Организация данных

Логическая структура

Темы

Базы данных и шаблоны

Схемы и путь поиска

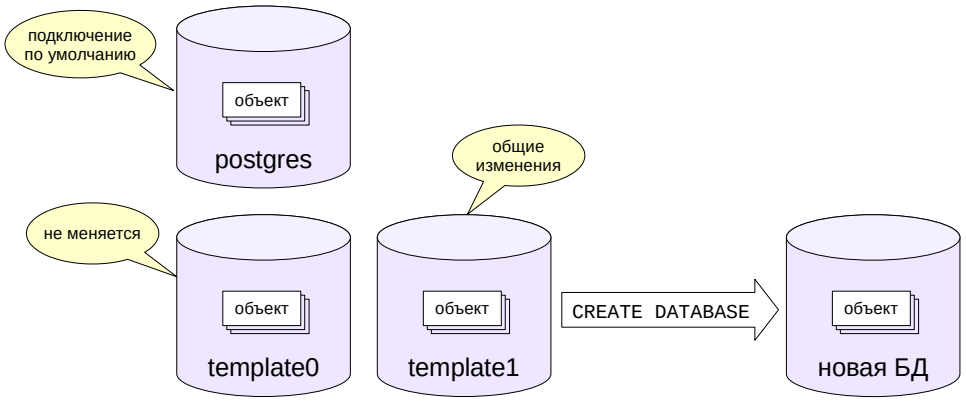
Специальные схемы

Системный каталог

Кластер баз данных

Инициализация кластера создает три базы данных

Новая база всегда клонируется из существующей



3

Экземпляр PostgreSQL управляет несколькими базами данных — кластером. При инициализации кластера (автоматически при установке PostgreSQL либо вручную командой initdb) создаются три одинаковые базы данных. Все остальные БД, создаваемые пользователем, клонируются из какой-либо существующей.

Шаблонная БД template1 используется по умолчанию для создания новых баз данных. В нее можно добавить объекты и расширения, которые будут копироваться в каждую новую базу данных.

Шаблон template0 не должен изменяться. Он нужен как минимум в двух ситуациях. Во-первых, для восстановления БД из резервной копии, выполненной утилитой pg_dump (это рассматривается в теме «Резервное копирование. Логическое резервирование»). Во-вторых, при создании новой БД с кодировкой, отличной от указанной при инициализации кластера (подробнее обсуждается в курсе DBA2).

База данных postgres используется при подключении по умолчанию пользователем postgres. Она не является обязательной, но некоторые утилиты предполагают ее наличие, поэтому ее не рекомендуется удалять, даже если она не нужна.

https://postgrespro.ru/docs/postgresql/16/manage-ag-templatedbs

Базы данных

Список баз данных можно получить в psql такой командой:

```
=> \l
```

```
                                                  List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU
Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+-------------+-------------+----------
--+-----------+-----------------------
 postgres  | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
  |           |
 student   | student  | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
  |           |
 template0 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
  |           | =c/postgres          +
           |          |          |                 |             |             |
  |           | postgres=CTc/postgres
 template1 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
  |           | =c/postgres          +
           |          |          |                 |             |             |
  |           | postgres=CTc/postgres
(4 rows)
```

База данных student была создана для удобства подключения одноименного пользователя, а bookstore понадобится нам в дальнейшем для разработки приложения. В выводе команды присутствует ряд столбцов, которые нас сейчас не интересуют.

Когда мы создаем новую базу данных, она (по умолчанию) копируется из шаблона template1.

```
=> CREATE DATABASE data_logical;
```

```
CREATE DATABASE
```

```
=> \c data_logical
```

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

```
=> \l
```

```
                                                    List of databases
     Name     |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU
Locale | ICU Rules |   Access privileges
--------------+----------+----------+-----------------+-------------+-------------+-------
-----+-----------+-----------------------
 data_logical | student  | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           |
 postgres     | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           |
 student      | student  | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           |
 template0    | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           | =c/postgres          +
              |          |          |                 |             |             |
     |           | postgres=CTc/postgres
 template1    | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           | =c/postgres          +
              |          |          |                 |             |             |
     |           | postgres=CTc/postgres
(5 rows)
```

Схемы

Пространство имен для объектов

- разделение объектов на логические группы
- предотвращение конфликта имен между приложениями

Схема и пользователь — разные сущности

Специальные схемы

- public — по умолчанию в ней создаются все объекты
- pg_catalog — системный каталог
- information_schema — вариант системного каталога
- pg_temp — для временных таблиц
- …

Схемы представляют собой пространства имен для объектов БД. Они позволяют разделить объекты на логические группы для управления ими, предотвратить конфликты имен при работе нескольких пользователей или при установке приложений и расширений.

В PostgreSQL *схема* и *пользователь* — разные сущности (хотя настройки по умолчанию позволяют пользователям удобно работать с одноименными схемами).

Существует несколько специальных схем, обычно присутствующих в каждой базе данных.

Схема public используется по умолчанию для хранения объектов, если не выполнены иные настройки.

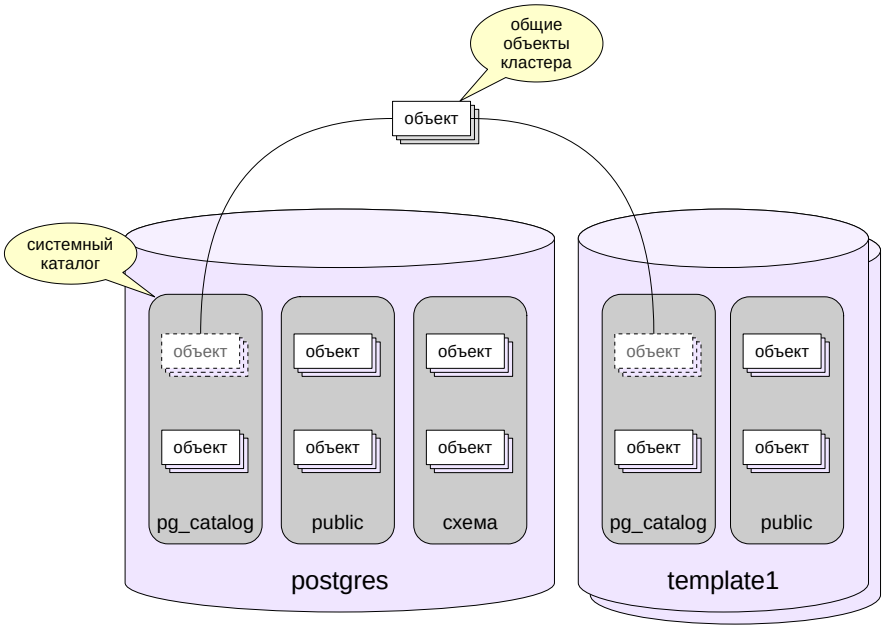
Схема pg_catalog хранит объекты *системного каталога*. Системный каталог — это метаинформация об объектах, принадлежащих кластеру, которая хранится в самом кластере в виде таблиц. Альтернативное представление системного каталога (определенное в стандарте SQL) дает схема information_schema.

Схема pg_temp служит для хранения временных таблиц. (На самом деле таблицы создаются в схемах pg_temp_1, pg_temp_2 и т. п. — у каждого пользователя своя схема. Но обращаются все пользователи к ней как к pg_temp.)

Есть и другие схемы, но они носят технический характер.

https://postgrespro.ru/docs/postgresql/16/ddl-schemas

Базы и схемы кластера



Схемы принадлежат базам данных, все объекты БД распределены по каким-либо схемам.

Однако несколько таблиц системного каталога хранят информацию, общую для всего кластера. Это список баз данных, список пользователей и некоторые другие сведения. Эти таблицы хранятся вне какой-либо конкретной базы данных, но при этом одинаково видны из каждой БД.

Таким образом, клиент, подключенный к какой-либо базе данных, видит в системном каталоге описание объектов не только данной базы, но и общих объектов кластера. Описание объектов других баз данных можно получить, только подключившись к ним.

Схемы

Для вывода списка схем в psql есть специальная команда (\dn = describe namespace):

```
=> \dn
```

```
      List of schemas
  Name  |       Owner
--------+-------------------
 public | pg_database_owner
(1 row)
```

Эта команда не показывает служебные схемы. Чтобы увидеть их, нужно добавить модификатор S (он работает аналогичным образом и для многих других команд):

```
=> \dnS
```

```
            List of schemas
        Name        |       Owner
--------------------+-------------------
 information_schema | postgres
 pg_catalog         | postgres
 pg_toast           | postgres
 public             | pg_database_owner
(4 rows)
```

Про некоторые из этих схем (public, pg_catalog, information_schema) мы уже говорили; про остальные поговорим позже в других темах.

Еще один полезный модификатор — знак «плюс», который выводит дополнительную информацию:

```
=> \dn+
```

```
                                     List of schemas
  Name  |       Owner       |           Access privileges            |      Description
--------+-------------------+----------------------------------------+--------------------
----
 public | pg_database_owner | pg_database_owner=UC/pg_database_owner+| standard public
schema
        |                   | =U/pg_database_owner                   |
(1 row)
```

Создадим новую схему:

```
=> CREATE SCHEMA special;
```

```
CREATE SCHEMA
```

```
=> \dn
```

```
       List of schemas
  Name   |       Owner
---------+-------------------
 public  | pg_database_owner
 special | student
(2 rows)
```

Создадим таблицу:

```
=> CREATE TABLE t(n integer);
```

```
CREATE TABLE
```

По умолчанию таблица будет создана в схеме public. Список таблиц в этой схеме можно получить командой \dt с указанием шаблона для имен схем и таблиц:

```
=> \dt public.*
```

```
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+---------
 public | t    | table | student
(1 row)
```

Таблицу (как и другие объекты) можно перемещать между схемами. Поскольку речь идет о логической организации, перемещение происходит только в системном каталоге; сами данные физически остаются на месте.

```
=> ALTER TABLE t SET SCHEMA special;
```

```
ALTER TABLE
```

Что останется в схеме public?

```
=> \dt public.*
```

```
Did not find any relation named "public.*".
```

Ничего. А в special?

```
=> \dt special.*
```

```
        List of relations
 Schema  | Name | Type  |  Owner
---------+------+-------+---------
 special | t    | table | student
(1 row)
```

Таблица переместилась. Теперь к ней можно обращаться с явным указанием схемы:

```
=> SELECT * FROM special.t;
```

```
 n
---
(0 rows)
```

Но если опустить имя схемы, таблица не будет найдена:

```
=> SELECT * FROM t;
```

```
ERROR:  relation "t" does not exist
LINE 1: SELECT * FROM t;
                      ^
```

Путь поиска

Определение схемы объекта

квалифицированное имя (*схема.имя*) явно определяет схему

имя без квалификатора проверяется в схемах, указанных в пути поиска

Путь поиска

определяется параметром *search_path*,
реальное значение — функция current_schemas

исключаются несуществующие схемы и схемы, к которым нет доступа

первая явно указанная в пути схема используется для создания объектов

схемы pg_temp и pg_catalog неявно включаются первыми,
если не указаны в *search_path*

При указании объекта надо определить, о какой схеме идет речь, ведь в разных схемах могут храниться объекты с одинаковыми именами.

Если имя объекта квалифицировано именем схемы, то используется явно указанная схема. Если схема не указана явным образом, то она определяется с помощью конфигурационного параметра *search_path*. Этот параметр содержит путь поиска — список схем, который просматривается последовательно слева направо, при этом из него исключаются несуществующие схемы и те, к которым у пользователя нет доступа.

При создании нового объекта с именем без квалификатора для выбора целевой схемы берется первая из оставшихся в списке, а при поиске объекта в начало пути неявно добавляются:

- схема pg_catalog, чтобы всегда иметь доступ к системному каталогу
- схема pg_temp, если пользователь создавал временные объекты

Реальный путь поиска, включающий неявные схемы, возвращает вызов функции: current_schemas(true).

Можно провести аналогию между путем поиска *search_path* и путем PATH в операционных системах.

https://postgrespro.ru/docs/postgresql/16/runtime-config-client#GUC-SEARCH-PATH

Путь поиска

Путь поиска по умолчанию имеет такое значение:

```
=> SHOW search_path;
```

```
   search_path
-----------------
 "$user", public
(1 row)
```

Конструкция «$user» обозначает схему с тем же именем, что и имя текущего пользователя (в нашем случае — student). Поскольку такой схемы нет, она игнорируется.

Чтобы не думать над тем, какие схемы есть, каких нет, а какие недоступны, можно воспользоваться функцией:

```
=> SELECT current_schemas(false);
```

```
 current_schemas
-----------------
 {public}
(1 row)
```

Передаваемый в функцию логический параметр управляет отображением системных схем, неявно включаемых при поиске. Мы можем увидеть, что кроме исключения несуществующей схемы PostgreSQL неявно включил в начало списка схему системного каталога:

```
=> SELECT current_schemas(true);
```

```
   current_schemas
---------------------
 {pg_catalog,public}
(1 row)
```

Установим путь поиска, например, так:

```
=> SET search_path = public, special;
```

```
SET
```

Теперь таблица будет найдена.

```
=> SELECT * FROM t;
```

```
 n
---
(0 rows)
```

Здесь мы установили конфигурационный параметр на уровне сеанса и при переподключении его значение пропадет. Устанавливать такое значение на уровне всего кластера тоже неправильно — возможно, этот путь нужен не всегда и не всем, к тому же в разных БД может быть разный набор схем.

Но параметр можно установить и на уровне отдельной базы данных:

```
=> ALTER DATABASE data_logical SET search_path = public, special;
```

```
ALTER DATABASE
```

Теперь он будет устанавливаться для всех новых подключений к БД data_logical. Проверим:

```
=> \c data_logical
```

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

```
=> SHOW search_path;
```

```
   search_path
-----------------
 public, special
(1 row)
```

Системный каталог

Описание всех объектов кластера

- набор таблиц в каждой базе данных (схема pg_catalog) и несколько глобальных объектов кластера
- набор представлений для удобства

Доступ

- запросы SQL, специальные команды psql

Правила организации

- названия таблиц начинаются с pg_
- имена столбцов содержат трехбуквенный префикс
- в качестве ключа используется столбец oid типа oid
- названия объектов хранятся в нижнем регистре

Системный каталог хранит метаинформацию об объектах кластера. В каждой базе данных доступен собственный набор таблиц, описывающих объекты этой конкретной БД, и нескольких таблиц, общих для всего кластера. Для удобства над таблицами также определены несколько представлений.

https://postgrespro.ru/docs/postgresql/16/catalogs

К системному каталогу можно обращаться с помощью обычных запросов SQL, а выполнение команд DDL приводит к изменению данных в системном каталоге. Кроме того, psql имеет целый ряд команд, позволяющих удобно просматривать системный каталог.

https://postgrespro.ru/docs/postgresql/16/app-psql

Все имена таблиц системного каталога начинаются с pg_, например, pg_database. Столбцы таблиц начинаются с префикса, обычно соответствующего имени таблицы, например, datname. Имена объектов хранятся в нижнем регистре, например, 'postgres'.

Таблицы системного каталога имеют первичные ключи — как правило, это столбцы с именем oid и типом oid (object identifier, целое 32-битное число). Эти идентификаторы встречаются и в других столбцах в виде отдельных значений или массивов, обеспечивая логические связи между таблицами. Внешние ключи в системном каталоге явно не определены.

https://postgrespro.ru/docs/postgresql/16/datatype-oid

Системный каталог

Для того, чтобы вывести информацию о любых объектах, psql (как и другие интерактивные пользовательские средства) обращается к таблицам системного каталога.

Например, команда \l для получения списка баз данных кластера, обращается к таблице:

```
=> SELECT datname FROM pg_database;
```

```
   datname
--------------
 postgres
 student
 template1
 template0
 data_logical
(5 rows)
```

Мы всегда можем посмотреть, какие запросы выполняет команда:

```
=> \set ECHO_HIDDEN on
```

```
=> \l
```

```
********* QUERY **********
SELECT
  d.datname as "Name",
  pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
  pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
  CASE d.datlocprovider WHEN 'c' THEN 'libc' WHEN 'i' THEN 'icu' END AS "Locale Provider",
  d.datcollate as "Collate",
  d.datctype as "Ctype",
  d.daticulocale as "ICU Locale",
  d.daticurules as "ICU Rules",
  pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges"
FROM pg_catalog.pg_database d
ORDER BY 1;
**************************
```

```
                                                      List of databases
     Name     |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU
Locale | ICU Rules |   Access privileges
--------------+----------+----------+-----------------+-------------+-------------+-------
-----+-----------+-----------------------
 data_logical | student  | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           |
 postgres     | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           |
 student      | student  | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           |
 template0    | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           | =c/postgres          +
              |          |          |                 |             |             |
     |           | postgres=CTc/postgres
 template1    | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
     |           | =c/postgres          +
              |          |          |                 |             |             |
     |           | postgres=CTc/postgres
(5 rows)
```

Таким образом можно исследовать системный каталог.

Отключим вывод команд.

```
=> \set ECHO_HIDDEN off
```

Список схем находится в таблице:

```
=> SELECT nspname FROM pg_namespace;
```

```
      nspname
--------------------
 pg_toast
 pg_catalog
 public
 information_schema
 special
(5 rows)
```

А на такие объекты, как таблицы и индексы, можно посмотреть так:

```
=> SELECT relname, relkind, relnamespace FROM pg_class WHERE relname = 't';
```

```
 relname | relkind | relnamespace
---------+---------+--------------
 t       | r       |        16387
(1 row)
```

Все столбцы здесь начинаются на rel (relation, отношение).

- relkind — тип объекта (r — таблица, i — индекс и т. п.);
- relnamespace — схема.

Поле relnamespace имеет тип oid; вот соответствующая строка таблицы pg_namespace:

```
=> SELECT oid, nspname FROM pg_namespace WHERE oid = 16387;
```

```
  oid  | nspname
-------+---------
 16387 | special
(1 row)
```

Для удобства преобразования между текстовым представлением и oid можно воспользоваться приведением к специальному типу-псевдониму regnamespace:

```
=> SELECT relname, relkind, relnamespace::regnamespace::text
FROM pg_class WHERE relname = 't';
```

```
 relname | relkind | relnamespace
---------+---------+--------------
 t       | r       | special
(1 row)
```

А вот как можно получить список объектов в схеме, например, pg_catalog:

```
=> SELECT relname, relkind FROM pg_class
WHERE relnamespace = 'pg_catalog'::regnamespace LIMIT 5;
```

```
            relname             | relkind
--------------------------------+---------
 pg_statistic                   | r
 pg_type                        | r
 pg_foreign_table               | r
 pg_proc_oid_index              | i
 pg_proc_proname_args_nsp_index | i
(5 rows)
```

Аналогичные reg-типы определены и для некоторых других таблиц системного каталога. Они позволяют упростить запросы и обойтись без явного соединения таблиц.

Удаление объектов

Можно ли удалить схему special?

```
=> DROP SCHEMA special;
```

```
ERROR:  cannot drop schema special because other objects depend on it
DETAIL:  table t depends on schema special
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

Схему нельзя удалить, если в ней находятся какие-либо объекты. Сначала надо удалить или перенести их.

Но можно удалить схему сразу вместе со всеми ее объектами:

```
=> DROP SCHEMA special CASCADE;
```

```
NOTICE:  drop cascades to table t
DROP SCHEMA
```

А что с удалением базы данных целиком? Во-первых, нельзя удалить базу, к которой вы подключены в данный момент, поэтому отключимся от нее.

```
=> \conninfo
```

```
You are connected to database "data_logical" as user "student" via socket in
"/var/run/postgresql" at port "5432".
```

```
=> \c postgres
```

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

Во-вторых, базу данных также нельзя удалить, если к ней есть активные подключения. Создадим такое подключение в отдельном сеансе и попробуем удалить ее:

```
=> \c data_logical
```

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

```
=> DROP DATABASE data_logical;
```

```
ERROR:  database "data_logical" is being accessed by other users
DETAIL:  There is 1 other session using the database.
```

Получили ошибку. Однако можно вызвать команду удаления с параметром FORCE, тогда она будет пытаться принудительно завершить все подключения к БД, а затем удалит ее:

```
=> DROP DATABASE data_logical WITH (FORCE);
```

```
DROP DATABASE
```

Итоги

Логически

кластер содержит базы данных,
базы данных — схемы,
схемы — конкретные объекты (таблицы, индексы и т. п.)

Базы данных создаются клонированием существующих

Схема объекта определяется по пути поиска

Полное описание содержимого кластера баз данных хранится в системном каталоге

Практика

1. В новой базе данных создайте схему, названную так же, как и пользователь. Создайте схему app.
   Создайте несколько таблиц в обеих схемах.
2. Получите в psql описание созданных схем и список всех таблиц в них.
3. Установите путь поиска так, чтобы при подключении к базе данных таблицы из обеих схем были доступны по неквалифицированному имени; приоритет должна иметь «пользовательская» схема.
   Проверьте правильность настройки.

1. База данных, схемы, таблицы

Создаем базу данных:

```
=> CREATE DATABASE data_logical;
```

```
CREATE DATABASE
```

```
=> \c data_logical
```

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

Схемы:

```
=> CREATE SCHEMA student;
```

```
CREATE SCHEMA
```

```
=> CREATE SCHEMA app;
```

```
CREATE SCHEMA
```

Таблицы для схемы student:

```
=> CREATE TABLE a(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO a VALUES ('student');
```

```
INSERT 0 1
```

```
=> CREATE TABLE b(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO b VALUES ('student');
```

```
INSERT 0 1
```

Таблицы для схемы app:

```
=> CREATE TABLE app.a(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO app.a VALUES ('app');
```

```
INSERT 0 1
```

```
=> CREATE TABLE app.c(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO app.c VALUES ('app');
```

```
INSERT 0 1
```

2. Описание схем и таблиц

Описание схем:

```
=> \dn
```

```
       List of schemas
  Name   |       Owner
---------+-------------------
 app     | student
 public  | pg_database_owner
 student | student
(3 rows)
```

Описание таблиц:

```
=> \dt student.*
```

```
        List of relations
 Schema  | Name | Type  |  Owner
---------+------+-------+---------
 student | a    | table | student
 student | b    | table | student
(2 rows)
```

```
=> \dt app.*
```

```
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+---------
 app    | a    | table | student
 app    | c    | table | student
(2 rows)
```

3. Путь поиска

С текущими настройками пути поиска видны только таблицы, находящиеся в схеме student:

```
=> SELECT * FROM a;
```

```
    s
---------
 student
(1 row)
```

```
=> SELECT * FROM b;
```

```
    s
---------
 student
(1 row)
```

```
=> SELECT * FROM c;
```

```
ERROR:  relation "c" does not exist
LINE 1: SELECT * FROM c;
                      ^
```

Изменим путь поиска на уровне базы.

```
=> ALTER DATABASE data_logical SET search_path = "$user",app,public;
```

```
ALTER DATABASE
```

```
=> \c
```

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

```
=> SHOW search_path;
```

```
     search_path
----------------------
 "$user", app, public
(1 row)
```

Теперь видны таблицы из обеих схем, но приоритет остается за student:

```
=> SELECT * FROM a;
```

```
    s
---------
 student
(1 row)
```

```
=> SELECT * FROM b;
```

```
    s
---------
 student
(1 row)
```

```
=> SELECT * FROM c;
```

```
  s
-----
 app
(1 row)
```

```
=> select usename, application_name from pg_stat_activity where datname = 'data_logical';
```

```
 usename | application_name
---------+------------------
 student | psql
(1 row)
```

```
=> \c postgres
```

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

```
=> DROP DATABASE data_logical;
```

```
DROP DATABASE
```

Приложение «Книжный магазин»

Схема данных и интерфейс

Темы

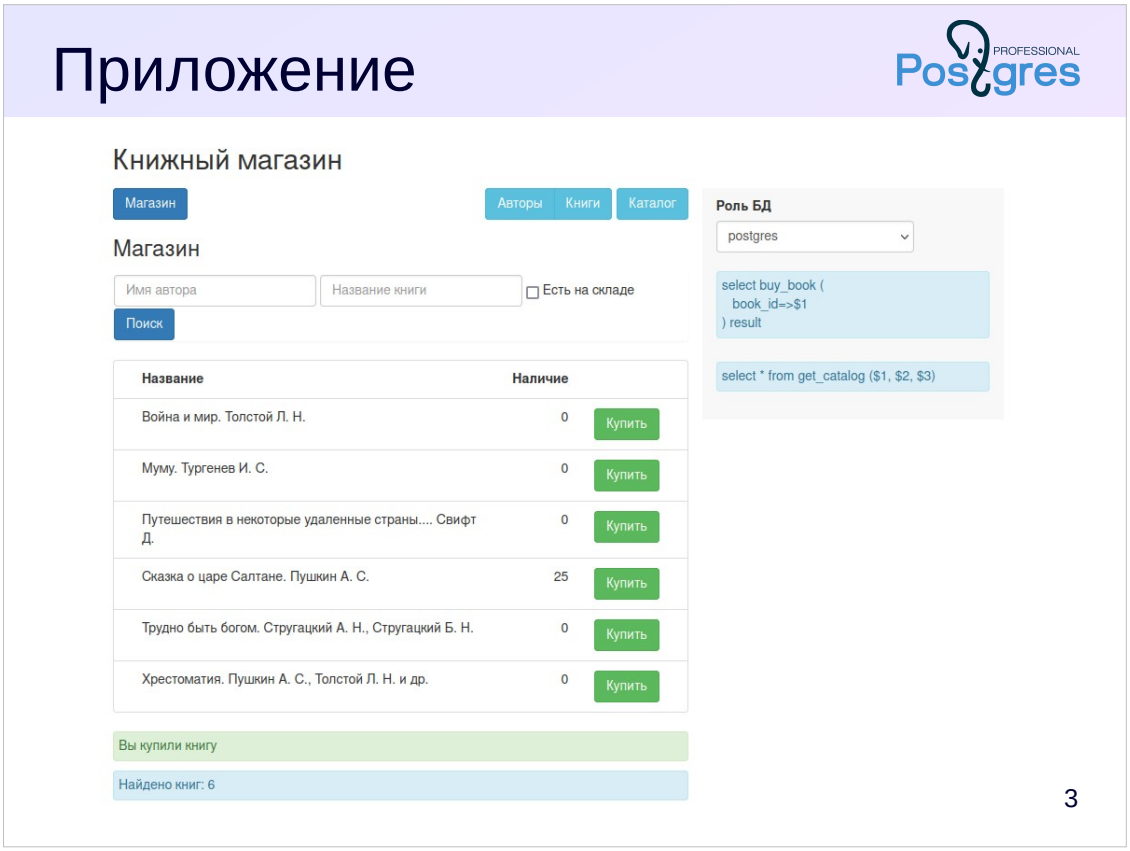
Обзор приложения «Книжный магазин»

Проектирование схемы данных, нормализация

Итоговая схема данных приложения

Организация интерфейса между клиентом и сервером

Приложение



Приложение состоит из нескольких частей, представленных вкладками.

«Магазин» — это интерфейс веб-пользователя, в котором он может покупать книги.

Остальные вкладки соответствуют внутреннему интерфейсу, доступному только сотрудникам («админка» сайта).

«Каталог» — интерфейс кладовщика, в котором он может заказывать закупку книг на склад магазина и просматривать операции поступлений и покупок.

«Книги» и «Авторы» — интерфейс библиотекаря, в котором он может регистрировать авторов и их книги.

В учебных целях вся функциональность представлена на одной общей веб-странице. Если какая-то часть функциональности недоступна из-за того, что на сервере нет подходящего объекта (таблицы, функции и т. п.), приложение сообщит об этом. Также приложение выводит текст запросов, которые оно посылает на сервер.

В ходе курса мы начнем с пустой базы данных и постепенно реализуем в ней все необходимые компоненты.

P. S. Исходный код приложения не является темой курса, но может быть получен в репозитории https://pubgit.postgrespro.ru/pub/dev1app.git

Демонстрация приложения

В этой демонстрации мы показываем приложение «Книжный магазин» в том виде, в котором оно будет после завершения всех практических заданий. Приложение откроется в отдельной вкладке браузера виртуальной машины курса:

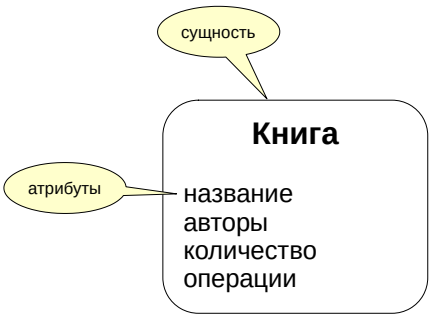
Открываем файл http://localhost...

```
student$ xdg-open http://localhost
```

Книги

ER-модель для высокоуровневого проектирования

сущности — понятия предметной области
связи — отношения между сущностями
атрибуты — свойства сущностей и связей



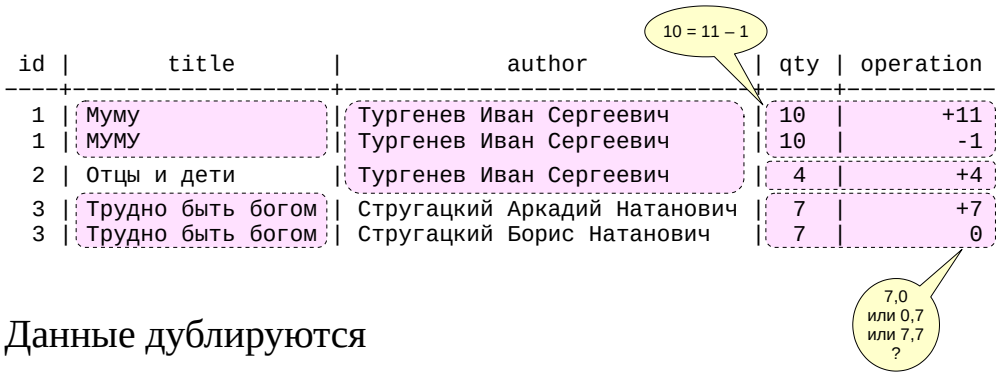
5

После того, как мы посмотрели на внешний вид приложения и поняли его функциональность, нам нужно разобраться со схемой данных. Мы не будем хоть сколько-нибудь глубоко вникать в вопросы проектирования баз данных — это отдельная дисциплина, не входящая в данный курс. Но совсем обойти вниманием эту тему тоже нельзя.

Часто для высокоуровневого проектирования баз данных используется модель «сущность — связи», или ER-модель (Entity — Relationship). Она оперирует *сущностями* (понятиями предметной области), *связями* между ними и *атрибутами* (свойствами сущностей и связей). Модель позволяет рассуждать на логическом уровне, не опускаясь до деталей представления данных на физическом (в виде таблиц).

Первым подходом к проектированию может служить диаграмма, представленная на слайде: можно представить одну большую сущность «Книга», а все остальное сделать ее атрибутами.

Схема данных



Данные дублируются

- сложно поддерживать согласованность
- сложно обновлять
- сложно писать запросы

Очевидно, что это неправильный подход. На диаграмме это может быть не так заметно, но давайте попробуем отобразить диаграмму на таблицы БД. Это можно сделать разными способами, например так, как показано на слайде: сущность становится таблицей, ее атрибуты — столбцами этой таблицы.

Здесь хорошо видно, что часть данных дублируется (эти фрагменты выделены на рисунке). Дублирование приводит к сложностям с обеспечением согласованности — а это едва ли не главная задача СУБД.

Например, каждая из двух строк, относящихся к книге 3, должна содержать общее количество (7 штук). Что нужно сделать, чтобы отразить покупку книги? С одной стороны, надо добавить строки для отражения операции покупки (а сколько строк добавлять, еще две?). С другой, во всех строках надо уменьшить количество с 7 до 6. А что делать, если в результате какой-нибудь ошибки значение количества в одной из этих строк будет отличаться от значения в другой строке? Как сформулировать ограничение целостности, которое запрещает такую ситуацию?

Также усложняются и многие запросы. Как найти общее число книг? Как найти всех уникальных авторов?

Итак, такая схема плохо работает для реляционных баз данных.

Схема данных (вариант)

```
 entity | attribute  | value
--------+------------+------------------------
      1 | title      | Муму
      1 | author     | Тургенев Иван Сергеевич
      1 | qty        | 10
      1 | operation  | +11
      1 | operation  | -1
      2 | title      | Отцы и дети
      2 | author     | Тургенев Иван Сергеевич
      2 | qty        | 4
      2 | operation  | +4
    ... | ...        | ...
```

Данные без схемы

- поддержка согласованности на стороне приложения
- сложно писать запросы
- низкая производительность (множественные соединения)

Другой вариант отображения сущности в таблицу — так называемая схема EAV: сущность — атрибут — значение. Она позволяет уложить вообще все, что угодно, в одну-единственную таблицу. Формально мы получаем реляционную базу данных, а фактически здесь отсутствует схема и СУБД не может гарантировать согласованность данных. Поддержка согласованности данных целиком ложится на приложение, что, безусловно, рано или поздно приведет к ее нарушению.

В такой схеме сложно писать запросы (хотя и довольно просто их генерировать), и в результате работа со сколько-нибудь большим объемом данных становится проблемой из-за постоянных многократных соединений таблицы самой с собой.

Однако следует отметить, что такой подход имеет и плюсы. Например, очевидно, что такую «схему» данных легко наращивать: при добавлении новой сущности или атрибута мы просто добавляем в нашу таблицу новые строки.

И все-таки так делать не стоит.

Схема данных (вариант)

```
 book_id |                        description
---------+-----------------------------------------------------------
       1 | {"title": "Муму",
         |  "authors": [ "Тургенев Иван Сергеевич" ],
         |  "qty": 10,
         |  "operations": [ +11, -1 ]}
       3 | {"title": "Трудно быть богом",
         |  "authors": [ "Стругацкий Аркадий Натанович",
         |               "Стругацкий Борис Натанович" ],
         |  "qty": 7,
         |  "operations": [ +7 ]}
     ... | ...
```

Данные без схемы

- поддержка согласованности на стороне приложения
- сложно писать запросы (специальный язык)
- индексная поддержка есть

Еще одна вариация на ту же тему — представление данных в виде JSON, в духе NoSQL. По сути, тут применимы все замечания, сделанные ранее.

Кроме того, запросы к такой структуре придется писать не на SQL, а используя какой-то специализированный язык (раньше скорее всего выбор пал бы на jsQuery, а в PostgreSQL 16 удобно пользоваться возможностями SQL/JSONPath, определенными в стандарте SQL:2016).

https://github.com/postgrespro/jsquery

https://postgrespro.ru/docs/postgresql/16/functions-json#FUNCTIONS-SQLJSON-PATH

Хотя у PostgreSQL есть индексная поддержка JSON, производительность все равно под вопросом.

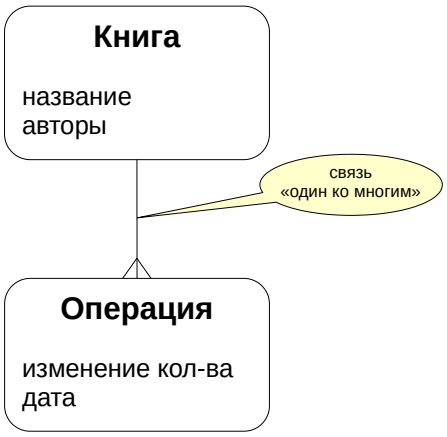
Такую схему удобно применять, когда от базы данных требуется только получение JSON по идентификатору, но не требуется серьезная работа с данными *внутри* JSON. Это не наш случай.

(Разумеется, это не категоричное утверждение. См. последнее задание в практике.)

Книги и операции

Нормализация — уменьшение избыточности данных

разбиение крупных сущностей на более мелкие



Итак, нам требуется устранить избыточность данных, чтобы привести их в вид, удобный для обработки в реляционной СУБД. Этот процесс называется нормализацией.

Возможно, вы знакомы с понятиями *нормальных форм* (первая, вторая, третья, Бойса-Кодда и т. д.). Мы не будем говорить о них; на неформальном уровне достаточно понимать, что весь этот математический аппарат преследует одну-единственную цель: устранение избыточности.

Средство для уменьшения избыточности — разбиение большой сущности на несколько меньших. Как именно это сделать, подскажет здравый смысл (который все равно нельзя заменить одним только знанием нормальных форм).

В нашем случае все достаточно просто. Давайте начнем с отделения книг от операций. Эти две сущности связаны отношением «один ко многим»: каждая книга может иметь несколько операций, но каждая операция относится только к одной книге.

Схема данных

books

book_id	title	author
1	Муму	Тургенев Иван Сергеевич
2	Отцы и дети	Тургенев Иван Сергеевич
3	Трудно быть богом	Стругацкий Аркадий Натанович
3	Трудно быть богом	Стругацкий Борис Натанович

operations

operation_id	book_id	qty_change	date_created
1	1	+10	2020-07-13
2	1	-1	2020-08-25
3	3	+7	2020-07-13
4	2	+4	2020-07-13

На уровне структуры базы данных это может быть представлено двумя таблицами: книги (books) и операции (operations).

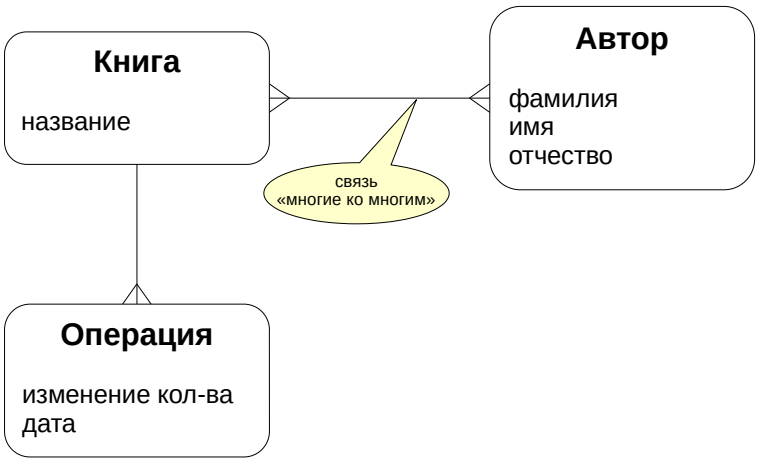
Операция состоит в изменении количества книг (положительном при заказе; отрицательном при покупке). Заметьте, что теперь у книги нет атрибута «количество». Вместо этого достаточно просуммировать изменения количества по всем операциям, относящимся к данной книге. Дополнительный атрибут «количество» у книги снова привел бы к избыточности данных.

Не исключено, что такое решение вас насторожило. Удобно ли вместо простого обращения к полю подсчитывать сумму? Но мы можем создать представление, которое для каждой книги будет показывать количество. Это не приведет к появлению избыточности, поскольку представление — это всего лишь запрос.

Второй момент — производительность. Если подсчет суммы приведет к ухудшению производительности, мы можем выполнить обратный процесс — денормализацию: добавить в таблицу книг поле «количество» и обеспечить его согласованность с таблицей операций. Надо ли этим заниматься — вопрос, ответ на который выходит за рамки этого курса (он рассматривается в курсе QPT «Оптимизация запросов»). Но из общих соображений понятно, что для нашего «игрушечного» магазина это не требуется. Однако мы еще вернемся к вопросу денормализации в теме «Триггеры».

Итак, как видно на слайде, выделение операций в отдельную сущность решило часть проблем дублирования, но не все.

Книги, авторы и операции



Поэтому надо сделать еще один шаг: отделить авторов от книг и связать их соотношением «многие ко многим»: и каждая книга может быть написана несколькими авторами, и каждый автор может написать несколько книг. На уровне таблиц такая связь реализуется с помощью дополнительной промежуточной таблицы.

Атрибутами автора можно сделать фамилию, имя и отчество. Это имеет смысл, поскольку нам может потребоваться работать отдельно с каждым из этих атрибутов, например, выводить фамилию и инициалы.

Схема данных приложения

```
student$ psql bookstore
```

```
=> \c bookstore
```

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

Итак, схема данных приложения состоит из четырех таблиц:

```
=> \dt
```

```
            List of relations
  Schema   |    Name    | Type  |  Owner
-----------+------------+-------+---------
 bookstore | authors    | table | student
 bookstore | authorship | table | student
 bookstore | books      | table | student
 bookstore | operations | table | student
(4 rows)
```

Книги

```
=> \d books
```

```
                        Table "bookstore.books"
 Column  |  Type   | Collation | Nullable |           Default
---------+---------+-----------+----------+-----------------------------
 book_id | integer |           | not null | generated always as identity
 title   | text    |           | not null |
Indexes:
    "books_pkey" PRIMARY KEY, btree (book_id)
Referenced by:
    TABLE "authorship" CONSTRAINT "authorship_book_id_fkey" FOREIGN KEY (book_id)
REFERENCES books(book_id)
    TABLE "operations" CONSTRAINT "operations_book_id_fkey" FOREIGN KEY (book_id)
REFERENCES books(book_id)
```

Здесь мы используем типы данных:

- integer — целое число;
- text — символьный, текстовая строка произвольной длины.

А также используем ограничение целостности:

- PRIMARY KEY — первичный ключ.

Конструкция GENERATED AS IDENTITY служит для автоматической генерации уникальных значений (в версиях до 10 для этого использовался псевдотип serial).

Строки столбцов, объявленных как GENERATED AS IDENTITY, получают значения из специальных объектов базы данных — последовательностей. Имя использованной последовательности можно узнать так:

```
=> SELECT pg_get_serial_sequence('books','book_id');
```

```
   pg_get_serial_sequence
----------------------------
 bookstore.books_book_id_seq
(1 row)
```

Последовательности можно при необходимости создавать вручную, а также обращаться к ним непосредственно:

```
=> SELECT nextval('books_book_id_seq');
```

```
 nextval
---------
       7
(1 row)
```

Последовательность — самый эффективный способ генерации уникальных номеров. Но следует иметь в виду, что последовательность не гарантирует:

- отсутствия пропусков в нумерации (так как изменение происходит не транзакционно);

- монотонного возрастания номеров (если используется кеширование значений в сеансах).

Вот какие данные находятся в таблице книг:

```
=> SELECT * FROM books \gx
```

```
-[ RECORD 1
]----------------------------------------------------------------------------------------------
----------------------------------------------------
book_id | 1
title   | Сказка о царе Салтане
-[ RECORD 2
]----------------------------------------------------------------------------------------------
----------------------------------------------------
book_id | 2
title   | Муму
-[ RECORD 3
]----------------------------------------------------------------------------------------------
----------------------------------------------------
book_id | 3
title   | Трудно быть богом
-[ RECORD 4
]----------------------------------------------------------------------------------------------
----------------------------------------------------
book_id | 4
title   | Война и мир
-[ RECORD 5
]----------------------------------------------------------------------------------------------
----------------------------------------------------
book_id | 5
title   | Путешествия в некоторые удаленные страны мира в четырех частях: сочинение
Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей
-[ RECORD 6
]----------------------------------------------------------------------------------------------
----------------------------------------------------
book_id | 6
title   | Хрестоматия
```

Обратите внимание, что названия могут быть длинными. Мы используем расширенный формат вывода psql, а появившийся в PostgreSQL 16 параметр xheader_width поможет обрезать строку заголовка "широкого" столбца:

```
=> \pset xheader_width 40
```

```
Expanded header width is 40.
```

Сравните вывод:

```
=> SELECT * FROM books \gx
```

```
-[ RECORD 1 ]--------------------------
book_id | 1
title   | Сказка о царе Салтане
-[ RECORD 2 ]--------------------------
book_id | 2
title   | Муму
-[ RECORD 3 ]--------------------------
book_id | 3
title   | Трудно быть богом
-[ RECORD 4 ]--------------------------
book_id | 4
title   | Война и мир
-[ RECORD 5 ]--------------------------
book_id | 5
title   | Путешествия в некоторые удаленные страны мира в четырех частях: сочинение
Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей
-[ RECORD 6 ]--------------------------
book_id | 6
title   | Хрестоматия
```

Авторы

```
=> \d authors
```

```
                        Table "bookstore.authors"
   Column    |  Type   | Collation | Nullable |           Default
-------------+---------+-----------+----------+-----------------------------
 author_id   | integer |           | not null | generated always as identity
 last_name   | text    |           | not null |
 first_name  | text    |           | not null |
 middle_name | text    |           |          |
Indexes:
    "authors_pkey" PRIMARY KEY, btree (author_id)
Referenced by:
    TABLE "authorship" CONSTRAINT "authorship_author_id_fkey" FOREIGN KEY (author_id)
REFERENCES authors(author_id)
```

Здесь дополнительно используется ограничение целостности:

- NOT NULL — обязательность, то есть недопустимость неопределенных значений.

```
=> SELECT * FROM authors;
```

```
 author_id | last_name  | first_name | middle_name
-----------+------------+------------+-------------
         1 | Пушкин     | Александр  | Сергеевич
         2 | Тургенев   | Иван       | Сергеевич
         3 | Стругацкий | Борис      | Натанович
         4 | Стругацкий | Аркадий    | Натанович
         5 | Толстой    | Лев        | Николаевич
         6 | Свифт      | Джонатан   |
(6 rows)
```

Обратите внимание, что отчество может отсутствовать (или быть заданным пустой строкой).

Ограничение PRIMARY KEY в выводе команды \d упоминалось вместе со словами «индекс» и «btree».

Btree (B-дерево) — основной тип индекса, используемый в базах данных для ускорения поиска и для поддержки ограничений целостности (первичного ключа и уникальности).

Представим себе, что в магазине продаются книги миллиона авторов-однофамильцев:

```
=> BEGIN; -- начнем транзакцию, чтобы откатить потом изменения
```

```
BEGIN
```

```
=> INSERT INTO authors(first_name, last_name)
    SELECT 'Графоман', 'Графоманов' FROM generate_series(1, 1_000_000);
```

```
INSERT 0 1000000
```

Сколько времени занимает поиск одного автора в такой таблице?

```
=> \timing on
```

```
Timing is on.
```

```
=> SELECT * FROM authors WHERE last_name = 'Пушкин';
```

```
 author_id | last_name | first_name | middle_name
-----------+-----------+------------+-------------
         1 | Пушкин    | Александр  | Сергеевич
(1 row)

Time: 72,795 ms
```

```
=> \timing off
```

```
Timing is off.
```

Если попросить оптимизатор показать план запроса, мы увидим в нем Seq Scan — последовательное сканирование всей таблицы в поисках нужного значения (Filter):

```
=> EXPLAIN (costs off)
SELECT * FROM authors WHERE last_name = 'Пушкин';
```

```
               QUERY PLAN
-----------------------------------------
 Seq Scan on authors
   Filter: (last_name = 'Пушкин'::text)
(2 rows)
```

А если искать по полю, которое проиндексировано?

```
=> \timing on
```

```
Timing is on.
```

```
=> SELECT * FROM authors WHERE author_id = 1;
```

```
 author_id | last_name | first_name | middle_name
-----------+-----------+------------+-------------
         1 | Пушкин    | Александр  | Сергеевич
(1 row)

Time: 0,225 ms
```

```
=> \timing off
```

```
Timing is off.
```

Время уменьшилось на несколько порядков.

А в плане запроса появился индекс:

```
=> EXPLAIN (costs off)
SELECT * FROM authors WHERE author_id = 1;
```

```
                QUERY PLAN
----------------------------------------
 Index Scan using authors_pkey on authors
   Index Cond: (author_id = 1)
(2 rows)
```

Можно создать индекс и по фамилии (и проанализировать таблицу, чтобы собрать актуальную статистику):

```
=> ANALYZE authors;
```

```
ANALYZE
```

```
=> CREATE INDEX ON authors(last_name);
```

```
CREATE INDEX
```

```
=> EXPLAIN (costs off)
SELECT * FROM authors WHERE last_name = 'Пушкин';
```

```
                    QUERY PLAN
--------------------------------------------------
 Index Scan using authors_last_name_idx on authors
   Index Cond: (last_name = 'Пушкин'::text)
(2 rows)
```

Однако индекс не является универсальным средством увеличения производительности. Обычно индекс очень полезен, если из таблицы требуется выбрать небольшую долю всех имеющихся строк. Если нужно прочитать много данных, индекс будет только мешать, и оптимизатор это понимает:

```
=> EXPLAIN (costs off)
SELECT * FROM authors WHERE last_name = 'Графоманов';
```

```
                 QUERY PLAN
-----------------------------------------
 Seq Scan on authors
   Filter: (last_name = 'Графоманов'::text)
(2 rows)
```

Кроме того, надо учитывать накладные расходы на обновление индексов при изменении таблицы и занимаемое ими место на диске.

Отменим все наши изменения (включая создание индекса):

```
=> ROLLBACK;
```

```
ROLLBACK
```

```
=> ANALYZE authors;
```

```
ANALYZE
```

Авторства

С помощью этой таблицы реализуется связь «многие ко многим».

```
=> \d authorship
```

```
            Table "bookstore.authorship"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 book_id   | integer |           | not null |
 author_id | integer |           | not null |
 seq_num   | integer |           | not null |
Indexes:
    "authorship_pkey" PRIMARY KEY, btree (book_id, author_id)
Foreign-key constraints:
    "authorship_author_id_fkey" FOREIGN KEY (author_id) REFERENCES authors(author_id)
    "authorship_book_id_fkey" FOREIGN KEY (book_id) REFERENCES books(book_id)
```

Здесь к уже использованным ограничениям добавляется ограничение ссылочной целостности:

- FOREIGN KEY — внешний ключ.

Фактически таблица содержит два внешних ключа: один ссылается на таблицу книг, другой — на таблицу авторов.

Столбец seq_num определяет последовательность, в которой должны перечисляться авторы одной книги, если их несколько.

Также обратите внимание на то, что первичный ключ — составной.

```
=> SELECT * FROM authorship;
```

```
 book_id | author_id | seq_num
---------+-----------+---------
       1 |         1 |       1
       2 |         2 |       1
       3 |         3 |       2
       3 |         4 |       1
       4 |         5 |       1
       5 |         6 |       1
       6 |         1 |       1
       6 |         5 |       2
       6 |         2 |       3
(9 rows)
```

Операции

```
=> \d operations
```

```
                        Table "bookstore.operations"
    Column    |  Type   | Collation | Nullable |           Default
--------------+---------+-----------+----------+------------------------------
 operation_id | integer |           | not null | generated always as identity
 book_id      | integer |           | not null |
 qty_change   | integer |           | not null |
 date_created | date    |           | not null | CURRENT_DATE
Indexes:
    "operations_pkey" PRIMARY KEY, btree (operation_id)
Foreign-key constraints:
    "operations_book_id_fkey" FOREIGN KEY (book_id) REFERENCES books(book_id)
```

В этой таблице используется еще один тип данных:

- date — дата (без указания времени).

Для столбца date_created указано значение по умолчанию (DEFAULT) — текущая дата.

```
=> SELECT * FROM operations;
```

```
 operation_id | book_id | qty_change | date_created
--------------+---------+------------+--------------
            1 |       1 |         10 | 2024-07-05
            2 |       1 |         10 | 2024-07-05
            3 |       1 |         -1 | 2024-07-05
(3 rows)
```

Кроме тех типов данных, которые используются в таблицах приложения, мы постоянно будем встречаться с логическим типом (boolean). Например, такой тип имеют логические выражения в условии WHERE.

Важно помнить, что, в отличие от традиционных языков программирования, SQL использует трехзначную логику: кроме значений true и false, логическое значение может принимать неопределенное значение NULL (которое можно понимать как «значение неизвестно»).

В примерах встречаются и некоторые другие типы данных, информацию о работе с ними можно найти в материалах

курса («Основные типы данных и функции», datatypes.pdf).

Кроме того, в курсе мы подробно рассмотрим более сложные типы:

- составной — записи, аналогичные строкам таблиц (в теме «SQL. Составные типы»);
- массивы (в теме «PL/pgSQL. Массивы»).

Организация интерфейса

Таблицы и триггеры

чтение данных напрямую из таблицы (представления);
запись данных напрямую в таблицу (представление),
плюс триггеры для изменения связанных таблиц

приложение должно быть в курсе модели данных,
максимальная гибкость

сложно поддерживать согласованность

Функции

чтение данных из табличных функций;
запись данных через вызов функций

приложение отделено от модели данных и ограничено API

большой объем работы по изготовлению функций-оберток,
потенциальные проблемы с производительностью

Есть несколько способов организации интерфейса между клиентской и серверной частями приложения.

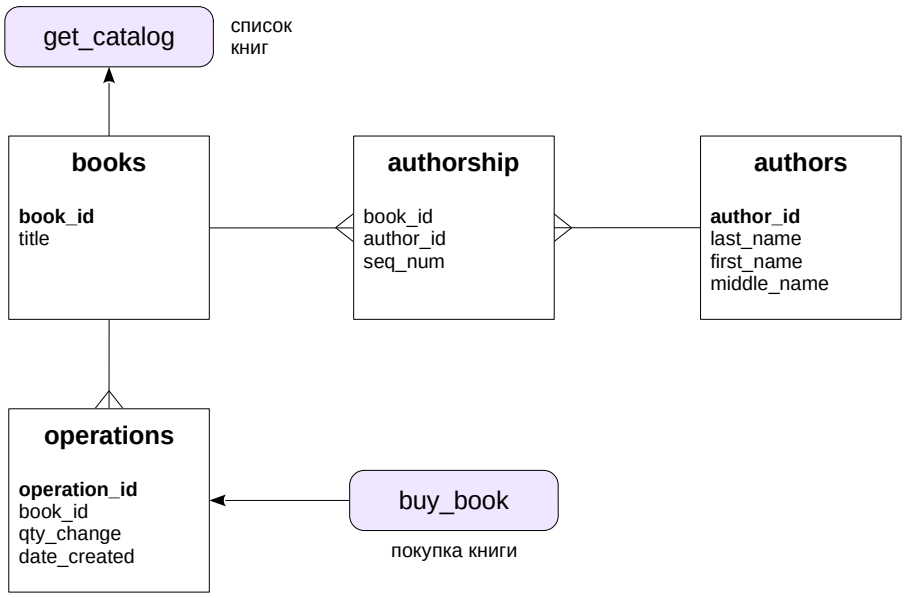
Один вариант — разрешить приложению напрямую обращаться к таблицам в базе данных и изменять их. При этом от приложения требуется детальное «знание» модели данных. Отчасти это требование можно ослабить за счет использования представлений (view).

Кроме того, от приложения требуется и определенная дисциплина — иначе очень сложно поддержать согласованность данных, защищаясь на уровне БД от любых возможных некорректных действий приложения. Но в этом случае достигается максимальная гибкость.

Другой вариант — запретить приложению доступ к таблицам и разрешить только вызовы функций. Чтение данных можно организовать с помощью табличных функций (которые возвращают набор строк). Изменение данных можно выполнять, вызывая другие функции и передавая им необходимые данные. В этом случае внутри функций можно реализовать все необходимые проверки согласованности — база данных будет защищена, но приложение сможет пользоваться только предоставленным и ограниченным набором возможностей. Такой вариант требует написания большого количества функций-оберток и может привести к потере производительности.

Вполне возможны и промежуточные варианты. Например, разрешить чтение данных непосредственно из таблиц, а изменение выполнять только через вызов функций.

Интерфейс покупателя

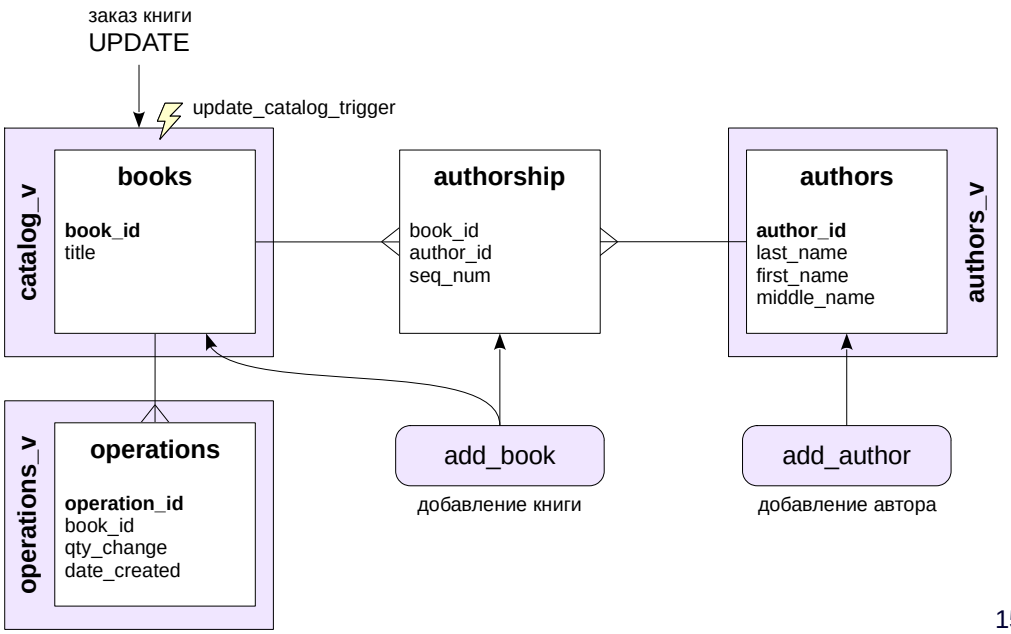


В нашем приложении мы попробуем разные варианты организации интерфейса (хотя в реальной жизни обычно лучше систематически придерживаться какого-то одного подхода).

Магазин будет использовать интерфейсные функции:

- для поиска книг — get_catalog (тема «SQL. Составные типы»);
- для покупки книг — buy_book («PL/pgSQL. Выполнение запросов»).

Интерфейс сотрудника



«Админка» для получения данных будет использовать представления (которые мы создадим в практике к этой теме):

- список книг — catalog_v;
- список авторов — authors_v;
- список операций — operations_v.

Добавление автора будет выполнять функция add_author (создадим ее в теме «PL/pgSQL. Выполнение запросов»), добавление книги — функция add_book («PL/pgSQL. Массивы»).

Для заказа книг сделаем представление catalog_v обновляемым («PL/pgSQL. Триггеры»).

Представления

Представление — запрос, у которого есть имя. Например, можно создать представление, которое показывает только авторов без отчества:

```
=> CREATE VIEW authors_no_middle_name AS
    SELECT author_id, first_name, last_name
    FROM authors
    WHERE nullif(middle_name,'') IS NULL;
```

```
CREATE VIEW
```

Теперь имя представления можно использовать в запросах практически так же, как и таблицу:

```
=> SELECT * FROM authors_no_middle_name;
```

```
 author_id | first_name | last_name
-----------+------------+-----------
         6 | Джонатан   | Свифт
(1 row)
```

В простом случае с представлением будут работать и другие операции, например:

```
=> UPDATE authors_no_middle_name SET last_name = initcap(last_name);
```

```
UPDATE 1
```

С помощью триггеров можно сделать так, чтобы и в сложных случаях для представлений работали вставка, обновление и удаление строк. Мы рассмотрим это в теме «PL/pgSQL. Триггеры».

При планировании запроса представление «разворачивается» до базовых таблиц:

```
=> EXPLAIN (costs off)
SELECT * FROM authors_no_middle_name;
```

```
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on authors
   Filter: (NULLIF(middle_name, ''::text) IS NULL)
(2 rows)
```

Приложение использует три представления. Сначала они будут очень простыми, но в следующих темах мы перенесем в них часть логики приложения.

Представление для авторов — конкатенация фамилии, имени и отчества (если оно есть):

```
=> SELECT * FROM authors_v;
```

```
 author_id |         display_name
-----------+------------------------------
         1 | Пушкин Александр Сергеевич
         2 | Тургенев Иван Сергеевич
         3 | Стругацкий Борис Натанович
         4 | Стругацкий Аркадий Натанович
         5 | Толстой Лев Николаевич
         6 | Свифт Джонатан
(6 rows)
```

Представление для каталога книг — пока просто название книги:

```
=> SELECT * FROM catalog_v \gx
```

```
-[ RECORD 1 ]+-----------------------------------------------------
book_id      | 1
display_name | Сказка о царе Салтане
-[ RECORD 2 ]+-----------------------------------------------------
book_id      | 2
display_name | Муму
-[ RECORD 3 ]+-----------------------------------------------------
book_id      | 3
display_name | Трудно быть богом
-[ RECORD 4 ]+-----------------------------------------------------
book_id      | 4
display_name | Война и мир
-[ RECORD 5 ]+-----------------------------------------------------
book_id      | 5
display_name | Путешествия в некоторые удаленные страны мира в четырех частях: сочинение
Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей
-[ RECORD 6 ]+-----------------------------------------------------
book_id      | 6
display_name | Хрестоматия
```

Представление для операций — дополнительно определяет тип операции (поступление или покупка):

```
=> SELECT * FROM operations_v;

 book_id |   op_type   | qty_change | date_created
---------+-------------+------------+--------------
       1 | Поступление |         10 | 05.07.2024
       1 | Поступление |         10 | 05.07.2024
       1 | Покупка     |          1 | 05.07.2024
(3 rows)
```

Итоги

Проектирование баз данных — отдельная тема

- теория важна, но не заменяет здравый смысл

Отсутствие избыточности в данных делает работу удобнее и упрощает поддержку согласованности

Для клиент-серверного интерфейса можно использовать таблицы, представления, функции, триггеры

Практика

1. В базе данных bookstore создайте схему bookstore. Настройте путь поиска к этой схеме на уровне подключения к БД.
2. В схеме bookstore создайте таблицы books, authors, authorship и operations с необходимыми ограничениями целостности так, чтобы они соответствовали показанным в демонстрации.
3. Вставьте в таблицы данные о нескольких книгах. Проверьте себя с помощью запросов.
4. В схеме bookstore создайте представления authors_v, catalog_v и operations_v так, чтобы они соответствовали показанным в демонстрации.
   Проверьте, что приложение стало показывать данные на вкладках «Книги», «Авторы» и «Каталог».

1. Вспомните материал темы «Организация данных. Логическая структура».

2. Ориентируйтесь на показанный в демонстрации вывод команд \d утилиты psql.

3. Вы можете использовать те же данные, что были показаны в демонстрации, или придумать свои собственные.

4. Попробуйте написать запросы к базовым таблицам, возвращающие тот же результат, что и показанные в демонстрации запросы к представлениям. Затем оформите запросы в виде представлений.

После выполнения практики обязательно сверьте свои запросы с решением, приведенным к этой теме. При необходимости внесите коррективы.

1. Схема и путь поиска

```
=> CREATE DATABASE bookstore;
```

```
CREATE DATABASE
```

```
=> \c bookstore
```

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

```
=> CREATE SCHEMA bookstore;
```

```
CREATE SCHEMA
```

```
=> ALTER DATABASE bookstore SET search_path = bookstore, public;
```

```
ALTER DATABASE
```

```
=> \c
```

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

```
=> SHOW search_path;
```

```
    search_path
------------------
 bookstore, public
(1 row)
```

2. Таблицы

Авторы:

```
=> CREATE TABLE authors(
    author_id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    last_name text NOT NULL,
    first_name text NOT NULL,
    middle_name text
);
```

```
CREATE TABLE
```

Книги:

```
=> CREATE TABLE books(
    book_id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    title text NOT NULL
);
```

```
CREATE TABLE
```

Авторство:

```
=> CREATE TABLE authorship(
    book_id integer REFERENCES books,
    author_id integer REFERENCES authors,
    seq_num integer NOT NULL,
    PRIMARY KEY (book_id,author_id)
);
```

```
CREATE TABLE
```

Операции:

```
=> CREATE TABLE operations(
    operation_id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    book_id integer NOT NULL REFERENCES books,
    qty_change integer NOT NULL,
    date_created date NOT NULL DEFAULT current_date
);
```

```
CREATE TABLE
```

3. Данные

Авторы:

```
=> INSERT INTO authors(last_name, first_name, middle_name)
VALUES
    ('Пушкин', 'Александр', 'Сергеевич'),
    ('Тургенев', 'Иван', 'Сергеевич'),
    ('Стругацкий', 'Борис', 'Натанович'),
    ('Стругацкий', 'Аркадий', 'Натанович'),
    ('Толстой', 'Лев', 'Николаевич'),
    ('Свифт', 'Джонатан', NULL);
```

```
INSERT 0 6
```

Книги:

```
=> INSERT INTO books(title)
VALUES
    ('Сказка о царе Салтане'),
    ('Муму'),
    ('Трудно быть богом'),
    ('Война и мир'),
    ('Путешествия в некоторые удаленные страны мира в четырех частях: сочинение Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей'),
    ('Хрестоматия');
```

```
INSERT 0 6
```

Авторство:

```
=> INSERT INTO authorship(book_id, author_id, seq_num)
VALUES
    (1, 1, 1),
    (2, 2, 1),
    (3, 3, 2),
    (3, 4, 1),
    (4, 5, 1),
    (5, 6, 1),
    (6, 1, 1),
    (6, 5, 2),
    (6, 2, 3);
```

```
INSERT 0 9
```

Операции:

```
=> INSERT INTO operations(book_id, qty_change)
VALUES
    (1, 10),
    (1, 10),
    (1, -1);
```

```
INSERT 0 3
```

4. Представления

Представление для авторов:

```
=> CREATE VIEW authors_v AS
SELECT a.author_id,
       a.last_name || ' ' ||
       a.first_name ||
       coalesce(' ' || nullif(a.middle_name, ''), '') AS display_name
FROM   authors a;
```

```
CREATE VIEW
```

Представление для каталога:

```
=> CREATE VIEW catalog_v AS
SELECT b.book_id,
       b.title AS display_name
FROM   books b;
```

```
CREATE VIEW
```

Представление для операций:

```
=> CREATE VIEW operations_v AS
SELECT book_id,
       CASE
           WHEN qty_change > 0 THEN 'Поступление'
           ELSE 'Покупка'
       END op_type,
       abs(qty_change) qty_change,
       to_char(date_created, 'DD.MM.YYYY') date_created
FROM   operations
ORDER BY operation_id;
```

```
CREATE VIEW
```

Практика+

1. Какие дополнительные атрибуты могут появиться у выделенных сущностей при развитии приложения?
2. Допустим, требуется хранить информацию об издательстве. Дополните ER-диаграмму и отобразите ее в таблицы.
3. Некоторые книги могут входить в серии (например, «Библиотека приключений»). Как изменится схема данных?
4. Пусть наш магазин стал торговать компьютерными комплектующими (материнскими платами, процессорами, памятью, жесткими дисками, мониторами и т. п.). Какие сущности и какие атрибуты вы бы выделили? Учтите, что на рынке постоянно появляются новые типы оборудования со своими характеристиками.

3. Разные издательства вполне могут иметь серии, названные одинаково.

1. Дополнительные атрибуты

Несколько примеров:

- Авторы: роль (автор, редактор, переводчик и т. п.);
- Книги: аннотация;
- Операции: текущий статус (оплачено, передано в службу доставки и т. п.).

2. Издательства

Надо добавить сущность «Издательство» с атрибутом «Название» (как минимум).

Книги связаны с издательствами отношением «многие ко многим»: книга может публиковаться в разных издательствах. Поэтому на физическом уровне потребуется промежуточная таблица «Публикации» с атрибутом «Год издания».

(Разумеется, это упрощенная модель; при желании ее можно уточнять еще очень долго.)

3. Серии

Добавим сущность «Серия». К серии относится не сама книга, а ее конкретная публикация, так что имеет смысл вывести «Публикацию» на уровень ER-модели и связать ее с серий отношением «один ко многим» (каждая публикация принадлежит к одной серии, каждая серия может включать несколько публикаций).

Также серия связана отношением «один ко многим» с издательством (у издательства может быть несколько серий, а каждая серия принадлежит конкретному издательству).

Остается вопрос о внесерийных изданиях. Его можно решить либо введением фиктивной серии «Без серии», либо возможностью не указывать для публикации внешний ключ серии.

4. Компьютерные комплектующие

Рассматривая каждый конкретный тип комплектующих, можно без труда выделить необходимые атрибуты. Какие-то атрибуты будут общими (скажем, фирма-производитель и название модели), а какие-то будут иметь смысл только для данного конкретного типа. Например:

- Процессор: частота;
- Монитор: диагональ, разрешение;
- Жесткий диск: типоразмер, емкость.

Проблема в том, что рынок комплектующих очень динамичен. Некоторое время назад жесткие диски определялись частотой вращения и емкостью, а сейчас важен тип (твердотельный, вращающийся, гибридный). Для мониторов во времена ЭЛТ была важна частота обновления, а сейчас важен тип матрицы. Дисководы уже никому не нужны, зато появились флеш-накопители. И так далее.

Таким образом, либо придется постоянно изменять схему данных (а, значит, и постоянно изменять приложение, которое работает с этими данными!), либо искать более универсальную модель за счета отказа от жесткой структуры и контроля согласованности. Некоторые универсальные модели (например, хранение части данных в JSON) мы затрагивали в презентации.

SQL
Функции

Темы

Функции и их особенности в базах данных

Параметры и возвращаемое значение

Способы передачи параметров при вызове

Категории изменчивости и оптимизация

Функции в базе данных

Основной мотив: упрощение задачи

- интерфейс (параметры) и реализация (тело функции)
- о функции можно думать вне контекста всей задачи

	Традиционные языки	*PostgreSQL*
побочные эффекты	глобальные переменные	вся база данных (категории изменчивости)
модули	со своим интерфейсом и реализацией	пространства имен, клиент и сервер
сложности	накладные расходы на вызов (подстановка)	скрытие запроса от планировщика (подстановка, подзапросы, представления)

Основная цель появления функций в программировании вообще — упростить решаемую задачу за счет ее декомпозиции на более мелкие подзадачи. Упрощение достигается за счет того, что о функции можно думать, абстрагировавшись от «большой» задачи. Для этого функция определяет четкий интерфейс с внешним миром (параметры и возвращаемое значение). Ее реализация (тело функции) может меняться; вызывающая сторона «не видит» этих изменений и не зависит от них. Этой идеальной ситуации может мешать глобальное состояние (глобальные переменные), и надо учитывать, что в случае БД таким состоянием является вся база данных.

В традиционных языках функции часто объединяются в модули (пакеты, классы для ООП и т. п.), имеющие собственный интерфейс и реализацию. Границы модулей могут проводиться более или менее произвольно. Для PostgreSQL есть жесткая граница между клиентской частью и серверной: серверный код работает с базой, клиентский — управляет транзакциями. Модули (пакеты) отсутствуют, есть только пространства имен.

Для традиционных языков единственный минус широкого использования функций состоит в накладных расходах на их вызов. Иногда его преодолевают с помощью подстановки (inlining) кода функции в вызывающую программу. Для БД последствия могут быть более серьезные: если в функцию выносится часть запроса, планировщик перестает видеть «общую картину» и не может построить хороший план. В некоторых случаях PostgreSQL умеет выполнять подстановку; альтернативные варианты — использование подзапросов или представлений.

Общие сведения

Объект базы данных

- определение хранится в системном каталоге

Основные составные части определения

- имя
- параметры
- тип возвращаемого значения
- тело

Доступны несколько языков, в том числе SQL

- код в виде строковой константы
- обычно интерпретируется при вызове

Вызывается в контексте выражения

Функции являются такими же объектами базы данных, как, например, таблицы и индексы. Определение функции сохраняется в системном каталоге; поэтому функции в базе данных называют *хранимыми*.

В PostgreSQL доступно большое количество стандартных функций, с некоторыми из них можно познакомиться в справочном материале «Основные типы данных и функции».

И, конечно, можно писать собственные функции на разных языках программирования. Материал этой темы относится к функциям на любом языке, но примеры будут использовать язык SQL.

Определение функции состоит из имени, необязательных параметров, типа возвращаемого значения и тела. Тело записывается в виде строковой константы, которая содержит код на выбранном языке. За счет этого определение функции выглядит одинаково независимо от выбранного языка. Тело-строка сохраняется в системном каталоге и интерпретируется каждый раз, когда функция вызывается. Начиная с версии PostgreSQL 14 для кода на SQL появилась возможность производить разбор заранее, в системном каталоге при этом сохраняется не исходный текст, а результат разбора. Еще один способ избежать интерпретации времени выполнения — написать функцию на языке Си, но в данном курсе эта тема не рассматривается.

Функция всегда вызывается в контексте какого-либо выражения. Например, в списке выражений команды SELECT, в условии WHERE, в ограничении целостности CHECK и т. п.

https://postgrespro.ru/docs/postgresql/16/sql-createfunction

https://postgrespro.ru/docs/postgresql/16/sql-syntax-calling-funcs

Функции без параметров

Вот простой пример функции без параметров, содержащей один оператор:

```
=> CREATE FUNCTION hello_world() -- имя и пустой список параметров
RETURNS text                     -- тип возвращаемого значения
AS $$ SELECT 'Hello, world!'; $$ -- тело
LANGUAGE sql;                    -- указание языка
```

```
CREATE FUNCTION
```

Тело удобно записывать в строке, заключенной в кавычки-доллары, как в приведенном примере. Иначе придется заботиться об экранировании кавычек, которые наверняка встретятся в теле функции. Сравните:

```
=> SELECT ' SELECT ''Hello, world!''; ';
```

```
         ?column?
--------------------------
  SELECT 'Hello, world!';
(1 row)
```

```
=> SELECT $$ SELECT 'Hello, world!'; $$;
```

```
         ?column?
--------------------------
  SELECT 'Hello, world!';
(1 row)
```

При необходимости кавычки-доллары могут быть вложенными. Для этого в каждой паре кавычек надо использовать разный текст между долларами:

```
=> SELECT $func$ SELECT $$Hello, world!$$; $func$;
```

```
          ?column?
----------------------------
  SELECT $$Hello, world!$$;
(1 row)
```

Функция вызывается в контексте выражения, например:

```
=> SELECT hello_world(); -- пустые скобки обязательны
```

```
  hello_world
---------------
 Hello, world!
(1 row)
```

Давайте взглянем на то, как тело функции хранится в системном каталоге.

```
=> \pset xheader_width 60
```

```
Expanded header width is 60.
```

```
=> SELECT proname, prosrc, prosqlbody FROM pg_proc
WHERE proname = 'hello_world' \gx
```

```
-[ RECORD 1 ]------------------------
proname    | hello_world
prosrc     |  SELECT 'Hello, world!';
prosqlbody |
```

Мы видим сохраненную в исходном виде тело-строку.

А теперь реализуем современную возможность пересоздать нашу функцию в другом виде — в стиле стандарта SQL. В нашем случае телом функции может быть один оператор вида RETURN <выражение>:

```
=> CREATE OR REPLACE FUNCTION hello_world() RETURNS text
LANGUAGE sql
RETURN 'Hello, world!';
```

```
CREATE FUNCTION
```

Снова заглянем в системный каталог — тело функции сохранено по-другому:

```
=> SELECT proname, prosrc, left(prosqlbody, 100) AS body FROM pg_proc WHERE proname = 'hello_world' \gx
```

```
-[ RECORD 1 ]-------------------------------------------
proname | hello_world
prosrc  |
body    | {QUERY :commandType 1 :querySource 0 :canSetTag true :utilityStmt <>
:resultRelation 0 :hasAggs fals
```

Исходный код в этом случае не хранится, получить его можно командой \sf:

```
=> \sf hello_world
```

```
CREATE OR REPLACE FUNCTION public.hello_world()
 RETURNS text
 LANGUAGE sql
RETURN 'Hello, world!'::text
```

В случае, если тело функции состоит из нескольких операторов SQL, в качестве результата возвращается значение из первой строки, которую вернул последний оператор. Если код такой функции написан в стиле стандарта SQL, потребуется использовать конструкцию BEGIN ATOMIC ... END, которая охватывает выполняемый блок операторов:

```
=> CREATE OR REPLACE FUNCTION hello_world() RETURNS text
LANGUAGE sql
BEGIN ATOMIC
  SELECT 'First Line';
  SELECT 'Second Line';
END;
```

```
CREATE FUNCTION
```

Пробуем вызов:

```
=> SELECT hello_world();
```

```
 hello_world
-------------
 Second Line
(1 row)
```

Обратите внимание на особенности синтаксиса стиля стандарта SQL — в отличие от традиционного «строчного»:

- нет конструкции AS, содержащей код функции в виде строки;
- может использоваться новое ключевое слово RETURN для возврата значения;
- указание LANGUAGE sql не является обязательным;
- при создании функции ее код разбирается, а результат разбора сохраняется в pg_proc.prosqlbody (в традиционной нотации текст функции сохраняется в pg_proc.prosrc).

Это лучше соответствует стандарту и в большей мере совместимо с другими реализациями SQL. Теперь при вызове функции ее команды заново не интерпретируются, а используется заранее разобранный вариант.

Не все операторы SQL можно использовать в функции. Запрещены:

- команды управления транзакциями (BEGIN, COMMIT, ROLLBACK и т. п.);
- служебные команды (такие, как VACUUM или CREATE INDEX).

Вот пример неправильной функции. Здесь мы использовали псевдотип void, который говорит о том, что функция не возвращает ничего.

```
=> CREATE FUNCTION do_commit() RETURNS void
LANGUAGE sql
BEGIN ATOMIC COMMIT; END;
```

```
ERROR:  COMMIT is not yet supported in unquoted SQL function body
```

Управлять транзакциями можно в процедурах, о чем мы будет говорить в следующей теме.

Функции с входными параметрами

Пример функции с одним параметром:

```
=> CREATE FUNCTION hello(name text) -- формальный параметр
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

При вызове функции мы указываем фактический параметр, соответствующий формальному:

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

При указании типа параметра можно указать и модификатор (например, varchar(10)), но он игнорируется.

Можно определить параметр функции без имени; тогда внутри тела функции на параметры придется ссылаться по номеру. Удалим функцию и создадим новую:

```
=> DROP FUNCTION hello(text); -- достаточно указать тип параметра

DROP FUNCTION

=> CREATE FUNCTION hello(text)
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || $1 || '!'; -- номер вместо имени

CREATE FUNCTION

=> SELECT hello('Alice');

     hello
---------------
 Hello, Alice!
(1 row)
```

Но так лучше не делать, это неудобно.

Удалим функцию и создадим заново, добавив еще два параметра — приветствие и обращение.

```
=> DROP FUNCTION hello(text);

DROP FUNCTION
```

Здесь мы используем необязательное ключевое слово IN, обозначающее входной параметр. Предложение DEFAULT позволяет определить значение по умолчанию для параметра:

```
=> CREATE FUNCTION hello(IN name text, IN greet text DEFAULT 'Dear', IN title text DEFAULT 'Mr')
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || greet || ' ' || title || ' ' || name || '!';

CREATE FUNCTION

=> SELECT hello('Alice', 'Charming', 'Mrs'); -- указаны второй и третий параметры

           hello
----------------------------
 Hello, Charming Mrs Alice!
(1 row)
```

Обратите внимание, что параметры со значениями по умолчанию должны идти в конце всего списка. При вызове функции значения фактических параметров, определенных как default, можно опускать, тогда остальные default-параметры, идущие в списке после, также получат значения по умолчанию

```
=> SELECT hello('Bob', 'Excellent'); -- указан только первый default-параметр

          hello
-------------------------
 Hello, Excellent Mr Bob!
(1 row)


=> SELECT hello('Bob'); -- опущены оба параметра, имеющие значение по умолчанию

        hello
--------------------
 Hello, Dear Mr Bob!
(1 row)
```

До сих пор мы вызывали функцию, указывая фактические параметры позиционным способом — в том порядке, в котором они определены при создании функции. Во многих стандартных функциях имена параметров не заданы, так что этот способ оказывается единственным.

Но если формальным параметрам даны имена, можно использовать их при указании фактических параметров. В этом случае параметры могут указываться в произвольном порядке:

```
=> SELECT hello(title => 'Mrs', name => 'Alice');
```

```
         hello
-----------------------
 Hello, Dear Mrs Alice!
(1 row)
```

Такой способ удобен, когда порядок параметров неочевиден, особенно если их много; также можно явно указать значение одного из параметров по умолчанию.

Можно совмещать оба способа: часть параметров (начиная с первого) указать позиционно, а оставшиеся — по имени:

```
=> SELECT hello('Alice', title => 'Mrs');
```

```
         hello
-----------------------
 Hello, Dear Mrs Alice!
(1 row)
```

Если функция должна возвращать неопределенное значение, когда хотя бы один из входных параметров не определен, ее можно объявить как строгую (STRICT). Тело функции при этом вообще не будет выполняться.

```
=> DROP FUNCTION hello(text, text, text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(IN name text, IN title text DEFAULT 'Mr')
RETURNS text
LANGUAGE sql STRICT
RETURN 'Hello, ' || title || ' ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice', NULL);
```

```
 hello
-------

(1 row)
```

Вход и выход

Входные значения

определяются параметрами с режимом IN и INOUT

Выходное значение

определяется либо предложением RETURNS,
либо параметрами с режимом INOUT и OUT

если одновременно указаны обе формы, они должны быть согласованы

Формальные параметры с режимом IN и INOUT считаются *входными*. Значения соответствующих фактических параметров должны быть указаны при вызове функции (либо должны быть определены значения по умолчанию).

Возвращаемое значение можно определить двумя способами:

- использовать предложение RETURNS для указания типа;
- определить *выходные* параметры с режимом INOUT или OUT.

Две эти формы записи эквивалентны. Например, функция с указанием RETURNS integer и функция с параметром OUT integer возвращают целое число.

Можно использовать и оба способа одновременно. В этом случае функция также будет возвращать *одно* целое число. Но при этом типы RETURNS и выходных параметров должны быть согласованы друг с другом.

Таким образом, нельзя написать функцию, которая будет возвращать одно значение, и при этом передавать другое значение в OUT-параметре — что позволяет большинство традиционных языков программирования.

Функции с выходными параметрами

Альтернативный способ вернуть значение — использовать выходной параметр.

```
=> DROP FUNCTION hello(text, text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(
    IN name text,
    OUT text -- имя можно не указывать, если оно не нужно
)
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Результат тот же самый.

Можно использовать и RETURNS, и OUT-параметр вместе — результат снова будет тем же:

```
=> DROP FUNCTION hello(text); -- OUT-параметры не указываем
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(IN name text, OUT text)
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Или даже так, использовав INOUT-параметр:

```
=> DROP FUNCTION hello(text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(INOUT name text)
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Обратите внимание, что, в отличие от многих языков программирования, фактическое значение, переданное SQL-функции в INOUT-параметре, никак не изменяется: мы передаем входное значение, а выходное возвращается функцией в качестве результата. Поэтому мы можем указать константу, хотя другие языки требовали бы переменную.

В то время как в RETURNS можно указать только одно значение, выходных параметров может быть несколько. Например:

```
=> DROP FUNCTION hello(text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(
    IN name text,
    OUT greeting text,
    OUT clock timetz)
LANGUAGE sql
RETURN ('Hello, ' || name || '!', current_time);
```

```
CREATE FUNCTION
```

Здесь возвращаемое RETURN выражение пришлось взять в скобки.

```
=> SELECT hello('Alice');
```

```
                hello
-------------------------------------
 ("Hello, Alice!",16:21:11.254097+03)
(1 row)
```

Действительно, наша функция вернула не одно значение, а сразу несколько.

Подробнее о такой возможности и составных типах мы будем говорить в теме «SQL. Составные типы».

Итоги

Можно создавать собственные функции
и использовать их так же, как и встроенные

Функции можно писать на разных языках, в том числе SQL

Изменчивость влияет на возможности оптимизации

Иногда функция на SQL может быть подставлена в запрос

Практика

1. Создайте функцию author_name для формирования имени автора. Функция принимает три параметра (фамилия, имя, отчество) и возвращает строку с фамилией и инициалами.

   Используйте эту функцию в представлении authors_v.

2. Создайте функцию book_name для формирования названия книги. Функция принимает два параметра (идентификатор книги и заголовок) и возвращает строку, составленную из заголовка и списка авторов в порядке seq_num.
   Имя каждого автора формируется функцией author_name.

   Используйте эту функцию в представлении catalog_v.

Проверьте изменения в приложении.

Напомним, что необходимые функции можно посмотреть в раздаточном материале «Основные типы данных и функции».

1. `FUNCTION author_name(`
```
    last_name text, first_name text, middle_name text
)
RETURNS text
```

Например: `author_name('Толстой','Лев','Николаевич') →`
`→ 'Толстой Л. Н.'`

3. `FUNCTION book_name(book_id integer, title text)`
```
RETURNS text
```

Например: `book_name(3,'Трудно быть богом') →`
`→ 'Трудно быть богом. Стругацкий А. Н., Стругацкий Б. Н.'`

Все инструменты позволяют «непосредственно» редактировать хранимые функции. Например, в psql есть команда \ef, открывающая текст функции в редакторе и сохраняющая изменения в базу.

Такой возможностью лучше не пользоваться (или как минимум не злоупотреблять). В нормально построенном процессе разработки весь код должен находиться в файлах под версионным контролем. При необходимости изменить функцию файл редактируется и выполняется (с помощью psql или средствами IDE). Если же менять определение функций сразу в БД, изменения легко потерять. (Вообще же вопрос организации процесса разработки намного сложнее и в курсе мы его не затрагиваем.)

1. Функция author_name

```
=> CREATE FUNCTION author_name(
    last_name text,
    first_name text,
    middle_name text
) RETURNS text
LANGUAGE sql IMMUTABLE
RETURN last_name || ' ' ||
       left(first_name, 1) || '.' ||
       CASE WHEN middle_name != '' -- подразумевает NOT NULL
           THEN ' ' || left(middle_name, 1) || '.'
           ELSE ''
       END;
```

```
CREATE FUNCTION
```

Категория изменчивости — immutable. Функция всегда возвращает одинаковое значение при одних и тех же входных параметрах.

```
=> CREATE OR REPLACE VIEW authors_v AS
SELECT a.author_id,
       author_name(a.last_name, a.first_name, a.middle_name) AS display_name
FROM   authors a
ORDER BY display_name;
```

```
CREATE VIEW
```

2. Функция book_name

```
=> CREATE FUNCTION book_name(book_id integer, title text)
RETURNS text
LANGUAGE sql STABLE
RETURN (
SELECT title || '. ' ||
       string_agg(
           author_name(a.last_name, a.first_name, a.middle_name), ', '
           ORDER BY ash.seq_num
       )
FROM   authors a
       JOIN authorship ash ON a.author_id = ash.author_id
WHERE  ash.book_id = book_name.book_id
);
```

```
CREATE FUNCTION
```

Категория изменчивости — stable. Функция возвращает одинаковое значение при одних и тех же входных параметрах, но только в рамках одного SQL-запроса.

```
=> CREATE OR REPLACE VIEW catalog_v AS
SELECT b.book_id,
       book_name(b.book_id, b.title) AS display_name
FROM   books b
ORDER BY display_name;
```

```
CREATE VIEW
```

Практика+

1. Напишите функцию, выдающую случайное время, равномерно распределенное в указанном отрезке.
   Начало отрезка задается временной отметкой (timestamptz), конец — либо временной отметкой, либо интервалом (interval).
2. В таблице хранятся номера автомобилей, введенные кое-как: встречаются как латинские, так и русские буквы в любом регистре; между буквами и цифрами могут быть пробелы. Считая, что формат номера «*буква три-цифры две-буквы*», напишите функцию, выдающую число уникальных номеров. Например, «K 123 XM» и «k123xm» считаются равными.
3. Напишите функцию, находящую корни квадратного уравнения.

Во всех заданиях обратите особое внимание на категорию изменчивости функций.

2. Сначала напишите функцию «нормализации» номера, то есть приводящую номер к какому-нибудь стандартному виду. Например, без пробелов и только заглавными латинскими буквами.

В номерах используются только 12 русских букв, имеющих латинские аналоги аналогичного начертания, а именно: АВЕКМНОРСТУХ.

3. Для уравнения вида $y = ax^2 + bx + c$ вычисляется дискриминант $D = b^2 - 4ac$:

- при $D > 0$ два корня $x_{1,2} = (-b \pm \sqrt{D}) / 2a$;
- при $D = 0$ один корень $x = -b / 2a$ (в качестве x_2 можно вернуть null);
- при $D < 0$ корней нет (оба корня null).

1. Случайная временна́я отметка

```
=> CREATE DATABASE sql_func;
```

```
CREATE DATABASE
```

```
=> \c sql_func
```

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

Функция с двумя временными отметками:

```
=> CREATE FUNCTION rnd_timestamp(t_start timestamptz, t_end timestamptz)
RETURNS timestamptz
LANGUAGE sql VOLATILE
RETURN t_start + (t_end - t_start) * random();
```

```
CREATE FUNCTION
```

Категория изменчивости — volatile. Используется функция random, поэтому функция будет возвращать разные значения при одних и тех же входных параметрах.

```
=> SELECT current_timestamp,
    rnd_timestamp(
        current_timestamp,
        current_timestamp + interval '1 hour'
    )
FROM generate_series(1,10);
```

```
       current_timestamp       |         rnd_timestamp
-------------------------------+-------------------------------
 2024-07-05 16:31:14.716763+03 | 2024-07-05 17:02:17.301088+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 17:07:26.010876+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 16:34:51.967748+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 17:25:05.538907+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 16:51:48.71589+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 16:48:45.241856+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 16:53:08.792748+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 16:58:49.581762+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 17:04:47.570722+03
 2024-07-05 16:31:14.716763+03 | 2024-07-05 17:26:12.353326+03
(10 rows)
```

Вторую функцию (с параметром-интервалом) можно определить через первую:

```
=> CREATE FUNCTION rnd_timestamp(t_start timestamptz, t_delta interval)
RETURNS timestamptz
LANGUAGE sql VOLATILE
RETURN rnd_timestamp(t_start, t_start + t_delta);
```

```
CREATE FUNCTION
```

```
=> SELECT rnd_timestamp(current_timestamp, interval '1 hour');
```

```
         rnd_timestamp
-------------------------------
 2024-07-05 16:40:09.956116+03
(1 row)
```

2. Автомобильные номера

Создадим таблицу с номерами.

```
=> CREATE TABLE cars(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    regnum text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO cars(regnum) VALUES
    ('K 123 XM'), ('k123xm'), ('A 098BC');
```

```
INSERT 0 3
```

Функция нормализации:

```
=> CREATE FUNCTION to_normal(regnum text) RETURNS text
LANGUAGE sql IMMUTABLE
RETURN upper(translate(regnum, 'АВЕКМНОРСТУХ ', 'ABEKMHOPCTYX'));
```

```
CREATE FUNCTION
```

Категория изменчивости — immutable. Функция всегда возвращает одинаковое значение при одних и тех же входных параметрах.

```
=> SELECT to_normal(regnum) FROM cars;
```

```
 to_normal
-----------
 K123XM
 K123XM
 A098BC
(3 rows)
```

Теперь легко исключить дубликаты:

```
=> CREATE FUNCTION num_unique() RETURNS bigint
LANGUAGE sql STABLE
RETURN (
SELECT count(DISTINCT to_normal(regnum))
FROM cars
);
```

```
CREATE FUNCTION
```

```
=> SELECT num_unique();
```

```
 num_unique
------------
          2
(1 row)
```

3. Корни квадратного уравнения

```
=> CREATE FUNCTION square_roots(
    a float,
    b float,
    c float,
    x1 OUT float,
    x2 OUT float
)
LANGUAGE sql IMMUTABLE
RETURN (
WITH discriminant(d) AS (
    SELECT b*b - 4*a*c
)
SELECT (CASE WHEN d >= 0.0 THEN (-b + sqrt(d))/2/a END,
        CASE WHEN d >  0.0 THEN (-b - sqrt(d))/2/a END)
FROM discriminant
);
```

```
CREATE FUNCTION
```

Категория изменчивости — immutable. Функция всегда возвращает одинаковое значение при одних и тех же входных параметрах.

```
=> SELECT square_roots(1,  0, -4);
```

```
 square_roots
--------------
 (2,-2)
(1 row)
```

```
=> SELECT square_roots(1, -4,  4);
```

```
 square_roots
--------------
 (2,)
(1 row)
```

```
=> SELECT square_roots(1,  1,  1);
```

```
 square_roots
--------------
 (,)
(1 row)
```

```
=> \c postgres
```

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

```
=> DROP DATABASE sql_func;
```

```
DROP DATABASE
```