Базы данных

Лекция 08 – Основы SQL. Модификация данных

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.03.26

Оглавление

Модификация данных

Добавление данных

1) Таблица после создания пуста.
2) Следующий шаг после создания — добавление в таблицу данных.
3) Данные добавляются в таблицу по одной строке — это «квант» данных в реляционной модели.
4) Полная строка создается даже если при добавлении будет указано только одно значение.

Для добавления строки используется команда **INSERT**.

```
CREATE TABLE products (
    product_no integer,
    name       varchar(120),
    price      numeric
);
```

Добавить в неё строку можно так:

```
INSERT INTO products VALUES (1, 'Мыло', 1.23);
```

Значения данных перечисляются в порядке столбцов в таблице и разделяются запятыми.

Обычно в качестве значений указываются константы, но это могут быть и скалярные выражения.

Показанная выше запись имеет один недостаток — вам необходимо знать порядок столбцов в таблице. Чтобы избежать этого, можно перечислить столбцы явно. Например, следующие две команды дадут тот же результат, что и показанная выше:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Мыло', 1.23);
INSERT INTO products (name, price, product_no) VALUES ('Мыло', 1.23, 1);
```

Если значения определяются не для всех столбцов, лишние столбцы можно опустить. В таком случае эти столбцы получат значения по умолчанию:

```
INSERT INTO products (product_no, name) VALUES (1, 'Мыло');
INSERT INTO products VALUES (1, 'Мыло');
```

Postgres заполняет столбцы слева направо по числу переданных значений, а все остальные столбцы принимают значения по умолчанию.

Можно явно указать значения по умолчанию для отдельных столбцов или всей строки:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Мыло', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

Одна команда может вставить сразу несколько строк:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Мыло', 1.23),
    (2, 'Зубная паста', 2.34),
    (3, 'Щетка', 3.45);
```

Можно вставлять результат запроса, который может не содержать строк вообще, либо содержать одну или несколько:

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products WHERE release_date = 'today';
```

Наследование

Стандарт SQL:1999 и более поздние версии определяют возможность ***наследования типов***.

PostgreSQL реализует ***наследование таблиц***.

Например, имеется организация, в здании которой могут находиться сотрудники и посетители.

У тех и других есть для учета имена, но у сотрудников дополнительно есть табельные номера, а у посетителей — номера регистрационной карты (гостиница, поликлиника, ...).

Это классическая ситуация «типы-подтипы» и разрешается она обычно объектными средствами.

Объектный подход дает разработчику не только приятные моменты — одной из оборотных сторон является более высокая сложность, нежели при работе с реляционными таблицами.

Не всегда получается придумать схему данных правильно и сразу, и оставить будущему только работу с самими данными. Часто приходится вносить изменения в схему при наличии уже имеющихся данных. Тут объектный подход обнаруживает больше проблем, чем можно было бы желать.

Простой пример наследования

Предположим, что мы создаём модель данных для городов.

В каждой области есть множество городов, но лишь один областной центр. Нужно иметь возможность быстро получать город-столицу региона для любой области. Это можно сделать, создав две таблицы — одну для областных центров, а другую для остальных городов:

```
CREATE TABLE cities (
    name            text,
    population      float,
    elevation       int    -- в
метрах
);
```

```
CREATE TABLE capitals (
    name            text,
    population      float,
    elevation       int     -- в
метрах
);
```

Если нам нужно получить информацию о любом городе, будь то областной центр или нет, у нас возникают небольшие сложности.

Здесь может помочь наследование — таблица **capitals** определяется, как наследник **cities**:

```
CREATE TABLE cities (
    name            text,
    population      float,
    elevation       int     -- в метрах
);

CREATE TABLE capitals (
    region           char(2)
) INHERITS (cities);
```

Таблица **capitals** наследует все столбцы своей родительской таблицы **cities**. Областные центры имеют дополнительный столбец **region**, в котором будет указан регион. Но можно и так:

```
CREATE TABLE region (
    region_id       integer PRIMARY KEY,
    region          text
);

CREATE TABLE cities2 (
    city_id         integer PRIMARY KEY,
    capital         bool DEFAULT no,
    name            text,
    region_id       integer REFERENCES region (region_id),
    population      float,
    elevation       int     -- в метрах
);
```

Консольный клиент psql не имеет специальных команд, генерирующих структуру таблицы на основе внутреннего представления таблицы. Это действие передано на откуп клиентским программам, которые могут на основе данных из системных таблиц построить структуру заданной таблицы.

Для решения этой задачи можно воспользоваться следующей командой:

```
SELECT column_name, column_default, data_type
    FROM INFORMATION_SCHEMA.COLUMNS
        WHERE table_name = 'cities';
 column_name | column_default |    data_type
-------------+----------------+------------------
 population  |                | double precision
 elevation   |                | integer
 name        |                | text
(3 строки)

SELECT column_name, column_default, data_type
    FROM INFORMATION_SCHEMA.COLUMNS
        WHERE table_name = 'cities2';
 column_name | column_default | data_type
-------------+----------------+-----------
 city_id     |                | integer
 capital     | false          | boolean
 region      |                | integer
 population  |                | bigint
 elevation   |                | integer
 name        |                | text
(6 строк)
```

В PostgreSQL таблица может наследоваться от нуля или нескольких других таблиц.

Запросы при этом могут выбирать все строки родительской таблицы или все строки родительской и всех дочерних таблиц. По умолчанию принят последний вариант.

Например, следующий запрос найдёт названия всех городов, включая центры регионов, расположенных выше 100 метров над уровнем моря:

```
SELECT name, elevation FROM cities WHERE elevation > 100;     -- cities & capitals
```

Следующий запрос находит все города, которые не являются столицами регионов, но также находятся на высоте выше 100 метров:

```
SELECT name, elevation FROM ONLY cities WHERE elevation > 100; -- только cities
```

Ключевое слово ONLY указывает, что запрос должен применяться только к таблице **cities**, но не к таблицам, расположенным ниже **cities** в иерархии наследования.

Ключевое слово ONLY поддерживается многими операторами, включая SELECT, UPDATE и DELETE, ограничивая действие предиката выбора.

Чтобы *явно* указать, что должны включаться и дочерние таблицы, следует после имени таблицы добавить *:

```
SELECT name, elevation FROM cities* WHERE elevation > 500;
```

Указывать * не обязательно, так как это поведение подразумевается по умолчанию.

Однако такая запись поддерживается для совместимости со старыми версиями, где поведение по умолчанию могло быть изменено.

В некоторых ситуациях бывает необходимо узнать, из какой таблицы выбрана конкретная строка. Для этого можно использовать системный столбец **tableoid**, присутствующий в каждой таблице:

```
SELECT tableoid, name, elevation
    FROM cities WHERE elevation > 100;
```

запрос выдаст:

```
 tableoid |   name     | elevation
----------+------------+-----------
   12345  | Пинск      |       123
   12345  | Барановичи |       234
   12378  | Брест      |       345
```

Имена таблиц вы можете получить, обратившись к pg_class:

```
SELECT p.relname, c.name, c.elevation
    FROM cities c, pg_class p WHERE c.elevation > 100 AND c.tableoid = p.oid;
 relname  |  name  | elevation
----------+--------+-----------
 cities   | Пинск  |       125
 cities   | Береза |       105
 capitals | Брест  |       135
```

Механизм наследования не способен автоматически распределять данные команд INSERT или COPY по таблицам в иерархии наследования.

Поэтому в следующем примере оператор INSERT не выполнится:

```
INSERT INTO cities (name, population, elevation, region) VALUES
    ('Могилев', NULL, NULL, '06');
```

INSERT всегда вставляет данные непосредственно в указанную таблицу, а таблица **cities** не содержит столбца **region**, поэтому команда будет отвергнута.

INSERT всегда вставляет данные непосредственно в указанную таблицу

Особенности

1) Дочерние таблицы автоматически наследуют от родительской таблицы ограничения-проверки CHECK и ограничения NOT NULL (если только для них не задано явно NO INHERIT).

Все остальные ограничения (уникальности, первичный ключ и внешние ключи) не наследуются.

2) Таблица может наследоваться от нескольких родительских таблиц, в этом случае она будет объединять в себе все столбцы этих таблиц, а также столбцы, описанные непосредственно в её определении.

3) Если в определениях родительских и дочерней таблиц встретятся столбцы с одним именем, эти столбцы будут «объединены», так что в дочерней таблице окажется только один столбец. Чтобы такое объединение было возможно, столбцы должны иметь одинаковый тип данных, в противном случае произойдёт ошибка.

4) Наследуемые ограничения-проверки CHECK и ограничения NOT NULL объединяются подобным образом.

5) Отношение наследования между таблицами обычно устанавливается при создании дочерней таблицы с использованием предложения INHERITS оператора CREATE TABLE.

6) Другой способ добавить такое отношение для таблицы, определённой подходящим образом, — использовать INHERIT с оператором ALTER TABLE.

```
ALTER TABLE capitals INHERIT cities;
```

Будущая дочерняя таблица должна уже включать те же столбцы (с совпадающими именами и типами), что и родительская таблица.

Также она должна включать аналогичные ограничения-проверки CHECK с теми же именами и выражениями.

Удалить отношение наследования можно с помощью указания NO INHERIT оператора ALTER TABLE.

```
ALTER TABLE capitals NO INHERIT cities
```

7) Для создания таблицы, которая затем может стать наследником другой, удобно воспользоваться предложением **LIKE** оператора **CREATE TABLE**. Такая команда создаст новую таблицу с теми же столбцами, что имеются в исходной. Если в исходной таблицы определены ограничения CHECK, для создания полностью совместимой таблицы их тоже нужно скопировать, и это можно сделать, добавив к предложению **LIKE** параметр **INCLUDING CONSTRAINTS**.

```
CREATE TABLE capitals ( LIKE cities [INCLUDING CONSTRAINTS] );
```

8) Родительскую таблицу нельзя удалить, пока существуют таблицы, унаследованные от неё.

9) В дочерних таблицах нельзя удалять или модифицировать столбцы или ограничения-проверки, унаследованные от родительских таблиц.

10) Для удаления таблицы вместе со всеми её потомками необходимо добавить в команду удаления родительской таблицы параметр CASCADE.

Изменение данных. Команда UPDATE

Модификация данных, сохранённых в БД, называется изменением.
Изменить можно:
- отдельные строки;
- все строки таблицы;
- некоторое подмножество всех строк.

Каждый столбец можно изменять независимо от других.
Для изменения данных в существующих строках используется команда UPDATE.
Ей требуется следующая информация:
- имя таблицы и изменяемого столбца;
- новое значение столбца
- критерий отбора изменяемых строк.

SQL не назначает строкам уникальные идентификаторы, соответственно, в таблице можно иметь несколько полностью идентичных строк. Такое дублирование обычно нежелательно и забота о том, чтобы такого не случалось, возлагается на программиста.

Итак, не всегда возможно явно указать на строку, которую требуется изменить. Поэтому необходимо указывать условия, каким должна соответствовать изменяемая строка.

Изменение отдельных строк

Однозначно адресовать отдельные строки можно только в том случае, если в таблице есть первичный ключ. В этом случае его можно использовать для определения вышеупомянутых условий соответствия. Именно таким образом работают многие интерактивные приложения для работы с базами данных, дающие возможность редактировать данные в индивидуальных строках.

Пример:

```
UPDATE products SET price = 10 WHERE price = 5;
```

В результате в таблице может измениться ноль, одна или несколько строк.

Команда изменения данных начинается с ключевого слова **UPDATE**. За ним идёт имя таблицы **products**. Затем идёт ключевое слово **SET**, за которым следует имя столбца, который следует изменить, знак равенства и новое значение столбца (10). Этим значением может быть не только константа, но и любое скалярное выражение. Например, если нужно поднять цену всех товаров на 10%:

```
UPDATE products SET price = price * 1.10;
```

Выражение нового значения может ссылаться на существующие значения столбцов в строке.

Предложение **WHERE price = 5** устанавливает условия, которым должна соответствовать изменяемая строка. Поскольку предложение WHERE во втором примере отсутствует, будут изменены все строки в таблице.

Если же это предложение присутствует, изменяются только строки, которые соответствуют условию WHERE.

В предложении SET знак равенства обозначает операцию присваивания, а в предложении WHERE он используется для сравнения и это не приводит к неоднозначности.

В условии WHERE не обязательно должна быть проверка равенства — могут применяться любые другие операторы. Необходимо только, чтобы это выражение было предикатом.

Предикат — выражение логического типа, использующее одну или более величин на входе.

В команде UPDATE можно изменить значения сразу нескольких столбцов, для этого их нужно все перечислить в предложении SET.

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

Удаление данных

Удалять данные можно только целыми строками.

Поскольку в SQL нет возможности напрямую адресовать отдельные строки, удалить избранные строки можно только сформулировав для них подходящие условия.

Если в таблице есть первичный ключ, с его помощью можно однозначно выделить для удаления определённую строку (интерактивные приложения).

Также можно удалить группы строк, соответствующие условию, либо сразу все строки таблицы.

Для удаления строк используется команда DELETE.

Её синтаксис очень похож на синтаксис команды UPDATE.

Удаление всех строк из таблицы с товарами, имеющими цену 10:

```
DELETE FROM products WHERE price = 10;
```

Удаление всех строк из таблицы

```
DELETE FROM products;
```

Возврат данных из изменённых строк

Иногда бывает полезно получать данные из модифицируемых строк в процессе их обработки.
Для этого используется предложение **RETURNING**.
Его можно задать для команд INSERT, UPDATE и DELETE.
RETURNING позволяет обойтись без дополнительного запроса к базе для выбора данных, особенно в тех случаях, когда как-то иначе трудно получить строки.

В предложении RETURNING используется то же, что и в выходном списке команды SELECT. Оно может содержать имена столбцов целевой таблицы команды или значения выражений с этими столбцами.
Часто применяется краткая запись RETURNING *, выбирающая все столбцы целевой таблицы по порядку.
В команде INSERT данные, выдаваемые в RETURNING, образуются из строки в том виде, в каком она была вставлена.
Если в таблице есть вычисляемые поля по умолчанию, RETURNING позволяет узнать их содержимое. Например, если в таблице есть столбец **serial**, в котором генерируются уникальные идентификаторы, команда RETURNING может возвратить идентификатор, назначенный новой строке:

```
CREATE TABLE users (id serial primary key, firstname text, lastname text);

INSERT INTO users (firstname, lastname) VALUES ('Peter', 'The First') RETURNING id;
```

RETURNING также очень полезно с INSERT ... SELECT.

В команде UPDATE данные, выдаваемые в RETURNING, образуются ***новым содержимым*** изменённой строки:

```
UPDATE products SET price = price * 1.10 WHERE price <= 99.99
    RETURNING name, price AS new_price;
```

В команде DELETE данные, выдаваемые в RETURNING, образуются ***содержимым удалённой строки***:

```
DELETE FROM products WHERE obsoletion_date = 'today'
    RETURNING *;
```

Если для целевой таблицы заданы триггеры, в RETURNING выдаются данные из строки, которая изменена триггерами. Поэтому RETURNING часто применяется и для того, чтобы проверить содержимое столбцов, изменяемых триггерами.

Функции и операторы

Postgres имеет огромное количество функций и операторов для встроенных типов данных.

Также пользователи могут определять свои функции и операторы. Все существующие функции и операторы в **psql** можно просмотреть с помощью команд **\df** и **\do**, соответственно.

```
==> \df pg_copy*
                           Список функций
Схема   |  Имя | Тип данных результата | Типы данных аргументов |  Тип
-----------------------------------------------------------------------------
...
(5 строк)
==> \df *
                           Список функций
Схема   |  Имя | Тип данных результата | Типы данных аргументов |  Тип
-----------------------------------------------------------------------------
...
(3191 строка)
==> \do *
                         Список операторов
Схема  | Имя | Тип левого аргумента | Тип правого аргумента | Результирующий тип |
Описание
-----------------------------------------------------------------------------
...
pg_cat |  +  | circle               | point                 | circle             | add
...
(804 строки)
```

В документации postgres типы аргументов и результата функции обозначаются как:

`foo(text, integer) -> text`

Функция **`foo`** принимает один текстовый и один целочисленный аргумент и возвращает результат текстового типа.

```
==> select repeat('Pg', 4);
repeat
------------
PgPgPgPg
(1 строка)
```

Важно:

1) практически все функции и операторы postgres, за исключением простейших арифметических и операторов сравнения, в стандарте SQL не описаны.

2) тем не менее они присутствуют и в других СУБД SQL, причем во многих случаях различные реализации одинаковых функций оказываются совместимыми.

Схемы

Кластер баз данных PostgreSQL содержит один или несколько именованных экземпляров баз.

Роли и некоторые другие объекты создаются на уровне кластера.

В рамках одного подключения к серверу можно обращаться к данным только одной базы — той, что была выбрана при установлении соединения (`$ psql database`).

База данных содержит одну или несколько именованных ***схем***, которые, в свою очередь, содержат таблицы.

Помимо таблиц схемы содержат именованные объекты других видов, включая типы данных, функции и операторы. При этом одно и то же имя объекта можно свободно использовать в разных схемах, например и **`schema1`**, и **`schema2`** могут содержать таблицы с именем **`sometable`**. Схемы нужны для:

- чтобы одну базу данных могли использовать несколько пользователей, не влияя друг на друга;
- чтобы объединить объекты базы данных в логические группы для облегчения управления ими;
- чтобы в одной БД существовали разные приложения, и при этом не возникало конфликтов имён.

Схемы в некотором смысле подобны каталогам в операционной системе, но они не могут быть вложенными.