

Data organization

Low level



Copyright

© Postgres Professional, 2015–2022

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Alexander Meleshko

Use of course materials

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

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

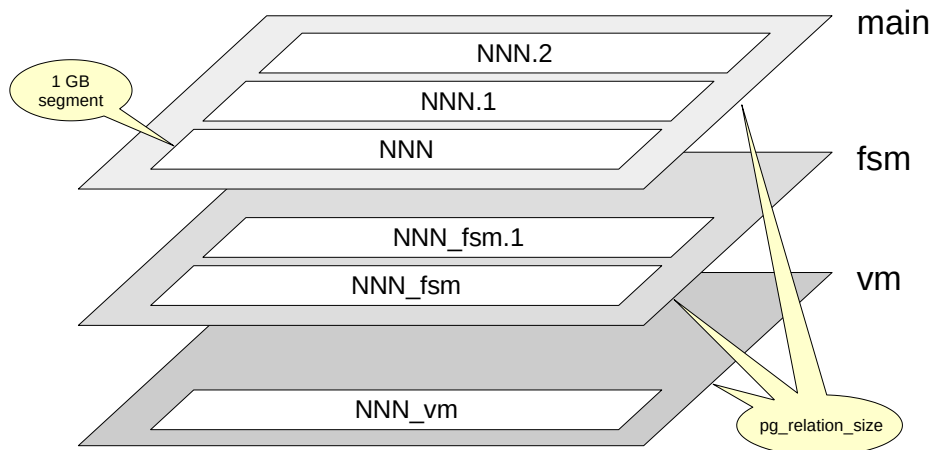
Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided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

Data files

Forks: main, visibility map, free space map

Oversized row versions and TOAST

Object forks



3

Usually, each database object that stores data (table, index, sequence, materialized view) has several corresponding forks. Each fork contains a specific type of data.

Initially, each fork contains a single file. The file name is a numeric identifier and may include a suffix derived from the fork name.

The file gradually increases in size until it reaches 1 GB, at which point the next file for the same fork is created. Such files are sometimes called *segments*. The segment sequence number is appended to the end of the file name. The `pg_relation_size` function displays the total size of a fork.

The 1 GB file size limit was established in the past to support file systems that cannot operate with larger file sizes. A different file size limit can be set during source code compilation with the `--with-segsize` flag.

So, a single database object may consist of multiple files on disk. A small table will have three corresponding files on disk, and an index will have two. All object files belonging to the same tablespace and the same database are stored in the same directory. This may become an issue as some file systems may perform poorly on directories with a large number of files.

Main fork

- actual data (row versions)
- exists for all objects

Initialization fork (init)

- A “template” of the main fork
- used in case of failure; exists only for unlogged tables

Visibility map (vm)

- exists only for tables

Free space map (fsm)

- exists for both tables and indexes

There are multiple types of forks.

The *main fork* contains the actual data, such as table row versions and index records. The main fork file names match the identifier. All objects have a main fork.

The file names of the *initialization fork* end with the “_init” suffix. This fork exists only for unlogged tables (created with the UNLOGGED keywords) and their indexes. Unlogged tables are no different from regular ones, except that actions performed on them are not logged in WAL. This makes operations on them faster, but their content cannot be recovered if a failure occurs. When recovering after a failure, PostgreSQL simply wipes all unlogged table forks and copies the initialization fork into the main fork. The result is an empty table.

<https://postgrespro.com/docs/postgresql/13/storage-init>

The vm (visibility map) fork’s filenames end in “_vm”. The fork exists only for tables. Separate MVCC for indexes is not supported.

The fsm (free space map) fork’s filenames end in “_fsm”. This fork exists for both tables and indexes.

These two maps were discussed in the Architecture module.

<https://postgrespro.com/docs/postgresql/13/storage-fsm>

<https://postgrespro.com/docs/postgresql/13/storage-vm>

File locations

```
=> CREATE DATABASE data_lowlevel;
```

```
CREATE DATABASE
```

```
=> \c data_lowlevel
```

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

Create a table and look where its files are.

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

```
CREATE TABLE
```

```
=> INSERT INTO t(n) SELECT id FROM generate_series(1,10000) AS id;
```

```
INSERT 0 10000
```

```
=> VACUUM t;
```

```
VACUUM
```

The path to the main file relative to PGDATA is shown with the following command:

```
=> SELECT pg_relation_filepath('t');
```

```
pg_relation_filepath
-----
base/16527/16530
(1 row)
```

Since the table is located in the pg_default tablespace, the path starts with "base", followed by the database directory:

```
=> SELECT oid FROM pg_database WHERE datname = 'data_lowlevel';
```

```
oid
-----
16527
(1 row)
```

Then follows the file name. To get it, use the command:

```
=> SELECT relfilenode FROM pg_class WHERE relname = 't';
```

```
relfilenode
-----
16530
(1 row)
```

Otherwise, use the function pg_relation_filepath to get the path in a single command.

Let's have a look at the files themselves. Only the OS user postgres has access to PGDATA, so run the ls on their behalf:

```
postgres$ ls -l /var/lib/postgresql/13/main/base/16527/16530*
```

```
-rw----- 1 postgres postgres 450560 Mar  7 13:50 /var/lib/postgresql/13/main/base/16527/16530
-rw----- 1 postgres postgres  24576 Mar  7 13:50 /var/lib/postgresql/13/main/base/16527/16530_fsm
-rw----- 1 postgres postgres   8192 Mar  7 13:50 /var/lib/postgresql/13/main/base/16527/16530_vm
```

There are three forks: the main fork, the free space map (fsm) and the visibility map (vm).

You can view the index files in a similar way:

```
=> \d t
```

```

              Table "public.t"
  Column | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
 id      | integer |           | not null | generated always as identity
 n       | numeric |           |          |
Indexes:
    "t_pkey" PRIMARY KEY, btree (id)
```

```
=> SELECT pg_relation_filepath('t_pkey');
```

```
pg_relation_filepath
-----
base/16527/16536
(1 row)
```

```
postgres$ ls -l /var/lib/postgresql/13/main/base/16527/16536*
```

```
-rw----- 1 postgres postgres 245760 Mar  7 13:50 /var/lib/postgresql/13/main/base/16527/16536
```

And the primary key sequence files:

```
=> SELECT pg_relation_filepath(pg_get_serial_sequence('t', 'id'));
```

```
pg_relation_filepath
-----
base/16527/16528
(1 row)
```

```
postgres$ ls -l /var/lib/postgresql/13/main/base/16527/16528*
```

```
-rw----- 1 postgres postgres 8192 Mar  7 13:50 /var/lib/postgresql/13/main/base/16527/16528
```

A stock extension oid2name lets you quickly and easily find out which database objects relate to which files.

You can view all databases:

```
student$ /usr/lib/postgresql/13/bin/oid2name
```

All databases:

Oid	Database Name	Tablespace
16527	data_lowlevel	pg_default
13485	postgres	pg_default
16385	student	pg_default
13484	template0	pg_default
1	template1	pg_default

All objects in a database:

```
student$ /usr/lib/postgresql/13/bin/oid2name -d data_lowlevel
```

From database "data_lowlevel":

Filenode	Table Name
16530	t

All tablespaces in a database:

```
student$ /usr/lib/postgresql/13/bin/oid2name -d data_lowlevel -s
```

All tablespaces:

Oid	Tablespace Name
1663	pg_default
1664	pg_global

Find the file name by table name:

```
student$ /usr/lib/postgresql/13/bin/oid2name -d data_lowlevel -t t
```

From database "data_lowlevel":

Filenode	Table Name
16530	t

Or the table name by file name:

```
student$ /usr/lib/postgresql/13/bin/oid2name -d data_lowlevel -f 16530
```

From database "data_lowlevel":

Filenode	Table Name
16530	t

Fork sizes

You can get the size of the files that comprise a fork from the file system, but there is an easier way to see the size of each fork individually:

```
=> SELECT pg_relation_size('t','main') main,  
         pg_relation_size('t','fsm') fsm,  
         pg_relation_size('t','vm') vm;
```

main	fsm	vm
450560	24576	8192

(1 row)

A row version must fit into one page

- some of the fields can be compressed
- some fields can be moved into a TOAST table
- fields can be both compressed and moved

TOAST table

- located in the `pg_toast` (`pg_toast_temp_N`) schema
- supported by its own index
- contains chunks of oversized values, each chunk is smaller than a page
- accessed by querying a corresponding oversized field
- has its own MVCC
- used transparently for the application

Any row version in PostgreSQL must fit entirely into one page. Oversized row versions are stored using TOAST, The Oversized Attributes Storage Technique. TOAST comprises several approaches to storing oversized field values. Firstly, the value can be compressed so that the row version fits into the page. Secondly, the value can be moved from the row version to a separate service table. Both strategies can be applied to the same row versions: some values would be compressed, some moved, some compressed and moved.

Any table can have a separate TOAST table (with a dedicated index) created for it, if necessary. The dedicated indexes are located in the `pg_toast` schema and therefore are usually not visible (temporary TOAST tables are stored in the `pg_toast_temp_N` schema, similarly to the regular `pg_temp_N`).

The row versions in the TOAST table must also fit into one page each, so longer values are split into multiple chunks, and are transparently “glued together” by PostgreSQL when the application demands.

TOAST tables are used only when oversized values are queried. The tables have their own versioning mechanism. Whenever a data update in the main table does not modify the oversized value in the TOAST table, the new row version in the table will refer to the same old TOAST value, saving disk space.

<https://postgrespro.com/docs/postgresql/13/storage-toast>

TOAST

The table `t` has a numeric type column. This type can hold very large numbers. For example:

```
=> SELECT length( (123456789::numeric ^ 12345::numeric)::text );

length
-----
 99907
(1 row)
```

However, when inserted into the table, this humongous value does not change the table size:

```
=> SELECT pg_relation_size('t', 'main');

pg_relation_size
-----
          450560
(1 row)
```

```
=> INSERT INTO t(n) SELECT 123456789::numeric ^ 12345::numeric;

INSERT 0 1
```

```
=> SELECT pg_relation_size('t', 'main');

pg_relation_size
-----
          450560
(1 row)
```

Since the row version does not fit into a single page, it is instead stored in a separate TOAST table. TOAST tables and their indexes are created automatically for all tables that include potentially “oversized” data types and are used as needed.

You can find the name and oid of a TOAST table:

```
=> SELECT relname, relfilenode FROM pg_class WHERE oid = (
    SELECT reltoastrelid FROM pg_class WHERE oid = 't'::regclass
);

 relname      | relfilenode
-----+-----
 pg_toast_16530 |          16533
(1 row)
```

And here are the TOAST table files:

```
postgres$ ls -l /var/lib/postgresql/13/main/base/16527/16533*
-rw----- 1 postgres postgres 57344 Mar  7 13:50 /var/lib/postgresql/13/main/base/16527/16533
-rw----- 1 postgres postgres 24576 Mar  7 13:50 /var/lib/postgresql/13/main/base/16527/16533_fsm
```

When it comes to oversized values, there are several strategies that can be employed. The name of the current strategy is listed in the Storage column:

```
=> \d+ t
```

Column	Type	Collation	Nullable	Table "public.t" Default	Storage	Stats target	Description
id	integer		not null	generated always as identity	plain		
n	numeric				main		

Indexes:
"t_pkey" PRIMARY KEY, btree (id)
Access method: heap

- plain — TOAST is not used (the type has a fixed length),
- extended — both compression and external storage are used,
- external — external storage but not compression,
- main — processed last, compression is preferred.

You can select what strategy to use. For example, if you know that data in a table is already compressed, you can switch

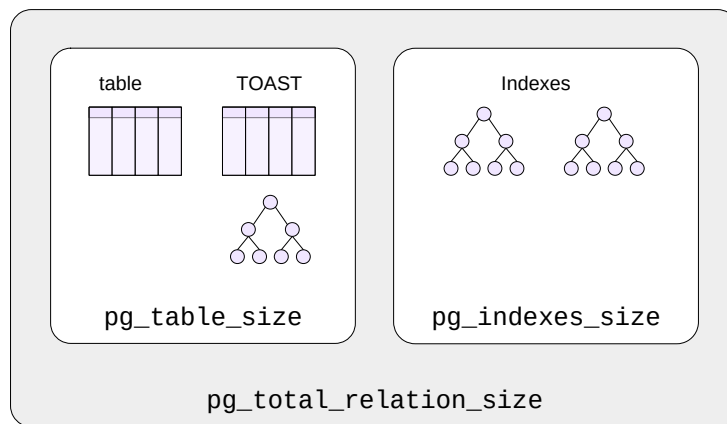
the strategy to external.

For example:

```
=> ALTER TABLE t ALTER COLUMN n SET STORAGE external;
```

```
ALTER TABLE
```

This operation does not change the data, but defines the strategy to be used for new row versions.



As already mentioned, the size of a single fork can be obtained by the `pg_relation_size` function. To get the total object size, other functions can be used:

- `pg_table_size` shows the size of the table and its TOAST part (the TOAST table and its index), but not the regular index sizes. The same function can be used to find the size of an individual index: both tables and indexes are relations, and despite the name, the function accepts any relation as input.
- `pg_indexes_size` sums up the sizes of all table indexes except the TOAST table index.
- `pg_total_relation_size` shows the full size of the table, along with all its indexes.

Table size

The size of a table (including the TOAST table and its index):

```
=> SELECT pg_table_size('t');
```

```
pg_table_size
-----
          581632
(1 row)
```

Total size of all table indexes:

```
=> SELECT pg_indexes_size('t');
```

```
pg_indexes_size
-----
          245760
(1 row)
```

You can get the size of a single index by using the `pg_table_size` function. Indexes have no TOASTs, so the function only shows the size of all index forks (main, fsm).

Currently, the table `t` has just the primary key index, so its size matches the size returned by `pg_indexes_size`:

```
=> SELECT pg_table_size('t_pkey') AS t_pkey;
```

```
t_pkey
-----
    245760
(1 row)
```

Total table size, including TOAST and all indexes:

```
=> SELECT pg_total_relation_size('t');
```

```
pg_total_relation_size
-----
          827392
(1 row)
```

An object comprises several forks

A fork consists of one or more segment files

Oversized row versions are stored using TOAST

1. Create an unlogged table in a custom tablespace and make sure that it has an init fork.
Delete the created tablespace.
2. Create a table with a column of the `text` type.
What storage strategy is used for this column?
Change the strategy to `external` and insert a short and a long row into the table.
Check if the rows are in the TOAST table by making a direct query to it. Explain why.

1. Unlogged tables

```
student$ sudo mkdir /var/lib/postgresql/ts_dir
student$ sudo chown postgres /var/lib/postgresql/ts_dir
=> CREATE TABLESPACE ts LOCATION '/var/lib/postgresql/ts_dir';
CREATE TABLESPACE
=> CREATE DATABASE data_lowlevel;
CREATE DATABASE
=> \c data_lowlevel
You are now connected to database "data_lowlevel" as user "student".
=> CREATE UNLOGGED TABLE u(n integer) TABLESPACE ts;
CREATE TABLE
=> INSERT INTO u(n) SELECT n FROM generate_series(1,1000) n;
INSERT 0 1000
=> SELECT pg_relation_filepath('u');
           pg_relation_filepath
-----
pg_tblspc/16705/PG_13_202007201/16706/16707
(1 row)
```

Let's look at the table files.

Note how the ls command is executed on behalf of the postgres user. You can open a second terminal window and switch to the new user with the following command:

```
student$ sudo su postgres
```

Now, in the same window, run:

```
postgres$ ls -l /var/lib/postgresql/13/main/pg_tblspc/16705/PG_13_202007201/16706/16707*
-rw----- 1 postgres postgres 40960 Mar  7 13:54 /var/lib/postgresql/13/main/pg_tblspc/16705/PG_13_202007201/16706/16707
-rw----- 1 postgres postgres 24576 Mar  7 13:54 /var/lib/postgresql/13/main/pg_tblspc/16705/PG_13_202007201/16706/16707_fsm
-rw----- 1 postgres postgres    0 Mar  7 13:54 /var/lib/postgresql/13/main/pg_tblspc/16705/PG_13_202007201/16706/16707_init
```

Drop the created tablespace:

```
=> DROP TABLE u;
DROP TABLE
=> DROP TABLESPACE ts;
DROP TABLESPACE
```

2. A table with a text column

```
=> CREATE TABLE t(s text);
CREATE TABLE
=> \d+ t
           Table "public.t"
  Column | Type  | Collation | Nullable | Default | Storage  | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+-----
s        | text  |           |          |         | extended |              |
Access method: heap
```

By default, the extended strategy is used for text data.

Change the strategy to external:

```
=> ALTER TABLE t ALTER COLUMN s SET STORAGE external;
ALTER TABLE
=> INSERT INTO t(s) VALUES ('Short string.');
```

```
INSERT 0 1
=> INSERT INTO t(s) VALUES (repeat('A',3456));
INSERT 0 1
```

Check the toast table:

```
=> SELECT relname FROM pg_class WHERE oid = (
      SELECT reltoastrelid FROM pg_class WHERE relname='t'
    );
```

```
      relname
-----
pg_toast_16710
(1 row)
```

The toast table is “hidden”, because it is located in a schema that is excluded from the search path. This is a good thing, because TOAST is intended to work transparently for the user. However, there still are ways to view the table:

```
=> SELECT chunk_id, chunk_seq, length(chunk_data)
FROM pg_toast.pg_toast_16710
ORDER BY chunk_id, chunk_seq;
```

```
 chunk_id | chunk_seq | length
-----+-----+-----
    16716 |         0 |   1996
    16716 |         1 |   1460
(2 rows)
```

Only the long string went into the toast table (two chunks, total size matches the string size). The short string wasn't toasted: there is no need, as it already fits into one page.

1. Create a database.

Compare the database size returned by the `pg_database_size` command with the total size of all tables in the database.

Explain the result.

1. You can get the list of database tables from the `pg_class` table.

1. Comparing the size of a database to the total size of its tables

```
=> CREATE DATABASE data_lowlevel;
```

```
CREATE DATABASE
```

```
=> \c data_lowlevel
```

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

Even an empty database contains some system catalog tables. The list of all tables is stored in `pg_class`. Exclude from the calculation:

- the cluster's shared tables (they don't belong to the database),
- indexes and TOAST tables (they will be included in the calculation automatically).

```
=> SELECT sum(pg_total_relation_size(oid))
FROM pg_class
WHERE NOT rellisshared -- local database objects
AND relkind = 'r'; -- regular tables
```

```
sum
-----
7995392
(1 row)
```

The size of the database is a bit larger:

```
=> SELECT pg_database_size('data_lowlevel');
```

```
pg_database_size
-----
8147503
(1 row)
```

This is because the `pg_database_size` function returns the size of the catalog in the file system, and the catalog contains some service files.

```
=> SELECT oid FROM pg_database WHERE datname = 'data_lowlevel';
```

```
oid
-----
16717
(1 row)
```

Note that the following `ls` command is executed on behalf of the `postgres` user. To follow along, open a new terminal window and switch to the `postgres` user:

```
student$ sudo su postgres
```

In the same window, run:

```
postgres$ ls -l /var/lib/postgresql/13/main/base/16717/[^0-9]*
```

```
-rw----- 1 postgres postgres  512 Mar  7 13:54 /var/lib/postgresql/13/main/base/16717/pg_filenode.map
-rw----- 1 postgres postgres 151596 Mar  7 13:54 /var/lib/postgresql/13/main/base/16717/pg_internal.init
-rw----- 1 postgres postgres    3 Mar  7 13:54 /var/lib/postgresql/13/main/base/16717/Pg_VERSION
```

- `pg_filenode.map` — mapping OIDs of some tables to file names,
- `pg_internal.init` — system catalog cache,
- `Pg_VERSION` — PostgreSQL version.

As some functions operate on the database object level, and others on the file system level, it is sometimes hard to compare the results directly. The same goes for the `pg_tablespace_size` function.