Bookstore Application

Application Schema and Interface

Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Bookstore application overview

Designing the application schema; normalization

The final version of the application schema

Setting up the client-server interface

Demo

In this demo, we are going to show the Bookstore application as it should appear after all practical assignments are complete. The application is available in the VM browser at http://localhost/.

The application consists of several parts that are provided as separate tabs.

“Store” is a web user interface for buying books online.

Other tabs represent the backend interface, which is available only to the bookstore employees (the admin panel).

“Catalog” is the storekeeper’s interface that is used for ordering books to the store and viewing new arrivals and purchases.

“Books” and “Authors” are interfaces for librarians, where they can register new arrivals.

For training purposes, all this functionality is exposed in a single web page. If any feature is unavailable because the required object (such as a table or a function) is missing, the application will report an error. It also displays the text of all queries sent to the server.

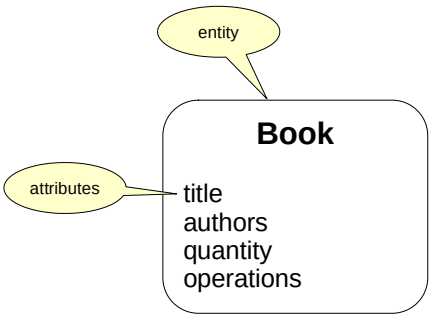
We will start with an empty database and will gradually implement all the required components as the course progresses.

The source code of the application frontend will not be discussed in this course, but you can download it from the following git repository: https://git.postgrespro.ru/pub/dev1app.git

Books

An ER-model for high-level design

entities are concepts of the subject domain
relationships are connections between entities
attributes are properties of entities and relationships



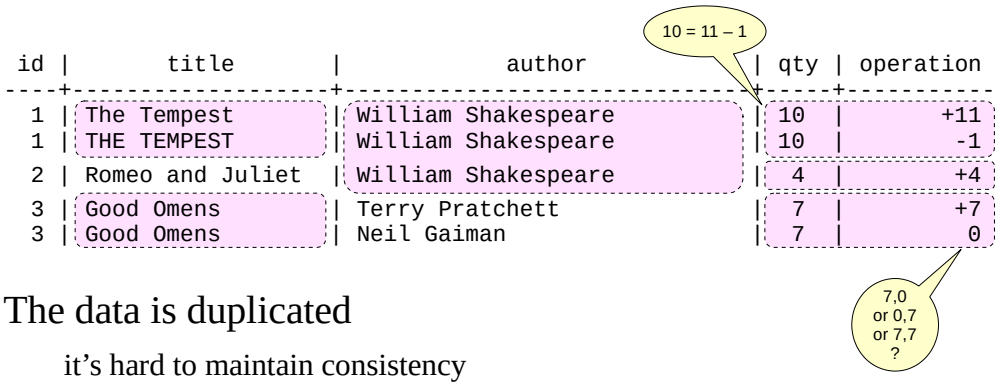
4

After taking a look at the application's interface and functionality, we have to deal with its schema. We will not go into details about the database design: it is a separate branch of knowledge, which is beyond the scope of this course. But we cannot ignore this topic entirely.

High-level database design often uses the ER-model (where ER stands for "Entity–Relationship"). It deals with *entities* (concepts of the subject area), their *relationships*, and *attributes* (the properties of entities and relationships). The model allows us to remain at the logical level, without getting down to data representation at the physical level (i.e., its table form).

The first approach to database design is creating a diagram as shown on this slide: a book is represented as a single big entity, and everything else becomes its attributes.

Application Schema (Ver. 1)



The data is duplicated

- it's hard to maintain consistency
- it's hard to perform updates
- it's hard to write queries

Clearly, this approach cannot be correct. It may be not quite obvious in the diagram itself, but let's try to project this diagram onto database tables. There are several ways to do it. One of them is shown on the slide: the table corresponds to the entity, and table columns represent the attributes of this entity.

This diagram is a good illustration that some data is duplicated (these fragments are highlighted). Data duplication makes it hard to maintain consistency, which is arguably the main objective of a database system.

For example, each of the two rows related to book 3 must list the total quantity (7 items). What should be done to reflect a purchase? On the one hand, we need to add some rows that reflect purchase operations. (But how many rows are required? Should we add one or two?) On the other hand, the quantity should be reduced from 7 to 6 in all rows. And what if an error leads to data discrepancy between these rows? How can we define a constraint that forbids such a situation?

Many queries will also become overcomplicated. How can we find the total number of books? How can we find all the unique authors?

Thus, such a schema will not work well for relational databases.

Application Schema (Ver. 2)

```
 entity | attribute  | value
--------+------------+-------
      1 | title      | The Tempest
      1 | author     | William Shakespeare
      1 | qty        | 10
      1 | operation  | +11
      1 | operation  | -1
      2 | title      | Romeo and Juliet
      2 | author     | William Shakespeare
      2 | qty        | 4
      2 | operation  | +4
    ... | ...        | ...
```

Data without a schema

- consistency is maintained at the application side
- it’s hard to write queries
- performance is low (multiple joins)

Another way to represent an entity as a table is a so-called EAV schema (“entity–attribute–value”). It allows storing anything at all in a single table. Technically, we get a relational database, but it has virtually no schema, and the database system cannot guarantee data consistency. Consistency has to be maintained by the application alone, and sooner or later it is bound to be compromised.

With such a schema, it is hard to write queries (although they are quite easy to generate). As a result, handling more or less significant data volumes becomes a problem because of multiple self-joins.

It is not an approach to follow.

Application Schema (Ver. 3)

```
 book_id |                      description
---------+-----------------------------------------------------------
       1 | {"title": "The Tempest",
         |  "authors": [ "William Shakespeare" ],
         |  "qty": 10,
         |  "operations": [ +11, -1 ]}
       3 | {"title": "Good Omens",
         |  "authors": [ "Terry Pratchett",
         |               "Neil Gaiman" ],
         |  "qty": 7,
         |  "operations": [ +7 ]}
     ... | ...
```

Data without a schema

- consistency is maintained at the application side
- it's hard to write queries (a special language is needed)
- there is index support

7

Another similar approach consists in representing data in the JSON format, NoSQL-style. All the previous considerations still apply here.

Besides, it will be impossible to query such a structure using SQL: you will have to use a special language (previously, jsQuery would be the most probable choice, but starting from PostgreSQL 12, it is convenient to use the SQL/JSONPath features defined in the SQL:2016 standard).

https://github.com/postgrespro/jsquery

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

Although PostgreSQL provides index support for JSON, performance is still a concern.

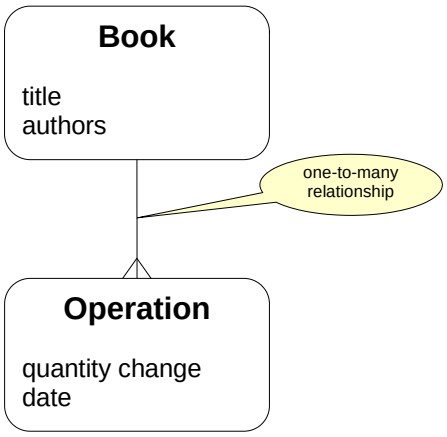
It is convenient to use such a schema if the database only needs to get JSON data by ID, and no serious data processing *within* the JSON structure is expected. But it is not our case.

(Naturally, nothing is set in stone here. See the last practical assignment for further discussion.)

Books and Operations

Normalization reduces data redundancy

Large entities are split into smaller ones



Thus, we need to eliminate redundancy, so that it is convenient to work with the data in a relational database system. This process is called *normalization*.

You might be familiar with various *normal form* concepts (first, second, third, Boyce–Codd, etc.). We are not going to discuss them here; speaking informally, it is enough to understand that all this math pursues one and the same goal: eliminating redundancy.

You can reduce redundancy by splitting a big entity into several smaller ones. The exact way to do it should be prompted by common sense (which cannot be replaced by the knowledge of normal forms alone anyway).

In our case, everything is quite straightforward. Let's start by separating books and operations. These two entities are connected by the one-to-many relationship: there can be several operations on each book, but each operation relates to a single book only.

Application Schema

books

book_id	title	author
1	The Tempest	William Shakespeare
2	Romeo and Juliet	William Shakespeare
3	Good Omens	Terry Pratchett
3	Good Omens	Neil Gaiman

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

At the physical level, the identified can be represented by two tables: `books` and `operations`.

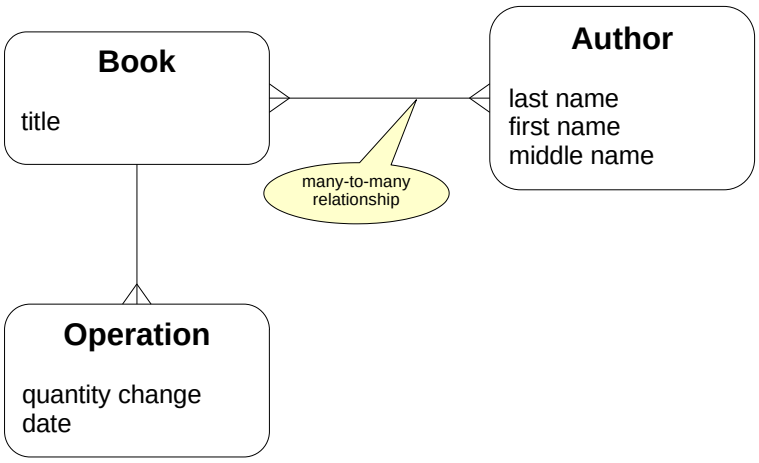
An operation changes the quantity of books. This change can be either positive or negative (the order operation adds some books, while the purchase operation subtracts them). Note that the book has no “quantity” attribute anymore. Instead, it is enough to sum up all quantity changes made by operations related to this book. Having an additional “quantity” attribute would lead to data redundancy again.

This solution might seem strange to you at first. Is it really convenient to calculate the sum instead of simply querying a separate field? But we can create a view to display the quantity of each book. It won’t result in redundancy: a view is just a query.

Another point to consider is performance. If summing up all changes brings too much overhead, we can perform the opposite process called denormalization: physically add the “quantity” field to the books table and ensure that it is consistent with the operations table. We are not going to discuss here whether it makes sense or not (this question is considered in the QPT course that covers query performance tuning). Common sense suggests that it’s not required for our “sandbox.” But we will get back to denormalization in the “Triggers” lecture.

Thus, as you can see on this slide, moving all operations into a separate entity resolves most of the duplication issues, but not all of them.

Books, Authors, Operations



That’s why we have to take one step further: separate books from authors and connect them with each other by a many-to-many relationship: each book can be written by several authors, and each author can have more than one book. At the table level, such relationship can be implemented using an additional intermediate table.

The first, last, and middle names can be the author’s attributes. It makes sense because we may need to work with each of these attributes separately, e.g., to display the author’s last name and initials.

Application Schema

```
student$ psql
```

```
=> \c bookstore
```

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

Our application schema consists of four tables:

```
=> \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)
```

Books

```
=> \d books
```

```
                         Table "bookstore.books"
   Column   |  Type   | Collation | Nullable |           Default
------------+---------+-----------+----------+-----------------------------
 book_id    | integer |           | not null | generated always as identity
 title      | text    |           | not null |
 onhand_qty | integer |           | not null | 0
Indexes:
    "books_pkey" PRIMARY KEY, btree (book_id)
Check constraints:
    "books_onhand_qty_check" CHECK (onhand_qty >= 0)
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)
```

We use the following data types here:

- integer;
- text, which is a text string of arbitrary length.

We also use the PRIMARY KEY constraint.

The GENERATED AS IDENTITY clause is used to automatically generate unique values (prior to version 10, the serial pseudotype was used for this purpose).

The values in the GENERATED AS IDENTITY columns take their values from special database objects called sequences. We can learn the name of the used sequence as follows:

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

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

If required, you can also create sequences manually and query them directly:

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

```
 nextval
---------
       8
(1 row)
```

A sequence is the most efficient way of generating unique IDs. But you should keep in mind that:

- there may be gaps in numbering (since the changes are not transactional);
- the numbers may not increase monotonically (if sessions cache values).

Here is the data stored in the books table:

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

```
-[ RECORD 1 ]----------------------------------------------------------------------------------------------------------------------------------------------------------
book_id    | 2
title      | Romeo and Juliet
onhand_qty | 0
-[ RECORD 2 ]----------------------------------------------------------------------------------------------------------------------------------------------------------
book_id    | 3
title      | Good Omens
onhand_qty | 0
-[ RECORD 3 ]----------------------------------------------------------------------------------------------------------------------------------------------------------
book_id    | 4
title      | Dark Avenues
onhand_qty | 0
-[ RECORD 4 ]----------------------------------------------------------------------------------------------------------------------------------------------------------
book_id    | 5
title      | Travels into Several Remote Nations of the World. In Four Parts. By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships
onhand_qty | 0
-[ RECORD 5 ]----------------------------------------------------------------------------------------------------------------------------------------------------------
book_id    | 6
title      | Three Men in a Boat (To Say Nothing of the Dog)
onhand_qty | 0
-[ RECORD 6 ]----------------------------------------------------------------------------------------------------------------------------------------------------------
book_id    | 7
title      | 101 Famous Poems
onhand_qty | 0
-[ RECORD 7 ]----------------------------------------------------------------------------------------------------------------------------------------------------------
book_id    | 1
title      | The Tale of Tsar Saltan
onhand_qty | 29
```

Note that book titles can be quite long.

Authors

```
=> \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)
```

In this table, we also use the NOT NULL constraint, which means that undefined values are not allowed.

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

```
 author_id |  last_name  | first_name | middle_name
-----------+-------------+------------+-------------
         1 | Pushkin     | Alexander  | Sergeyevich
         2 | Shakespeare | William    |
         3 | Pratchett   | Terry      |
         4 | Gaiman      | Neil       |
         5 | Bunin       | Ivan       | Alekseyevich
         6 | Swift       | Jonathan   |
         7 | Jerome      | Jerome     | Klapka
(7 rows)
```

Note that the middle name might be missing (or defined by an empty string).

The PRIMARY KEY constraint was mentioned in the \d output together with the terms “index” and “btree”.

Btree is the main index type used in databases to speed up search and provide support for constraints (primary key and unique).

Suppose that our bookstore sells books written by a million of different authors with the same last name:

```
=> BEGIN; -- let's explicitly start a transaction to roll back the changes later
```

```
BEGIN
```

```
=> INSERT INTO authors(first_name, last_name)
    SELECT 'John', 'Wordsmith' FROM generate_series(1,1000000);
```

```
INSERT 0 1000000
```

How long will it take to find an author in such a table?

```
=> \timing on
```

```
Timing is on.
```

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

```
 author_id | last_name | first_name | middle_name
-----------+-----------+------------+-------------
         1 | Pushkin   | Alexander  | Sergeyevich
(1 row)

Time: 102.994 ms
```

```
=> \timing off
```

```
Timing is off.
```

If we ask the optimizer to display the query plan, we will see that Seq Scan is used; it means that the whole table is scanned sequentially using a Filter to find the required value:

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

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

And what if we perform the search by an indexed field?

```
=> \timing on
```

```
Timing is on.
```

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

```
 author_id | last_name | first_name | middle_name
-----------+-----------+------------+-------------
         1 | Pushkin   | Alexander  | Sergeyevich
(1 row)
```

```
Time: 0.375 ms
```

```
=> \timing off
```

```
Timing is off.
```

The query time has been reduced by an order of magnitude.

And the query plan now contains an index:

```
=> 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)
```

We can also create an index by the last name (and analyze the table to gather up-to-date statistics):

```
=> ANALYZE authors;
```

```
ANALYZE
```

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

```
CREATE INDEX
```

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

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

However, the index is not a universal performance tuning tool. Having an index is usually very useful if the query needs to select only a small portion of all table rows. But if it is required to read a lot of data, the index will only add overhead, and the optimizer is smart enough to understand it:

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

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

Besides, you have to keep in mind that indexes take extra disk space, and index updates caused by table modifications bring extra overhead.

Let's cancel all our changes (including index creation):

```
=> ROLLBACK;
```

```
ROLLBACK
```

```
=> ANALYZE authors;
```

```
ANALYZE
```

Authorship

This table implements many-to-many relationship.

```
=> \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)
```

In addition to all the previously used constraints, this table also uses FOREIGN KEY, which is a referential integrity constraint.

In fact, this table contains two foreign keys: one of them refers to the books table, and the other refers to the authors table.

The seq_num column defines the order in which multiple authors of the same book should be listed.

Note that we have a composite primary key here.

```
=> 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 |         7 |       1
       7 |         1 |       1
       7 |         5 |       2
       7 |         2 |       3
(10 rows)
```

Operations

```
=> \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)
Triggers:
    update_onhand_qty_trigger AFTER INSERT ON operations FOR EACH ROW EXECUTE FUNCTION update_onhand_qty()
```

This table uses one more data type: date, which defines the date without timestamp.

For the date_created column, the current date is specified as the default value (using the DEFAULT clause).

```
=> SELECT * FROM operations;

 operation_id | book_id | qty_change | date_created
--------------+---------+------------+--------------
            1 |       1 |         10 | 2021-10-19
            2 |       1 |         10 | 2021-10-19
            3 |       1 |         -1 | 2021-10-19
            4 |       1 |         10 | 2021-10-19
(4 rows)
```

Apart from the data types used in application tables, we are going to come across the boolean type all the time. For example, the expressions in WHERE clauses are of the boolean type.

It's important to remember that, unlike traditional programming languages, SQL uses three-valued logic: in addition to true and false, there is also the NULL value (which can be interpreted as “the value is unknown”).

We will also cover some other types that are more complex:

- the composite type, which represents a record similar to a table row (in “SQL. Composite Types”);
- arrays (in “PL/pgSQL. Arrays”).

See also the “Basic Data Types and Functions” handout.

Designing the Interface

Tables and triggers

reading data directly from tables (views);
writing data directly to tables (views),
using triggers for changing related tables

the application must be aware of the data model;
this approach provides maximum flexibility

consistency is hard to maintain

Functions

reading data via table functions;
writing data by calling functions

the application is separated from the data model and is limited by API

you have to write a lot of wrapper-functions,
potential performance issues

There are several ways to set up an application's client-server interface.

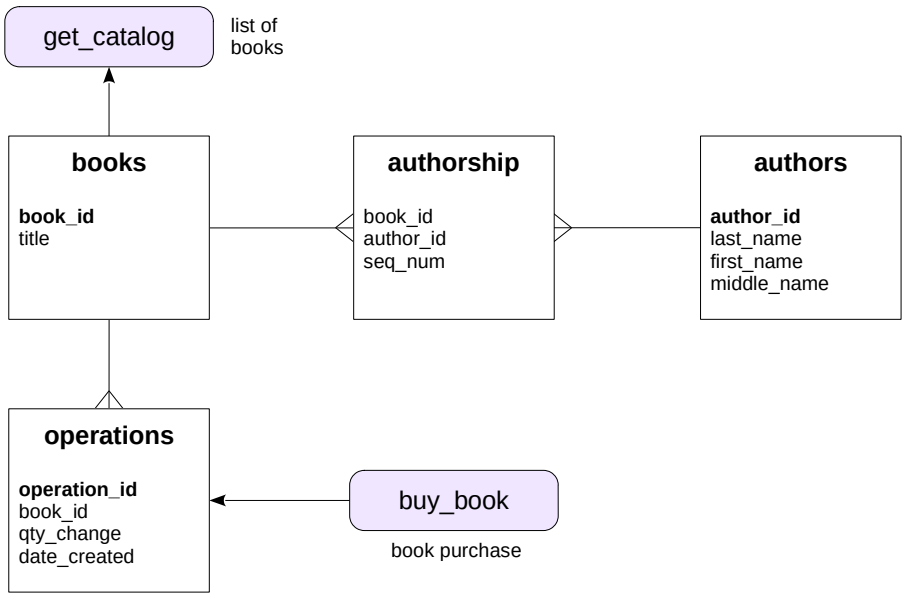
The first option is to allow the application to access and modify database tables directly. In this case, the application must have the precise "knowledge" of the data model. This requirement can be relaxed to some extent by using views.

Another limitation of this approach is that the application has to follow certain rules; otherwise, it is very hard to maintain data consistency if you have to address all possible inappropriate operations at the database level. But this approach is the most flexible one.

Another option is to forbid direct table access from the application and allow only function calls. Reading can be performed by table functions (which return a set of rows). Writing can be performed by calling other functions and passing the required data to them. In this case, all the necessary consistency checks can be implemented within functions: the database will be protected, but the application will be able to use only a limited set of features that we provide. It requires writing many wrapper functions and can lead to performance degradation.

You can also combine these two approaches. For example, you can allow the application to read data from tables directly, but perform modifications only by calling special functions.

Bookstore Interface

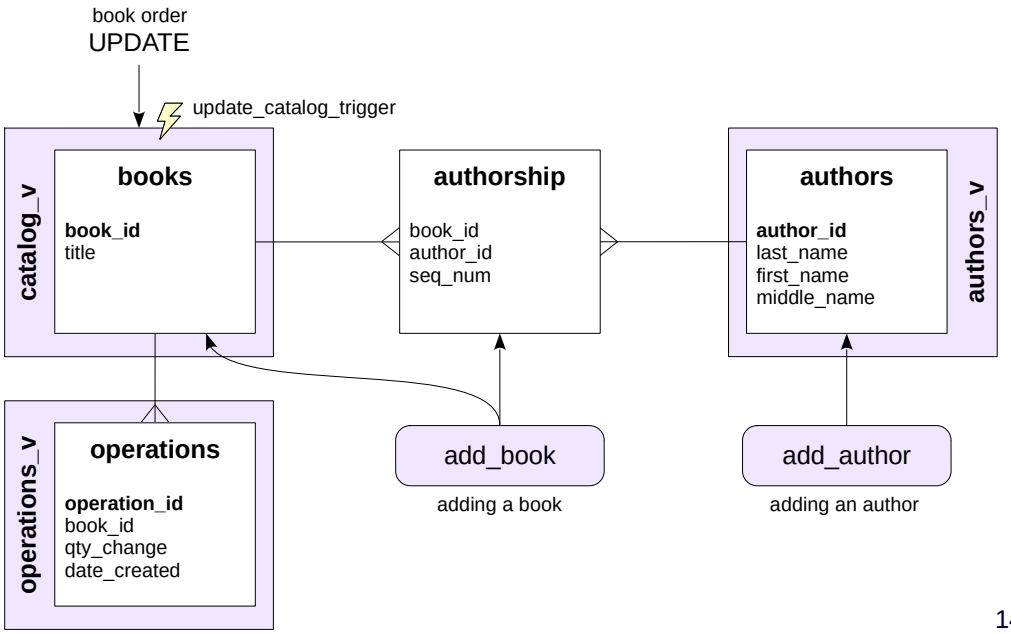


In this application, we will try different ways of setting up the interface (although it's usually better to stick to one approach when developing real applications).

The store will use interface functions:

- `get_catalog` for searching books (see "SQL. Composite Types")
- `buy_books` for making a purchase (see "PL/pgSQL. Executing Queries")

Admin Panel Interface



The admin panel is going to retrieve data by accessing the following views (which we have to create as part of the practice for this lecture):

- `catalog_v` for the list of books
- `authors_v` for the list of authors
- `operations_v` for the list of operations

Authors will be added using the `add_author` function (we will create it once we get to the “PL/pgSQL. Executing Queries” lecture). For adding books, we will implement the add_book function (“PL/pgSQL. Arrays”).

To enable book purchase, we will make the `catalog_v` view updatable (“PL/pgSQL. Triggers”).

Views

A view is a named query. For example, you can create a view that displays only those authors who do not have a middle name, as follows:

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

```
CREATE VIEW
```

Now this view can be used in queries almost like a regular table:

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

```
 author_id | first_name |  last_name
-----------+------------+-------------
         2 | William    | Shakespeare
         3 | Terry      | Pratchett
         4 | Neil       | Gaiman
         6 | Jonathan   | Swift
(4 rows)
```

In a simple case, other operations can also be applied to a view, for example:

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

```
UPDATE 4
```

In complex cases, you can use triggers to enable insert, update, and delete operations. We will explain it in the “PL/pgSQL. Triggers” lecture.

At the planning stage, the view “unfolds,” revealing the base tables:

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

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

The application uses three views. They will be very simple at first, but later we’ll move some application logic into them.

The authors view displays a concatenation of the first, last, and middle names of each author (if available):

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

```
 author_id |     display_name
-----------+---------------------
         1 | Alexander S. Pushkin
         5 | Ivan A. Bunin
         7 | Jerome K. Jerome
         6 | Jonathan Swift
         4 | Neil Gaiman
         3 | Terry Pratchett
         2 | William Shakespeare
(7 rows)
```

The catalog view displays only the book title for now:

```
=> SELECT * FROM catalog_v;
```

```
 book_id |                                                                   title                                                                    | onhand_qty |
---------+--------------------------------------------------------------------------------------------------------------------------------------------+------------+--------------------
       7 | 101 Famous Poems                                                                                                                           |          0 | 101 Famous Poems. Al
       4 | Dark Avenues                                                                                                                               |          0 | Dark Avenues. Ivan A
       3 | Good Omens                                                                                                                                 |          0 | Good Omens. Neil Gai
       2 | Romeo and Juliet                                                                                                                           |          0 | Romeo and Juliet. Wi
       1 | The Tale of Tsar Saltan                                                                                                                    |         29 | The Tale of Tsar Sal
       6 | Three Men in a Boat (To Say Nothing of the Dog)                                                                                            |          0 | Three Men in a Boat 
       5 | Travels into Several Remote Nations of the World. In Four Parts. By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships |          0 | Travels into Several
(7 rows)
```

The operations view specifies the operation type (new arrival or purchase):

```
=> SELECT * FROM operations_v;
```

```
 book_id | op_type  | qty_change | date_created
---------+----------+------------+-------------
       1 | Arrival  |         10 | 19.10.2021
       1 | Arrival  |         10 | 19.10.2021
       1 | Purchase |          1 | 19.10.2021
       1 | Arrival  |         10 | 19.10.2021
(4 rows)
```

Summary

Database design is a separate complex topic

- theory is important, but it should not replace common sense

Normalized data makes your life easier and facilitates consistency support

The client-server interface can use tables, views, functions, and triggers

Practice

1. Create the `bookstore` schema in the `bookstore` database.
   Set up the search path to this schema at the database level.
2. In the `bookstore` schema, create `books`, `authors`, `authorship`, and `operations` tables with all the necessary constraints, exactly as shown in the demo.
3. Insert the data about several books into the tables.
   Check the result by running some queries.
4. In the `bookstore` schema, create `authors_v`, `catalog_v`, and `operations_v` views, so that they look exactly like shown in the demo.
   Check that the application now shows some data in “Books”, “Authors”, and “Catalog” tabs.

17

Task 1. Recall the contents of the topic “Data Organization. Logical Structure.”

Task 2. Use the demonstrated output of psql’s \d commands as a reference.

Task 3. You can use the data shown in the demo, or come up with your own data.

Task 4. Try writing queries to the base tables that return the same results as the queries to views shown in the demo. Then save these queries as views.

After completing the assignments, make sure to compare your queries with those in the provided keys. Make corrections if required.

Task 1. Schema and Search Path

```
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)
```

Task 2. Tables

Authors:

```
=> 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
```

Books:

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

```
CREATE TABLE
```

Authorship:

```
=> 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
```

Operations:

```
=> 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
```

Task 3. Data

Authors:

```
=> INSERT INTO authors(last_name, first_name, middle_name)
VALUES
    ('Pushkin', 'Alexander', 'Sergeyevich'),
    ('Shakespeare', 'William', NULL),
    ('Pratchett', 'Terry', NULL),
    ('Gaiman', 'Neil', NULL),
    ('Bunin', 'Ivan', 'Alekseyevich'),
    ('Swift', 'Jonathan', NULL),
    ('Jerome', 'Jerome', 'Klapka');
```

```
INSERT 0 7
```

Books:

```
=> INSERT INTO books(title)
VALUES
    ('The Tale of Tsar Saltan'),
    ('Romeo and Juliet'),
    ('Good Omens'),
    ('Dark Avenues'),
    ('Travels into Several Remote Nations of the World. In Four Parts. By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships'),
    ('Three Men in a Boat (To Say Nothing of the Dog)'),
    ('101 Famous Poems');
```

```
INSERT 0 7
```

Authorship:

```
=> 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, 7, 1),
    (7, 1, 1),
    (7, 5, 2),
    (7, 2, 3);
```

```
INSERT 0 10
```

The operations table will be filled in by the COPY command. It is an alternative way of inserting data into a table, which is commonly used for loading large batches of data. In this case, you have to remember to increment the sequence value:

```
=> 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)
```

Task 4. Views

Authors View:

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

```
CREATE VIEW
```

Catalog View:

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

```
CREATE VIEW
```

Operations View:

```
=> CREATE VIEW operations_v AS
SELECT book_id,
       CASE
           WHEN qty_change > 0 THEN 'Arrival'
           ELSE 'Purchase'
       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
```

Practice

1. What additional attributes can appear for these entities as the application evolves?
2. Suppose you have to store the information about the publisher. Extend the ER-diagram accordingly and create the corresponding tables.
3. Some books can belong to a series (such as “The Adventure Collection”). How will it affect the schema?
4. Suppose our store started selling hardware equipment (motherboards, CPUs, memory, storage devices, monitors, etc.). What entities and attributes would you single out?
   Keep in mind that new types of equipment constantly appear in the market, and they can have their own specific characteristics.

Task 3. Different publishers can easily have different book series with the same title.

Task 1. Additional Attributes

Some examples:

- Authors: role (author, editor, translator, etc.)
- Books: abstract
- Operations: current status (paid, out for delivery, etc.)

Task 2. Publishers

We need to add the “Publisher” entity, which will have (at least) the “Name” attribute.

Books are connected with publishers by the many-to-many relationship: a book can be printed by different publishers. That’s why an intermediate table at the physical level is required: we have to create the “Publications” table with the “Publication Year” attribute.

(Sure enough, this model is simplified; it can be further elaborated on if required.)

Task 3. Series

Let’s add the “Series” entity. It’s not a book itself, but a particular publication of the book that belongs to a series, so it is a good idea to make “Publication” a full-fledged entity of the ER-model and connect it with the series by the one-to-many relationship (each publication belongs to one series, and each series can contain several publications).

The one-to-many relationship also connects the series with a publisher (a publisher can print several series, but each series belongs to a particular publisher).

But what if a book is not a part of any series? We can either introduce a fake series like “No series,” or allow omitting the series foreign key for publications.

Task 4. Hardware Equipment

Looking at each type of hardware equipment, it’s not hard to single out the required attributes. Some attributes (like the model or the manufacturer) will be common, while others will make sense only for a particular type. For example:

- CPU: frequency
- Monitor: size, resolution
- Storage Device: type, capacity

The problem is that the hardware market is highly dynamic. Not so long ago we cared about hard drives’ spin rate and capacity, and now we are more interested in the type of the storage device (SSD, HDD, or SSHD). In the CRT times, the monitor’ refresh rate was important, and now we would like to know the matrix type. Nobody needs CD-ROM drives anymore as USB flash drives have appeared instead. And so on.

Thus, we either have to change the schema all the time (which means changing the application that works with the data), or look for a more flexible model without a rigid structure and consistency control. We have touched upon such models in the presentation (for example, storing some part of the data in the JSON format).