Приложение «Книжный магазин»
Схема данных и интерфейс

Авторы: Егор Рогов, Павел Лузанов

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

Темы

Обзор приложения «Книжный магазин»

Проектирование схемы данных, нормализация

Итоговая схема данных приложения

Организация интерфейса между клиентом и сервером

Демонстрация

В этой демонстрации мы показываем приложение «Книжный магазин» в том виде, в котором оно будет после завершения всех практических заданий. Приложение доступно в браузере виртуальной машины курса по адресу http://localhost/.

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 новые поступления.

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.

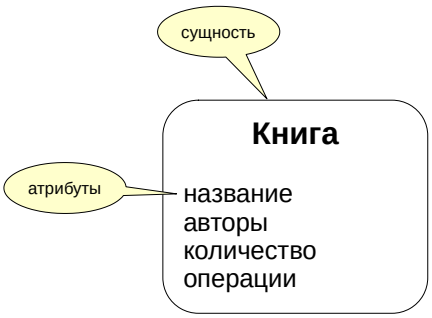
Мы начнем с пустой базы данных и в ходе курса постепенно реализуем все необходимые компоненты.

Исходный код приложения не является темой курса, но его можно найти в git-репозитории https://git.postgrespro.ru/pub/dev1app.git

Книги

ER-модель для высокоуровневого проектирования

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

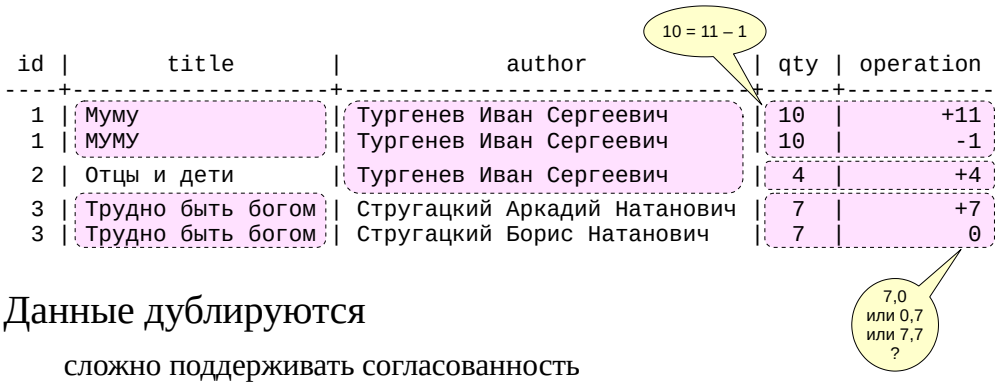


После того как мы посмотрели внешний вид приложения и поняли его функциональность, нам нужно разобраться со схемой данных. Мы не будем хоть сколько-нибудь глубоко вникать в вопросы проектирования баз данных — это отдельная дисциплина, не входящая в этот курс. Но совсем обойти вниманием эту тему тоже 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 уровне (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, 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. Проверка целиком ложится на приложение, что, безусловно, рано или поздно приведет к тому, что согласованность данных будет нарушена.

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.

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 12 удобно пользоваться возможностями SQL/JSONPath, определенными в стандарте SQL:2016).

https://github.com/postgrespro/jsquery

https://postgrespro.ru/docs/postgresql/12/functions-json#FUNCTIONS-SQLJSON-PATH

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

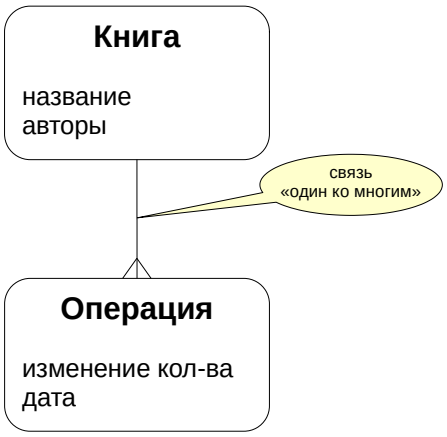
Такую схему удобно применять, когда от базы данных требуется только получение JSON по идентификатору, но не требуется серьезная работа с данными *внутри* JSON. Это не наш случай.

(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 физическом уровне выделенные нами сущности могут быть представлены двумя таблицами: книги (books) и операции (operations).

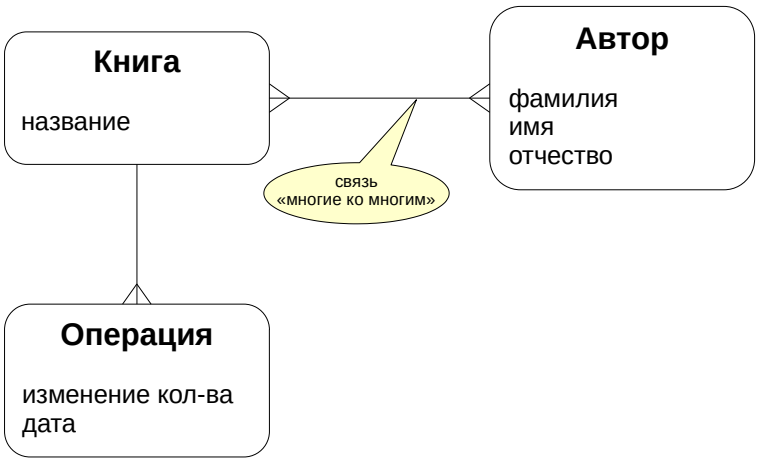
Операция состоит в изменении количества книг (положительном при заказе, отрицательном при покупке). Заметьте, что теперь у книги нет атрибута «количество». Вместо этого достаточно просуммировать изменения количества по всем операциям, относящимся к данной книге. Дополнительный атрибут «количество» у 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 приводит 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: физически добавить 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 теме 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 от книг авторов 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
```

```
=> \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:

- отсутствие 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.

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 вместо 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(1,1000000);
```

```
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: 127,368 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: 0,351 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 | 2022-11-17
            2 |       1 |         10 | 2022-11-17
            3 |       1 |         -1 | 2022-11-17
(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»).

Кроме того, мы подробно рассмотрим более сложные типы:

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

См. также раздаточный материал «Основные типы данных и функции».

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.

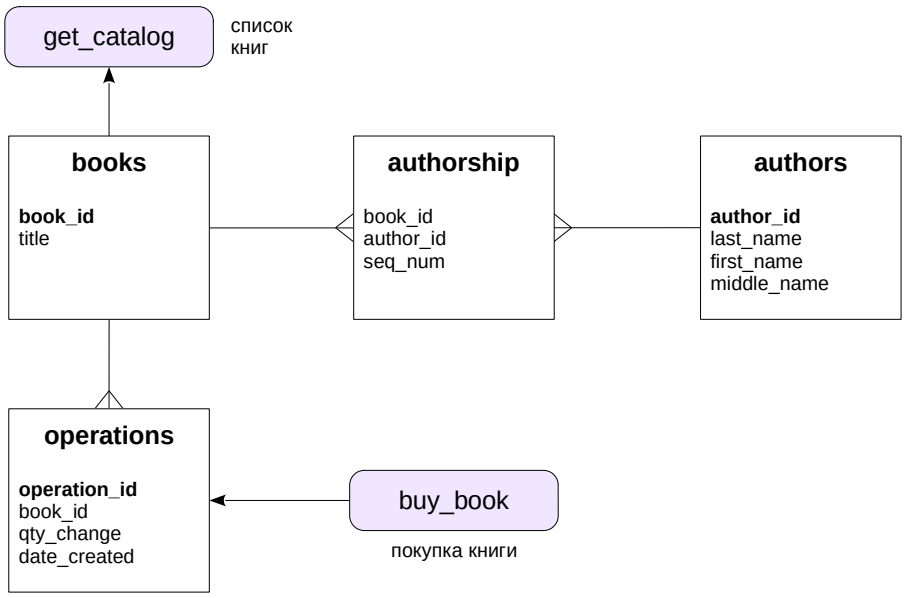
Один вариант — разрешить приложению напрямую обращаться к таблицам в базе данных и изменять их. При этом от приложения требуется детальное «знание» модели данных. Отчасти это требование можно ослабить за счет использования представлений (view).

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 тоже можно выполнять, вызывая функции и передавая необходимые параметры. В этом случае внутри функций можно реализовать все необходимые проверки согласованности — база данных будет защищена, но приложение сможет пользоваться только предоставленным ему 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 магазина

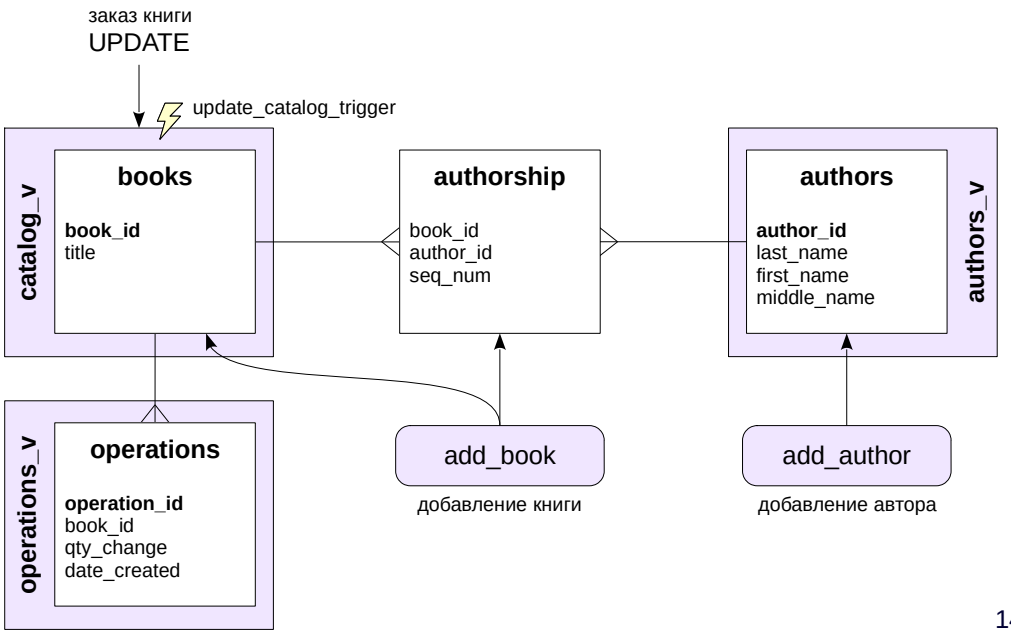


В нашем приложении мы попробуем разные варианты организации интерфейса (хотя в реальной жизни обычно лучше систематически придерживаться какого-то одного подхода).

Магазин будет использовать интерфейсные функции:

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

Интерфейс «админки»



«Администрирование» 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;
```

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

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 | 17.11.2022
       1 | Поступление |         10 | 17.11.2022
       1 | Покупка     |          1 | 17.11.2022
(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 приведенным решением. При необходимости внесите коррективы.

1. Схема и путь поиска

```
student$ psql bookstore
```

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

```
CREATE SCHEMA
```

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

```
ALTER DATABASE
```

```
=> \c bookstore
```

```
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.

Другой способ вставки данных в таблицу — команда COPY. Она обычно используется, если нужно загрузить большой объем информации. Но в этом случае надо не забыть «передвинуть» значение последовательности:

```
=> COPY operations (operation_id, book_id, qty_change) FROM stdin;
1       1       10
2       1       10
3       1       -1
\.
```

```
COPY 3
```

```
=> SELECT pg_catalog.setval('operations_operation_id_seq', 3, true);
```

```
 setval
--------
      3
(1 row)
```

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.