

Организация данных Низкий уровень



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Алексей Береснев

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

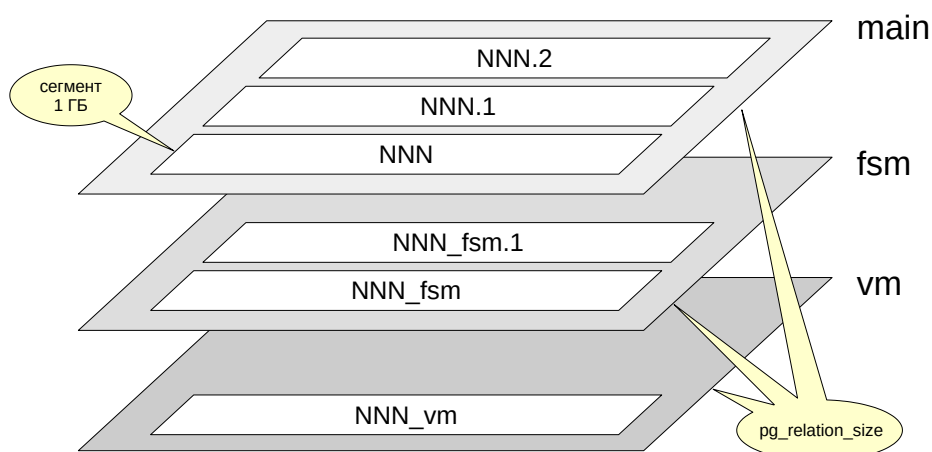
Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Файлы данных

Слои: данные, карты видимости и свободного пространства

Длинные версии строк и TOAST



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 достигает 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.

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/16/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 версияность.

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/16/storage-fsm>

<https://postgrespro.ru/docs/postgresql/16/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(1,10_000) AS id;
```

```
INSERT 0 10000
```

Чтобы сформировались дополнительные слои, выполним очистку:

```
=> VACUUM t;
```

```
VACUUM
```

Путь до основного файла относительно PGDATA можно получить функцией:

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

```
pg_relation_filepath  
-----  
base/16386/16388  
(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  
-----  
16386  
(1 row)
```

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

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

```
relfilenode  
-----  
16388  
(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/16/main/base/16386/16388*
```

```
-rw----- 1 postgres postgres 450560 июн 23 23:25  
/var/lib/postgresql/16/main/base/16386/16388  
-rw----- 1 postgres postgres 24576 июн 23 23:25  
/var/lib/postgresql/16/main/base/16386/16388_fsm  
-rw----- 1 postgres postgres 8192 июн 23 23:25  
/var/lib/postgresql/16/main/base/16386/16388_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/16386/16393
(1 row)
```

```
postgres$ ls -l /var/lib/postgresql/16/main/base/16386/16393*
```

```
-rw----- 1 postgres postgres 245760 июн 23 23:25
/var/lib/postgresql/16/main/base/16386/16393
```

И на файлы последовательности, созданной для первичного ключа:

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

```
pg_relation_filepath
-----
base/16386/16387
(1 row)
```

```
postgres$ ls -l /var/lib/postgresql/16/main/base/16386/16387*
```

```
-rw----- 1 postgres postgres 8192 июн 23 23:25
/var/lib/postgresql/16/main/base/16386/16387
```

Для индекса карта свободного пространства строится при наличии пустых страниц, а для последовательности существует только основной слой.

Временные таблицы хранятся так же, как и постоянные.

```
=> CREATE TEMP TABLE temp AS SELECT * FROM t;
```

```
SELECT 10000
```

```
=> VACUUM temp;
```

```
VACUUM
```

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

```
pg_relation_filepath
-----
base/16386/t4_16397
(1 row)
```

К имени файла добавляется префикс, соответствующий номеру схемы для временных объектов.

```
postgres$ ls -l /var/lib/postgresql/16/main/base/16386/t4_16397*
```

```
-rw----- 1 postgres postgres 450560 июн 23 23:25
/var/lib/postgresql/16/main/base/16386/t4_16397
-rw----- 1 postgres postgres 24576 июн 23 23:25
/var/lib/postgresql/16/main/base/16386/t4_16397_fsm
-rw----- 1 postgres postgres 8192 июн 23 23:25
/var/lib/postgresql/16/main/base/16386/t4_16397_vm
```

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

Можно посмотреть все базы данных:

```
student$ /usr/lib/postgresql/16/bin/oid2name
```

All databases:

Oid	Database Name	Tablespace
16386	data_lowlevel	pg_default
5	postgres	pg_default
16385	student	pg_default
4	template0	pg_default

```
1      template1  pg_default
```

Можно посмотреть все объекты в базе:

```
student$ /usr/lib/postgresql/16/bin/oid2name -d data_lowlevel
```

From database "data_lowlevel":

Filenode	Table Name
16388	t
16397	temp

Или все табличные пространства в базе:

```
student$ /usr/lib/postgresql/16/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/16/bin/oid2name -d data_lowlevel -t t
```

From database "data_lowlevel":

Filenode	Table Name
16388	t

Или наоборот, по номеру файла узнать таблицу:

```
student$ /usr/lib/postgresql/16/bin/oid2name -d data_lowlevel -f 16388
```

From database "data_lowlevel":

Filenode	Table Name
16388	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)

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 версиюность
- 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 версияность: 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/16/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
-----
 99890
(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 поместиться на одну страницу, значение атрибута `n` 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_16388 |          16391
(1 row)
```

Вот и файлы toast-таблицы:

```
postgres$ ls -l /var/lib/postgresql/16/main/base/16386/16391*
```

```
-rw----- 1 postgres postgres 57344 июн 23 23:25
/var/lib/postgresql/16/main/base/16386/16391
-rw----- 1 postgres postgres 24576 июн 23 23:25
/var/lib/postgresql/16/main/base/16386/16391_fsm
```

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

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

Column	Type	Collation	Nullable	Table "public.t"	Default	Storage
Compression	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` — такие поля обрабатываются в последнюю очередь и выносятся в `toast`-таблицу, только если сжатия недостаточно.

Стратегия назначается для каждого столбца при создании таблицы. Ее можно указать явно, а значение по умолчанию зависит от типа данных.

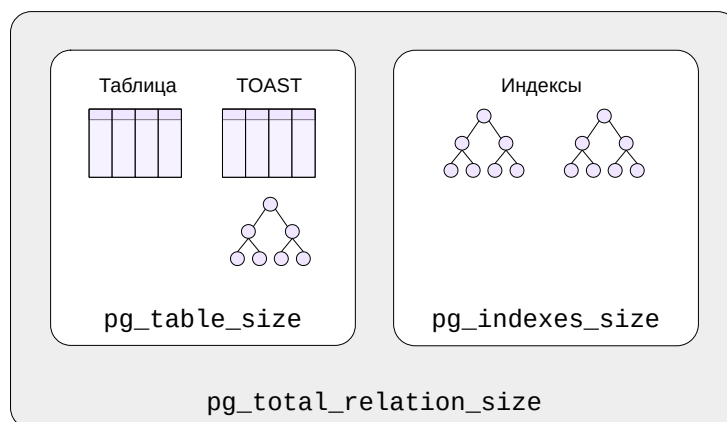
При необходимости стратегию можно впоследствии изменить. Например, если известно, что в столбце хранятся уже сжатые данные, разумно поставить стратегию `external`.

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.



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 есть только 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)
```

Объект представлен несколькими слоями

Слой состоит из одного или нескольких файлов-сегментов

Для «длинных» версий строк используется TOAST

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 -u postgres mkdir /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/16386/PG_16_202307071/16387/16388
(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 -i -u postgres
```

И затем в этом же окне выполнить:

```
postgres$ ls -l /var/lib/postgresql/16/main/pg_tblspc/16386/PG_16_202307071/16387/16388*
-rw----- 1 postgres postgres 40960 июн 23 23:33
/var/lib/postgresql/16/main/pg_tblspc/16386/PG_16_202307071/16387/16388
-rw----- 1 postgres postgres 24576 июн 23 23:33
/var/lib/postgresql/16/main/pg_tblspc/16386/PG_16_202307071/16387/16388_fsm
-rw----- 1 postgres postgres    0 июн 23 23:33
/var/lib/postgresql/16/main/pg_tblspc/16386/PG_16_202307071/16387/16388_init
```

Удалим созданное табличное пространство:

```
=> DROP TABLE u;
DROP TABLE
=> DROP TABLESPACE ts;
DROP TABLESPACE
student$ sudo -u postgres rm -rf /var/lib/postgresql/ts_dir
```

2. Таблица с текстовым столбцом

```
=> CREATE TABLE t(s text);
CREATE TABLE
=> \d+ t
```

```

              Table "public.t"
  Column | Type  | Collation | Nullable | Default | Storage  | Compression | Stats target |
-----+-----+-----+-----+-----+-----+-----+-----+
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_16391  
(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_16391  
ORDER BY chunk_id, chunk_seq;
```

```
 chunk_id | chunk_seq | length  
-----+-----+-----  
    16396 |         0 |    1996  
    16396 |         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.

1. Создайте базу данных.
Сравните размер базы данных, возвращаемый функцией `pg_database_size`, с общим размером всех таблиц в этой базе. Объясните полученный результат.
2. В TOAST поддерживаются два метода сжатия: `pglz` и `lz4`. Проверьте средствами SQL, был ли PostgreSQL скомпилирован с поддержкой этих методов.
3. Создайте текстовый файл размером больше 10 Мбайт. Загрузите его содержимое в таблицу с текстовым полем, сначала без сжатия, а затем используя каждый из алгоритмов. Сравните итоговый размер таблицы и время загрузки данных для трех вариантов.

1. Список таблиц базы данных можно получить из таблицы `pg_class` системного каталога.
2. С помощью представления `pg_config` можно узнать, какие параметры были установлены скрипту `configure` при сборке ПО сервера. Строка, содержащая список параметров, велика; выделить необходимые параметры можно функцией `string_to_table()`.
3. Чтобы получить текст для эксперимента, можно взять достаточно большой двоичный файл (например, исполняемый файл `postgres`) и преобразовать его в текст. Для преобразования можно использовать алгоритм Base32 (ключ `-w0` отменяет переносы строк):
`base32 -w0 < двоичный-файл > текстовый-файл`

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 системному 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
-----
7536640
(1 row)
```

Размер базы данных оказывается несколько больше:

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

pg_database_size
-----
7696867
(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
-----
16386
(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 -i -u postgres
```

И затем в этом же окне выполнить:

```
postgres$ ls -l /var/lib/postgresql/16/main/base/16386/[^0-9]*

-rw----- 1 postgres postgres 524 июн 23 23:33
/var/lib/postgresql/16/main/base/16386/pg_filenode.map
-rw----- 1 postgres postgres 159700 июн 23 23:33
/var/lib/postgresql/16/main/base/16386/pg_internal.init
-rw----- 1 postgres postgres 3 июн 23 23:33
/var/lib/postgresql/16/main/base/16386/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.

2. Поддержка методов сжатия TOAST

Представление pg_config показывает параметры, которые были переданы скрипту configure при сборке PostgreSQL.

```
=> SELECT * FROM (
  SELECT string_to_table(setting, ' ' ' ') AS setting
  FROM pg_config WHERE name = 'CONFIGURE'
)
WHERE setting ~ '(lz|zs)';

   setting
-----
--with-lz4
--with-zstd
(2 rows)
```

Какой метод сжатия TOAST используется по умолчанию?

```
=> \dconfig *toast*

List of configuration parameters
      Parameter      | Value
-----+-----
default_toast_compression | pglz
(1 row)
```

Какие методы можно применять?

```
=> SELECT setting, enumvals FROM pg_settings WHERE name = 'default_toast_compression';

 setting | enumvals
-----+-----
 pglz    | {pglz,lz4}
(1 row)
```

3. Сравнение методов сжатия

Сравним методы сжатия на примере текстовых данных.

Чтобы получить текст большого объема, возьмем исполняемый файл postgres и преобразуем его в текст с помощью алгоритма Base 32, который применяется в электронной почте.

```
student$ sudo cat /usr/lib/postgresql/16/bin/postgres | base32 -w0 > /tmp/gram.input
```

Получившийся текстовый файл имеет достаточный размер.

```
student$ ls -l --block-size=K /tmp/gram.input

-rw-rw-r-- 1 student student 16379K июн 23 23:33 /tmp/gram.input
```

Создадим таблицу для загрузки текстовых данных.

Для столбца txt установим стратегию хранения EXTERNAL, которая допускает отдельное хранение, но не сжатие.

```
=> CREATE TABLE t (
  txt text STORAGE EXTERNAL
);
```

```
CREATE TABLE
```

Загрузим данные из текстового файла.

```
=> \timing on

Timing is on.

=> COPY t FROM '/tmp/gram.input';
```

```
COPY 1
Time: 441,252 ms
```

```
=> \timing off

Timing is off.
```

Проверим размер таблицы, включая TOAST.

```
=> SELECT pg_table_size('t')/1024;

?column?
-----
    17040
(1 row)
```

Опустошим таблицу и зададим сжатие с помощью pglz.

```
=> TRUNCATE TABLE t;
```

```
TRUNCATE TABLE
```

```
=> ALTER TABLE t
ALTER COLUMN txt SET STORAGE EXTENDED,
ALTER COLUMN txt SET COMPRESSION pglz;
```

```
ALTER TABLE
```

Теперь используется стратегия EXTENDED, которая допускает как сжатие, так и отдельное хранение.

Снова загрузим данные.

```
=> \timing on
```

```
Timing is on.
```

```
=> COPY t FROM '/tmp/gram.input';
```

```
COPY 1
```

```
Time: 830,392 ms
```

```
=> \timing off
```

```
Timing is off.
```

```
=> SELECT pg_table_size('t')/1024;
```

```
?column?
-----
    10368
(1 row)
```

Размер таблицы значительно уменьшился, но при этом заметно выросло время загрузки.

Снова опустошим таблицу и зададим теперь сжатие с помощью lz4.

```
=> TRUNCATE TABLE t;
```

```
TRUNCATE TABLE
```

```
=> ALTER TABLE t ALTER COLUMN txt SET COMPRESSION lz4;
```

```
ALTER TABLE
```

Еще раз загрузим данные и сравним.

```
=> \timing on
```

```
Timing is on.
```

```
=> COPY t FROM '/tmp/gram.input';
```

```
COPY 1
```

```
Time: 293,715 ms
```

```
=> \timing off
```

```
Timing is off.
```

```
=> SELECT pg_table_size('t')/1024;
```

```
?column?
-----
    10712
(1 row)
```

Алгоритм lz4 слегка уступает pglz по степени сжатия, однако работает значительно быстрее.

Удалим текстовый файл.

```
student$ sudo rm -f /tmp/gram.input
```