Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

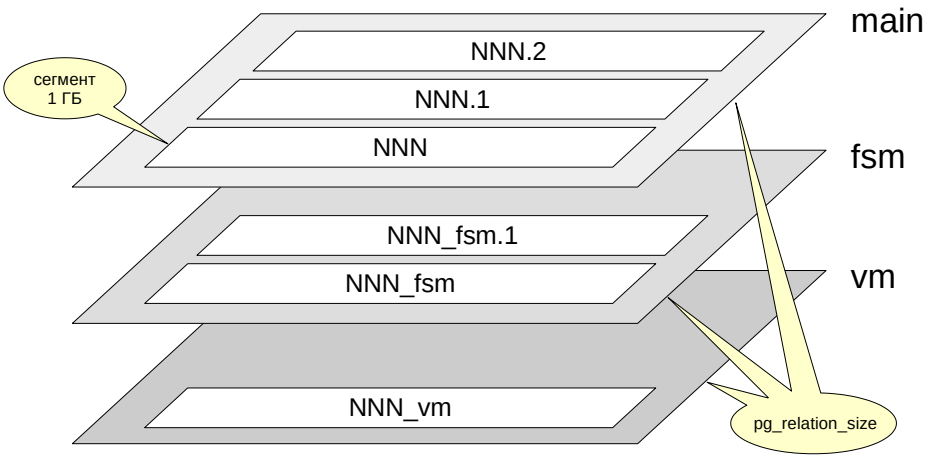
Темы

Файлы и страницы

Слои: данные, карты видимости и свободного пространства

Длинные версии строк и TOAST

Слои объекта



Обычно каждому объекту БД, хранящему данные (таблице, индексу, последовательности, материализованному представлению), соответствует несколько *слоев* (forks). Каждый слой содержит определенный вид данных.

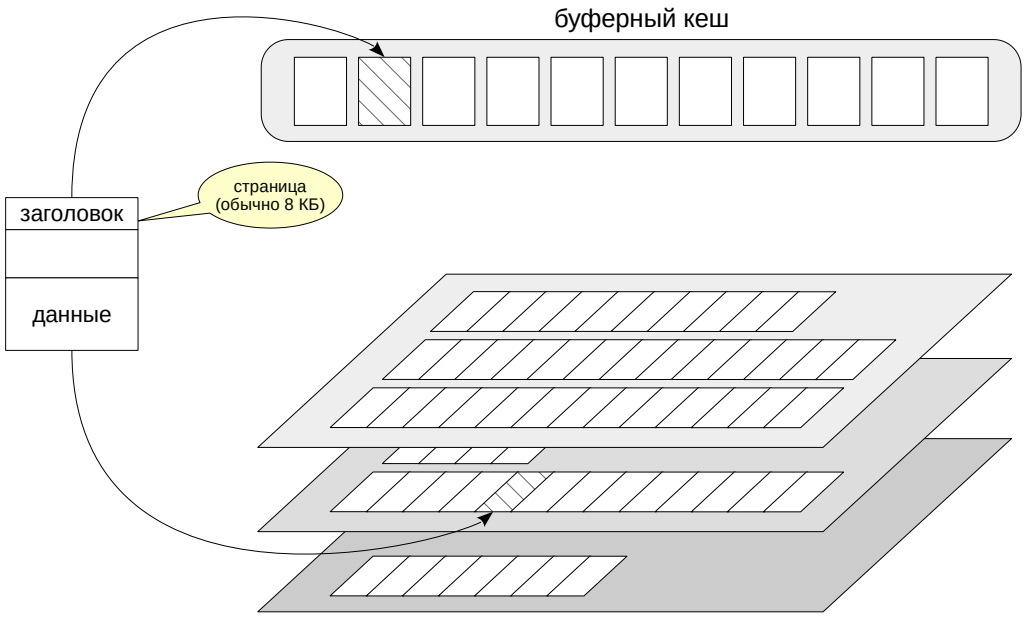
Вначале слой содержит один-единственный файл. Имя файла состоит из числового идентификатора, к которому может быть добавлено окончание, соответствующее имени слоя.

Файл постепенно растет и, когда его размер доходит до 1 Гбайта, создается следующий файл этого же слоя. Такие файлы иногда называют *сегментами.* Порядковый номер сегмента добавляется в конец имени файла. Общий размер любого слоя показывает функция pg_relation_size.

Ограничение размера файла в 1 Гбайт возникло исторически для поддержки различных файловых систем, некоторые из которых не умеют работать с файлами большого размера. Установить другой размер можно только при сборке сервера из исходных кодов (--with-segsize).

Таким образом, одному объекту БД на диске может соответствовать несколько файлов. Для небольшой таблицы их будет 3, для индекса — два. Все файлы объектов, принадлежащих одному табличному пространству и одной БД, будут помещены в один каталог. Это необходимо учитывать, потому что файловые системы могут не очень хорошо работать с большим количеством файлов в каталоге.

Страницы файлов



Файлы, в свою очередь, разделены на *страницы* (иногда используется термин *блок*). Страница обычно имеет размер 8 Кбайт. Его в некоторых пределах можно поменять (16 или 32 Кбайта), но только при сборке. Собранный и запущенный кластер может работать со страницами только одного размера.

Независимо от того, к какому слою принадлежат файлы, они используются буферным менеджером примерно одинаково. Страницы сначала читаются в буферный кеш, там их могут читать и изменять процессы PostgreSQL, затем при необходимости страницы вытесняются обратно на диск.

Каждая страница имеет внутреннюю разметку. Она содержит заголовок и полезные данные; между ними может находиться свободное пространство, если страница занята не полностью.

Слои

Основной слой

- собственно данные (версии строк)
- существует для всех объектов

Слой инициализации (init)

- «пустышка» для основного слоя
- используется при сбое; только для нежурналируемых таблиц

Карта видимости (vm)

- существует только для таблиц

Карта свободного пространства (fsm)

- существует и для таблиц, и для индексов

Посмотрим теперь на типы слоев.

Основной слой — это собственно данные: версии строк таблиц или индексные записи. Имена файлов основного слоя совпадают с идентификатором. Основной слой существует для любых объектов.

Имена файлов *слоя инициализации* оканчиваются на «_init». Этот слой существует только для нежурналируемых таблиц (созданных с указанием UNLOGGED) и их индексов. Такие объекты ничем не отличаются от обычных, но действия с ними не записываются в журнал упреждающей записи. За счет этого работа с ними происходит быстрее, но в случае сбоя их содержимое невозможно восстановить. При восстановлении PostgreSQL просто удаляет все слои таких объектов и записывает слой инициализации на место основного слоя. В результате получается пустая таблица.

https://postgrespro.ru/docs/postgresql/13/storage-init

Слой vm (visibility map) — битовая карта видимости. Имена файлов этого слоя оканчиваются на «_vm». Слой существует только для таблиц; для индексов не поддерживается отдельная версионность.

Слой fsm (free space map) — *карта свободного пространства*. Имена файлов этого слоя оканчиваются на «_fsm». Этот слой существует и для таблиц, и для индексов.

Про две эти карты рассказывалось в модуле «Архитектура».

https://postgrespro.ru/docs/postgresql/13/storage-fsm

https://postgrespro.ru/docs/postgresql/13/storage-vm

Расположение файлов

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

```
CREATE DATABASE
```

```
=> \c data_lowlevel
```

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

Создадим таблицу и посмотрим на файлы, принадлежащие ей.

```
=> 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
```

Путь до основного файла относительно PGDATA можно получить функцией:

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

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

Поскольку таблица находится в табличном пространстве pg_default, путь начинается с base. Затем идет имя каталога для базы данных:

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

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

Затем — собственно имя файла. Его можно узнать следующим образом:

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

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

Тем и удобна функция pg_relation_filepath, что выдает готовый путь без необходимости выполнять несколько запросов к системному каталогу.

Посмотрим на файлы. Доступ к каталогу PGDATA имеет только пользователь ОС postgres, поэтому команда ls выдается от его имени:

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

```
-rw------- 1 postgres postgres 450560 апр 11 22:58 /var/lib/postgresql/13/main/base/16527/16530
-rw------- 1 postgres postgres  24576 апр 11 22:58 /var/lib/postgresql/13/main/base/16527/16530_fsm
-rw------- 1 postgres postgres   8192 апр 11 22:58 /var/lib/postgresql/13/main/base/16527/16530_vm
```

Мы видим три слоя: основной слой, карту свободного пространства (fsm) и карту видимости (vm).

Аналогично можно посмотреть и на файлы индекса:

```
=> \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 апр 11 22:58 /var/lib/postgresql/13/main/base/16527/16536
```

И на файлы последовательности, созданной для первичного ключа:

```
=> 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 апр 11 22:58 /var/lib/postgresql/13/main/base/16527/16528
```

Существует полезное расширение oid2name, входящее в стандартную поставку, с помощью которого можно легко связать объекты БД и файлы.

Можно посмотреть все базы данных:

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

All databases:
    Oid  Database Name  Tablespace
----------------------------------
  16527  data_lowlevel  pg_default
  13450       postgres  pg_default
  16385        student  pg_default
  13449      template0  pg_default
      1      template1  pg_default
```

Можно посмотреть все объекты в базе:

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

From database "data_lowlevel":
  Filenode  Table Name
----------------------
     16530           t
```

Или все табличные пространства в базе:

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

All tablespaces:
   Oid  Tablespace Name
-----------------------
  1663       pg_default
  1664        pg_global
```

Можно по имени таблицы узнать имя файла:

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

From database "data_lowlevel":
  Filenode  Table Name
----------------------
     16530           t
```

Или наоборот, по номеру файла узнать таблицу:

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

From database "data_lowlevel":
  Filenode  Table Name
```

```
----------------------
     16530           t
```

Размер слоев

Размер файлов, входящих в слой, можно, конечно, посмотреть в файловой системе, но существует специальная функция для получения размера каждого слоя в отдельности:

```
=> 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)
```

TOAST

Версия строки должна помещаться на одну страницу

- можно сжать часть полей
- можно вынести часть полей в отдельную toast-таблицу
- можно сжать и вынести одновременно

Toast-таблица

- находится в схеме pg_toast (pg_toast_temp_*N*)
- поддержана собственным индексом
- содержит фрагменты «длинных» значений размером меньше страницы
- читается только при обращении к «длинному» полю
- имеет собственную версионность
- используется прозрачно для приложения

Любая версия строки в PostgreSQL должна целиком помещаться на одну страницу. Для «длинных» версий строк применяется технология TOAST — The Oversized Attributes Storage Technique. Она подразумевает несколько стратегий работы с «длинными» полями. Значение поля может быть сжато так, чтобы версия строки поместилась на страницу. Значение может быть убрано из версии и перемещено в отдельную служебную таблицу. Могут применяться и оба подхода: какие-то значения будут сжаты, какие-то — перемещены, какие-то — сжаты и перемещены одновременно.

Для каждой основной таблицы при необходимости создается отдельная toast-таблица (и к ней специальный индекс). Такие таблицы и индексы располагаются в отдельной схеме pg_toast и поэтому обычно не видны (для временных таблиц используется схема pg_toast_temp_*N* аналогично обычной pg_temp_*N*).

Версии строк в toast-таблице тоже должны помещаться на одну страницу, поэтому длинные значения хранятся порезанными на фрагменты. Из этих фрагментов PostgreSQL прозрачно для приложения «склеивает» необходимое значение.

Toast-таблица используется только при обращении к длинному значению. Кроме того, для toast-таблицы поддерживается своя версионность: если обновление данных не затрагивает «длинное» значение, новая версия строки будет ссылаться на то же самое значение в toast-таблице — это экономит место.

https://postgrespro.ru/docs/postgresql/13/storage-toast

TOAST

В таблице t есть столбец типа numeric. Этот тип может работать с очень большими числами. Например, с такими:

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

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

При этом, если вставить такое значение в таблицу, размер файлов не изменится:

```
=> 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)
```

Поскольку версия строки не может поместить на одну страницу, она хранится в отдельной toast-таблице. Toast-таблица и индекс к ней создаются автоматически для каждой таблицы, в которой есть потенциально «длинный» тип данных, и используются по необходимости.

Имя и идентификатор такой таблицы можно найти следующим образом:

```
=> 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)
```

Вот и файлы toast-таблицы:

```
postgres$ ls -l /var/lib/postgresql/13/main/base/16527/16533*
```

```
-rw------- 1 postgres postgres 57344 апр 11 22:58 /var/lib/postgresql/13/main/base/16527/16533
-rw------- 1 postgres postgres 24576 апр 11 22:58 /var/lib/postgresql/13/main/base/16527/16533_fsm
```

Существуют несколько стратегий работы с длинными значениями. Название стратегии указывается в поле Storage:

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

```
                                             Table "public.t"
 Column |  Type   | Collation | Nullable |           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 не применяется (тип имеет фиксированную длину);
- extended — применяется как сжатие, так и отдельное хранение;
- external — сжатие не используется, только отдельное хранение;
- main — обрабатываются в последнюю очередь с приоритетом сжатия.

Стратегию можно изменить, если это необходимо. Например, если известно, что в столбце хранятся уже сжатые

данные, разумно поставить стратегию external.

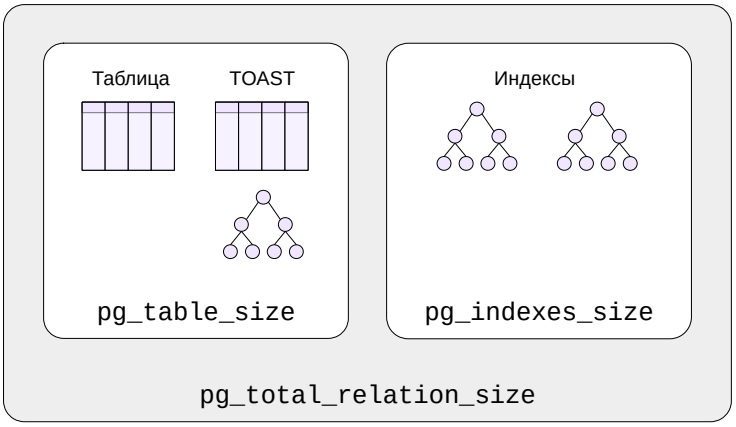
Просто для примера:

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

```
ALTER TABLE
```

Эта операция не меняет существующие данные в таблице, но определяет стратегию работы с новыми версиями строк.

Размер таблицы



Как уже говорилось, размер отдельного слоя можно получить функцией pg_relation_size. Чтобы не суммировать размеры отдельных слоев, есть несколько функций, показывающих размеры таблицы:

- pg_table_size показывает размер таблицы и ее toast-части (toast-таблицы и обслуживающего ее индекса), но без обычных индексов. Эту же функцию можно использовать для вычисления размера отдельного индекса: и таблицы, и индексы являются отношениями, и, несмотря на название, функция принимает любое отношение.
- pg_indexes_size суммирует размеры всех индексов таблицы, кроме индекса toast-таблицы.
- pg_total_relation_size показывает полный размер таблицы, вместе со всеми ее индексами.

Размер таблицы

Размер таблицы, включая toast-таблицу и обслуживающий ее индекс:

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

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

Общий размер всех индексов таблицы:

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

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

Для получения размера отдельного индекса можно воспользоваться функцией pg_table_size. Toast-части у индексов нет, поэтому функция покажет только размер всех слоев индекса (main, fsm).

Сейчас у таблицы только один индекс по первичному ключу, поэтому размер этого индекса совпадает со значением pg_indexes_size:

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

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

Общий размер таблицы, включающий TOAST и все индексы:

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

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

Итоги

Объект представлен несколькими слоями

Слой состоит из одного или нескольких файлов-сегментов

Каждый файл разбит на страницы

Для «длинных» версий строк используется TOAST

Практика

1. Создайте нежурналируемую таблицу в пользовательском табличном пространстве и убедитесь, что для таблицы существует слой init.

   Удалите созданное табличное пространство.

2. Создайте таблицу со столбцом типа text.
   Какая стратегия хранения применяется для этого столбца?

   Измените стратегию на external и вставьте в таблицу короткую и длинную строки.

   Проверьте, попали ли строки в toast-таблицу, выполнив прямой запрос к ней. Объясните, почему.

1. Нежурналируемая таблица

```
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)
```

Посмотрим на файлы таблицы.

Обратите внимание, что следующая команда ls выполняется от имени пользователя postgres. Чтобы повторить такую команду, удобно сначала открыть еще одно окно терминала и переключиться в нем на другого пользователя командой:

```
student$ sudo su postgres
```

И затем в этом же окне выполнить:

```
postgres$ ls -l /var/lib/postgresql/13/main/pg_tblspc/16705/PG_13_202007201/16706/16707*
```

```
-rw------- 1 postgres postgres 40960 апр 11 23:01 /var/lib/postgresql/13/main/pg_tblspc/16705/PG_13_202007201/16706/16707
-rw------- 1 postgres postgres 24576 апр 11 23:01 /var/lib/postgresql/13/main/pg_tblspc/16705/PG_13_202007201/16706/16707_fsm
-rw------- 1 postgres postgres     0 апр 11 23:01 /var/lib/postgresql/13/main/pg_tblspc/16705/PG_13_202007201/16706/16707_init
```

Удалим созданное табличное пространство:

```
=> DROP TABLE u;
```

```
DROP TABLE
```

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

```
DROP TABLESPACE
```

2. Таблица с текстовым столбцом

```
=> 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
```

По умолчанию для типа text используется стратегия extended.

Изменим стратегию на external:

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

```
ALTER TABLE
```

```
=> INSERT INTO t(s) VALUES ('Короткая строка.');
```

```
INSERT 0 1
```

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

```
INSERT 0 1
```

Проверим toast-таблицу:

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

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

Toast-таблица «спрятана», так как находится в схеме, которой нет в пути поиска. И это правильно, поскольку TOAST работает прозрачно для пользователя. Но заглянуть в таблицу все-таки можно:

```
=> 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)
```

Видно, что в TOAST-таблицу попала только длинная строка (два фрагмента, общий размер совпадает с длиной строки). Короткая строка не вынесена в TOAST просто потому, что в этом нет необходимости — версия строки и без этого помещается в страницу.

Практика+

1. Создайте базу данных.

   Сравните размер базы данных, возвращаемый функцией pg_database_size, с общим размеров всех таблиц в этой базе.

   Объясните полученный результат.

1. Список таблиц базы данных можно получить из таблицы pg_class системного каталога.

1. Сравнение размеров базы данных и таблиц в ней

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

```
CREATE DATABASE
```

```
=> \c data_lowlevel
```

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

Даже пустая база данных содержит таблицы, относящиеся к системного каталогу. Полный список отношений можно получить из таблицы pg_class. Из выборки надо исключить:

- таблицы, общие для всего кластера (они не относятся к текущей базе данных);
- индексы и toast-таблицы (они будут автоматически учтены при подсчета размера).

```
=> SELECT sum(pg_total_relation_size(oid))
FROM pg_class
WHERE NOT relisshared -- локальные объекты базы
AND relkind = 'r';  -- обычные таблицы
```

```
   sum
---------
 7962624
(1 row)
```

Размер базы данных оказывается несколько больше:

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

```
 pg_database_size
------------------
          8114735
(1 row)
```

Дело в том, что функция pg_database_size возвращает размер каталога файловой системы, а в этом каталоге находятся несколько служебных файлов.

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

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

Обратите внимание, что следующая команда ls выполняется от имени пользователя postgres. Чтобы повторить такую команду, удобно сначала открыть еще одно окно терминала и переключиться в нем на другого пользователя командой:

```
student$ sudo su postgres
```

И затем в этом же окне выполнить:

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

```
-rw------- 1 postgres postgres    512 апр 11 23:01 /var/lib/postgresql/13/main/base/16717/pg_filenode.map
-rw------- 1 postgres postgres 151596 апр 11 23:01 /var/lib/postgresql/13/main/base/16717/pg_internal.init
-rw------- 1 postgres postgres      3 апр 11 23:01 /var/lib/postgresql/13/main/base/16717/PG_VERSION
```

- pg_filenode.map — отображение oid некоторых таблиц в имена файлов;
- pg_internal.init — кеш системного каталога;
- PG_VERSION — версия PostgreSQL.

Из-за того, что одни функции работают на уровне объектов базы данных, а другие — на уровне файловой системы, бывает сложно точно сопоставить возвращаемые размеры. Это относится и к функции pg_tablespace_size.