

Расширяемость Пользовательские типы данных



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

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.

Составные типы (краткое повторение)

Типы перечислений

Диапазонные типы и мультидиапазоны

Базовые типы

Домены

Приведение типов, операторы

Богатый набор существующих типов

числа, строки, даты, слабоструктурированные, геометрия, ...
массивы

Средства для создания новых типов

простые: на основе уже имеющихся типов
сложные: новый тип «с нуля»

PostgreSQL имеет множество встроенных типов данных (и поддержку массивов элементов любых типов данных).

<https://postgrespro.ru/docs/postgresql/16/datatype>

Вместе с тем имеются и средства для создания своих собственных типов данных.

В простом случае новые типы создаются по одной из предопределенных схем из других, уже имеющихся, типов. Это составные типы, перечисления, диапазоны, домены.

Если же необходимый тип данных не может быть выражен с помощью этих простых средств, можно создать новый тип «с нуля», детально описав его поведение на низком уровне.

Набор именованных атрибутов (табличная строка)

Создание нового типа

вручную: CREATE TYPE AS

автоматически при создании отношения

Составные типы представляют собой набор именованных атрибутов каких-либо других типов данных. В разных языках программирования такие типы могут называться структурами или записями.

По сути, составной тип описывает табличную строку и поэтому создается автоматически при создании любой таблицы или представления.

Новый составной тип можно создать и вручную.

Составные типы определены в стандарте SQL. Они подробно рассматриваются в курсе DEV1, поэтому здесь мы не будем на них останавливаться.

<https://postgrespro.ru/docs/postgresql/16/rowtypes>

<https://postgrespro.ru/docs/postgresql/16/functions-comparisons#COMPOSITE-TYPE-COMPARISON>

Упорядоченный набор значений

значения выглядят как текстовые строки
хранятся как 4-байтовые числа (oid)

Создание нового типа

только вручную: CREATE TYPE AS ENUM

Тип перечисления создается простым перечислением всех допустимых значений.

Значения задаются и используются в виде текстовых строк, которые хранятся в системном каталоге как упорядоченный набор значений. При использовании перечисления, например, в качестве типа столбца таблицы, каждое его значение будет представлено в виде 4-байтного целого числа (фактически, тип oid). Это позволяет использовать длинные (до 63 символов) описательные значения, не экономя на месте в таблице.

Стандартных типов перечисления в PostgreSQL нет.

<https://postgrespro.ru/docs/postgresql/16/datatype-enum>

<https://postgrespro.ru/docs/postgresql/16/functions-enum>

Типы перечислений

```
=> CREATE DATABASE ext_datatypes;
```

```
CREATE DATABASE
```

```
=> \c ext_datatypes
```

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

Создадим новые типы перечислений для старшинства карт:

```
=> CREATE TYPE ranks AS ENUM (  
    '6', '7', '8', '9', '10', 'Queen', 'King', 'Ace'  
);
```

```
CREATE TYPE
```

И для мастей:

```
=> CREATE TYPE suits AS ENUM (  
    'clubs', 'diamonds', 'hearts', 'spades'  
);
```

```
CREATE TYPE
```

Напомним, что тип данных — отдельный объект базы данных, который хранится в системном каталоге:

```
=> \dT public.*
```

```
      List of data types  
Schema | Name  | Description  
-----+-----+-----  
public | ranks |  
public | suits |  
(2 rows)
```

Чтобы вспомнить и составные типы, определим еще тип для карты:

```
=> CREATE TYPE cards AS (  
    rank ranks,  
    suit suits  
);
```

```
CREATE TYPE
```

```
=> DO $$  
DECLARE  
    card cards;  
BEGIN  
    card := ('Ace', 'spades');  
    RAISE NOTICE '%', card;  
    RAISE NOTICE 'Старшинство: % ... %',  
        enum_first(card.rank), enum_last(card.rank);  
    RAISE NOTICE 'Масти: % ... %',  
        enum_first(card.suit), enum_last(card.suit);  
END;  
$$;
```

```
NOTICE: (Ace,spades)  
NOTICE: Старшинство: 6 ... Ace  
NOTICE: Масти: clubs ... spades  
DO
```

Обратите внимание, что названия значений перечислимого типа регистрозависимы.

Все значения в перечислении упорядочены, и этим можно пользоваться:

```
=> SELECT '6'::ranks < 'Ace'::ranks;
```

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

Однако сами типы перечислений считаются уникальными и поэтому значения разных перечислений нельзя сравнивать, даже если они определены одинаково:

```
=> CREATE TYPE suits_bis AS ENUM (
    'clubs', 'diamonds', 'hearts', 'spades'
);
```

CREATE TYPE

```
=> SELECT 'hearts'::suits = 'hearts'::suits_bis;
```

```
ERROR: operator does not exist: suits = suits_bis
LINE 1: SELECT 'hearts'::suits = 'hearts'::suits_bis;
          ^
```

HINT: No operator matches the given name and argument types. You might need to add explicit type casts.

При необходимости выполнить подобное сравнение придется либо реализовать собственный оператор, либо явно преобразовать типы в запросе, например так:

```
=> SELECT 'hearts'::suits::text = 'hearts'::suits_bis::text;

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

Можно получить список значений в виде массива (функция `enum_range` смотрит только на тип параметра; само значение может быть любым):

```
=> DO $$
DECLARE
    suit suits;
    rank ranks;
    deck cards[];
BEGIN
    RAISE NOTICE '%', enum_range(NULL::ranks);
    RAISE NOTICE '%', enum_range(NULL::suits);
    FOREACH rank IN ARRAY enum_range(NULL::ranks) LOOP
        FOREACH suit IN ARRAY enum_range(NULL::suits) LOOP
            deck := deck || (rank,suit)::cards;
        END LOOP;
    END LOOP;
END;
$$;

NOTICE:  {6,7,8,9,10,Queen,King,Ace}
NOTICE:  {clubs,diamonds,hearts,spades}
DO
```

Кажется, мы забыли валета. Не беда, его можно добавить к перечислению:

```
=> ALTER TYPE ranks ADD VALUE 'Jack' BEFORE 'Queen';
```

ALTER TYPE

А вот удалить значение из типа невозможно. Единственный вариант — удалить тип и создать его заново.

Если мы заглянем в системный каталог, то увидим, что упорядоченность значений перечисления обеспечивает столбец типа `real`. Обратите внимание на его дробное значение для добавленного нами валета:

```
=> SELECT enumsortorder, enumlabel FROM pg_enum WHERE enumtypeid = 'ranks'::regtype ORDER BY enumsortorder;
```

enumsortorder	enumlabel
1	6
2	7
3	8
4	9
5	10
5.5	Jack
6	Queen
7	King
8	Ace

(9 rows)

Непрерывный отрезок значений от начальной до конечной точки

открытый ($>$, $<$), закрытый (\leq , \geq), неограниченный

Создание нового типа

`CREATE TYPE AS RANGE`

уже есть для `integer`, `bigint`, `numeric`, `timestamp`, `date`

Операции

сравнение

проверка на включение диапазона или элемента, на пересечение

объединение, пересечение, вычитание

и др.

Диапазонный тип представляет непрерывный отрезок значений скалярного типа. Концы отрезка могут быть открытыми (граничное значение не включается) или закрытыми (наоборот, включается), или отрезок может быть бесконечным, то есть не иметь одной или обеих границ.

Для диапазонов определено довольно много удобных операций, в том числе:

- проверки на включение одного диапазона в другой, на принадлежность значения диапазона, на пересечение двух диапазонов;
- обычные операции над множествами: объединение, пересечение, вычитание.

Диапазоны предопределены для различных числовых типов и дат (например, `int4range` для `integer` и др.). Можно создать и свой собственный диапазонный тип.

<https://postgrespro.ru/docs/postgresql/16/rangetypes>

<https://postgrespro.ru/docs/postgresql/16/functions-range>

Множество диапазонов

не должны пересекаться

Создание нового типа

автоматически при создании диапазона

Операции

сравнение

проверка на включение мультидиапазона, диапазона или элемента

проверка на пересечение

объединение, пересечение, вычитание

и др.

Для каждого диапазонного типа определен соответствующий мультидиапазонный тип. Такой тип по сути представляет собой массив отдельных диапазонов. Диапазоны внутри мультидиапазонного типа не должны пересекаться между собой, но между диапазонами могут быть пропуски. Большинство диапазонных операторов работают и с мультидиапазонами. Кроме того, есть функции для работы именно с мультидиапазонными типами.

Мультидиапазонный тип определяется автоматически при определении диапазонного типа. По умолчанию имя мультидиапазонного типа генерируется PostgreSQL, но можно задать его и явно.

Диапазонные типы

Начнем с нескольких примеров работы с уже имеющимися диапазонными типами.

Диапазон целых чисел от 1 до 9, обе границы включаются (об этом говорят квадратные скобки). Что будет выведено?

```
=> SELECT '[1,9]':::int4range;
```

```
int4range
-----
[1,10)
(1 row)
```

Результат выводится в каноническом виде, в котором левая граница включается, а правая — нет (круглая скобка).

Конечно, значение правой границы при этом увеличивается на единицу. Канонический вид определен не для всех типов.

Диапазон может быть и открытым. Проверим, принадлежит ли число диапазону:

```
=> SELECT -100 <@ '[1,)'::int4range, 100 <@ '[1,)'::int4range;
```

```
?column? | ?column?
-----+-----
f         | t
(1 row)
```

Точно так же можно проверить включение одного диапазона в другой:

```
=> SELECT '[2,3]':::int4range <@ '[1,4]':::int4range;
```

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

Удобны функции, позволяющие работать с диапазонами как с множествами. Например, можно легко найти пересечение (общие точки) двух диапазонов:

```
=> SELECT '[1,3]':::int4range * '[2,4]':::int4range;
```

```
?column?
-----
[2,4)
(1 row)
```

При создании нового диапазонного типа указываются:

- базовый тип, значения которого должны быть сортируемыми;
- функция разности двух значений, которая должна возвращать результат типа double precision.

Например, в PostgreSQL нет типа для диапазонов времени (без даты). Чтобы определить его, нам потребуется функция, возвращающая разность между двумя временами. Обычная разность возвращает значение типа interval:

```
=> SELECT '10:00':::time - '9:59':::time;
```

```
?column?
-----
00:01:00
(1 row)
```

Но мы можем воспользоваться функцией extract epoch, которая вернет длину диапазона в секундах:

```
=> CREATE FUNCTION time_diff(a time, b time) RETURNS double precision
LANGUAGE sql STRICT IMMUTABLE
RETURN extract(epoch FROM (a - b));
```

```
CREATE FUNCTION
```

```
=> SELECT time_diff('10:00', '9:59');
```

```
time_diff
-----
60
(1 row)
```

Теперь можно определить тип диапазона, а вместе с ним — и мультидиапазона:

```
=> CREATE TYPE timerange AS RANGE (  
    subtype = time,  
    subtype_diff = time_diff,  
    multirange_type_name = timemultirange  
);
```

CREATE TYPE

Последний параметр — необязательный, в его отсутствие соответствующий мультидиапазонный тип будет создан автоматически.

Новый тип можно использовать аналогично уже существующим. Например, найдем объединение (здесь показаны два способа записи констант диапазонного типа):

```
=> SELECT timerange('9:00', '11:00', '[]') + '[10:00,12:00]':timerange;  
  
?column?  
-----  
[09:00:00,12:00:00]  
(1 row)
```

Заметьте, что диапазон не приводится к каноническому виду. Чтобы это работало, необходимо реализовать соответствующую функцию и указать ее в определении типа. Мы не будем этого делать.

Новый мультидиапазонный тип позволяет представить несколько диапазонов как одно значение:

```
=> SELECT timemultirange(  
    timerange('09:00', '13:00', '[]'),  
    timerange('08:00', '10:00', '[]'),  
    timerange('14:00', '18:00', '[]')  
) AS working_hours;  
  
working_hours  
-----  
{[08:00:00,13:00:00],[14:00:00,18:00:00]}  
(1 row)
```

Обратите внимание, что пересекающиеся диапазоны были объединены.

Интервал значений определенной длины

Операции

арифметика дат и времени

функции корректировки

В PostgreSQL есть тип `interval`, на первый взгляд похожий на диапазон. Но, в отличие от диапазонных типов, интервал определяется длиной, а не начальной и конечной точкой. Кроме того, интервалы существуют только для дат и времени.

Отметим также, что при работе с этим типом есть ряд тонкостей, обусловленных особенностями хранения значений интервалов. Например, существуют специальные функции для корректировки числа дней и часов при их выходе за обычные границы (одну из них мы увидим в демонстрации).

Создание собственных типов, аналогичных `interval`, невозможно. Здесь мы показываем этот тип только для полноты картины и не будем останавливаться на нем подробно.

<https://postgrespro.ru/docs/postgresql/16/datatype-datetime>

<https://postgrespro.ru/docs/postgresql/16/functions-datetime>

Интервалы

Значения типа `interval` определяют длину временного отрезка, в отличие от диапазоного типа, значения которого состоят из начальной и конечной точек.

Интервал появляется естественным образом при вычитании двух моментов времени:

```
=> SELECT timestamp '01.02.2020' - timestamp '01.01.2020';
```

```
?column?
-----
31 days
(1 row)
```

(Хотя та же операция для дат (тип `date`) возвращает целое число.)

Интервалы используются в арифметике дат и времени. Например:

```
=> SELECT now(), now() + 2 * interval '1 month';
```

```
now | ?column?
-----+-----
2025-06-24 05:09:39.918058+03 | 2025-08-24 05:09:39.918058+03
(1 row)
```

Обратите внимание, что интервал `'1 month'` может иметь разную длину — от 28 до 31 дней в зависимости от месяца.

При необходимости можно преобразовать интервал так, чтобы, например, каждый 24-часовой период считался одним днем:

```
=> SELECT interval '29 hours', justify_hours(interval '29 hours');
```

```
interval | justify_hours
-----+-----
29:00:00 | 1 day 05:00:00
(1 row)
```

Ограничение допустимых значений существующего типа

ограничение NOT NULL

значение по умолчанию DEFAULT

проверка CHECK

Создание нового типа

только вручную: CREATE DOMAIN

Домен создается на базе любого существующего типа данных и служит для ограничения множества допустимых значений этого типа.

Определяя домен, можно запретить неопределенные значения (и определить значение по умолчанию), можно также указать произвольную проверку допустимости значения.

По сути, это похоже на ограничения целостности, накладываемые на столбцы при создании таблицы. Создав тип домена, его можно использовать в нескольких таблицах, не указывая каждый раз одни и те же ограничения.

Домены определены в стандарте SQL.

<https://postgrespro.ru/docs/postgresql/16/domains>

<https://postgrespro.ru/docs/postgresql/16/sql-createdomain>

Домены

Создадим доменный тип, ограничивающий временной диапазон рабочими часами и запрещающий неопределенные значения:

```
=> CREATE DOMAIN work_timerange AS timerange
NOT NULL
CHECK (VALUE <@ '[10:00,19:00]':timerange);
```

CREATE DOMAIN

Его можно использовать при создании таблицы:

```
=> CREATE TABLE work_events(
    event_name text,
    event_range work_timerange
);
```

CREATE TABLE

```
=> INSERT INTO work_events VALUES (
    'обед', '[13:00,14:00]'
);
```

INSERT 0 1

А так получится?

```
=> INSERT INTO work_events VALUES (
    'труд', '[14:00,22:00]'
);
```

ERROR: value for domain work_timerange violates check constraint "work_timerange_check"

Нет — нарушено условие проверки. А так?

```
=> INSERT INTO work_events VALUES (
    'лень', NULL
);
```

ERROR: domain work_timerange does not allow null values

Нет — неопределенные значения не допускаются. А если так?

```
=> INSERT INTO work_events VALUES (
    'лень', (SELECT event_range FROM work_events WHERE false)
);
```

INSERT 0 1

А так получится. Запрет неопределенных значений на уровне домена работает не так, как на уровне столбца. Ограничение NOT NULL надежнее указывать при создании таблицы.

Еще одна проблема связана с тем, что при любых операциях значения приводятся к базовому типу. Это не позволяет использовать домены для контроля типов в операциях. Вот пример:

```
=> CREATE DOMAIN distance AS float;
```

CREATE DOMAIN

```
=> CREATE DOMAIN weight AS float;
```

CREATE DOMAIN

Можно ли складывать метры с граммами?

```
=> SELECT 2::distance + 3::weight;
```

?column?

5

(1 row)

Оказывается, можно.

Поэтому практическое применение доменных типов довольно ограничено.

Тип, не сводящийся к комбинации существующих

Создание нового типа

CREATE TYPE, требует программирования на языке C
множество типов предоставляются расширениями

Иногда может возникнуть потребность в совершенно новом типе, который нельзя представить как комбинацию уже существующих.

В этом случае потребуется определить все низкоуровневые детали внутреннего устройства типа, что можно сделать только на языке C. Поэтому мы не будем рассматривать эту возможность.

Впрочем, множество различных типов данных доступны в виде расширений (в том числе сторонних), так что вероятность того, что потребуется совершенно новый тип, очень мала.

<https://postgrespro.ru/docs/postgresql/16/sql-createtype>

Функции, принимающие или возвращающие значения типа

Приведение типов

преобразование значения одного типа к другому типу

Операторы

унарные (префиксные)

бинарные (инфиксные)

Понятно, что мало определить новый тип данных — надо уметь его как-то использовать.

Написанию функций, принимающих или возвращающих значения произвольных (в том числе определенных пользователем) типов посвящен курс DEV1, поэтому на этом мы не будем специально останавливаться.

На функциях основаны и другие важные объекты — приведения типов и операторы, — которые могут определяться пользователем.

Приведения типов позволяют преобразовывать значения одного типа к другому.

Операторы позволяют вместо вызова функции $f(x, y)$ использовать инфиксную нотацию, например $x+y$, что позволяет сделать код проще и нагляднее. Кроме того, как мы увидим в следующей теме «Классы операторов», операторы играют важную роль в индексировании.

Приведение типов в документации:

<https://postgrespro.ru/docs/postgresql/16/sql-createcast>

Операторы:

<https://postgrespro.ru/docs/postgresql/16/xoper>

<https://postgrespro.ru/docs/postgresql/16/xoper-optimization>

<https://postgrespro.ru/docs/postgresql/16/sql-createoperator>

Приведение типов

В качестве примера рассмотрим приведение типа `timerange` к типу `interval`.

Сработает ли такое приведение само по себе?

```
=> SELECT '[10:00,12:00]':::timerange::interval;

ERROR:  cannot cast type timerange to interval
LINE 1: SELECT '[10:00,12:00]':::timerange::interval;
               ^
```

Увы, нет. Чтобы создать такое приведение, сначала напомним функцию:

```
=> CREATE FUNCTION timerange_to_interval(a timerange) RETURNS interval
LANGUAGE sql STRICT IMMUTABLE
RETURN make_interval(
    secs => extract( epoch FROM (upper(a)-lower(a)) )
);

CREATE FUNCTION

=> SELECT timerange_to_interval('[10:00,12:00]');

 timerange_to_interval
-----
02:00:00
(1 row)
```

Теперь можно создать приведение типов. Здесь мы указываем `AS IMPLICIT`, чтобы приведение срабатывало и неявно.

```
=> CREATE CAST (timerange AS interval)
WITH FUNCTION timerange_to_interval
AS IMPLICIT;

CREATE CAST

=> SELECT '[10:00,12:00]':::timerange::interval;

 interval
-----
02:00:00
(1 row)
```

Операторы

Пусть требуется узнать, сколько раз заданный интервал (тип `interval`) содержится в другом интервале. Например, сколько в интервале часов, или минут, или пятисекундных отрезков. Иными словами, нужно деление одного интервала на другой. Но в PostgreSQL для интервалов определен только оператор деления на целое число.

Напишем соответствующую функцию:

```
=> CREATE FUNCTION interval_div(a interval, b interval) RETURNS double precision
LANGUAGE sql STRICT IMMUTABLE
RETURN extract(epoch FROM a) / extract(epoch FROM b);

CREATE FUNCTION
```

Теперь мы можем узнать, сколько раз можно послушать композицию *Your Latest Trick*, длящуюся 6 минут 33 секунды, если у нас есть полтора часа времени:

```
=> SELECT interval_div('1:30:00':::interval, '0:06:33':::interval);

 interval_div
-----
13.740458015267176
(1 row)
```

Определим бинарный оператор, используя эту функцию:

```
=> CREATE OPERATOR / (
    FUNCTION = interval_div,
    LEFTARG = interval,
    RIGHTARG = interval
);
```

CREATE OPERATOR

Имя оператора можно составлять только из специальных символов, список которых приведен в документации к команде CREATE OPERATOR. Если не указать LEFTARG, будет определен префиксный оператор. Постфиксные операторы не поддерживаются начиная с PostgreSQL 14.

Теперь можно пользоваться удобной инфиксной формой записи (здесь также показан другой способ записи интервалов):

```
=> SELECT interval '1 hour 30 min' / interval '6 min 33 sec';
```

```
      ?column?
```

```
-----
```

```
13.740458015267176
```

```
(1 row)
```

PostgreSQL позволяет создавать новые типы данных

составные, перечисления, диапазоны и мультдиапазоны, домены
базовые типы

Для работы со значениями типов могут создаваться
функции, операторы и приведения типов



1. Реализуйте в приложении возможность установки розничной цены книг с указанной пользователем даты (сейчас дата просто игнорируется).
Предыдущая история изменений цен должна сохраняться. Для этого добавьте в таблицу `retail_prices` информацию о периоде действия цены и внесите необходимые изменения в функции `get_retail_price` и `set_retail_price`.
2. Создайте составной тип для формата издания, состоящего из размеров типографского листа и доли листа. Также создайте приведение для этого нового типа к текстовому. Замените тип данных столбца `format` таблицы `books` и убедитесь, что интерфейс с приложением не изменился.

18

1. Интервал действия розничной цены можно представить по-разному:
 - двумя столбцами типа `timestampz` («дата с» и «дата по»);
 - одним столбцом типа `timestampz` («дата с»), при этом цена считается действующей до следующей даты в другой строке таблицы;
 - одним столбцом диапазонного типа `tstzrange`.

Реализуйте последний вариант. Обратите внимание:

- В функции `set_retail_price` придется изменять две строки таблицы, и это должно корректно работать в случае, если несколько пользователей устанавливают цену одновременно.
- Функция `get_retail_price` должна получать цену на текущий момент. Используйте функцию `current_timestamp` (возвращающую время начала транзакции), а не `clock_timestamp`. Это будет важно при последующих изменениях.

2. Формат издания записывается в виде $W \times H / N$, где W и H — ширина и высота типографского листа, с которым работает печатная машина, и который разрезается потом на N одинаковых частей (страниц книги). Поэтому N как правило представляет собой степень двойки.

Несмотря на то, что отношение длины страницы к ширине может быть разным, форматы книг можно упорядочить, сравнивая площади страниц, которые равны $W \times H / N$.

При замене типа столбца обратите внимание на блокировку, которая при этом удерживается.

1. Розничная цена на книги

Добавим в таблицу retail_prices столбец с диапазоном дат. Значение по умолчанию — неограниченный диапазон.

```
=> ALTER TABLE public.retail_prices
    ADD effective tstzrange NOT NULL DEFAULT '(),';
```

ALTER TABLE

Функция получения текущей цены должна выбрать тот диапазон, в который входит текущая дата:

```
=> CREATE OR REPLACE FUNCTION public.get_retail_price(book_id bigint) RETURNS numeric
LANGUAGE sql STABLE SECURITY DEFINER
RETURN (SELECT rp.price
        FROM retail_prices rp
        WHERE rp.book_id = get_retail_price.book_id
        AND rp.effective @> current_timestamp);
```

CREATE FUNCTION

Функция установки цены должна изменить одну строку и добавить одну новую. Использование уровня изоляции Read Committed в этом случае может приводить к аномалиям, поэтому сначала заблокируем соответствующую запись в таблице books, чтобы в один момент времени только одна транзакция могла устанавливать цену одной и той же книги.

```
=> CREATE OR REPLACE FUNCTION empapi.set_retail_price (
    book_id bigint,
    price numeric,
    at timestamptz
)
RETURNS void
AS $$
DECLARE
    lower_bound timestamptz;
    upper_bound timestamptz;
BEGIN
    PERFORM FROM books b
    WHERE b.book_id = set_retail_price.book_id
    FOR UPDATE;

    SELECT lower(rp.effective), upper(rp.effective)
    INTO lower_bound, upper_bound
    FROM retail_prices rp
    WHERE rp.book_id = set_retail_price.book_id
    AND at <@ rp.effective;

    IF at = lower_bound THEN
        -- только обновляем цену в существующем диапазоне
        UPDATE retail_prices rp
        SET price = set_retail_price.price
        WHERE rp.book_id = set_retail_price.book_id
        AND at <@ rp.effective
        ;
    ELSE
        -- закрываем существующий диапазон...
        UPDATE retail_prices rp
        SET effective = tstzrange(lower_bound, at, '[]')
        WHERE rp.book_id = set_retail_price.book_id
        AND at <@ rp.effective;
        -- ...и добавляем новый
        INSERT INTO retail_prices (
            book_id,
            price,
            effective
        ) VALUES (
            book_id,
            price,
            tstzrange(at, upper_bound, '[]')
        );
    END IF;
END;
$$ LANGUAGE plpgsql VOLATILE SECURITY DEFINER;
```

CREATE FUNCTION

Проверим.

```
=> BEGIN;
```

BEGIN

```
=> SELECT * FROM retail_prices WHERE book_id = 1 ORDER BY effective;
```

book_id	price	effective
1	640	(,)

(1 row)

Вставка новой цены в конец:

```
=> SELECT empapi.set_retail_price(1, 200.00, '2025-06-25 05:27:00+03');
```

set_retail_price

(1 row)

```
=> SELECT * FROM retail_prices WHERE book_id = 1 ORDER BY effective;
```

book_id	price	effective
1	640	(,"2025-06-25 05:27:00+03")
1	200.00	["2025-06-25 05:27:00+03",)

(2 rows)

Вставка не в конец:

```
=> SELECT empapi.set_retail_price(1, 300.00, '2025-06-24 05:27:00+03');
```

set_retail_price

(1 row)

```
=> SELECT * FROM retail_prices WHERE book_id = 1 ORDER BY effective;
```

book_id	price	effective
1	640	(,"2025-06-24 05:27:00+03")
1	300.00	["2025-06-24 05:27:00+03","2025-06-25 05:27:00+03")
1	200.00	["2025-06-25 05:27:00+03",)

(3 rows)

Цена с уже существующей датой (новый интервал не появляется):

```
=> SELECT empapi.set_retail_price(1, 400.00, '2025-06-24 05:27:00+03');
```

set_retail_price

(1 row)

```
=> SELECT * FROM retail_prices WHERE book_id = 1 ORDER BY effective;
```

book_id	price	effective
1	640	(,"2025-06-24 05:27:00+03")
1	400.00	["2025-06-24 05:27:00+03","2025-06-25 05:27:00+03")
1	200.00	["2025-06-25 05:27:00+03",)

(3 rows)

```
=> ROLLBACK;
```

ROLLBACK

2. Тип данных для формата издания

Тип данных:

```
=> CREATE TYPE book_format AS (  
  width integer,  
  height integer,  
  parts integer  
);
```

CREATE TYPE

Преобразование в текст:

```
=> CREATE FUNCTION book_format_to_text(f book_format) RETURNS text
LANGUAGE sql STRICT IMMUTABLE
RETURN f.width || 'x' || f.height || '/' || f.parts;
```

CREATE FUNCTION

```
=> CREATE CAST (book_format AS text)
WITH FUNCTION book_format_to_text AS IMPLICIT;
```

CREATE CAST

Проверим:

```
=> SELECT (90,60,16)::book_format::text;
```

```
      row
-----
 90x60/16
(1 row)
```

И обратно:

```
=> CREATE FUNCTION text_to_book_format(f text) RETURNS book_format
LANGUAGE sql STRICT IMMUTABLE
RETURN (SELECT (m[1],m[2],m[3])::book_format
        FROM regexp_match(f, '(\d+)x(\d+)/(\d+)') m);
```

CREATE FUNCTION

```
=> CREATE CAST (text AS book_format)
WITH FUNCTION text_to_book_format AS IMPLICIT;
```

CREATE CAST

Проверим:

```
=> SELECT '90x60/16'::text::book_format;
```

```
      book_format
-----
  (90,60,16)
(1 row)
```

Обратите внимание, что если написать просто:

```
=> SELECT '90x60/16'::book_format;
```

```
ERROR: malformed record literal: "90x60/16"
LINE 1: SELECT '90x60/16'::book_format;
              ^
```

DETAIL: Missing left parenthesis.

то произойдет ошибка: здесь срабатывает не приведение типа text в book_format, а создание типа book_format из литерала, которое использует фиксированный формат для всех составных типов:

```
=> SELECT '(90,60,16)'::book_format;
```

```
      book_format
-----
  (90,60,16)
(1 row)
```

Теперь заменим тип столбца. Это тяжелая операция, которая перезаписывает всю таблицу, полностью блокируя работу с ней. Поэтому использовать ее надо с осторожностью, особенно для больших таблиц.

Вот в каком файле находятся данные сейчас:

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

```
      pg_relation_filepath
-----
base/16388/16415
(1 row)
```

Выполняем замену типа. Преобразование будет выполнено автоматически благодаря созданному ранее приведению типов:

```
=> BEGIN;
```

BEGIN

```
=> ALTER TABLE books ALTER COLUMN format SET DATA TYPE book_format;
```

```
ALTER TABLE
```

```
=> SELECT relation::regclass, mode
FROM pg_locks
WHERE relation = 'books'::regclass;
```

relation	mode
books	ShareLock
books	AccessExclusiveLock

(2 rows)

Операция выполняется в несколько шагов, но при перезаписи таблицы используется исключительная блокировка.

В этой же транзакции надо внести изменения и в интерфейсные функции приложения. Формат возвращают две функции: `webapi.get_catalog` и `empapi.get_catalog`, но обе они вызывают одну и ту же функцию `public.get_catalog`. К счастью, в этой функции столбец `format` явно приводится к типу `text`, поэтому никаких изменений не требуется.

```
=> COMMIT;
```

```
COMMIT
```

Теперь данные таблицы находятся в другом файле:

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

pg_relation_filepath
base/16388/16676

(1 row)

1. Производственные смены каждого сотрудника фабрики хранятся в таблице. Напишите запрос, который покажет, сколько человеко-часов пропадет даром, если в некоторый момент времени на 15 минут будет отключено электричество.
2. В таблице, содержащей много данных, используется столбец статуса, имеющий тип перечисления. Проверьте, какие блокировки будут установлены на таблицу при необходимости добавления нового статуса, и какое время занимает эта операция.
Сравните с альтернативным подходом, при котором перечисление реализуется как обычный текстовый тип с ограничением CHECK на уровне столбца или домена.

19

1. Производственные смены могут храниться в таблице вида:

```
CREATE TABLE shifts (
    employee_name text,
    work_hours tstzrange
);
```

2. Блокировки можно посмотреть в таблице pg_locks, как говорилось в теме «Обзор блокировок».

Кроме обычной проверки

```
status text CHECK (status IN (...))
```

рассмотрите также вариант с использованием домена.

1. Человеко-часы

```
=> CREATE DATABASE ext_datatypes;
```

CREATE DATABASE

```
=> \c ext_datatypes
```

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

Таблица с производственными сменами:

```
=> CREATE TABLE shifts (
    employee_name text,
    work_hours tstzrange
);
```

CREATE TABLE

Добавим несколько записей (рабочий день с перерывом на обед):

```
=> INSERT INTO shifts VALUES
    ('alice', '[2020-04-01 09:00,2020-04-01 13:00)'),
    ('alice', '[2020-04-01 14:00,2020-04-01 18:00)'),
    ('bob', '[2020-04-01 10:00,2020-04-01 14:00)'),
    ('bob', '[2020-04-01 15:00,2020-04-01 17:00)'),
    ('charlie', '[2020-04-01 08:30,2020-04-01 12:30)'),
    ('charlie', '[2020-04-01 13:30,2020-04-01 17:30)');
```

INSERT 0 6

Результат:

```
=> WITH intersection(r) AS (
    SELECT work_hours * '[2020-04-01 09:50,2020-04-01 10:05)>:::tstzrange'
    FROM shifts
)
SELECT sum(upper(r) - lower(r))
FROM intersection;

    sum
-----
00:35:00
(1 row)
```

2. Блокировки при добавлении значения

Тип перечисления и таблица:

```
=> CREATE TYPE statuses AS ENUM ('todo', 'done');
```

CREATE TYPE

```
=> CREATE TABLE process (
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    status statuses
);
```

CREATE TABLE

```
=> INSERT INTO process(status)
    SELECT 'todo' FROM generate_series(1,1_000_000);
```

INSERT 0 1000000

Начинаем транзакцию:

```
=> BEGIN;
```

BEGIN

```
=> \timing on
```

Timing is on.

```
=> ALTER TYPE statuses ADD VALUE 'in progress';
```

ALTER TYPE

Time: 0,622 ms

```
=> \timing off
```

Timing is off.

Выведем только блокировки таблицы:

```
=> SELECT relation::regclass, mode
FROM pg_locks
WHERE relation = 'process'::regclass;

relation | mode
-----+-----
(0 rows)
```

```
=> COMMIT;
```

COMMIT

Итак:

- добавление выполняется быстро;
- таблица не блокируется.

Вариант с проверкой CHECK.

```
=> DROP TABLE process;
```

DROP TABLE

```
=> DROP TYPE statuses;
```

DROP TYPE

```
=> CREATE TABLE process (
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    status text CHECK (status IN ('todo', 'done'))
);
```

CREATE TABLE

```
=> INSERT INTO process(status)
    SELECT 'todo' FROM generate_series(1,1_000_000);
```

INSERT 0 1000000

```
=> BEGIN;
```

BEGIN

```
=> \timing on
```

Timing is on.

```
=> ALTER TABLE process
    DROP CONSTRAINT process_status_check,
    ADD CHECK (status IN ('todo', 'done', 'in progress'));

ALTER TABLE
Time: 100,954 ms
```

```
=> \timing off
```

Timing is off.

```
=> SELECT relation::regclass, mode
FROM pg_locks
WHERE relation = 'process'::regclass;

relation | mode
-----+-----
process  | AccessExclusiveLock
(1 row)
```

```
=> COMMIT;
```

COMMIT

Здесь:

- добавление выполняется дольше — поскольку требуется перепроверка всех значений в таблице;
- таблица полностью блокируется на время изменения ограничения.

Это плохой вариант.

Вариант с проверкой на домене.

```
=> DROP TABLE process;

DROP TABLE

=> CREATE DOMAIN statuses AS text CHECK (VALUE IN ('todo', 'done'));

CREATE DOMAIN

=> CREATE TABLE process (
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    status statuses
);

CREATE TABLE

=> INSERT INTO process(status)
    SELECT 'todo' FROM generate_series(1,1_000_000);

INSERT 0 1000000

=> BEGIN;

BEGIN

=> \timing on

Timing is on.

=> ALTER DOMAIN statuses
    DROP CONSTRAINT statuses_check;

ALTER DOMAIN
Time: 0,156 ms

=> ALTER DOMAIN statuses
    ADD CHECK (VALUE IN ('todo', 'done', 'in progress'));

ALTER DOMAIN
Time: 99,482 ms

=> \timing off

Timing is off.

=> SELECT relation::regclass, mode
FROM pg_locks
WHERE relation = 'process'::regclass;

 relation | mode
-----+-----
 process  | ShareLock
(1 row)

=> COMMIT;

COMMIT
```

В этом случае:

- изменение также выполняется долго;
- таблица блокируется в более мягком режиме (конфликтует с изменением данных, но разрешает чтение).

Это вариант несколько лучше, чем CHECK.