Расширяемость

Типы для больших значений

Авторы: Егор Рогов, Павел Лузанов

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

Темы

Хранение в файловой системе

Технология TOAST

Тип bytea

Большие объекты

Варианты хранения

Файловая система

в базе данных только ссылки на файлы

Встроенные типы

	PostgreSQL	*стандарт SQL*
двоичные данные	bytea	blob
символьные данные	text	clob

Подсистема «больших объектов»

Для хранения и обработки больших объемов данных в стандарте SQL предусмотрены типы:

- clob (character large object) — для символьных данных;
- blob (binary large object) — для двоичных данных.

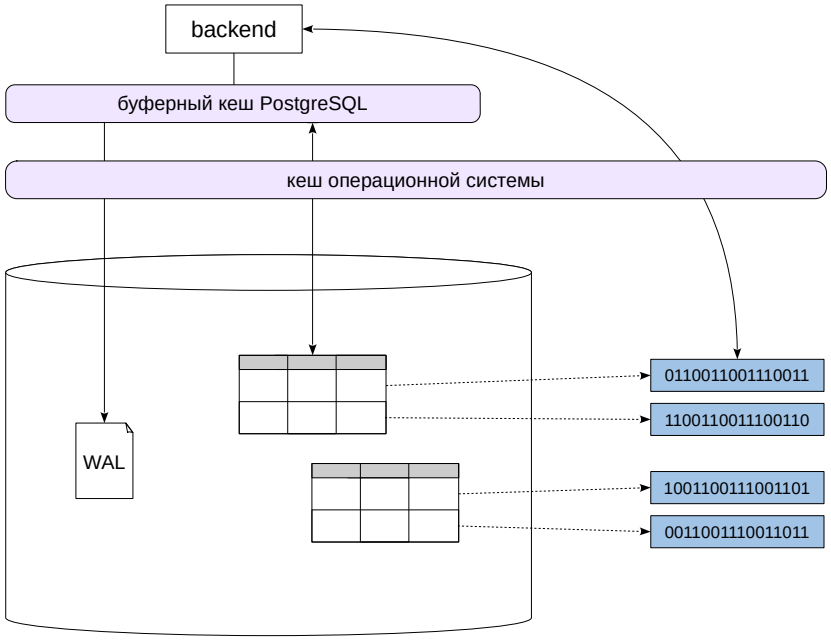
В PostgreSQL типы clob и blob отсутствуют. Вместо них для символьных данных можно использовать тип text, а для двоичных — bytea (byte array).

Работа с данными типа text не зависит от размера значения, поэтому в дальнейшем будем говорить только о двоичных данных. Например, об изображениях, видео, аудио и пр. Как правило такие данные представляют собой файлы определенных форматов.

В качестве альтернативы типу bytea, можно использовать «отчасти устаревшую» (цитата из документации) подсистему работы с большими объектами.

Но прежде чем размещать файлы внутри базы данных, следует рассмотреть возможность хранения их в файловой системе.

Файловая система



Если говорить о файлах, то один из вариантов — отказаться от размещения их в СУБД. В этом случае файлы располагаются непосредственно в файловой системе, а внутри базы данных поддерживаются только ссылки на них.

Файловая система

Преимущества

скорость чтения и записи файлов

не увеличивается размер базы данных

Недостатки

не используются возможности СУБД: транзакции, управление доступом

отсутствуют гарантии согласованности, долговечности данных

усложняется архитектура системы,
например, в части резервного копирования и восстановления

Такой подход может быть оправдан в первую очередь производительностью. Скорость чтения и записи файлов увеличивается за счет избежания затрат, характерных для СУБД:

- двойная запись на диск (в WAL и файлы данных);
- двойное кеширование в оперативной памяти (файловый кеш и буферный кеш СУБД);
- накладные расходы на хранение в базе данных и обработку: файлы для хранения нарезаются на короткие фрагменты, а при чтении их приходится обратно склеивать;
- возможное распухание таблиц и индексов приводит к дополнительному чтению устаревших фрагментов значения.

Кроме того, удается сократить расходы на обслуживание. Не требуется очистка и переиндексация таблиц с большими данными. Копирование файлов, в том числе инкрементальное, можно выполнять средствами ОС. А резервные копии только базы данных будут создаваться быстрее, меньшего размера и займут меньше времени на восстановление.

С другой стороны, становится невозможным использовать преимущества СУБД:

- транзакционная обработка: атомарность и согласованность записи, конкурентный доступ, восстановление после сбоя;
- управление доступом пользователей на чтение и запись файлов.

А усложнение архитектуры системы потребует пересмотра и доработки таких процедур, как резервное копирование и восстановление, использование реплик, обнаружение неиспользуемых файлов и пр.

TOAST

Хранение «длинных» атрибутов в отдельной таблице

- применяется для типов переменной длины: text, bytea, xml, json и др.
- «длинные» атрибуты нарезаются на фрагменты меньше страницы
- возможно сжатие
- размер значения до 1 Гбайта

TOAST-таблица

- читается только при обращении к «длинному» атрибуту
- возможна частичная декомпрессия значения при чтении
- собственная версионность
- работает 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. Точнее TOAST применяется к отдельным атрибутам, имеющим тип переменной длины, например, text и bytea, а также xml и json, которые будут рассмотрены позже в этом курсе. В любом случае размер одного значения (возможно сжатого) не должен превышать 1 Гбайта.

Для каждой основной таблицы при необходимости создается отдельная TOAST-таблица (и к ней специальный индекс). Версии строк в TOAST-таблице тоже должны помещаться на одну страницу, поэтому «длинные» значения хранятся порезанными на части, обычно около 2 Кбайт. Из этих частей PostgreSQL прозрачно для приложения «склеивает» необходимое значение.

TOAST-таблица используется только при обращении к «длинному» значению. Если требуется прочитать начальную часть сжатого значения (например функцией substr), то после считывания и склеивания полного значения будет распакована только первая часть, достаточная для выдачи результата. А в версии 13 начальные фрагменты будут считываться порциями и сразу распаковываться, позволяя избежать считывания «длинного» значения целиком. Но если требуется изменить несколько байт значения, то оно будет считано и записано полностью.

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.

Стратегии хранения

	основная таблица	сжатие данных	перенос в TOAST
Extended	да	да	да
External	да	—	да
Main	да	да	в последнюю очередь
Plain	да	—	—

Столбцы таблиц используют одну из четырех стратегий хранения значений. Вне зависимости от выбранной стратегии, значения небольшого размера будут оставаться в основной таблице.

Первые три стратегии относятся к TOAST:

- EXTENDED. Допускается сжатие и перенос в таблицу TOAST. Эта стратегия используется по умолчанию для типов, поддерживающих перенос в TOAST.

- EXTERNAL. Допускается только перенос в таблицу TOAST, но не сжатие. Если загружаемые данные уже сжаты, имеет смысл выбрать стратегию EXTERNAL для экономии ресурсов на малоэффективное повторное сжатие.

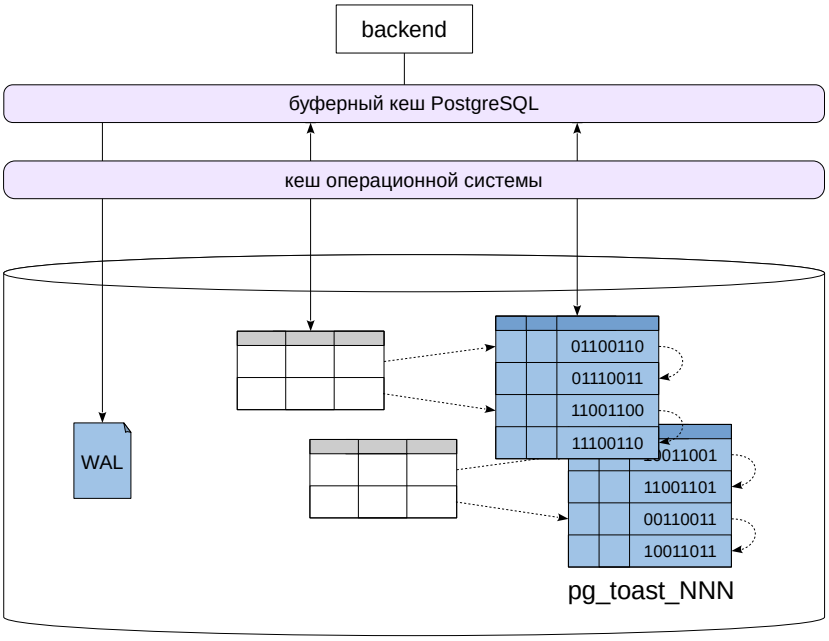
- MAIN. Допускается сжатие. Перенос в таблицу TOAST возможен как крайняя мера, если перенос других атрибутов (external и extended) не уменьшил строку так, чтобы она уместилась на странице.

- PLAIN. Хранение всегда в основной таблице. Используется для типов, чьи значения не могут быть большими.

Стратегии хранения у столбцов можно изменить только после создания таблицы.

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

Тип bytea



Хранить двоичные объекты в базе данных можно двумя способами.

Первый способ заключается в создании столбцов с типом bytea в таблицах.

Тип bytea использует технологию хранения TOAST. Поэтому значения большого размера размещаются не в самой таблице, а нарезаются на фрагменты и хранятся в связанной таблице TOAST.

Тип bytea

Преимущества

- транзакционность, управление доступом
- согласованность и долговечность больших данных
- интерфейс SQL

Недостатки

- увеличение размера базы данных
- меньшая скорость, чем при работе с файлами
- размер значения до 1 Гбайта
- значения считываются и записываются целиком

Главное преимущество хранения больших значений в базе данных — это использование возможностей СУБД. Транзакционный механизм гарантирует согласованность данных и защиту от сбоев.
А разграничивать доступ пользователей к значениям можно при помощи средств СУБД.

Для работы с типом bytea используется язык SQL, что делает эту работу удобной.

Двоичные типы данных, функции и операторы для работы с ними:

https://postgrespro.ru/docs/postgresql/12/datatype-binary

https://postgrespro.ru/docs/postgresql/12/functions-binarystring

Как ранее было рассмотрено, хранение внутри базы данных увеличивает размер базы данных и расходы на ее сопровождение. А также уменьшает скорость чтения и записи файлов из-за необходимости нарезать значение на фрагменты и склеивать их, и помещать в дополнительный кеш.

К недостатком использования типа bytea можно отнести ограничение TOAST на размер одного значения в 1 Гбайт.

Также работа с TOAST, как правило, осуществляется со значениями целиком. Если требуется прочитать или изменить небольшую часть значения, то придется читать и записывать в буферный кеш полное значение. Как было сказано ранее, ситуация частично улучшится в 13-й версии PostgreSQL.

Тип bytea и TOAST

```
=> CREATE DATABASE ext_lob;
```

```
CREATE DATABASE
```

```
=> \c ext_lob
```

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

По умолчанию двоичные данные выводятся в шестнадцатеричном формате.

```
=> SHOW bytea_output;
```

```
 bytea_output
--------------
 hex
(1 row)
```

Значения начинаются с '\x', далее каждый байт представлен двумя шестнадцатеричными цифрами:

```
=> SELECT 'Hello'::bytea;
```

```
    bytea
--------------
 \x48656c6c6f
(1 row)
```

Добавим нулевой символ к строке:

```
=> SELECT 'Hello'::bytea || '\x00'::bytea;
```

```
    ?column?
----------------
 \x48656c6c6f00
(1 row)
```

Шестнадцатеричный формат появился в версии 9.0. До этого был доступен только формат «спецпоследовательностей».

```
=> SET bytea_output = 'escape';
```

```
SET
```

В этом формате ASCII-символы отображаются как есть, а остальные представлены спецпоследовательностями:

```
=> SELECT 'Hello'::bytea || '\x00'::bytea;
```

```
 ?column?
-----------
 Hello\000
(1 row)
```

Параметр bytea_output определяет только формат вывода двоичных данных. Входные данные принимаются в любом из этих двух форматов.

```
=> RESET bytea_output;
```

```
RESET
```

Теперь создадим таблицу со столбцом типа bytea.

```
=> CREATE TABLE demo_bytea (filename text, data bytea);
```

```
CREATE TABLE
```

Таблица TOAST создается автоматически для хранения больших значений data и filename (значения типа text также могут быть большими). Найдем служебную таблицу запросом:

```
=> SELECT reltoastrelid::regclass AS toast_table
   FROM pg_class
   WHERE oid = 'demo_bytea'::regclass;
```

```
       toast_table
-------------------------
 pg_toast.pg_toast_18346
(1 row)
```

Служебные таблицы TOAST всегда располагаются в специальной схеме pg_toast чтобы не пересекаться с обычными объектами базы данных. Структура таблицы:

```
=> \d pg_toast.pg_toast_18346
```

```
TOAST table "pg_toast.pg_toast_18346"
   Column   |  Type
------------+---------
 chunk_id   | oid
 chunk_seq  | integer
 chunk_data | bytea
```

- chunk_id — идентификатор значения,
- chunk_seq — порядковый номер фрагмента значения,
- chunk_data — данные фрагмента.

Можно обратить внимание, что вывод команды \d для TOAST-таблиц отличается от обычных. Не показан первичный ключ по первым двум столбцам и соответствующий индекс:

```
=> \d pg_toast.pg_toast_18346_index
```

```
  Index "pg_toast.pg_toast_18346_index"
  Column   |  Type   | Key? | Definition
-----------+---------+------+------------
 chunk_id  | oid     | yes  | chunk_id
 chunk_seq | integer | yes  | chunk_seq
primary key, btree, for table "pg_toast.pg_toast_18346"
```

Доступ к значениям в TOAST всегда осуществляется по индексу. Это самый быстрый способ получить все фрагменты одного значения для склейки, но доступ ко всем значениям будет заведомо неэффективен.

Кроме того, чтение большого объема данных из TOAST-таблиц может приводить к вытеснению полезных данных из буферного кеша. Механизм буферного кольца, предотвращающий массовое вытеснение, для TOAST-таблиц не используется, так как задействуется только при полном последовательном сканировании таблицы, но не при индексном доступе.

Добавим в demo_bytea строку. В качестве двоичных данных возьмем текстовый файл bookstore2.sql, который используется для создания и первоначального наполнения базы данных приложения.

```
student$ ls -l /home/student/dev2/bookstore2.sql
```

```
-rw-rw-r-- 1 student student 1748732 июл 26 10:30 /home/student/dev2/bookstore2.sql
```

Для считывания файла воспользуемся встроенной функцией pg_read_binary_file.

```
=> INSERT INTO demo_bytea(filename, data) VALUES (
    'bookstore2.sql',
    pg_read_binary_file('/home/student/dev2/bookstore2.sql')
);
```

```
INSERT 0 1
```

Использование TOAST прозрачно для приложения. Нам не нужно обращаться к служебной таблице в запросах. Вот первые 16 байт загруженного файла в двоичном виде:

```
=> SELECT substring(data,1,16) FROM demo_bytea;
```

```
             substring
-----------------------------------
 \x2d2d20d0add182d0bed18220d181d0ba
(1 row)
```

Общий размер загруженного значения соответствует размеру файла:

```
=> SELECT length(data) FROM demo_bytea;
```

```
 length
---------
 1748732
(1 row)
```

Однако размер таблицы demo_bytea составляет всего одну страницу, данных загруженного файла в ней нет:

```
=> SELECT pg_relation_size('demo_bytea');
```

```
 pg_relation_size
------------------
             8192
(1 row)
```

Значение столбца data попало в TOAST-таблицу. Можно убедиться, что в служебной таблице появилось одно значение:

```
=> SELECT count(distinct(chunk_id)) FROM pg_toast.pg_toast_18346;
```

```
 count
-------
     1
(1 row)
```

Сколько места требуется для хранения TOAST-таблицы, если сравнивать с размером файла: меньше, больше или ровно столько же?

```
=> SELECT pg_relation_size('pg_toast.pg_toast_18346');
```

```
 pg_relation_size
------------------
           778240
(1 row)
```

Почему потребовалось меньше места?

```
=> \d+ demo_bytea
```

```
                                Table "public.demo_bytea"
  Column  | Type  | Collation | Nullable | Default | Storage  | Stats target | Description
----------+-------+-----------+----------+---------+----------+--------------+-------------
 filename | text  |           |          |         | extended |              |
 data     | bytea |           |          |         | extended |              |
Access method: heap
```

Стратегия хранения extended предполагает сжатие данных при помещении в TOAST. А текстовые данные хорошо сжимаются.

Сжатие можно запретить, выбрав стратегию external:

```
=> ALTER TABLE demo_bytea ALTER COLUMN data SET STORAGE external;
```

```
ALTER TABLE
```

Это изменение будет действовать только для новых строк, поэтому очистим таблицу и загрузим файл заново.

```
=> TRUNCATE demo_bytea;
```

```
TRUNCATE TABLE
```

```
=> INSERT INTO demo_bytea(filename, data) VALUES (
    'bookstore2.sql',
    pg_read_binary_file('/home/student/dev2/bookstore2.sql')
);
```

```
INSERT 0 1
```

Сколько теперь потребуется места для хранения TOAST-таблицы, если сравнивать с размером файла: меньше, больше или ровно столько же?

```
=> SELECT pg_relation_size('pg_toast.pg_toast_18346');
```

```
 pg_relation_size
------------------
          1802240
(1 row)
```

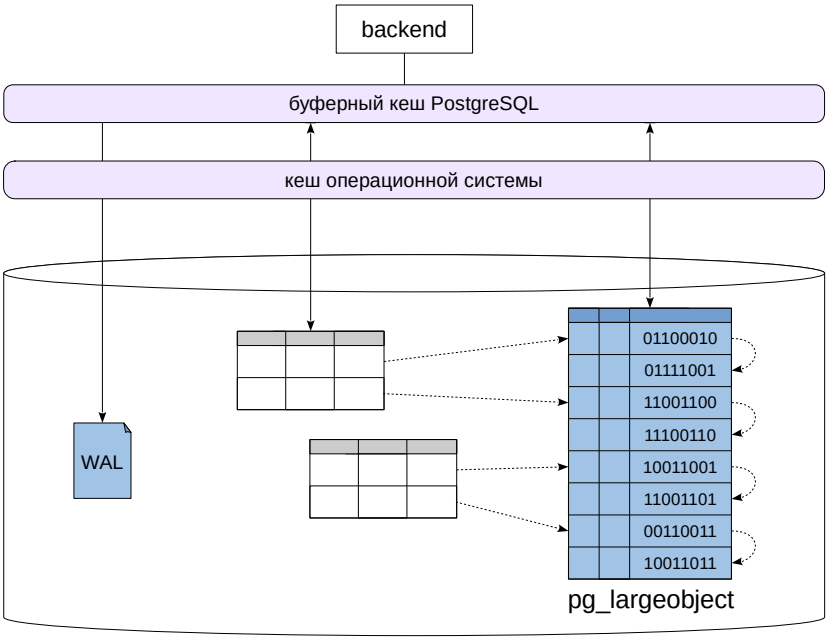
Теперь для хранения используется немного больше места. Почему?

```
=> SELECT chunk_id, chunk_seq, substring(chunk_data,1,16),
    length(chunk_data)
FROM pg_toast.pg_toast_18346
ORDER BY 1,2 LIMIT 3;
```

```
 chunk_id | chunk_seq |             substring              | length
----------+-----------+------------------------------------+--------
    18356 |         0 | \x2d2d20d0add182d0bed18220d181d0ba |   1996
    18356 |         1 | \x746578742c0a20202020636f7665725f |   1996
    18356 |         2 | \x4553207075626c69632e70726f677261 |   1996
(3 rows)
```

На размер повлияли накладные расходы на хранение фрагментов в отдельных строках с дополнительными столбцами и служебной информацией.

Большие объекты



Другой вариант хранения двоичных данных большого размера внутри базы данных — использование подсистемы «больших объектов», появившейся задолго до TOAST.

Все большие объекты сохраняются в отдельной таблице системного каталога pg_largeobject. Для ссылок на большие объекты в таблицах предоставляются идентификаторы типа oid. Доступ к объектам осуществляется по специальному интерфейсу, похожему на интерфейс работы с файлами.

https://postgrespro.ru/docs/postgresql/12/largeobjects

Большие объекты

Преимущества

- транзакционность, управление доступом
- согласованность и долговечность больших данных
- размер значения до 4 Тбайт
- потоковое чтение и запись фрагментов значения

Недостатки

- увеличение размера базы данных
- меньшая скорость, чем при работе с файлами
- специальный интерфейс
- ограничение на количество больших объектов (2^{32})
- риск появления осиротевших объектов
- не поддерживается логической репликацией

Большие объекты наследуют те же преимущества и недостатки, что и тип bytea, относительно хранения в файловой системе.

По сравнению с типом bytea у больших объектов есть два преимущества. Во-первых, размер одного объекта может достигать 4 Тбайта. А во-вторых, интерфейс для работы с большими объектами предоставляет возможности чтения и записи произвольных фрагментов, а не значения целиком, как обычно происходит с TOAST.

С другой стороны, использование специального интерфейса делает работу с большими объектами не такой удобной, как с чистым SQL. К тому же драйвер PostgreSQL должен поддерживать этот интерфейс.

Идентификатор большого объекта хранится в 4-байтовом типе oid, что ограничивает количество объектов в системе числом 4 294 967 296. Таблицы TOAST тоже используют oid, но ограничение на количество значений действует на отдельные таблицы, а не всю систему.

Кроме того, хранение всех больших объектов в отдельной таблице увеличивает риск появления осиротевших объектов, на которые не осталось ссылок в обычных таблицах. Для поиска и удаления осиротевших объектов можно использовать утилиту vacuumlo, а для предотвращения их появления – расширение lo.

https://postgrespro.ru/docs/postgresql/12/lo

https://postgrespro.ru/docs/postgresql/12/vacuumlo

В 12-й версии логическая репликация не поддерживает изменения в таблицах системного каталога, в том числе в pg_largeobject.

Использование large objects

Большие объекты хранятся в таблице системного каталога pg_largeobject, структура которой похожа на структуру TOAST-таблицы.

```
=> \d pg_largeobject
```

```
         Table "pg_catalog.pg_largeobject"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 loid   | oid     |           | not null |
 pageno | integer |           | not null |
 data   | bytea   |           | not null |
Indexes:
    "pg_largeobject_loid_pn_index" UNIQUE, btree (loid, pageno)
```

Создадим таблицу для хранения ссылок на большие объекты.

```
=> CREATE TABLE demo_largeobject (filename text, link oid);
```

```
CREATE TABLE
```

Для работы с большими объектами будем использовать интерфейсные функции SQL.

```
=> INSERT INTO demo_largeobject VALUES (
    'bookstore2.sql',
    lo_import('/home/student/dev2/bookstore2.sql')
);
```

```
INSERT 0 1
```

Функция lo_import загружает файл с сервера в pg_largeobject и возвращает указатель на него (OID).

Функция lo_get считывает указанную часть значения:

```
=> SELECT filename, link, lo_get(link,1,16) FROM demo_largeobject;
```

```
    filename    | link  |               lo_get
----------------+-------+-----------------------------------
 bookstore2.sql | 18363 | \x2d20d0add182d0bed18220d181d0bad1
(1 row)
```

Что будет, если удалить строку из таблицы demo_largeobject?

```
=> DELETE FROM demo_largeobject;
```

```
DELETE 1
```

Строка удалится, а большой объект станет «потерянным»:

```
=> \lo_list
```

```
         Large objects
  ID   |  Owner  | Description
-------+---------+-------------
 18363 | student |
(1 row)
```

Дополнительная утилита vacuumlo, поставляемая с сервером, позволяет найти большие объекты, на которые не осталось ссылок, и удалить их:

```
student$ vacuumlo --verbose ext_lob
```

```
Connected to database "ext_lob"
Checking link in public.demo_largeobject
Removing lo  18363   Successfully removed 1 large objects from database "ext_lob".
```

```
=> \lo_list
```

```
      Large objects
 ID | Owner | Description
----+-------+-------------
(0 rows)
```

Для предотвращения потери ссылок также можно воспользоваться расширением lo.

```
=> CREATE EXTENSION lo;
```

```
CREATE EXTENSION
```

Расширение создает тип данных lo (обертка над oid) и функцию lo_manage для использования в триггерных функциях.

```
=> CREATE TABLE demo_lo (filename text, link lo);

CREATE TABLE

=> CREATE TRIGGER t_link
BEFORE UPDATE OR DELETE ON demo_lo
FOR EACH ROW
EXECUTE FUNCTION lo_manage(link);

CREATE TRIGGER
```

Загрузим большой объект и поместим ссылку на него в таблицу.

```
=> INSERT INTO demo_lo VALUES (
    'bookstore2.sql',
    lo_import('/home/student/dev2/bookstore2.sql')
);

INSERT 0 1
```

Убедимся, что всё на месте и удалим.

```
=> SELECT filename, lo_get(link,1,16) FROM demo_lo;

    filename    |               lo_get
----------------+------------------------------------
 bookstore2.sql | \x2d20d0add182d0bed18220d181d0bad1
(1 row)


=> DELETE FROM demo_lo;

DELETE 1

=> \lo_list

      Large objects
 ID | Owner | Description
----+-------+-------------
(0 rows)
```

Табличный триггер удалил связанный большой объект.

Итоги

Для работы с двоичными данными предоставлены

- тип bytea, использующий технологию TOAST
- подсистема «больших объектов»

Использование файловой системы позволяет добиться высокой скорости за счет отказа от возможностей СУБД

Практика

1. Добавьте в таблицу books поле cover с типом bytea. Загрузите в поле cover обложки книг из файлов формата jpeg. Файлы находятся в каталоге /home/student/covers.
2. Создайте функцию webapi.get_image, возвращающую обложку книги по переданному идентификатору.
3. Сравните время, за которое выполняется запрос ко всем столбцам таблицы books (SELECT *) и к столбцам без cover.

1. Для чтения файлов можно использовать функцию pg_read_binary_file.

2. Функция get_image уже создана в базе данных, но ничего не возвращает. Заголовок функции:

```
CREATE FUNCTION webapi.get_image(book_id integer) RETURNS bytea
```

```
student$ psql bookstore2
```

1. Перекладывание сканов обложек в базу данных

Сканы обложек находятся в каталоге covers:

```
student$ ls ~/covers/ | head -n 10
```

```
abelson_prog.jpg
adelson_chess.jpg
aho_compilers.jpg
aho_structalgo.jpg
ben-ari_pl.jpg
bogdesko_calligraphy.jpg
boswell_readable.jpg
bringhurst_style.jpg
brooks_myth.jpg
camp_no.jpg
```

Названия файлов хранятся в таблице books в столбце cover_filename. Расширим таблицу столбцом для хранения картинок:

```
=> ALTER TABLE books ADD cover bytea;
```

```
ALTER TABLE
```

Используемую по умолчанию компрессию данных стоит отключить: дополнительно сжать файлы JPEG не удастся, а ресурсы на упаковку/распаковку можно сэкономить.

Установить нужную стратегию хранения можно только отдельной командой после добавления столбца:

```
=> ALTER TABLE books ALTER COLUMN cover SET STORAGE external;
```

```
ALTER TABLE
```

Переложим файлы в этот столбец:

```
=> UPDATE books b
SET cover = pg_read_binary_file(
    '/home/student/covers/' || b.cover_filename
);
```

```
UPDATE 96
```

2. Функция, возвращающая скан обложки

```
=> CREATE OR REPLACE FUNCTION webapi.get_image(book_id bigint)
RETURNS bytea
AS $$
    SELECT b.cover
    FROM books b
    WHERE b.book_id = get_image.book_id;
$$ LANGUAGE sql STABLE SECURITY DEFINER;
```

```
CREATE FUNCTION
```

3. Время выполнения SELECT *

Сравните время самостоятельно; здесь мы не приводим результат выполнения запросов из-за большого размера.

Вывод должен быть очевиден: в промышленном коде следует обращаться только к тем столбцам, которые действительно необходимы.

Практика

1. Создайте таблицу с полем типа bytea.
   Загрузите в таблицу любой файл, этот же файл загрузите как большой объект.
   Выполните контрольную точку, чтобы в кеше не осталось грязных буферов.
2. Добавьте один произвольный байт к значению в столбце типа bytea таблицы.
   Сколько грязных буферов, относящихся к TOAST-таблице, появилось в буферном кеше?
3. Добавьте один произвольный байт к значению большого объекта.
   Сколько грязных буферов, относящихся к таблице pg_largeobject, появилось в буферном кеше?

2 и 3. Чтобы получить количество грязных буферов для таблицы, установите расширение pg_buffercache и воспользуйтесь запросом:

```
SELECT count(*)
FROM pg_buffercache b
WHERE b.relfilenode = pg_relation_filenode('table_name'::regclass)
AND isdirty;
```

3. Можно воспользоваться функцией lo_put, записывающей данные по заданному смещению.

1. Подготовка

```
=> CREATE DATABASE ext_lob;
```

```
CREATE DATABASE
```

```
=> \c ext_lob
```

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

Расширение для просмотра буферного кеша:

```
=> CREATE EXTENSION pg_buffercache;
```

```
CREATE EXTENSION
```

Создадим таблицу со столбцом типа bytea:

```
=> CREATE TABLE demo_bytea (filename text, data bytea);
```

```
CREATE TABLE
```

Загрузим файл с одной из обложек книг в таблицу:

```
=> INSERT INTO demo_bytea VALUES (
    'novikov_dbtech.jpg',
    pg_read_binary_file('/home/student/covers/novikov_dbtech.jpg')
);
```

```
INSERT 0 1
```

Этот же файл загрузим как большой объект и запомним его oid:

```
=> SELECT lo_import('/home/student/covers/novikov_dbtech.jpg') AS "oid";
```

```
  oid
--------
 340197
(1 row)
```

Для чистоты эксперимента сбросим все грязные буферы на диск:

```
=> CHECKPOINT;
```

```
CHECKPOINT
```

2. Изменение bytea

Добавим к значению нулевой байт:

```
=> UPDATE demo_bytea SET data = data || '\x00'::bytea;
```

```
UPDATE 1
```

Смотрим на грязные буферы, относящиеся к основному слою данных таблицы TOAST:

```
=> SELECT reltoastrelid::regclass AS toast_table
FROM pg_class
WHERE oid = 'demo_bytea'::regclass;
```

```
       toast_table
--------------------------
 pg_toast.pg_toast_340190
(1 row)
```

```
=> SELECT count(*)
FROM pg_buffercache b
WHERE b.relfilenode = pg_relation_filenode('pg_toast.pg_toast_340190')
AND b.relforknumber = 0 /* 0: main fork, 1: fsm, 2: vm */
AND isdirty;
```

```
 count
-------
   294
(1 row)
```

Несмотря на то, что был изменен всего один байт, все страницы TOAST-таблицы оказались грязными.

```
=> SELECT pg_relation_size('pg_toast.pg_toast_340190', 'main') / 8192
    AS "Table, pages";
```

```
 Table, pages
--------------
          294
(1 row)
```

3. Изменение большого объекта

Добавим в конец объекта нулевой байт:

```
=> SELECT lo_put(340197, 16777216, '\x00'::bytea);
```

```
 lo_put
--------

(1 row)
```

Посчитаем грязные буферы таблицы pg_largeobject:

```
=> SELECT count(*)
FROM pg_buffercache b
WHERE b.relfilenode = pg_relation_filenode('pg_largeobject')
AND isdirty;
```

```
 count
-------
     1
(1 row)
```

Изменена всего одна страница таблицы pg_largeobject.