tmp/target
/tmp/target
/tmp/target/pgcopydb.pid
/tmp/target/schema
/tmp/target/schema/post.dump
/tmp/target/schema/pre.dump
/tmp/target/run
/tmp/target/run/tables
/tmp/target/run/indexes
```

Then we have almost the same thing when using the other forms.

We can see that `pgcopydb dump pre-data` only does the pre-data section of the dump.

```
$ pgcopydb dump pre-data --source "port=5501 dbname=demo" --target /tmp/target
09:35:21 3926 INFO  Dumping database from "port=5501 dbname=demo"
09:35:21 3926 INFO  Dumping database into directory "/tmp/target"
09:35:21 3926 INFO  Found a stale pidfile at "/tmp/target/pgcopydb.pid"
09:35:21 3926 WARN  Removing the stale pid file "/tmp/target/pgcopydb.pid"
09:35:21 3926 INFO  Using pg_dump for Postgres "12.9" at "/Applications/Postgres.app/Contents/Versions/12/bin/
↪pg_dump"
09:35:21 3926 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_dump -Fc --section pre-data --file␣
↪/tmp/target/schema/pre.dump 'port=5501 dbname=demo'
```

And then `pgcopydb dump post-data` only does the post-data section of the dump.

```
$ pgcopydb dump post-data --source "port=5501 dbname=demo" --target /tmp/target
09:35:21 3926 INFO  Dumping database from "port=5501 dbname=demo"
09:35:21 3926 INFO  Dumping database into directory "/tmp/target"
09:35:21 3926 INFO  Found a stale pidfile at "/tmp/target/pgcopydb.pid"
09:35:21 3926 WARN  Removing the stale pid file "/tmp/target/pgcopydb.pid"
09:35:21 3926 INFO  Using pg_dump for Postgres "12.9" at "/Applications/Postgres.app/Contents/Versions/12/bin/
↪pg_dump"
09:35:21 3926 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_dump -Fc --section post-data --
↪file /tmp/target/schema/post.dump 'port=5501 dbname=demo'
```

4.7 pgcopydb restore

pgcopydb restore - Restore database objects into a Postgres instance

This command prefixes the following sub-commands:

```
pgcopydb restore
  schema      Restore a database schema from custom files to target database
  pre-data    Restore a database pre-data schema from custom file to target database
  post-data   Restore a database post-data schema from custom file to target database
  roles       Restore database roles from SQL file to target database
  parse-list  Parse pg_restore --list output from custom file
```

4.7.1 pgcopydb restore schema

pgcopydb restore schema - Restore a database schema from custom files to target database

The command `pgcopydb restore schema` uses pg_restore to create the SQL schema definitions from the given `pgcopydb dump schema` export directory. This command is not compatible with using Postgres files directly, it must be fed with the directory output from the `pgcopydb dump ...` commands.

```
pgcopydb restore schema: Restore a database schema from custom files to target database
usage: pgcopydb restore schema  --dir <dir> [ --source <URI> ] --target <URI>

  --source             Postgres URI to the source database
  --target             Postgres URI to the target database
  --dir                Work directory to use
  --drop-if-exists     On the target database, clean-up from a previous run first
  --no-owner           Do not set ownership of objects to match the original database
  --no-acl             Prevent restoration of access privileges (grant/revoke commands).
  --no-comments        Do not output commands to restore comments
  --filters <filename> Use the filters defined in <filename>
  --restart            Allow restarting when temp files exist already
  --resume             Allow resuming operations after a failure
  --not-consistent     Allow taking a new snapshot on the source database
```

4.7.2 pgcopydb restore pre-data

pgcopydb restore pre-data - Restore a database pre-data schema from custom file to target database

The command `pgcopydb restore pre-data` uses pg_restore to create the SQL schema definitions from the given `pgcopydb dump schema` export directory. This command is not compatible with using Postgres files directly, it must be fed with the directory output from the `pgcopydb dump ...` commands.

```
pgcopydb restore pre-data: Restore a database pre-data schema from custom file to target database
usage: pgcopydb restore pre-data  --dir <dir> [ --source <URI> ] --target <URI>

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
  --drop-if-exists      On the target database, clean-up from a previous run first
  --no-owner            Do not set ownership of objects to match the original database
  --no-acl              Prevent restoration of access privileges (grant/revoke commands).
  --no-comments         Do not output commands to restore comments
  --filters <filename> Use the filters defined in <filename>
  --restart             Allow restarting when temp files exist already
  --resume              Allow resuming operations after a failure
  --not-consistent      Allow taking a new snapshot on the source database
```

4.7.3 pgcopydb restore post-data

pgcopydb restore post-data - Restore a database post-data schema from custom file to target database

The command `pgcopydb restore post-data` uses pg_restore to create the SQL schema definitions from the given `pgcopydb dump schema` export directory. This command is not compatible with using Postgres files directly, it must be fed with the directory output from the `pgcopydb dump ...` commands.

```
pgcopydb restore post-data: Restore a database post-data schema from custom file to target database
usage: pgcopydb restore post-data  --dir <dir> [ --source <URI> ] --target <URI>

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
  --no-owner            Do not set ownership of objects to match the original database
  --no-acl              Prevent restoration of access privileges (grant/revoke commands).
  --no-comments         Do not output commands to restore comments
  --filters <filename> Use the filters defined in <filename>
  --restart             Allow restarting when temp files exist already
  --resume              Allow resuming operations after a failure
  --not-consistent      Allow taking a new snapshot on the source database
```

4.7.4 pgcopydb restore roles

pgcopydb restore roles - Restore database roles from SQL file to target database

The command `pgcopydb restore roles` runs the commands from the SQL script obtained from the command `pgcopydb dump roles`. Roles that already exist on the target database are skipped.

The `pg_dumpall` command issues two lines per role, the first one is a `CREATE ROLE` SQL command, the second one is an `ALTER ROLE` SQL command. Both those lines are skipped when the role already exists on the target database.

```
pgcopydb restore roles: Restore database roles from SQL file to target database
usage: pgcopydb restore roles  --dir <dir> [ --source <URI> ] --target <URI>

  --source              Postgres URI to the source database
  --target              Postgres URI to the target database
  --dir                 Work directory to use
```

4.7.5 pgcopydb restore parse-list

pgcopydb restore parse-list - Parse pg_restore –list output from custom file

The command `pgcopydb restore parse-list` outputs pg_restore to list the archive catalog of the custom file format file that has been exported for the post-data section.

When using the `--filters` option , then the source database connection is used to grab all the dependend objects that should also be filtered, and the output of the command shows those pg_restore catalog entries commented out.

A pg_restore archive catalog entry is commented out when its line starts with a semi-colon character (;).

```
pgcopydb restore parse-list: Parse pg_restore --list output from custom file
usage: pgcopydb restore parse-list  --dir <dir> [ --source <URI> ] --target <URI>

  --source             Postgres URI to the source database
  --target             Postgres URI to the target database
  --dir                Work directory to use
  --filters <filename> Use the filters defined in <filename>
  --skip-extensions    Skip restoring extensions
  --restart            Allow restarting when temp files exist already
  --resume             Allow resuming operations after a failure
  --not-consistent     Allow taking a new snapshot on the source database
```

4.7.6 Description

The `pgcopydb restore schema` command implements the creation of SQL objects in the target database, second and last steps of a full database migration.

When the command runs, it calls `pg_restore` on the files found at the expected location within the `--target` directory, which has typically been created with the `pgcopydb dump schema` command.

The `pgcopydb restore pre-data` and `pgcopydb restore post-data` are limiting their action to respectively the pre-data and the post-data files in the source directory..

4.7.7 Options

The following options are available to `pgcopydb restore schema`:

--source Connection string to the source Postgres instance. See the Postgres documentation for connection strings for the details. In short both the quoted form `"host=... dbname=..."` and the URI form `postgres://user@host:5432/dbname` are supported.

--target Connection string to the target Postgres instance.

--dir During its normal operations pgcopydb creates a lot of temporary files to track sub-processes progress. Temporary files are created in the directory location given by this option, or defaults to `${TMPDIR}/pgcopydb` when the environment variable is set, or then to `/tmp/pgcopydb`.

--drop-if-exists When restoring the schema on the target Postgres instance, `pgcopydb` actually uses `pg_restore`. When this options is specified, then the following pg_restore options are also used: `--clean --if-exists`.

This option is useful when the same command is run several times in a row, either to fix a previous mistake or for instance when used in a continuous integration system.

This option causes `DROP TABLE` and `DROP INDEX` and other DROP commands to be used. Make sure you understand what you're doing here!

--no-owner Do not output commands to set ownership of objects to match the original database. By default, `pg_restore` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail unless the initial connection to the database is made by a superuser (or the same user that owns all of the objects in the script). With `--no-owner`, any user name can be used for the initial connection, and this user will own all the created objects.

--filters <filename> This option allows to exclude table and indexes from the copy operations. See *Filtering* for details about the expected file format and the filtering options available.

--skip-extensions Skip copying extensions from the source database to the target database.

When used, schema that extensions depend-on are also skipped: it is expected that creating needed extensions on the target system is then the responsibility of another command (such as *pgcopydb copy extensions*), and schemas that extensions depend-on are part of that responsibility.

Because creating extensions require superuser, this allows a multi-steps approach where extensions are dealt with superuser privileges, and then the rest of the pgcopydb operations are done without superuser privileges.

--restart When running the pgcopydb command again, if the work directory already contains information from a previous run, then the command refuses to proceed and delete information that might be used for diagnostics and forensics.

In that case, the `--restart` option can be used to allow pgcopydb to delete traces from a previous run.

--resume When the pgcopydb command was terminated before completion, either by an interrupt signal (such as C-c or SIGTERM) or because it crashed, it is possible to resume the database migration.

When resuming activity from a previous run, table data that was fully copied over to the target server is not sent again. Table data that was interrupted during the COPY has to be started from scratch even when using `--resume`: the COPY command in Postgres is transactional and was rolled back.

Same reasonning applies to the CREATE INDEX commands and ALTER TABLE commands that pgcopydb issues, those commands are skipped on a `--resume` run only if known to have run through to completion on the previous one.

Finally, using `--resume` requires the use of `--not-consistent`.

--not-consistent In order to be consistent, pgcopydb exports a Postgres snapshot by calling the pg_export_snapshot() function on the source database server. The snapshot is then re-used in all the connections to the source database server by using the `SET TRANSACTION SNAPSHOT` command.

Per the Postgres documentation about `pg_export_snapshot`:

> Saves the transaction's current snapshot and returns a text string identifying the snapshot. This string must be passed (outside the database) to clients that want to import the snapshot. The snapshot is available for import only until the end of the transaction that exported it.

Now, when the pgcopydb process was interrupted (or crashed) on a previous run, it is possible to resume operations, but the snapshot that was exported does not exists anymore. The pgcopydb command can only resume operations with a new snapshot, and thus can not ensure consistency of the whole data set, because each run is now using their own snapshot.

--snapshot Instead of exporting its own snapshot by calling the PostgreSQL function `pg_export_snapshot()` it is possible for pgcopydb to re-use an already exported snapshot.

--verbose Increase current verbosity. The default level of verbosity is INFO. In ascending order pgcopydb knows about the following verbosity levels: FATAL, ERROR, WARN, INFO, NOTICE, DEBUG, TRACE.

--debug Set current verbosity to DEBUG level.

--trace Set current verbosity to TRACE level.

--quiet Set current verbosity to ERROR level.

4.7.8 Environment

PGCOPYDB_TARGET_PGURI

Connection string to the target Postgres instance. When `--target` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_DROP_IF_EXISTS

When true (or *yes*, or *on*, or 1, same input as a Postgres boolean) then pgcopydb uses the pg_restore options `--clean --if-exists` when creating the schema on the target Postgres instance.

4.7.9 Examples

First, using `pgcopydb restore schema`

```
$ PGCOPYDB_DROP_IF_EXISTS=on pgcopydb restore schema --source /tmp/target/ --target "port=54314 dbname=demo"
09:54:37 20401 INFO  Restoring database from "/tmp/target/"
09:54:37 20401 INFO  Restoring database into "port=54314 dbname=demo"
09:54:37 20401 INFO  Found a stale pidfile at "/tmp/target//pgcopydb.pid"
09:54:37 20401 WARN  Removing the stale pid file "/tmp/target//pgcopydb.pid"
09:54:37 20401 INFO  Using pg_restore for Postgres "12.9" at "/Applications/Postgres.app/Contents/Versions/12/
↪bin/pg_restore"
09:54:37 20401 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_restore --dbname 'port=54314␣
↪dbname=demo' --clean --if-exists /tmp/target//schema/pre.dump
09:54:38 20401 INFO   /Applications/Postgres.app/Contents/Versions/12/bin/pg_restore --dbname 'port=54314␣
↪dbname=demo' --clean --if-exists --use-list /tmp/target//schema/post.list /tmp/target//schema/post.dump
```

Then the `pgcopydb restore pre-data` and `pgcopydb restore post-data` would look the same with just a single call to pg_restore instead of the both of them.

Using `pgcopydb restore parse-list` it's possible to review the filtering options and see how pg_restore catalog entries are being commented-out.

```
$ cat ./tests/filtering/include.ini
[include-only-table]
public.actor
public.category
public.film
public.film_actor
public.film_category
public.language
public.rental

[exclude-index]
public.idx_store_id_film_id

[exclude-table-data]
public.rental

$ pgcopydb restore parse-list --dir /tmp/pagila/pgcopydb --resume --not-consistent --filters ./tests/filtering/
```

(continues on next page)

(continued from previous page)

```
↪include.ini
11:41:22 75175 INFO  Running pgcopydb version 0.5.8.ge0d2038 from "/Users/dim/dev/PostgreSQL/pgcopydb/./src/
↪bin/pgcopydb/pgcopydb"
11:41:22 75175 INFO  [SOURCE] Restoring database from "postgres://@:54311/pagila?"
11:41:22 75175 INFO  [TARGET] Restoring database into "postgres://@:54311/plop?"
11:41:22 75175 INFO  Using work dir "/tmp/pagila/pgcopydb"
11:41:22 75175 INFO  Removing the stale pid file "/tmp/pagila/pgcopydb/pgcopydb.pid"
11:41:22 75175 INFO  Work directory "/tmp/pagila/pgcopydb" already exists
11:41:22 75175 INFO  Schema dump for pre-data and post-data section have been done
11:41:22 75175 INFO  Restoring database from existing files at "/tmp/pagila/pgcopydb"
11:41:22 75175 INFO  Using pg_restore for Postgres "12.9" at "/Applications/Postgres.app/Contents/Versions/12/
↪bin/pg_restore"
11:41:22 75175 INFO  Exported snapshot "00000003-0003209A-1" from the source database
3242; 2606 317973 CONSTRAINT public actor actor_pkey postgres
;3258; 2606 317975 CONSTRAINT public address address_pkey postgres
3245; 2606 317977 CONSTRAINT public category category_pkey postgres
;3261; 2606 317979 CONSTRAINT public city city_pkey postgres
;3264; 2606 317981 CONSTRAINT public country country_pkey postgres
;3237; 2606 317983 CONSTRAINT public customer customer_pkey postgres
3253; 2606 317985 CONSTRAINT public film_actor film_actor_pkey postgres
3256; 2606 317987 CONSTRAINT public film_category film_category_pkey postgres
3248; 2606 317989 CONSTRAINT public film film_pkey postgres
;3267; 2606 317991 CONSTRAINT public inventory inventory_pkey postgres
3269; 2606 317993 CONSTRAINT public language language_pkey postgres
3293; 2606 317995 CONSTRAINT public rental rental_pkey postgres
;3295; 2606 317997 CONSTRAINT public staff staff_pkey postgres
;3298; 2606 317999 CONSTRAINT public store store_pkey postgres
3246; 1259 318000 INDEX public film_fulltext_idx postgres
3243; 1259 318001 INDEX public idx_actor_last_name postgres
;3238; 1259 318002 INDEX public idx_fk_address_id postgres
;3259; 1259 318003 INDEX public idx_fk_city_id postgres
;3262; 1259 318004 INDEX public idx_fk_country_id postgres
;3270; 1259 318005 INDEX public idx_fk_customer_id postgres
3254; 1259 318006 INDEX public idx_fk_film_id postgres
3290; 1259 318007 INDEX public idx_fk_inventory_id postgres
3249; 1259 318008 INDEX public idx_fk_language_id postgres
3250; 1259 318009 INDEX public idx_fk_original_language_id postgres
;3272; 1259 318010 INDEX public idx_fk_payment_p2020_01_customer_id postgres
;3271; 1259 318011 INDEX public idx_fk_staff_id postgres
;3273; 1259 318012 INDEX public idx_fk_payment_p2020_01_staff_id postgres
;3275; 1259 318013 INDEX public idx_fk_payment_p2020_02_customer_id postgres
;3276; 1259 318014 INDEX public idx_fk_payment_p2020_02_staff_id postgres
;3278; 1259 318015 INDEX public idx_fk_payment_p2020_03_customer_id postgres
;3279; 1259 318016 INDEX public idx_fk_payment_p2020_03_staff_id postgres
;3281; 1259 318017 INDEX public idx_fk_payment_p2020_04_customer_id postgres
;3282; 1259 318018 INDEX public idx_fk_payment_p2020_04_staff_id postgres
;3284; 1259 318019 INDEX public idx_fk_payment_p2020_05_customer_id postgres
;3285; 1259 318020 INDEX public idx_fk_payment_p2020_05_staff_id postgres
;3287; 1259 318021 INDEX public idx_fk_payment_p2020_06_customer_id postgres
;3288; 1259 318022 INDEX public idx_fk_payment_p2020_06_staff_id postgres
;3239; 1259 318023 INDEX public idx_fk_store_id postgres
;3240; 1259 318024 INDEX public idx_last_name postgres
;3265; 1259 318025 INDEX public idx_store_id_film_id postgres
3251; 1259 318026 INDEX public idx_title postgres
;3296; 1259 318027 INDEX public idx_unq_manager_staff_id postgres
3291; 1259 318028 INDEX public idx_unq_rental_rental_date_inventory_id_customer_id postgres
;3274; 1259 318029 INDEX public payment_p2020_01_customer_id_idx postgres
;3277; 1259 318030 INDEX public payment_p2020_02_customer_id_idx postgres
;3280; 1259 318031 INDEX public payment_p2020_03_customer_id_idx postgres
;3283; 1259 318032 INDEX public payment_p2020_04_customer_id_idx postgres
;3286; 1259 318033 INDEX public payment_p2020_05_customer_id_idx postgres
;3289; 1259 318034 INDEX public payment_p2020_06_customer_id_idx postgres
;3299; 0 0 INDEX ATTACH public idx_fk_payment_p2020_01_staff_id postgres
;3301; 0 0 INDEX ATTACH public idx_fk_payment_p2020_02_staff_id postgres
;3303; 0 0 INDEX ATTACH public idx_fk_payment_p2020_03_staff_id postgres
;3305; 0 0 INDEX ATTACH public idx_fk_payment_p2020_04_staff_id postgres
;3307; 0 0 INDEX ATTACH public idx_fk_payment_p2020_05_staff_id postgres
;3309; 0 0 INDEX ATTACH public idx_fk_payment_p2020_06_staff_id postgres
;3300; 0 0 INDEX ATTACH public payment_p2020_01_customer_id_idx postgres
;3302; 0 0 INDEX ATTACH public payment_p2020_02_customer_id_idx postgres
;3304; 0 0 INDEX ATTACH public payment_p2020_03_customer_id_idx postgres
;3306; 0 0 INDEX ATTACH public payment_p2020_04_customer_id_idx postgres
;3308; 0 0 INDEX ATTACH public payment_p2020_05_customer_id_idx postgres
;3310; 0 0 INDEX ATTACH public payment_p2020_06_customer_id_idx postgres
3350; 2620 318035 TRIGGER public film film_fulltext_trigger postgres
3348; 2620 318036 TRIGGER public actor last_updated postgres
;3354; 2620 318037 TRIGGER public address last_updated postgres
3349; 2620 318038 TRIGGER public category last_updated postgres
```

(continues on next page)

(continued from previous page)

```
;3355; 2620 318039 TRIGGER public city last_updated postgres
;3356; 2620 318040 TRIGGER public country last_updated postgres
;3347; 2620 318041 TRIGGER public customer last_updated postgres
3351; 2620 318042 TRIGGER public film last_updated postgres
3352; 2620 318043 TRIGGER public film_actor last_updated postgres
3353; 2620 318044 TRIGGER public film_category last_updated postgres
;3357; 2620 318045 TRIGGER public inventory last_updated postgres
3358; 2620 318046 TRIGGER public language last_updated postgres
3359; 2620 318047 TRIGGER public rental last_updated postgres
;3360; 2620 318048 TRIGGER public staff last_updated postgres
;3361; 2620 318049 TRIGGER public store last_updated postgres
;3319; 2606 318050 FK CONSTRAINT public address address_city_id_fkey postgres
;3320; 2606 318055 FK CONSTRAINT public city city_country_id_fkey postgres
;3311; 2606 318060 FK CONSTRAINT public customer customer_address_id_fkey postgres
;3312; 2606 318065 FK CONSTRAINT public customer customer_store_id_fkey postgres
3315; 2606 318070 FK CONSTRAINT public film_actor film_actor_actor_id_fkey postgres
3316; 2606 318075 FK CONSTRAINT public film_actor film_actor_film_id_fkey postgres
3317; 2606 318080 FK CONSTRAINT public film_category film_category_category_id_fkey postgres
3318; 2606 318085 FK CONSTRAINT public film_category film_category_film_id_fkey postgres
3313; 2606 318090 FK CONSTRAINT public film film_language_id_fkey postgres
3314; 2606 318095 FK CONSTRAINT public film film_original_language_id_fkey postgres
;3321; 2606 318100 FK CONSTRAINT public inventory inventory_film_id_fkey postgres
;3322; 2606 318105 FK CONSTRAINT public inventory inventory_store_id_fkey postgres
;3323; 2606 318110 FK CONSTRAINT public payment_p2020_01 payment_p2020_01_customer_id_fkey postgres
;3324; 2606 318115 FK CONSTRAINT public payment_p2020_01 payment_p2020_01_rental_id_fkey postgres
;3325; 2606 318120 FK CONSTRAINT public payment_p2020_01 payment_p2020_01_staff_id_fkey postgres
;3326; 2606 318125 FK CONSTRAINT public payment_p2020_02 payment_p2020_02_customer_id_fkey postgres
;3327; 2606 318130 FK CONSTRAINT public payment_p2020_02 payment_p2020_02_rental_id_fkey postgres
;3328; 2606 318135 FK CONSTRAINT public payment_p2020_02 payment_p2020_02_staff_id_fkey postgres
;3329; 2606 318140 FK CONSTRAINT public payment_p2020_03 payment_p2020_03_customer_id_fkey postgres
;3330; 2606 318145 FK CONSTRAINT public payment_p2020_03 payment_p2020_03_rental_id_fkey postgres
;3331; 2606 318150 FK CONSTRAINT public payment_p2020_03 payment_p2020_03_staff_id_fkey postgres
;3332; 2606 318155 FK CONSTRAINT public payment_p2020_04 payment_p2020_04_customer_id_fkey postgres
;3333; 2606 318160 FK CONSTRAINT public payment_p2020_04 payment_p2020_04_rental_id_fkey postgres
;3334; 2606 318165 FK CONSTRAINT public payment_p2020_04 payment_p2020_04_staff_id_fkey postgres
;3335; 2606 318170 FK CONSTRAINT public payment_p2020_05 payment_p2020_05_customer_id_fkey postgres
;3336; 2606 318175 FK CONSTRAINT public payment_p2020_05 payment_p2020_05_rental_id_fkey postgres
;3337; 2606 318180 FK CONSTRAINT public payment_p2020_05 payment_p2020_05_staff_id_fkey postgres
;3338; 2606 318185 FK CONSTRAINT public payment_p2020_06 payment_p2020_06_customer_id_fkey postgres
;3339; 2606 318190 FK CONSTRAINT public payment_p2020_06 payment_p2020_06_rental_id_fkey postgres
;3340; 2606 318195 FK CONSTRAINT public payment_p2020_06 payment_p2020_06_staff_id_fkey postgres
;3341; 2606 318200 FK CONSTRAINT public rental rental_customer_id_fkey postgres
;3342; 2606 318205 FK CONSTRAINT public rental rental_inventory_id_fkey postgres
;3343; 2606 318210 FK CONSTRAINT public rental rental_staff_id_fkey postgres
;3344; 2606 318215 FK CONSTRAINT public staff staff_address_id_fkey postgres
;3345; 2606 318220 FK CONSTRAINT public staff staff_store_id_fkey postgres
;3346; 2606 318225 FK CONSTRAINT public store store_address_id_fkey postgres
```

4.8 pgcopydb list

pgcopydb list - List database objects from a Postgres instance

This command prefixes the following sub-commands:

```
pgcopydb list
  databases    List databases
  extensions   List all the source extensions to copy
  collations   List all the source collations to copy
  tables       List all the source tables to copy data from
  table-parts  List a source table copy partitions
  sequences    List all the source sequences to copy data from
  indexes      List all the indexes to create again after copying the data
  depends      List all the dependencies to filter-out
  schema       List the schema to migrate, formatted in JSON
  progress     List the progress
```

4.8.1 pgcopydb list databases

pgcopydb list databases - List databases

The command `pgcopydb list databases` connects to the source database and executes a SQL query using the Postgres catalogs to get a list of all the databases there.

```
pgcopydb list databases: List databases
usage: pgcopydb list databases  --source ...

  --source            Postgres URI to the source database
```

4.8.2 pgcopydb list extensions

pgcopydb list extensions - List all the source extensions to copy

The command `pgcopydb list extensions` connects to the source database and executes a SQL query using the Postgres catalogs to get a list of all the extensions to COPY to the target database.

```
pgcopydb list extensions: List all the source extensions to copy
usage: pgcopydb list extensions  --source ...

  --source            Postgres URI to the source database
```

4.8.3 pgcopydb list collations

pgcopydb list collations - List all the source collations to copy

The command `pgcopydb list collations` connects to the source database and executes a SQL query using the Postgres catalogs to get a list of all the collations to COPY to the target database.

```
pgcopydb list collations: List all the source collations to copy
usage: pgcopydb list collations  --source ...

  --source            Postgres URI to the source database
```

The SQL query that is used lists the database collation, and then any non-default collation that's used in a user column or a user index.

4.8.4 pgcopydb list tables

pgcopydb list tables - List all the source tables to copy data from

The command `pgcopydb list tables` connects to the source database and executes a SQL query using the Postgres catalogs to get a list of all the tables to COPY the data from.

```
pgcopydb list tables: List all the source tables to copy data from
usage: pgcopydb list tables  --source ...

  --source            Postgres URI to the source database
  --filter <filename> Use the filters defined in <filename>
  --cache             Cache table size in relation pgcopydb.pgcopydb_table_size
  --drop-cache        Drop relation pgcopydb.pgcopydb_table_size
  --list-skipped      List only tables that are setup to be skipped
  --without-pkey      List only tables that have no primary key
```

The `--cache` option allows caching the pg_table_size() result in the newly created table `pgcopydb.pgcopydb_table_size`. This is only useful in Postgres deployments where this computation is quite slow, and when the pgcopydb operation is going to be run multiple times.

4.8.5 pgcopydb list table-parts

pgcopydb list table-parts - List a source table copy partitions

The command `pgcopydb list table-parts` connects to the source database and executes a SQL query using the Postgres catalogs to get detailed information about the given source table, and then another SQL query to compute how to split this source table given the size threshold argument.

```
pgcopydb list table-parts: List a source table copy partitions
usage: pgcopydb list table-parts  --source ...

  --source                    Postgres URI to the source database
  --schema-name               Name of the schema where to find the table
  --table-name                Name of the target table
  --split-tables-larger-than  Size threshold to consider partitioning
```

4.8.6 pgcopydb list sequences

pgcopydb list sequences - List all the source sequences to copy data from

The command `pgcopydb list sequences` connects to the source database and executes a SQL query using the Postgres catalogs to get a list of all the sequences to COPY the data from.

```
pgcopydb list sequences: List all the source sequences to copy data from
usage: pgcopydb list sequences  --source ...

  --source            Postgres URI to the source database
  --filter <filename> Use the filters defined in <filename>
  --list-skipped      List only tables that are setup to be skipped
```

4.8.7 pgcopydb list indexes

pgcopydb list indexes - List all the indexes to create again after copying the data

The command `pgcopydb list indexes` connects to the source database and executes a SQL query using the Postgres catalogs to get a list of all the indexes to COPY the data from.

```
pgcopydb list indexes: List all the indexes to create again after copying the data
usage: pgcopydb list indexes  --source ... [ --schema-name [ --table-name ] ]

  --source            Postgres URI to the source database
  --schema-name       Name of the schema where to find the table
  --table-name        Name of the target table
  --filter <filename> Use the filters defined in <filename>
  --list-skipped      List only tables that are setup to be skipped
```

4.8.8 pgcopydb list depends

pgcopydb list depends - List all the dependencies to filter-out

The command `pgcopydb list depends` connects to the source database and executes a SQL query using the Postgres catalogs to get a list of all the objects that depend on excluded objects from the filtering rules.

```
pgcopydb list depends: List all the dependencies to filter-out
usage: pgcopydb list depends  --source ... [ --schema-name [ --table-name ] ]

  --source            Postgres URI to the source database
  --schema-name       Name of the schema where to find the table
  --table-name        Name of the target table
  --filter <filename> Use the filters defined in <filename>
  --list-skipped      List only tables that are setup to be skipped
```

4.8.9 pgcopydb list schema

pgcopydb list schema - List the schema to migrate, formatted in JSON

The command `pgcopydb list schema` connects to the source database and executes a SQL queries using the Postgres catalogs to get a list of the tables, indexes, and sequences to migrate. The command then outputs a JSON formatted string that contains detailed information about all those objects.

```
pgcopydb list schema: List the schema to migrate, formatted in JSON
usage: pgcopydb list schema  --source ...

  --source            Postgres URI to the source database
  --filter <filename> Use the filters defined in <filename>
```

4.8.10 pgcopydb list progress

pgcopydb list progress - List the progress

The command `pgcopydb list progress` reads the `schema.json` file in the work directory, parses it, and then computes how many tables and indexes are planned to be copied and created on the target database, how many have been done already, and how many are in-progress.

When using the option `--json` the JSON formatted output also includes a list of all the tables and indexes that are currently being processed.

```
pgcopydb list progress: List the progress
usage: pgcopydb list progress  --source ...

  --source  Postgres URI to the source database
  --summary List the summary, requires --json
  --json    Format the output using JSON
  --dir     Work directory to use
```

4.8.11 Options

The following options are available to `pgcopydb dump schema`:

--source Connection string to the source Postgres instance. See the Postgres documentation for connection strings for the details. In short both the quoted form `"host=... dbname=..."` and the URI form `postgres://user@host:5432/dbname` are supported.

--schema-name Filter indexes from a given schema only.

--table-name Filter indexes from a given table only (use `--schema-name` to fully qualify the table).

--without-pkey List only tables from the source database when they have no primary key attached to their schema.

--filter <filename> This option allows to skip objects in the list operations. See *Filtering* for details about the expected file format and the filtering options available.

--list-skipped Instead of listing objects that are selected for copy by the filters installed with the `--filter` option, list the objects that are going to be skipped when using the filters.

--summary Instead of listing current progress when the command is still running, instead list the summary with timing details for each step and for all tables, indexes, and constraints.

This options requires the `--json` option too: at the moment only this output format is supported.

--json The output of the command is formatted in JSON, when supported. Ignored otherwise.

--verbose Increase current verbosity. The default level of verbosity is INFO. In ascending order pgcopydb knows about the following verbosity levels: FATAL, ERROR, WARN, INFO, NOTICE, DEBUG, TRACE.

--debug Set current verbosity to DEBUG level.

--trace Set current verbosity to TRACE level.

--quiet Set current verbosity to ERROR level.

4.8.12 Environment

PGCOPYDB_SOURCE_PGURI

Connection string to the source Postgres instance. When `--source` is ommitted from the command line, then this environment variable is used.

4.8.13 Examples

Listing the tables:

```
$ pgcopydb list tables
14:35:18 13827 INFO  Listing ordinary tables in "port=54311 host=localhost dbname=pgloader"
14:35:19 13827 INFO  Fetched information for 56 tables
     OID |          Schema Name |           Table Name |  Est. Row Count |    On-disk size
---------+----------------------+----------------------+-----------------+----------------
   17085 |                  csv |                track |            3503 |          544 kB
   17098 |             expected |                track |            3503 |          544 kB
   17290 |             expected |           track_full |            3503 |          544 kB
   17276 |               public |           track_full |            3503 |          544 kB
   17016 |             expected |            districts |             440 |           72 kB
   17007 |               public |            districts |             440 |           72 kB
   16998 |                  csv |               blocks |             460 |           48 kB
   17003 |             expected |               blocks |             460 |           48 kB
   17405 |                  csv |              partial |               7 |           16 kB
   17323 |                  err |               errors |               0 |           16 kB
   16396 |             expected |              allcols |               0 |           16 kB
   17265 |             expected |                  csv |               0 |           16 kB
   17056 |             expected |      csv_escape_mode |               0 |           16 kB
   17331 |             expected |               errors |               0 |           16 kB
   17116 |             expected |                group |               0 |           16 kB
   17134 |             expected |                 json |               0 |           16 kB
   17074 |             expected |             matching |               0 |           16 kB
   17201 |             expected |               nullif |               0 |           16 kB
   17229 |             expected |                nulls |               0 |           16 kB
   17417 |             expected |              partial |               0 |           16 kB
   17313 |             expected |              reg2013 |               0 |           16 kB
   17437 |             expected |               serial |               0 |           16 kB
   17247 |             expected |                 sexp |               0 |           16 kB
   17378 |             expected |                test1 |               0 |           16 kB
   17454 |             expected |                  udc |               0 |           16 kB
   17471 |             expected |                xzero |               0 |           16 kB
   17372 |               nsitra |                test1 |               0 |           16 kB
   16388 |               public |              allcols |               0 |           16 kB
   17256 |               public |                  csv |               0 |           16 kB
   17047 |               public |      csv_escape_mode |               0 |           16 kB
   17107 |               public |                group |               0 |           16 kB
   17125 |               public |                 json |               0 |           16 kB
   17065 |               public |             matching |               0 |           16 kB
   17192 |               public |               nullif |               0 |           16 kB
   17219 |               public |                nulls |               0 |           16 kB
   17307 |               public |              reg2013 |               0 |           16 kB
```

(continues on next page)

(continued from previous page)

```
   17428 |              public |              serial |              0 |          16 kB
   17238 |              public |                sexp |              0 |          16 kB
   17446 |              public |                 udc |              0 |          16 kB
   17463 |              public |               xzero |              0 |          16 kB
   17303 |            expected |             copyhex |              0 |      8192 bytes
   17033 |            expected |          dateformat |              0 |      8192 bytes
   17366 |            expected |               fixed |              0 |      8192 bytes
   17041 |            expected |             jordane |              0 |      8192 bytes
   17173 |            expected |          missingcol |              0 |      8192 bytes
   17396 |            expected |            overflow |              0 |      8192 bytes
   17186 |            expected |             tab_csv |              0 |      8192 bytes
   17213 |            expected |                temp |              0 |      8192 bytes
   17299 |              public |             copyhex |              0 |      8192 bytes
   17029 |              public |          dateformat |              0 |      8192 bytes
   17362 |              public |               fixed |              0 |      8192 bytes
   17037 |              public |             jordane |              0 |      8192 bytes
   17164 |              public |          missingcol |              0 |      8192 bytes
   17387 |              public |            overflow |              0 |      8192 bytes
   17182 |              public |             tab_csv |              0 |      8192 bytes
   17210 |              public |                temp |              0 |      8192 bytes
```

Listing a table list of COPY partitions:

```
$ pgcopydb list table-parts --table-name rental --split-at 300kB
16:43:26 73794 INFO  Running pgcopydb version 0.8.8.g0838291.dirty from "/Users/dim/dev/PostgreSQL/pgcopydb/
↪src/bin/pgcopydb/pgcopydb"
16:43:26 73794 INFO  Listing COPY partitions for table "public"."rental" in "postgres://@:/pagila?"
16:43:26 73794 INFO  Table "public"."rental" COPY will be split 5-ways
      Part |        Min |        Max |      Count
-----------+------------+------------+-----------
       1/5 |          1 |       3211 |       3211
       2/5 |       3212 |       6422 |       3211
       3/5 |       6423 |       9633 |       3211
       4/5 |       9634 |      12844 |       3211
       5/5 |      12845 |      16049 |       3205
```

Listing the indexes:

```
$ pgcopydb list indexes
14:35:07 13668 INFO  Listing indexes in "port=54311 host=localhost dbname=pgloader"
14:35:07 13668 INFO  Fetching all indexes in source database
14:35:07 13668 INFO  Fetched information for 12 indexes
     OID |     Schema |           Index Name |         conname |                 Constraint | DDL
---------+------------+----------------------+-----------------+----------------------------+-------------------
↪--
   17002 |        csv |      blocks_ip4r_idx |                 |                            | CREATE INDEX␣
↪blocks_ip4r_idx ON csv.blocks USING gist (iprange)
   17415 |        csv |        partial_b_idx |                 |                            | CREATE INDEX␣
↪partial_b_idx ON csv.partial USING btree (b)
   17414 |        csv |        partial_a_key |   partial_a_key |                UNIQUE (a) | CREATE UNIQUE␣
↪INDEX partial_a_key ON csv.partial USING btree (a)
   17092 |        csv |           track_pkey |      track_pkey |     PRIMARY KEY (trackid) | CREATE UNIQUE␣
↪INDEX track_pkey ON csv.track USING btree (trackid)
   17329 |        err |          errors_pkey |     errors_pkey |           PRIMARY KEY (a) | CREATE UNIQUE␣
↪INDEX errors_pkey ON err.errors USING btree (a)
   16394 |     public |         allcols_pkey |    allcols_pkey |           PRIMARY KEY (a) | CREATE UNIQUE␣
↪INDEX allcols_pkey ON public.allcols USING btree (a)
   17054 |     public | csv_escape_mode_pkey | csv_escape_mode_pkey |          PRIMARY KEY (id) | CREATE␣
↪UNIQUE INDEX csv_escape_mode_pkey ON public.csv_escape_mode USING btree (id)
   17199 |     public |          nullif_pkey |     nullif_pkey |          PRIMARY KEY (id) | CREATE UNIQUE␣
↪INDEX nullif_pkey ON public."nullif" USING btree (id)
   17435 |     public |          serial_pkey |     serial_pkey |           PRIMARY KEY (a) | CREATE UNIQUE␣
↪INDEX serial_pkey ON public.serial USING btree (a)
   17288 |     public |      track_full_pkey | track_full_pkey |     PRIMARY KEY (trackid) | CREATE UNIQUE␣
↪INDEX track_full_pkey ON public.track_full USING btree (trackid)
   17452 |     public |            udc_pkey |       udc_pkey |           PRIMARY KEY (b) | CREATE UNIQUE␣
↪INDEX udc_pkey ON public.udc USING btree (b)
   17469 |     public |          xzero_pkey |     xzero_pkey |           PRIMARY KEY (a) | CREATE UNIQUE␣
↪INDEX xzero_pkey ON public.xzero USING btree (a)
```

Listing the schema in JSON:

```
$ pgcopydb list schema --split-at 200kB
```

This gives the following JSON output:

```
{
    "setup": {
        "snapshot": "00000003-00051AAE-1",
        "source_pguri": "postgres:\/\/@:\/pagila?",
        "target_pguri": "postgres:\/\/@:\/plop?",
        "table-jobs": 4,
        "index-jobs": 4,
        "split-tables-larger-than": 204800
    },
    "tables": [
        {
            "oid": 317934,
            "schema": "public",
            "name": "rental",
            "reltuples": 16044,
            "bytes": 1253376,
            "bytes-pretty": "1224 kB",
            "exclude-data": false,
            "restore-list-name": "public rental postgres",
            "part-key": "rental_id",
            "parts": [
                {
                    "number": 1,
                    "total": 7,
                    "min": 1,
                    "max": 2294,
                    "count": 2294
                },
                {
                    "number": 2,
                    "total": 7,
                    "min": 2295,
                    "max": 4588,
                    "count": 2294
                },
                {
                    "number": 3,
                    "total": 7,
                    "min": 4589,
                    "max": 6882,
                    "count": 2294
                },
                {
                    "number": 4,
                    "total": 7,
                    "min": 6883,
                    "max": 9176,
                    "count": 2294
                },
                {
                    "number": 5,
                    "total": 7,
                    "min": 9177,
                    "max": 11470,
                    "count": 2294
                },
                {
                    "number": 6,
                    "total": 7,
                    "min": 11471,
                    "max": 13764,
                    "count": 2294
                },
                {
                    "number": 7,
                    "total": 7,
                    "min": 13765,
                    "max": 16049,
                    "count": 2285
                }
            ]
        },
        {
            "oid": 317818,
            "schema": "public",
            "name": "film",
            "reltuples": 1000,
            "bytes": 483328,
            "bytes-pretty": "472 kB",
```


(continues on next page)

(continued from previous page)


```
            "exclude-data": false,
            "restore-list-name": "public film postgres",
            "part-key": "film_id",
            "parts": [
                {
                    "number": 1,
                    "total": 3,
                    "min": 1,
                    "max": 334,
                    "count": 334
                },
                {
                    "number": 2,
                    "total": 3,
                    "min": 335,
                    "max": 668,
                    "count": 334
                },
                {
                    "number": 3,
                    "total": 3,
                    "min": 669,
                    "max": 1000,
                    "count": 332
                }
            ]
        },
        {
            "oid": 317920,
            "schema": "public",
            "name": "payment_p2020_04",
            "reltuples": 6754,
            "bytes": 434176,
            "bytes-pretty": "424 kB",
            "exclude-data": false,
            "restore-list-name": "public payment_p2020_04 postgres",
            "part-key": ""
        },
        {
            "oid": 317916,
            "schema": "public",
            "name": "payment_p2020_03",
            "reltuples": 5644,
            "bytes": 368640,
            "bytes-pretty": "360 kB",
            "exclude-data": false,
            "restore-list-name": "public payment_p2020_03 postgres",
            "part-key": ""
        },
        {
            "oid": 317830,
            "schema": "public",
            "name": "film_actor",
            "reltuples": 5462,
            "bytes": 270336,
            "bytes-pretty": "264 kB",
            "exclude-data": false,
            "restore-list-name": "public film_actor postgres",
            "part-key": ""
        },
        {
            "oid": 317885,
            "schema": "public",
            "name": "inventory",
            "reltuples": 4581,
            "bytes": 270336,
            "bytes-pretty": "264 kB",
            "exclude-data": false,
            "restore-list-name": "public inventory postgres",
            "part-key": "inventory_id",
            "parts": [
                {
                    "number": 1,
                    "total": 2,
                    "min": 1,
                    "max": 2291,
                    "count": 2291
                },
```


(continues on next page)

(continued from previous page)


```
                {
                    "number": 2,
                    "total": 2,
                    "min": 2292,
                    "max": 4581,
                    "count": 2290
                }
            ]
        },
        {
            "oid": 317912,
            "schema": "public",
            "name": "payment_p2020_02",
            "reltuples": 2312,
            "bytes": 163840,
            "bytes-pretty": "160 kB",
            "exclude-data": false,
            "restore-list-name": "public payment_p2020_02 postgres",
            "part-key": ""
        },
        {
            "oid": 317784,
            "schema": "public",
            "name": "customer",
            "reltuples": 599,
            "bytes": 106496,
            "bytes-pretty": "104 kB",
            "exclude-data": false,
            "restore-list-name": "public customer postgres",
            "part-key": "customer_id"
        },
        {
            "oid": 317845,
            "schema": "public",
            "name": "address",
            "reltuples": 603,
            "bytes": 98304,
            "bytes-pretty": "96 kB",
            "exclude-data": false,
            "restore-list-name": "public address postgres",
            "part-key": "address_id"
        },
        {
            "oid": 317908,
            "schema": "public",
            "name": "payment_p2020_01",
            "reltuples": 1157,
            "bytes": 98304,
            "bytes-pretty": "96 kB",
            "exclude-data": false,
            "restore-list-name": "public payment_p2020_01 postgres",
            "part-key": ""
        },
        {
            "oid": 317855,
            "schema": "public",
            "name": "city",
            "reltuples": 600,
            "bytes": 73728,
            "bytes-pretty": "72 kB",
            "exclude-data": false,
            "restore-list-name": "public city postgres",
            "part-key": "city_id"
        },
        {
            "oid": 317834,
            "schema": "public",
            "name": "film_category",
            "reltuples": 1000,
            "bytes": 73728,
            "bytes-pretty": "72 kB",
            "exclude-data": false,
            "restore-list-name": "public film_category postgres",
            "part-key": ""
        },
        {
            "oid": 317798,
            "schema": "public",
```


(continues on next page)

(continued from previous page)


```
            "name": "actor",
            "reltuples": 200,
            "bytes": 49152,
            "bytes-pretty": "48 kB",
            "exclude-data": false,
            "restore-list-name": "public actor postgres",
            "part-key": "actor_id"
        },
        {
            "oid": 317924,
            "schema": "public",
            "name": "payment_p2020_05",
            "reltuples": 182,
            "bytes": 40960,
            "bytes-pretty": "40 kB",
            "exclude-data": false,
            "restore-list-name": "public payment_p2020_05 postgres",
            "part-key": ""
        },
        {
            "oid": 317808,
            "schema": "public",
            "name": "category",
            "reltuples": 0,
            "bytes": 16384,
            "bytes-pretty": "16 kB",
            "exclude-data": false,
            "restore-list-name": "public category postgres",
            "part-key": "category_id"
        },
        {
            "oid": 317865,
            "schema": "public",
            "name": "country",
            "reltuples": 109,
            "bytes": 16384,
            "bytes-pretty": "16 kB",
            "exclude-data": false,
            "restore-list-name": "public country postgres",
            "part-key": "country_id"
        },
        {
            "oid": 317946,
            "schema": "public",
            "name": "staff",
            "reltuples": 0,
            "bytes": 16384,
            "bytes-pretty": "16 kB",
            "exclude-data": false,
            "restore-list-name": "public staff postgres",
            "part-key": "staff_id"
        },
        {
            "oid": 378280,
            "schema": "pgcopydb",
            "name": "sentinel",
            "reltuples": 1,
            "bytes": 8192,
            "bytes-pretty": "8192 bytes",
            "exclude-data": false,
            "restore-list-name": "pgcopydb sentinel dim",
            "part-key": ""
        },
        {
            "oid": 317892,
            "schema": "public",
            "name": "language",
            "reltuples": 0,
            "bytes": 8192,
            "bytes-pretty": "8192 bytes",
            "exclude-data": false,
            "restore-list-name": "public language postgres",
            "part-key": "language_id"
        },
        {
            "oid": 317928,
            "schema": "public",
            "name": "payment_p2020_06",
```


(continues on next page)

(continued from previous page)


```
            "reltuples": 0,
            "bytes": 8192,
            "bytes-pretty": "8192 bytes",
            "exclude-data": false,
            "restore-list-name": "public payment_p2020_06 postgres",
            "part-key": ""
        },
        {
            "oid": 317957,
            "schema": "public",
            "name": "store",
            "reltuples": 0,
            "bytes": 8192,
            "bytes-pretty": "8192 bytes",
            "exclude-data": false,
            "restore-list-name": "public store postgres",
            "part-key": "store_id"
        }
    ],
    "indexes": [
        {
            "oid": 378283,
            "schema": "pgcopydb",
            "name": "sentinel_expr_idx",
            "isPrimary": false,
            "isUnique": true,
            "columns": "",
            "sql": "CREATE UNIQUE INDEX sentinel_expr_idx ON pgcopydb.sentinel USING btree ((1))",
            "restore-list-name": "pgcopydb sentinel_expr_idx dim",
            "table": {
                "oid": 378280,
                "schema": "pgcopydb",
                "name": "sentinel"
            }
        },
        {
            "oid": 318001,
            "schema": "public",
            "name": "idx_actor_last_name",
            "isPrimary": false,
            "isUnique": false,
            "columns": "last_name",
            "sql": "CREATE INDEX idx_actor_last_name ON public.actor USING btree (last_name)",
            "restore-list-name": "public idx_actor_last_name postgres",
            "table": {
                "oid": 317798,
                "schema": "public",
                "name": "actor"
            }
        },
        {
            "oid": 317972,
            "schema": "public",
            "name": "actor_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "actor_id",
            "sql": "CREATE UNIQUE INDEX actor_pkey ON public.actor USING btree (actor_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317798,
                "schema": "public",
                "name": "actor"
            },
            "constraint": {
                "oid": 317973,
                "name": "actor_pkey",
                "sql": "PRIMARY KEY (actor_id)"
            }
        },
        {
            "oid": 317974,
            "schema": "public",
            "name": "address_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "address_id",
            "sql": "CREATE UNIQUE INDEX address_pkey ON public.address USING btree (address_id)",
```


(continues on next page)

(continued from previous page)


```
            "restore-list-name": "",
            "table": {
                "oid": 317845,
                "schema": "public",
                "name": "address"
            },
            "constraint": {
                "oid": 317975,
                "name": "address_pkey",
                "sql": "PRIMARY KEY (address_id)"
            }
        },
        {
            "oid": 318003,
            "schema": "public",
            "name": "idx_fk_city_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "city_id",
            "sql": "CREATE INDEX idx_fk_city_id ON public.address USING btree (city_id)",
            "restore-list-name": "public idx_fk_city_id postgres",
            "table": {
                "oid": 317845,
                "schema": "public",
                "name": "address"
            }
        },
        {
            "oid": 317976,
            "schema": "public",
            "name": "category_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "category_id",
            "sql": "CREATE UNIQUE INDEX category_pkey ON public.category USING btree (category_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317808,
                "schema": "public",
                "name": "category"
            },
            "constraint": {
                "oid": 317977,
                "name": "category_pkey",
                "sql": "PRIMARY KEY (category_id)"
            }
        },
        {
            "oid": 317978,
            "schema": "public",
            "name": "city_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "city_id",
            "sql": "CREATE UNIQUE INDEX city_pkey ON public.city USING btree (city_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317855,
                "schema": "public",
                "name": "city"
            },
            "constraint": {
                "oid": 317979,
                "name": "city_pkey",
                "sql": "PRIMARY KEY (city_id)"
            }
        },
        {
            "oid": 318004,
            "schema": "public",
            "name": "idx_fk_country_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "country_id",
            "sql": "CREATE INDEX idx_fk_country_id ON public.city USING btree (country_id)",
            "restore-list-name": "public idx_fk_country_id postgres",
            "table": {
                "oid": 317855,
```


(continues on next page)

(continued from previous page)


```
                "schema": "public",
                "name": "city"
            }
        },
        {
            "oid": 317980,
            "schema": "public",
            "name": "country_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "country_id",
            "sql": "CREATE UNIQUE INDEX country_pkey ON public.country USING btree (country_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317865,
                "schema": "public",
                "name": "country"
            },
            "constraint": {
                "oid": 317981,
                "name": "country_pkey",
                "sql": "PRIMARY KEY (country_id)"
            }
        },
        {
            "oid": 318024,
            "schema": "public",
            "name": "idx_last_name",
            "isPrimary": false,
            "isUnique": false,
            "columns": "last_name",
            "sql": "CREATE INDEX idx_last_name ON public.customer USING btree (last_name)",
            "restore-list-name": "public idx_last_name postgres",
            "table": {
                "oid": 317784,
                "schema": "public",
                "name": "customer"
            }
        },
        {
            "oid": 318002,
            "schema": "public",
            "name": "idx_fk_address_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "address_id",
            "sql": "CREATE INDEX idx_fk_address_id ON public.customer USING btree (address_id)",
            "restore-list-name": "public idx_fk_address_id postgres",
            "table": {
                "oid": 317784,
                "schema": "public",
                "name": "customer"
            }
        },
        {
            "oid": 317982,
            "schema": "public",
            "name": "customer_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "customer_id",
            "sql": "CREATE UNIQUE INDEX customer_pkey ON public.customer USING btree (customer_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317784,
                "schema": "public",
                "name": "customer"
            },
            "constraint": {
                "oid": 317983,
                "name": "customer_pkey",
                "sql": "PRIMARY KEY (customer_id)"
            }
        },
        {
            "oid": 318023,
            "schema": "public",
            "name": "idx_fk_store_id",
```


(continues on next page)

(continued from previous page)


```
            "isPrimary": false,
            "isUnique": false,
            "columns": "store_id",
            "sql": "CREATE INDEX idx_fk_store_id ON public.customer USING btree (store_id)",
            "restore-list-name": "public idx_fk_store_id postgres",
            "table": {
                "oid": 317784,
                "schema": "public",
                "name": "customer"
            }
        },
        {
            "oid": 318009,
            "schema": "public",
            "name": "idx_fk_original_language_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "original_language_id",
            "sql": "CREATE INDEX idx_fk_original_language_id ON public.film USING btree (original_language_id)
↪",
            "restore-list-name": "public idx_fk_original_language_id postgres",
            "table": {
                "oid": 317818,
                "schema": "public",
                "name": "film"
            }
        },
        {
            "oid": 318026,
            "schema": "public",
            "name": "idx_title",
            "isPrimary": false,
            "isUnique": false,
            "columns": "title",
            "sql": "CREATE INDEX idx_title ON public.film USING btree (title)",
            "restore-list-name": "public idx_title postgres",
            "table": {
                "oid": 317818,
                "schema": "public",
                "name": "film"
            }
        },
        {
            "oid": 318000,
            "schema": "public",
            "name": "film_fulltext_idx",
            "isPrimary": false,
            "isUnique": false,
            "columns": "fulltext",
            "sql": "CREATE INDEX film_fulltext_idx ON public.film USING gist (fulltext)",
            "restore-list-name": "public film_fulltext_idx postgres",
            "table": {
                "oid": 317818,
                "schema": "public",
                "name": "film"
            }
        },
        {
            "oid": 317988,
            "schema": "public",
            "name": "film_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "film_id",
            "sql": "CREATE UNIQUE INDEX film_pkey ON public.film USING btree (film_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317818,
                "schema": "public",
                "name": "film"
            },
            "constraint": {
                "oid": 317989,
                "name": "film_pkey",
                "sql": "PRIMARY KEY (film_id)"
            }
        },
        {
```


(continues on next page)

(continued from previous page)


```
            "oid": 318008,
            "schema": "public",
            "name": "idx_fk_language_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "language_id",
            "sql": "CREATE INDEX idx_fk_language_id ON public.film USING btree (language_id)",
            "restore-list-name": "public idx_fk_language_id postgres",
            "table": {
                "oid": 317818,
                "schema": "public",
                "name": "film"
            }
        },
        {
            "oid": 317984,
            "schema": "public",
            "name": "film_actor_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "actor_id,film_id",
            "sql": "CREATE UNIQUE INDEX film_actor_pkey ON public.film_actor USING btree (actor_id, film_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317830,
                "schema": "public",
                "name": "film_actor"
            },
            "constraint": {
                "oid": 317985,
                "name": "film_actor_pkey",
                "sql": "PRIMARY KEY (actor_id, film_id)"
            }
        },
        {
            "oid": 318006,
            "schema": "public",
            "name": "idx_fk_film_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "film_id",
            "sql": "CREATE INDEX idx_fk_film_id ON public.film_actor USING btree (film_id)",
            "restore-list-name": "public idx_fk_film_id postgres",
            "table": {
                "oid": 317830,
                "schema": "public",
                "name": "film_actor"
            }
        },
        {
            "oid": 317986,
            "schema": "public",
            "name": "film_category_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "film_id,category_id",
            "sql": "CREATE UNIQUE INDEX film_category_pkey ON public.film_category USING btree (film_id,␣
↪category_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317834,
                "schema": "public",
                "name": "film_category"
            },
            "constraint": {
                "oid": 317987,
                "name": "film_category_pkey",
                "sql": "PRIMARY KEY (film_id, category_id)"
            }
        },
        {
            "oid": 318025,
            "schema": "public",
            "name": "idx_store_id_film_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "film_id,store_id",
            "sql": "CREATE INDEX idx_store_id_film_id ON public.inventory USING btree (store_id, film_id)",
```


(continues on next page)

(continued from previous page)


```
            "restore-list-name": "public idx_store_id_film_id postgres",
            "table": {
                "oid": 317885,
                "schema": "public",
                "name": "inventory"
            }
        },
        {
            "oid": 317990,
            "schema": "public",
            "name": "inventory_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "inventory_id",
            "sql": "CREATE UNIQUE INDEX inventory_pkey ON public.inventory USING btree (inventory_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317885,
                "schema": "public",
                "name": "inventory"
            },
            "constraint": {
                "oid": 317991,
                "name": "inventory_pkey",
                "sql": "PRIMARY KEY (inventory_id)"
            }
        },
        {
            "oid": 317992,
            "schema": "public",
            "name": "language_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "language_id",
            "sql": "CREATE UNIQUE INDEX language_pkey ON public.language USING btree (language_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317892,
                "schema": "public",
                "name": "language"
            },
            "constraint": {
                "oid": 317993,
                "name": "language_pkey",
                "sql": "PRIMARY KEY (language_id)"
            }
        },
        {
            "oid": 318010,
            "schema": "public",
            "name": "idx_fk_payment_p2020_01_customer_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_01_customer_id ON public.payment_p2020_01 USING btree␣
↪(customer_id)",
            "restore-list-name": "public idx_fk_payment_p2020_01_customer_id postgres",
            "table": {
                "oid": 317908,
                "schema": "public",
                "name": "payment_p2020_01"
            }
        },
        {
            "oid": 318029,
            "schema": "public",
            "name": "payment_p2020_01_customer_id_idx",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX payment_p2020_01_customer_id_idx ON public.payment_p2020_01 USING btree␣
↪(customer_id)",
            "restore-list-name": "public payment_p2020_01_customer_id_idx postgres",
            "table": {
                "oid": 317908,
                "schema": "public",
                "name": "payment_p2020_01"
            }
```


(continues on next page)

(continued from previous page)


```
        },
        {
            "oid": 318012,
            "schema": "public",
            "name": "idx_fk_payment_p2020_01_staff_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "staff_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_01_staff_id ON public.payment_p2020_01 USING btree
↪(staff_id)",
            "restore-list-name": "public idx_fk_payment_p2020_01_staff_id postgres",
            "table": {
                "oid": 317908,
                "schema": "public",
                "name": "payment_p2020_01"
            }
        },
        {
            "oid": 318013,
            "schema": "public",
            "name": "idx_fk_payment_p2020_02_customer_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_02_customer_id ON public.payment_p2020_02 USING btree
↪(customer_id)",
            "restore-list-name": "public idx_fk_payment_p2020_02_customer_id postgres",
            "table": {
                "oid": 317912,
                "schema": "public",
                "name": "payment_p2020_02"
            }
        },
        {
            "oid": 318014,
            "schema": "public",
            "name": "idx_fk_payment_p2020_02_staff_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "staff_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_02_staff_id ON public.payment_p2020_02 USING btree
↪(staff_id)",
            "restore-list-name": "public idx_fk_payment_p2020_02_staff_id postgres",
            "table": {
                "oid": 317912,
                "schema": "public",
                "name": "payment_p2020_02"
            }
        },
        {
            "oid": 318030,
            "schema": "public",
            "name": "payment_p2020_02_customer_id_idx",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX payment_p2020_02_customer_id_idx ON public.payment_p2020_02 USING btree
↪(customer_id)",
            "restore-list-name": "public payment_p2020_02_customer_id_idx postgres",
            "table": {
                "oid": 317912,
                "schema": "public",
                "name": "payment_p2020_02"
            }
        },
        {
            "oid": 318016,
            "schema": "public",
            "name": "idx_fk_payment_p2020_03_staff_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "staff_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_03_staff_id ON public.payment_p2020_03 USING btree
↪(staff_id)",
            "restore-list-name": "public idx_fk_payment_p2020_03_staff_id postgres",
            "table": {
                "oid": 317916,
                "schema": "public",
```


(continues on next page)

(continued from previous page)


```
                "name": "payment_p2020_03"
            }
        },
        {
            "oid": 318031,
            "schema": "public",
            "name": "payment_p2020_03_customer_id_idx",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX payment_p2020_03_customer_id_idx ON public.payment_p2020_03 USING btree
↪(customer_id)",
            "restore-list-name": "public payment_p2020_03_customer_id_idx postgres",
            "table": {
                "oid": 317916,
                "schema": "public",
                "name": "payment_p2020_03"
            }
        },
        {
            "oid": 318015,
            "schema": "public",
            "name": "idx_fk_payment_p2020_03_customer_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_03_customer_id ON public.payment_p2020_03 USING btree
↪(customer_id)",
            "restore-list-name": "public idx_fk_payment_p2020_03_customer_id postgres",
            "table": {
                "oid": 317916,
                "schema": "public",
                "name": "payment_p2020_03"
            }
        },
        {
            "oid": 318032,
            "schema": "public",
            "name": "payment_p2020_04_customer_id_idx",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX payment_p2020_04_customer_id_idx ON public.payment_p2020_04 USING btree
↪(customer_id)",
            "restore-list-name": "public payment_p2020_04_customer_id_idx postgres",
            "table": {
                "oid": 317920,
                "schema": "public",
                "name": "payment_p2020_04"
            }
        },
        {
            "oid": 318018,
            "schema": "public",
            "name": "idx_fk_payment_p2020_04_staff_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "staff_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_04_staff_id ON public.payment_p2020_04 USING btree
↪(staff_id)",
            "restore-list-name": "public idx_fk_payment_p2020_04_staff_id postgres",
            "table": {
                "oid": 317920,
                "schema": "public",
                "name": "payment_p2020_04"
            }
        },
        {
            "oid": 318017,
            "schema": "public",
            "name": "idx_fk_payment_p2020_04_customer_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_04_customer_id ON public.payment_p2020_04 USING btree
↪(customer_id)",
            "restore-list-name": "public idx_fk_payment_p2020_04_customer_id postgres",
            "table": {
```


(continues on next page)

(continued from previous page)


```
                "oid": 317920,
                "schema": "public",
                "name": "payment_p2020_04"
            }
        },
        {
            "oid": 318019,
            "schema": "public",
            "name": "idx_fk_payment_p2020_05_customer_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_05_customer_id ON public.payment_p2020_05 USING btree
↪(customer_id)",
            "restore-list-name": "public idx_fk_payment_p2020_05_customer_id postgres",
            "table": {
                "oid": 317924,
                "schema": "public",
                "name": "payment_p2020_05"
            }
        },
        {
            "oid": 318020,
            "schema": "public",
            "name": "idx_fk_payment_p2020_05_staff_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "staff_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_05_staff_id ON public.payment_p2020_05 USING btree
↪(staff_id)",
            "restore-list-name": "public idx_fk_payment_p2020_05_staff_id postgres",
            "table": {
                "oid": 317924,
                "schema": "public",
                "name": "payment_p2020_05"
            }
        },
        {
            "oid": 318033,
            "schema": "public",
            "name": "payment_p2020_05_customer_id_idx",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX payment_p2020_05_customer_id_idx ON public.payment_p2020_05 USING btree
↪(customer_id)",
            "restore-list-name": "public payment_p2020_05_customer_id_idx postgres",
            "table": {
                "oid": 317924,
                "schema": "public",
                "name": "payment_p2020_05"
            }
        },
        {
            "oid": 318022,
            "schema": "public",
            "name": "idx_fk_payment_p2020_06_staff_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "staff_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_06_staff_id ON public.payment_p2020_06 USING btree
↪(staff_id)",
            "restore-list-name": "public idx_fk_payment_p2020_06_staff_id postgres",
            "table": {
                "oid": 317928,
                "schema": "public",
                "name": "payment_p2020_06"
            }
        },
        {
            "oid": 318034,
            "schema": "public",
            "name": "payment_p2020_06_customer_id_idx",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX payment_p2020_06_customer_id_idx ON public.payment_p2020_06 USING btree
↪(customer_id)",
```


(continues on next page)

(continued from previous page)


```
            "restore-list-name": "public payment_p2020_06_customer_id_idx postgres",
            "table": {
                "oid": 317928,
                "schema": "public",
                "name": "payment_p2020_06"
            }
        },
        {
            "oid": 318021,
            "schema": "public",
            "name": "idx_fk_payment_p2020_06_customer_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "customer_id",
            "sql": "CREATE INDEX idx_fk_payment_p2020_06_customer_id ON public.payment_p2020_06 USING btree
↪(customer_id)",
            "restore-list-name": "public idx_fk_payment_p2020_06_customer_id postgres",
            "table": {
                "oid": 317928,
                "schema": "public",
                "name": "payment_p2020_06"
            }
        },
        {
            "oid": 318028,
            "schema": "public",
            "name": "idx_unq_rental_rental_date_inventory_id_customer_id",
            "isPrimary": false,
            "isUnique": true,
            "columns": "rental_date,inventory_id,customer_id",
            "sql": "CREATE UNIQUE INDEX idx_unq_rental_rental_date_inventory_id_customer_id ON public.rental
↪USING btree (rental_date, inventory_id, customer_id)",
            "restore-list-name": "public idx_unq_rental_rental_date_inventory_id_customer_id postgres",
            "table": {
                "oid": 317934,
                "schema": "public",
                "name": "rental"
            }
        },
        {
            "oid": 317994,
            "schema": "public",
            "name": "rental_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "rental_id",
            "sql": "CREATE UNIQUE INDEX rental_pkey ON public.rental USING btree (rental_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317934,
                "schema": "public",
                "name": "rental"
            },
            "constraint": {
                "oid": 317995,
                "name": "rental_pkey",
                "sql": "PRIMARY KEY (rental_id)"
            }
        },
        {
            "oid": 318007,
            "schema": "public",
            "name": "idx_fk_inventory_id",
            "isPrimary": false,
            "isUnique": false,
            "columns": "inventory_id",
            "sql": "CREATE INDEX idx_fk_inventory_id ON public.rental USING btree (inventory_id)",
            "restore-list-name": "public idx_fk_inventory_id postgres",
            "table": {
                "oid": 317934,
                "schema": "public",
                "name": "rental"
            }
        },
        {
            "oid": 317996,
            "schema": "public",
            "name": "staff_pkey",
```


(continues on next page)

(continued from previous page)


```
            "isPrimary": true,
            "isUnique": true,
            "columns": "staff_id",
            "sql": "CREATE UNIQUE INDEX staff_pkey ON public.staff USING btree (staff_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317946,
                "schema": "public",
                "name": "staff"
            },
            "constraint": {
                "oid": 317997,
                "name": "staff_pkey",
                "sql": "PRIMARY KEY (staff_id)"
            }
        },
        {
            "oid": 318027,
            "schema": "public",
            "name": "idx_unq_manager_staff_id",
            "isPrimary": false,
            "isUnique": true,
            "columns": "manager_staff_id",
            "sql": "CREATE UNIQUE INDEX idx_unq_manager_staff_id ON public.store USING btree (manager_staff_id)
↪",
            "restore-list-name": "public idx_unq_manager_staff_id postgres",
            "table": {
                "oid": 317957,
                "schema": "public",
                "name": "store"
            }
        },
        {
            "oid": 317998,
            "schema": "public",
            "name": "store_pkey",
            "isPrimary": true,
            "isUnique": true,
            "columns": "store_id",
            "sql": "CREATE UNIQUE INDEX store_pkey ON public.store USING btree (store_id)",
            "restore-list-name": "",
            "table": {
                "oid": 317957,
                "schema": "public",
                "name": "store"
            },
            "constraint": {
                "oid": 317999,
                "name": "store_pkey",
                "sql": "PRIMARY KEY (store_id)"
            }
        }
    ],
    "sequences": [
        {
            "oid": 317796,
            "schema": "public",
            "name": "actor_actor_id_seq",
            "last-value": 200,
            "is-called": true,
            "restore-list-name": "public actor_actor_id_seq postgres"
        },
        {
            "oid": 317843,
            "schema": "public",
            "name": "address_address_id_seq",
            "last-value": 605,
            "is-called": true,
            "restore-list-name": "public address_address_id_seq postgres"
        },
        {
            "oid": 317806,
            "schema": "public",
            "name": "category_category_id_seq",
            "last-value": 16,
            "is-called": true,
            "restore-list-name": "public category_category_id_seq postgres"
        },
```


(continues on next page)

(continued from previous page)


```
        {
            "oid": 317853,
            "schema": "public",
            "name": "city_city_id_seq",
            "last-value": 600,
            "is-called": true,
            "restore-list-name": "public city_city_id_seq postgres"
        },
        {
            "oid": 317863,
            "schema": "public",
            "name": "country_country_id_seq",
            "last-value": 109,
            "is-called": true,
            "restore-list-name": "public country_country_id_seq postgres"
        },
        {
            "oid": 317782,
            "schema": "public",
            "name": "customer_customer_id_seq",
            "last-value": 599,
            "is-called": true,
            "restore-list-name": "public customer_customer_id_seq postgres"
        },
        {
            "oid": 317816,
            "schema": "public",
            "name": "film_film_id_seq",
            "last-value": 1000,
            "is-called": true,
            "restore-list-name": "public film_film_id_seq postgres"
        },
        {
            "oid": 317883,
            "schema": "public",
            "name": "inventory_inventory_id_seq",
            "last-value": 4581,
            "is-called": true,
            "restore-list-name": "public inventory_inventory_id_seq postgres"
        },
        {
            "oid": 317890,
            "schema": "public",
            "name": "language_language_id_seq",
            "last-value": 6,
            "is-called": true,
            "restore-list-name": "public language_language_id_seq postgres"
        },
        {
            "oid": 317902,
            "schema": "public",
            "name": "payment_payment_id_seq",
            "last-value": 32099,
            "is-called": true,
            "restore-list-name": "public payment_payment_id_seq postgres"
        },
        {
            "oid": 317932,
            "schema": "public",
            "name": "rental_rental_id_seq",
            "last-value": 16050,
            "is-called": true,
            "restore-list-name": "public rental_rental_id_seq postgres"
        },
        {
            "oid": 317944,
            "schema": "public",
            "name": "staff_staff_id_seq",
            "last-value": 2,
            "is-called": true,
            "restore-list-name": "public staff_staff_id_seq postgres"
        },
        {
            "oid": 317955,
            "schema": "public",
            "name": "store_store_id_seq",
            "last-value": 2,
            "is-called": true,
```


(continues on next page)

(continued from previous page)

```
            "restore-list-name": "public store_store_id_seq postgres"
        }
    ]
}
```

Listing current progress (log lines removed):

```
$ pgcopydb list progress 2>/dev/null
             |  Total Count |  In Progress |         Done
-------------+--------------+--------------+-------------
      Tables |           21 |            4 |            7
     Indexes |           48 |           14 |            7
```

Listing current progress, in JSON:

```
$ pgcopydb list progress --json 2>/dev/null
{
    "table-jobs": 4,
    "index-jobs": 4,
    "tables": {
        "total": 21,
        "done": 9,
        "in-progress": [
            {
                "oid": 317908,
                "schema": "public",
                "name": "payment_p2020_01",
                "reltuples": 1157,
                "bytes": 98304,
                "bytes-pretty": "96 kB",
                "exclude-data": false,
                "restore-list-name": "public payment_p2020_01 postgres",
                "part-key": "",
                "process": {
                    "pid": 75159,
                    "start-time-epoch": 1662476249,
                    "start-time-string": "2022-09-06 16:57:29 CEST",
                    "command": "COPY \"public\".\"payment_p2020_01\""
                }
            },
            {
                "oid": 317855,
                "schema": "public",
                "name": "city",
                "reltuples": 600,
                "bytes": 73728,
                "bytes-pretty": "72 kB",
                "exclude-data": false,
                "restore-list-name": "public city postgres",
                "part-key": "city_id",
                "process": {
                    "pid": 75157,
                    "start-time-epoch": 1662476249,
                    "start-time-string": "2022-09-06 16:57:29 CEST",
                    "command": "COPY \"public\".\"city\""
                }
            }
        ]
    },
       "indexes": {
        "total": 48,
        "done": 39,
        "in-progress": [
            {
                "oid": 378283,
                "schema": "pgcopydb",
                "name": "sentinel_expr_idx",
                "isPrimary": false,
                "isUnique": true,
                "columns": "",
                "sql": "CREATE UNIQUE INDEX sentinel_expr_idx ON pgcopydb.sentinel USING btree ((1))",
                "restore-list-name": "pgcopydb sentinel_expr_idx dim",
                "table": {
                    "oid": 378280,
                    "schema": "pgcopydb",
                    "name": "sentinel"
```

(continues on next page)

(continued from previous page)


```
                },
                "process": {
                    "pid": 74372,
                    "start-time-epoch": 1662476080,
                    "start-time-string": "2022-09-06 16:54:40 CEST"
                }
            },
            {
                "oid": 317980,
                "schema": "public",
                "name": "country_pkey",
                "isPrimary": true,
                "isUnique": true,
                "columns": "country_id",
                "sql": "CREATE UNIQUE INDEX country_pkey ON public.country USING btree (country_id)",
                "restore-list-name": "public country_pkey postgres",
                "table": {
                    "oid": 317865,
                    "schema": "public",
                    "name": "country"
                },
                "constraint": {
                    "oid": 317981,
                    "name": "country_pkey",
                    "sql": "PRIMARY KEY (country_id)",
                    "restore-list-name": ""
                },
                "process": {
                    "pid": 74358,
                    "start-time-epoch": 1662476080,
                    "start-time-string": "2022-09-06 16:54:40 CEST"
                }
            },
            {
                "oid": 317996,
                "schema": "public",
                "name": "staff_pkey",
                "isPrimary": true,
                "isUnique": true,
                "columns": "staff_id",
                "sql": "CREATE UNIQUE INDEX staff_pkey ON public.staff USING btree (staff_id)",
                "restore-list-name": "public staff_pkey postgres",
                "table": {
                    "oid": 317946,
                    "schema": "public",
                    "name": "staff"
                },
                "constraint": {
                    "oid": 317997,
                    "name": "staff_pkey",
                    "sql": "PRIMARY KEY (staff_id)",
                    "restore-list-name": ""
                },
                "process": {
                    "pid": 74368,
                    "start-time-epoch": 1662476080,
                    "start-time-string": "2022-09-06 16:54:40 CEST"
                }
            }
        ]
    }
}
```

4.9 pgcopydb stream

pgcopydb stream - Stream changes from source database

> **Warning: This mode of operations has been designed for unit testing only.**
>
> Consider using the *pgcopydb clone* (with the `--follow` option) or the *pgcopydb follow* command instead.

Note: Some *pgcopydb stream* commands are still designed for normal operations, rather than unit testing only.

The *pgcopydb stream sentinel set startpos*, *pgcopydb stream sentinel set endpos*, *pgcopydb stream sentinel set apply*, and *pgcopydb stream sentinel set prefetch* commands are necessary to communicate with the main `pgcopydb clone --follow` or `pgcopydb follow` process. See *Change Data Capture Example 1* for a detailed example using *pgcopydb stream sentinel set endpos*.

Also the commands *pgcopydb stream setup* and *pgcopydb stream cleanup* might be used directly in normal operations. See *Change Data Capture Example 2* for a detailed example.

This command prefixes the following sub-commands:

```
pgcopydb stream
  setup      Setup source and target systems for logical decoding
  cleanup    cleanup source and target systems for logical decoding
  prefetch   Stream JSON changes from the source database and transform them to SQL
  catchup    Apply prefetched changes from SQL files to the target database
  replay     Replay changes from the source to the target database, live
+ sentinel   Maintain a sentinel table on the source database
  receive    Stream changes from the source database
  transform  Transform changes from the source database into SQL commands
  apply      Apply changes from the source database into the target database

pgcopydb stream create
  slot    Create a replication slot in the source database
  origin  Create a replication origin in the target database

pgcopydb stream drop
  slot    Drop a replication slot in the source database
  origin  Drop a replication origin in the target database

pgcopydb stream sentinel
  create  Create the sentinel table on the source database
  drop    Drop the sentinel table on the source database
  get     Get the sentinel table values on the source database
+ set     Maintain a sentinel table on the source database

pgcopydb stream sentinel set
  startpos  Set the sentinel start position LSN on the source database
  endpos    Set the sentinel end position LSN on the source database
  apply     Set the sentinel apply mode on the source database
  prefetch  Set the sentinel prefetch mode on the source database
```

Those commands implement a part of the whole database replay operation as detailed in section *pgcopydb follow*. Only use those commands to debug a specific part, or because you know that you just want to implement that step.

Note: The sub-commands `stream setup` then `stream prefetch` and `stream catchup` are higher level commands, that use internal information to know which files to process. Those commands also keep track of their progress.

The sub-commands `stream receive`, `stream transform`, and `stream apply` are lower level interface that work on given files. Those commands still keep track of their progress, but have to be given more information to work.

4.9.1 pgcopydb stream setup

pgcopydb stream setup - Setup source and target systems for logical decoding

The command `pgcopydb stream setup` connects to the source database and creates creates a `pgcopydb.sentinel` table, and then connects to the target database and creates a replication origin positioned at the LSN position of the logical decoding replication slot that must have been created already. See *pgcopydb snapshot* to create the replication slot and export a snapshot.

```
pgcopydb stream setup: Setup source and target systems for logical decoding
usage: pgcopydb stream setup

  --source          Postgres URI to the source database
  --target          Postgres URI to the target database
  --dir             Work directory to use
  --restart         Allow restarting when temp files exist already
  --resume          Allow resuming operations after a failure
  --not-consistent  Allow taking a new snapshot on the source database
  --snapshot        Use snapshot obtained with pg_export_snapshot
  --plugin          Output plugin to use (test_decoding, wal2json)
  --slot-name       Stream changes recorded by this slot
  --origin          Name of the Postgres replication origin
```

4.9.2 pgcopydb stream cleanup

pgcopydb stream cleanup - cleanup source and target systems for logical decoding

The command `pgcopydb stream cleanup` connects to the source and target databases to delete the objects created in the `pgcopydb stream setup` step.

```
pgcopydb stream cleanup: cleanup source and target systems for logical decoding
usage: pgcopydb stream cleanup

  --source          Postgres URI to the source database
  --target          Postgres URI to the target database
  --restart         Allow restarting when temp files exist already
  --resume          Allow resuming operations after a failure
  --not-consistent  Allow taking a new snapshot on the source database
  --snapshot        Use snapshot obtained with pg_export_snapshot
  --slot-name       Stream changes recorded by this slot
  --origin          Name of the Postgres replication origin
```

4.9.3 pgcopydb stream prefetch

pgcopydb stream prefetch - Stream JSON changes from the source database and transform them to SQL

The command `pgcopydb stream prefetch` connects to the source database using the logical replication protocl and the given replication slot.

The prefetch command receives the changes from the source database in a streaming fashion, and writes them in a series of JSON files named the same as their origin WAL filename (with the `.json` extension). Each time a JSON file is closed, a subprocess is started to transform the JSON into an SQL file.

```
pgcopydb stream prefetch: Stream JSON changes from the source database and transform them to SQL
usage: pgcopydb stream prefetch

  --source          Postgres URI to the source database
  --dir             Work directory to use
  --restart         Allow restarting when temp files exist already
  --resume          Allow resuming operations after a failure
  --not-consistent  Allow taking a new snapshot on the source database
  --slot-name       Stream changes recorded by this slot
  --endpos          LSN position where to stop receiving changes
```

4.9.4 pgcopydb stream catchup

pgcopydb stream catchup - Apply prefetched changes from SQL files to the target database

The command `pgcopydb stream catchup` connects to the target database and applies changes from the SQL files that have been prepared with the `pgcopydb stream prefetch` command.

```
pgcopydb stream catchup: Apply prefetched changes from SQL files to the target database
usage: pgcopydb stream catchup

  --source         Postgres URI to the source database
  --target         Postgres URI to the target database
  --dir            Work directory to use
  --restart        Allow restarting when temp files exist already
  --resume         Allow resuming operations after a failure
  --not-consistent Allow taking a new snapshot on the source database
  --slot-name      Stream changes recorded by this slot
  --endpos         LSN position where to stop receiving changes
      --origin         Name of the Postgres replication origin
```

4.9.5 pgcopydb stream replay

pgcopydb stream replay - Replay changes from the source to the target database, live

The command `pgcopydb stream replay` connects to the source database and streams changes using the logical decoding protocol, and internally streams those changes to a transform process and then a replay process, which connects to the target database and applies SQL changes.

```
pgcopydb stream replay: Replay changes from the source to the target database, live
usage: pgcopydb stream replay

  --source         Postgres URI to the source database
  --target         Postgres URI to the target database
  --dir            Work directory to use
  --restart        Allow restarting when temp files exist already
  --resume         Allow resuming operations after a failure
  --not-consistent Allow taking a new snapshot on the source database
  --slot-name      Stream changes recorded by this slot
  --endpos         LSN position where to stop receiving changes
  --origin         Name of the Postgres replication origin
```

This command is equivalent to running the following script:

```
pgcopydb stream receive --to-stdout
| pgcopydb stream transform - -
| pgcopydb stream apply -
```

4.9.6 pgcopydb stream sentinel create

pgcopydb stream sentinel create - Create the sentinel table on the source database

The `pgcopydb.sentinel` table allows to remote control the prefetch and catchup processes of the logical decoding implementation in pgcopydb.

```
pgcopydb stream sentinel create: Create the sentinel table on the source database
usage: pgcopydb stream sentinel create

  --source      Postgres URI to the source database
  --startpos    Start replaying changes when reaching this LSN
  --endpos      Stop replaying changes when reaching this LSN
```

4.9.7 pgcopydb stream sentinel drop

pgcopydb stream sentinel drop - Drop the sentinel table on the source database

The `pgcopydb.sentinel` table allows to remote control the prefetch and catchup processes of the logical decoding implementation in pgcopydb.

```
pgcopydb stream sentinel drop: Drop the sentinel table on the source database
usage: pgcopydb stream sentinel drop

  --source      Postgres URI to the source database
```

4.9.8 pgcopydb stream sentinel get

pgcopydb stream sentinel get - Get the sentinel table values on the source database

```
pgcopydb stream sentinel get: Get the sentinel table values on the source database
usage: pgcopydb stream sentinel get

  --source      Postgres URI to the source database
  --json        Format the output using JSON
```

4.9.9 pgcopydb stream sentinel set startpos

pgcopydb stream sentinel set startpos - Set the sentinel start position LSN on the source database

```
pgcopydb stream sentinel set startpos: Set the sentinel start position LSN on the source database
usage: pgcopydb stream sentinel set startpos <start LSN>

  --source      Postgres URI to the source database
```

4.9.10 pgcopydb stream sentinel set endpos

pgcopydb stream sentinel set endpos - Set the sentinel end position LSN on the source database

```
pgcopydb stream sentinel set endpos: Set the sentinel end position LSN on the source database
usage: pgcopydb stream sentinel set endpos <end LSN>

  --source      Postgres URI to the source database
  --current     Use pg_current_wal_flush_lsn() as the endpos
```

4.9.11 pgcopydb stream sentinel set apply

pgcopydb stream sentinel set apply - Set the sentinel apply mode on the source database

```
pgcopydb stream sentinel set apply: Set the sentinel apply mode on the source database
usage: pgcopydb stream sentinel set apply

  --source      Postgres URI to the source database
```

4.9.12 pgcopydb stream sentinel set prefetch

pgcopydb stream sentinel set prefetch - Set the sentinel prefetch mode on the source database

```
pgcopydb stream sentinel set prefetch: Set the sentinel prefetch mode on the source database
usage: pgcopydb stream sentinel set prefetch

  --source      Postgres URI to the source database
```

4.9.13 pgcopydb stream receive

pgcopydb stream receive - Stream changes from the source database

The command `pgcopydb stream receive` connects to the source database using the logical replication protocl and the given replication slot.

The receive command receives the changes from the source database in a streaming fashion, and writes them in a series of JSON files named the same as their origin WAL filename (with the `.json` extension).

```
pgcopydb stream receive: Stream changes from the source database
usage: pgcopydb stream receive  --source ...

  --source         Postgres URI to the source database
  --dir            Work directory to use
  --to-stdout      Stream logical decoding messages to stdout
  --restart        Allow restarting when temp files exist already
  --resume         Allow resuming operations after a failure
  --not-consistent Allow taking a new snapshot on the source database
  --slot-name      Stream changes recorded by this slot
  --endpos         LSN position where to stop receiving changes
```

4.9.14 pgcopydb stream transform

pgcopydb stream transform - Transform changes from the source database into SQL commands

The command `pgcopydb stream transform` transforms a JSON file as received by the `pgcopydb stream receive` command into an SQL file with one query per line.

```
pgcopydb stream transform: Transform changes from the source database into SQL commands
usage: pgcopydb stream transform  <json filename> <sql filename>

  --source         Postgres URI to the source database
  --dir            Work directory to use
  --restart        Allow restarting when temp files exist already
  --resume         Allow resuming operations after a failure
  --not-consistent Allow taking a new snapshot on the source database
```

The command supports using – as the filename for either the JSON input or the SQL output, or both. In that case reading from standard input and/or writing to standard output is implemented, in a streaming fashion. A classic use case is to use Unix Pipes, see *pgcopydb stream replay* too.

4.9.15 pgcopydb stream apply

pgcopydb stream apply - Apply changes from the source database into the target database

The command `pgcopydb stream apply` applies a SQL file as prepared by the `pgcopydb stream transform` command in the target database. The apply process tracks progress thanks to the Postgres API for Replication Progress Tracking.

```
pgcopydb stream apply: Apply changes from the source database into the target database
usage: pgcopydb stream apply <sql filename>

  --target         Postgres URI to the target database
  --dir            Work directory to use
  --restart        Allow restarting when temp files exist already
  --resume         Allow resuming operations after a failure
  --not-consistent Allow taking a new snapshot on the source database
  --origin         Name of the Postgres replication origin
```

This command supports using − as the filename to read from, and in that case reads from the standard input in a streaming fashion instead.

4.9.16 Options

The following options are available to `pgcopydb stream` sub-commands:

--source Connection string to the source Postgres instance. See the Postgres documentation for connection strings for the details. In short both the quoted form `"host=... dbname=..."` and the URI form `postgres://user@host:5432/dbname` are supported.

--target Connection string to the target Postgres instance.

--dir During its normal operations pgcopydb creates a lot of temporary files to track sub-processes progress. Temporary files are created in the directory location given by this option, or defaults to `${TMPDIR}/pgcopydb` when the environment variable is set, or then to `/tmp/pgcopydb`.

Change Data Capture files are stored in the `cdc` sub-directory of the `--dir` option when provided, otherwise see XDG_DATA_HOME environment variable below.

--restart When running the pgcopydb command again, if the work directory already contains information from a previous run, then the command refuses to proceed and delete information that might be used for diagnostics and forensics.

In that case, the `--restart` option can be used to allow pgcopydb to delete traces from a previous run.

--resume When the pgcopydb command was terminated before completion, either by an interrupt signal (such as C-c or SIGTERM) or because it crashed, it is possible to resume the database migration.

To be able to resume a streaming operation in a consistent way, all that's required is re-using the same replication slot as in previous run(s).

--plugin Logical decoding output plugin to use. The default is test_decoding which ships with Postgres core itself, so is probably already available on your source server.

It is possible to use wal2json instead. The support for wal2json is mostly historical in pgcopydb, it should not make a user visible difference whether you use the default test_decoding or wal2json.

--slot-name Logical decoding slot name to use.

--endpos Logical replication target LSN to use. Automatically stop replication and exit with normal exit status 0 when receiving reaches the specified LSN. If there's a record with LSN exactly equal to lsn, the record will be output.

The `--endpos` option is not aware of transaction boundaries and may truncate output partway through a transaction. Any partially output transaction will not be consumed and will be replayed again when the slot is next read from. Individual messages are never truncated.

See also documentation for pg_recvlogical.

--origin Logical replication target system needs to track the transactions that have been applied already, so that in case we get disconnected or need to resume operations we can skip already replayed transaction.

Postgres uses a notion of an origin node name as documented in Replication Progress Tracking. This option allows to pick your own node name and defaults to "pgcopydb". Picking a different name is useful in some advanced scenarios like migrating several sources in the same target, where each source should have their own unique origin node name.

--startpos Logical replication target system registers progress by assigning a current LSN to the `--origin` node name. When creating an origin on the target database system, it is required to provide the current LSN from the source database system, in order to properly bootstrap pgcopydb logical decoding.

--verbose Increase current verbosity. The default level of verbosity is INFO. In ascending order pgcopydb knows about the following verbosity levels: FATAL, ERROR, WARN, INFO, NOTICE, DEBUG, TRACE.

--debug Set current verbosity to DEBUG level.

--trace Set current verbosity to TRACE level.

--quiet Set current verbosity to ERROR level.

4.9.17 Environment

PGCOPYDB_SOURCE_PGURI

Connection string to the source Postgres instance. When `--source` is ommitted from the command line, then this environment variable is used.

PGCOPYDB_TARGET_PGURI

Connection string to the target Postgres instance. When `--target` is ommitted from the command line, then this environment variable is used.

TMPDIR

The pgcopydb command creates all its work files and directories in `${TMPDIR}/pgcopydb`, and defaults to `/tmp/pgcopydb`.

XDG_DATA_HOME

The pgcopydb command creates Change Data Capture files in the standard place XDG_DATA_HOME, which defaults to `~/.local/share`. See the XDG Base Directory Specification.

4.9.18 Examples

As an example here is the output generated from running the cdc test case, where a replication slot is created before the initial copy of the data, and then the following INSERT statement is executed:

```
begin;

with r as
 (
   insert into rental(rental_date, inventory_id, customer_id, staff_id, last_update)
        select '2022-06-01', 371, 291, 1, '2022-06-01'
     returning rental_id, customer_id, staff_id
 )
 insert into payment(customer_id, staff_id, rental_id, amount, payment_date)
      select customer_id, staff_id, rental_id, 5.99, '2020-06-01'
        from r;

commit;
```

The command then looks like the following, where the `--endpos` has been extracted by calling the `pg_current_wal_lsn()` SQL function:

```
$ pgcopydb stream receive --slot-name test_slot --restart --endpos 0/236D668 -vv
16:01:57 157 INFO  Running pgcopydb version 0.7 from "/usr/local/bin/pgcopydb"
16:01:57 157 DEBUG copydb.c:406 Change Data Capture data is managed at "/var/lib/postgres/.local/share/pgcopydb
↪"
16:01:57 157 INFO  copydb.c:73 Using work dir "/tmp/pgcopydb"
16:01:57 157 DEBUG pidfile.c:143 Failed to signal pid 34: No such process
16:01:57 157 DEBUG pidfile.c:146 Found a stale pidfile at "/tmp/pgcopydb/pgcopydb.pid"
16:01:57 157 INFO  pidfile.c:147 Removing the stale pid file "/tmp/pgcopydb/pgcopydb.pid"
16:01:57 157 INFO  copydb.c:254 Work directory "/tmp/pgcopydb" already exists
16:01:57 157 INFO  copydb.c:258 A previous run has run through completion
16:01:57 157 INFO  copydb.c:151 Removing directory "/tmp/pgcopydb"
16:01:57 157 DEBUG copydb.c:445 rm -rf "/tmp/pgcopydb" && mkdir -p "/tmp/pgcopydb"
16:01:57 157 DEBUG copydb.c:445 rm -rf "/tmp/pgcopydb/schema" && mkdir -p "/tmp/pgcopydb/schema"
16:01:57 157 DEBUG copydb.c:445 rm -rf "/tmp/pgcopydb/run" && mkdir -p "/tmp/pgcopydb/run"
16:01:57 157 DEBUG copydb.c:445 rm -rf "/tmp/pgcopydb/run/tables" && mkdir -p "/tmp/pgcopydb/run/tables"
16:01:57 157 DEBUG copydb.c:445 rm -rf "/tmp/pgcopydb/run/indexes" && mkdir -p "/tmp/pgcopydb/run/indexes"
16:01:57 157 DEBUG copydb.c:445 rm -rf "/var/lib/postgres/.local/share/pgcopydb" && mkdir -p "/var/lib/
↪postgres/.local/share/pgcopydb"
16:01:57 157 DEBUG pgsql.c:2476 starting log streaming at 0/0 (slot test_slot)
16:01:57 157 DEBUG pgsql.c:485 Connecting to [source] "postgres://postgres@source:/postgres?password=****&
↪replication=database"
16:01:57 157 DEBUG pgsql.c:2009 IDENTIFY_SYSTEM: timeline 1, xlogpos 0/236D668, systemid 7104302452422938663
16:01:57 157 DEBUG pgsql.c:3188 RetrieveWalSegSize: 16777216
16:01:57 157 DEBUG pgsql.c:2547 streaming initiated
16:01:57 157 INFO  stream.c:237 Now streaming changes to "/var/lib/postgres/.local/share/pgcopydb/
↪000000010000000000000002.json"
16:01:57 157 DEBUG stream.c:341 Received action B for XID 488 in LSN 0/236D638
16:01:57 157 DEBUG stream.c:341 Received action I for XID 488 in LSN 0/236D178
16:01:57 157 DEBUG stream.c:341 Received action I for XID 488 in LSN 0/236D308
16:01:57 157 DEBUG stream.c:341 Received action C for XID 488 in LSN 0/236D638
16:01:57 157 DEBUG pgsql.c:2867 pgsql_stream_logical: endpos reached at 0/236D668
16:01:57 157 DEBUG stream.c:382 Flushed up to 0/236D668 in file "/var/lib/postgres/.local/share/pgcopydb/
↪000000010000000000000002.json"
16:01:57 157 INFO  pgsql.c:3030 Report write_lsn 0/236D668, flush_lsn 0/236D668
16:01:57 157 DEBUG pgsql.c:3107 end position 0/236D668 reached by WAL record at 0/236D668
16:01:57 157 DEBUG pgsql.c:408 Disconnecting from [source] "postgres://postgres@source:/postgres?password=****&
↪replication=database"
16:01:57 157 DEBUG stream.c:414 streamClose: closing file "/var/lib/postgres/.local/share/pgcopydb/
↪000000010000000000000002.json"
16:01:57 157 INFO  stream.c:171 Streaming is now finished after processing 4 messages
```

The JSON file then contains the following content, from the *wal2json* logical replication plugin. Note that you're seeing diffent LSNs here because each run produces different ones, and the captures have not all been made from the same run.

```
$ cat /var/lib/postgres/.local/share/pgcopydb/000000010000000000000002.json
{"action":"B","xid":489,"timestamp":"2022-06-27 13:24:31.460822+00","lsn":"0/236F5A8","nextlsn":"0/236F5D8"}
{"action":"I","xid":489,"timestamp":"2022-06-27 13:24:31.460822+00","lsn":"0/236F0E8","schema":"public","table
↪":"rental","columns":[{"name":"rental_id","type":"integer","value":16050},{"name":"rental_date","type":
↪"timestamp with time zone","value":"2022-06-01 00:00:00+00"},{"name":"inventory_id","type":"integer","value
↪":371},{"name":"customer_id","type":"integer","value":291},{"name":"return_date","type":"timestamp with␣
↪time zone","value":null},{"name":"staff_id","type":"integer","value":1},{"name":"last_update","type":
↪"timestamp with time zone","value":"2022-06-01 00:00:00+00"}]}
```

(continues on next page)

(continued from previous page)

```
{"action":"I","xid":489,"timestamp":"2022-06-27 13:24:31.460822+00","lsn":"0/236F278","schema":"public","table
↪":"payment_p2020_06","columns":[{"name":"payment_id","type":"integer","value":32099},{"name":"customer_id",
↪"type":"integer","value":291},{"name":"staff_id","type":"integer","value":1},{"name":"rental_id","type":
↪"integer","value":16050},{"name":"amount","type":"numeric(5,2)","value":5.99},{"name":"payment_date","type":
↪"timestamp with time zone","value":"2020-06-01 00:00:00+00"}]}
{"action":"C","xid":489,"timestamp":"2022-06-27 13:24:31.460822+00","lsn":"0/236F5A8","nextlsn":"0/236F5D8"}
```

It's then possible to transform the JSON into SQL:

```
$ pgcopydb stream transform  ./tests/cdc/00000001000000000000002.json /tmp/00000001000000000000002.sql
```

And the SQL file obtained looks like this:

```
$ cat /tmp/00000001000000000000002.sql
BEGIN; -- {"xid":489,"lsn":"0/236F5A8"}
INSERT INTO "public"."rental" (rental_id, rental_date, inventory_id, customer_id, return_date, staff_id, last_
↪update) VALUES (16050, '2022-06-01 00:00:00+00', 371, 291, NULL, 1, '2022-06-01 00:00:00+00');
INSERT INTO "public"."payment_p2020_06" (payment_id, customer_id, staff_id, rental_id, amount, payment_date)␣
↪VALUES (32099, 291, 1, 16050, 5.99, '2020-06-01 00:00:00+00');
COMMIT; -- {"xid": 489,"lsn":"0/236F5A8"}
```

4.10 pgcopydb configuration

Manual page for the configuration of pgcopydb. The `pgcopydb` command accepts sub-commands and command line options, see the manual for those commands for details. The only setup that `pgcopydb` commands accept is the filtering.

4.10.1 Filtering

Filtering allows to skip some object definitions and data when copying from the source to the target database. The pgcopydb commands that accept the option `--filter` (or `--filters`) expect an existing filename as the option argument. The given filename is read in the INI file format, but only uses sections and option keys. Option values are not used.

Here is an inclusion based filter configuration example:

```
[include-only-table]
public.allcols
public.csv
public.serial
public.xzero

[exclude-index]
public.foo_gin_tsvector

[exclude-table-data]
public.csv
```

Here is an exclusion based filter configuration example:

```
[exclude-schema]
foo
bar
expected

[exclude-table]
"schema"."name"
schema.othername
err.errors
public.serial

[exclude-index]
schema.indexname

```

(continues on next page)

(continued from previous page)

```
[exclude-table-data]
public.bar
nsitra.test1
```

Filtering can be done with pgcopydb by using the following rules, which are also the name of the sections of the INI file.

include-only-tables

This section allows listing the exclusive list of the source tables to copy to the target database. No other table will be processed by pgcopydb.

Each line in that section should be a schema-qualified table name. Postgres identifier quoting rules can be used to avoid ambiguity.

When the section `include-only-tables` is used in the filtering configuration then the sections `exclude-schema` and `exclude-table` are disallowed. We would not know how to handle tables that exist on the source database and are not part of any filter.

exclude-schema

This section allows adding schemas (Postgres namespaces) to the exclusion filters. All the tables that belong to any listed schema in this section are going to be ignored by the pgcopydb command.

This section is not allowed when the section `include-only-tables` is used.

exclude-table

This section allows to add a list of qualified table names to the exclusion filters. All the tables that are listed in the `exclude-table` section are going to be ignored by the pgcopydb command.

This section is not allowed when the section `include-only-tables` is used.

exclude-index

This section allows to add a list of qualified index names to the exclusion filters. It is then possible for pgcopydb to operate on a table and skip a single index definition that belong to a table that is still processed.

exclude-table-data

This section allows to skip copying the data from a list of qualified table names. The schema, index, constraints, etc of the table are still copied over.

4.10.2 Reviewing and Debugging the filters

Filtering a `pg_restore` archive file is done through rewriting the archive catalog obtained with `pg_restore --list`. That's a little hackish at times, and we also have to deal with dependencies in pgcopydb itself.

The following commands can be used to explore a set of filtering rules:

- *pgcopydb list depends*
- *pgcopydb restore parse-list*

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search