Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

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

Agenda

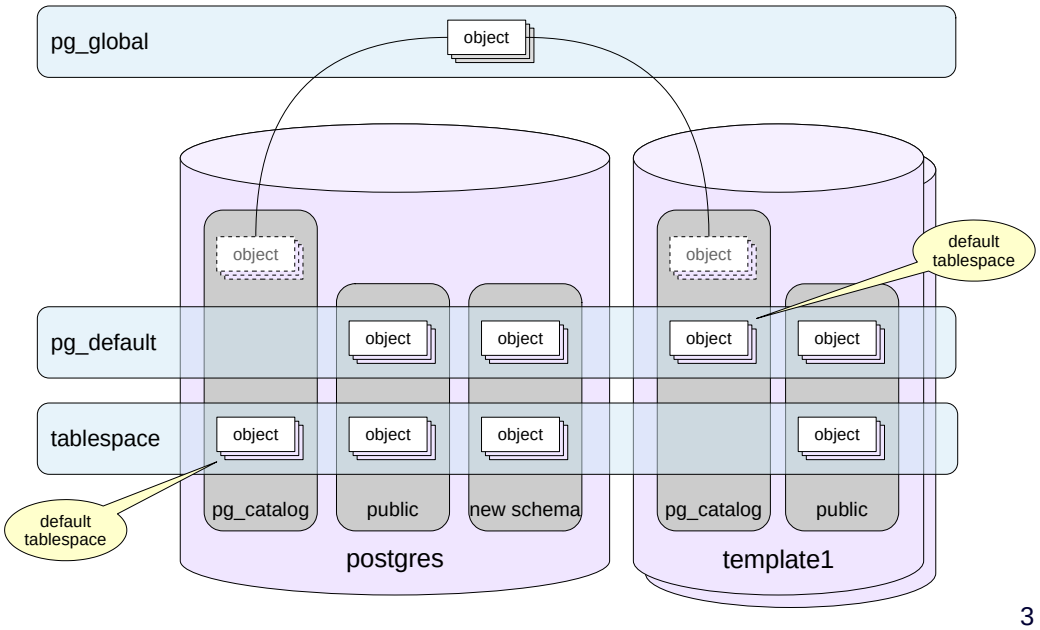
Tablespaces and directories

Files and data pages

Forks: data, visibility map, free space map

TOAST

Tablespaces



Tablespaces are used to manage physical storage of data; they define the layout of files in the file system.

For example, one tablespace can be created on slow disks to store archive data, while another tablespace located on fast disks will be used for the data that is being actively updated.

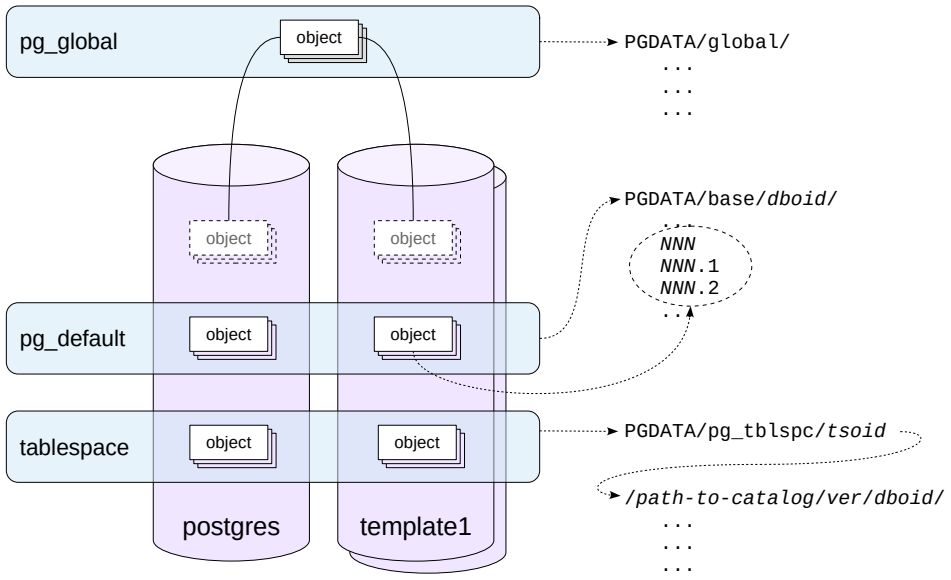
During cluster initialization, two tablespaces are created: `pg_default` and `pg_global`.

One and the same tablespace can be used by several databases, and one database can store data in several tablespaces.

Besides, each database has a so-called “default tablespace,” in which all database objects are created unless another location is specified. This tablespace also stores system catalog objects. Initially, the `pg_default` tablespace is used as the default one, but you can change this behavior.

The `pg_global` tablespace is special: it stores those system catalog objects that are common to the whole cluster.

Catalogs



Basically, a tablespace is a reference to the catalog that stores data. The pre-defined `pg_global` and `pg_default` tablespaces are always located in the PGDATA/global/ and PGDATA/base/ directories, respectively. When creating a custom tablespace, you can specify an arbitrary directory; the server always uses relative links, so it creates an additional symlink in PGDATA/pg_tblspc/ that points to this custom directory.

Within the PGDATA/base/ directory, all data is distributed between subdirectories of different databases (PGDATA/global/ has no such subdirectories because it contains the data related to the cluster as a whole).

Custom tablespaces have one more nesting level to reflect the PostgreSQL version. It is done to facilitate server upgrades.

Actual database objects are stored in files within these directories: each object has its own files.

Each file, which is called a *segment*, takes no more than 1 GB of disk space (this value can be altered at build time). That's why each object can have several corresponding files. Thus, a database can potentially have plenty of files, and their impact on the file system has to be taken into account.

https://postgrespro.com/docs/postgresql/12/storage-file-layout

Using Tablespaces

Initially, there are two tablespaces in a cluster. They are listed in the following table of the system catalog:

```
=> SELECT spcname FROM pg_tablespace;

  spcname
------------
 pg_default
 pg_global
(2 rows)
```

Naturally, it's one of the tables common for the whole cluster.

You can also get the same information with the following psql command:

```
=> \db

       List of tablespaces
    Name    |  Owner   | Location
------------+----------+----------
 pg_default | postgres |
 pg_global  | postgres |
(2 rows)
```

To add a new tablespace, it is required to create an empty directory owned by the user that has started the database server (here we run this command on behalf of the postgres user):

```
postgres$ mkdir /var/lib/postgresql/ts_dir
```

Now run the following command specifying the created directory:

```
=> CREATE TABLESPACE ts LOCATION '/var/lib/postgresql/ts_dir';

CREATE TABLESPACE

=> \db

               List of tablespaces
    Name    |  Owner   |          Location
------------+----------+----------------------------
 pg_default | postgres |
 pg_global  | postgres |
 ts         | student  | /var/lib/postgresql/ts_dir
(3 rows)
```

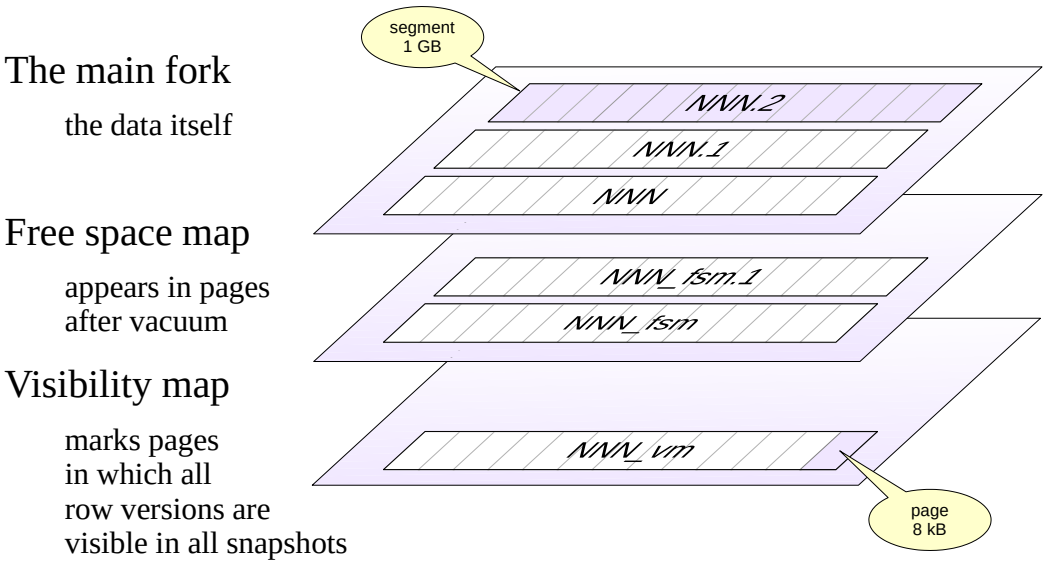
When creating a database, you can set the default tablespace:

```
=> CREATE DATABASE data_physical TABLESPACE ts;

CREATE DATABASE

=> \c data_physical

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

It means that all objects will be created in this tablespace unless another tablespace is explicitly specified.

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

CREATE TABLE

=> INSERT INTO t(id, s)
    SELECT id, id::text FROM generate_series(1,100000) id;

INSERT 0 100000

=> VACUUM t;

VACUUM
```

Files and Forks



6

Each object is usually represented by several *forks*. Each fork is a set of *segments* (i.e., a file or several files). All segments are split into separate *pages*. The page size is usually 8 kB; it can be configured for the whole cluster, but only at build time. The pages of different objects are read from disk using the same buffer cache mechanism.

The **main fork** is the data itself: index rows or different versions of table rows (which are called *tuples*).

The vm fork is a **visibility map** (a bitmap). It marks the pages that contain only the current tuples visible in all data snapshots. In other words, these are the pages that have not been modified in a while and have no outdated tuples left.

Visibility map is used to optimize vacuum (the pages marked in the visibility map do not have to be vacuumed) and to speed up index access. The point is that versioning information is stored only for tables, but not for indexes (that's why indexes have no visibility maps). Once a reference to a tuple is found in index, the corresponding table page has to be read to check the visibility status of this tuple. But this step can be skipped if the index itself already contains all the required columns, and the page is present in the visibility map.

The fsm fork is a **free space map**. It tracks free space within pages, which can appear after vacuum. This map is used to quickly find a page that is suitable for inserting a new tuple.

Files and Forks

You can determine the location of the files related to a particular object as follows:

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

```
            pg_relation_filepath
-----------------------------------------------
 pg_tblspc/16411/PG_12_201909212/16412/16413
(1 row)
```

Let's take a look at these files (their name and size in bytes):

```
student$ sudo find /var/lib/postgresql/12/main/pg_tblspc/16411/PG_12_201909212/16412/16413* -printf '%f\t%s\n'
```

```
find: '/var/lib/postgresql/12/main/pg_tblspc/16411/PG_12_201909212/16412/16413*': No such file or directory
```

We can see that they belong to three forks: the main one, fsm, and vm.

Objects can be moved between tablespaces, but (unlike moving between schemas) it results in a physical move of the data:

```
=> ALTER TABLE t SET TABLESPACE pg_default;
```

```
ALTER TABLE
```

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

```
 pg_relation_filepath
----------------------
 base/16412/16421
(1 row)
```

Object Size

There are several functions that return the size of the database and its objects.

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

```
 pg_database_size
------------------
         14902127
(1 row)
```

To facilitate comprehension, the value can be displayed in a human-readable format:

```
=> SELECT pg_size_pretty(pg_database_size('data_physical'));
```

```
 pg_size_pretty
----------------
 14 MB
(1 row)
```

The full size of the table (including all indexes):

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

```
 pg_size_pretty
----------------
 6568 kB
(1 row)
```

The size of the table itself...

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

```
 pg_size_pretty
----------------
 4360 kB
(1 row)
```

...and the size of the indexes:

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

```
 pg_size_pretty
----------------
 2208 kB
(1 row)
```

You can also learn the size of separate table forks, if you like. For example:

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

 pg_size_pretty
----------------
 4320 kB
(1 row)
```

Here is another function that returns the tablespace size:

```
=> SELECT pg_size_pretty(pg_tablespace_size('ts'));

 pg_size_pretty
----------------
 10209 kB
(1 row)
```

TOAST

A tuple must fit one page

- some attributes can be compressed,
- moved to a separate TOAST table,
- or both compressed and moved

A TOAST table

- is stored in the `pg_toast` schema
- provides its own index
- splits “oversized” attributes into chunks that are smaller than a page
- is read only when an “oversized” attribute is queried
- uses its own versioning
- is seamlessly used by applications

In PostgreSQL, any row version must fit a single page. For “oversized” tuples, the TOAST mechanism is used, which stands for “The Oversized Attributes Storage Technique”. It implies several strategies. Some oversized attributes can be compressed for the tuple to fit a page. If it is impossible, the attribute can be moved into a separate service table. These two approaches can also be combined.

If required, for each main table, PostgreSQL creates a separate TOAST table (with a special index). Such tables and indexes are located in a separate schema called `pg_toast`, so they are usually invisible.

Tuples in TOAST tables must also fit a single page, so to store oversized values, PostgreSQL splits them into chunks. When required by an application, these chunks are seamlessly put together to produce a full value.

TOAST tables are accessed only if the oversized value has to be returned. Besides, TOAST tables have their own versioning: if an update does not affect the oversized value, the new tuple refers to the same value in the TOAST table. This approach allows us to save some space.

https://postgrespro.com/docs/postgresql/12/storage-toast

TOAST

Let's insert a very long row into the table:

```
=> INSERT INTO t(id, s)
SELECT 0, string_agg(id::text,'.') FROM generate_series(1,10000) AS id;
```

```
INSERT 0 1
```

```
=> VACUUM;
```

```
VACUUM
```

Will the table size change?

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

```
 pg_size_pretty
----------------
 4440 kB
(1 row)
```

Yes. And what about the main fork that stores the data?

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

```
 pg_size_pretty
----------------
 4320 kB
(1 row)
```

The size remains the same.

Since the row does not fit into a single page, the value of the s attribute will be split into chunks and stored in a separate TOAST table. You can find it in the system catalog (we use the regclass type to convert oid into the relation name):

```
=> SELECT reltoastrelid::regclass::text FROM pg_class WHERE relname='t';
```

```
      reltoastrelid
-------------------------
 pg_toast.pg_toast_16413
(1 row)
```

Our row is split into chunks to be stored; PostgreSQL puts these chunks together to get the full value when required:

```
=> SELECT chunk_id, chunk_seq, left(chunk_data::text,45) AS chuck_data
FROM pg_toast.pg_toast_16413 LIMIT 5;
```

```
 chunk_id | chunk_seq |                  chuck_data
----------+-----------+-----------------------------------------------
    16424 |         0 | \xfdbe000000312e322e332e342e00352e362e372e382
    16424 |         1 | \x392e353161ff31002e3531322e353133002e3531342
    16424 |         2 | \xe215e216e217e218abe219e11a30e11b30e11c30e11
    16424 |         3 | \x11f4aa3611f43611f43611f43611f4aa3611f43611f
    16424 |         4 | \xf43211f4325511f43211f43211f43211f4325511f43
(5 rows)
```

Let's delete this database since we no longer need it.

```
=> \c postgres
```

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

```
=> DROP DATABASE data_physical;
```

```
DROP DATABASE
```

Once there are no objects left in the tablespace, you can delete it, too:

```
=> DROP TABLESPACE ts;
```

```
DROP TABLESPACE
```

Summary

At the physical level

- the data is distributed between tablespaces (directories)
- an object is represented by several forks
- each fork consists of one or more segments

Tablespaces are managed by DBAs

Files, forks, and TOAST are managed internally by PostgreSQL

Practice

1. Create a new database and establish a connection with it.
   Create a tablespace called `ts`.
   Create table `t` in tablespace `ts`
   and insert several rows into this table.
2. Calculate the size of the database, the table, as well as the size of `ts` and `pg_default` tablespaces.
3. Move the table into the `pg_default` tablespace.
   How has the tablespace size changed?
4. Delete the `ts` tablespace.

Task 1. Tablespaces and Tables

Let's create a database:

```
=> CREATE DATABASE data_physical;

CREATE DATABASE

=> \c data_physical

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

Create a tablespace:

```
postgres$ mkdir /var/lib/postgresql/ts_dir

=> CREATE TABLESPACE ts LOCATION '/var/lib/postgresql/ts_dir';

CREATE TABLESPACE
```

Create a table:

```
=> CREATE TABLE t(n integer) TABLESPACE ts;

CREATE TABLE

=> INSERT INTO t SELECT 1 FROM generate_series(1,1000);

INSERT 0 1000
```

Task 2. Data Size

The database size:

```
=> SELECT pg_size_pretty(pg_database_size('data_physical')) AS db_size;

 db_size
---------
 8041 kB
(1 row)
```

The table size:

```
=> SELECT pg_size_pretty(pg_total_relation_size('t')) AS t_size;

 t_size
--------
 64 kB
(1 row)
```

The tablespace size:

```
=> SELECT
    pg_size_pretty(pg_tablespace_size('pg_default')) AS pg_default_size,
    pg_size_pretty(pg_tablespace_size('ts')) AS ts_size;

 pg_default_size | ts_size
-----------------+---------
 164 MB          | 68 kB
(1 row)
```

The tablespace size is a bit bigger than the table size because the tablespace directory also contains some service files.

Task 3. Moving a Table

Let's move our table:

```
=> ALTER TABLE t SET TABLESPACE pg_default;

ALTER TABLE
```

Check the new tablespace size:

```
=> SELECT
    pg_size_pretty(pg_tablespace_size('pg_default')) AS pg_default_size,
    pg_size_pretty(pg_tablespace_size('ts')) AS ts_size;
```

```
 pg_default_size |  ts_size
-----------------+------------
 164 MB          | 4096 bytes
(1 row)
```

Task 4. Deleting a Tablespace

Let’s delete our tablespace:

```
=> DROP TABLESPACE ts;
```

```
DROP TABLESPACE
```