

# The Bookstore App Data Schema and API

 PostgresPro

17

## Copyright

© Postgres Professional, 2017–2025

Authors Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by Liudmila Mantrova and Alexander Meleshko

Photo by Oleg Bartunov (Tukuche peak, Nepal)

## Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

## Feedback

Please send your feedback, comments and suggestions to:

[edu@postgrespro.com](mailto:edu@postgrespro.com)

## Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

# Topics



Bookstore app overview

Data schema design, normalization

Data schema, the final version

Setting up the API

# The application



Bookstore

Store Authors Books Catalog

Store

Author name Book title ☐ In stock Search

Title	In stock	
101 Famous Poems. Alexander S. Pushkin, Ivan A. Bunin, William Shakespeare	0	Buy
Dark Avenues. Ivan A. Bunin	0	Buy
Good Omens. Neil Gaiman, Terry Pratchett	⊕ 0	Buy
Romeo and Juliet. William Shakespeare	0	Buy
The Tale of Tsar Saltan. Alexander S. Pushkin	24	Buy
Three Men in a Boat (To Say Nothing of the... Jerome K. Jerome	0	Buy
Travels into Several Remote Nations of the... Jonathan Swift	0	Buy

You have bought a book

Books found: 7

DB Role

postgres

```
select buy_book (
  book_id->$1
) result
[1]
```

```
select * from get_catalog ($1, $2, $3)
[null,null,false]
```

NOTICE: SELECT cv.book\_id, cv.display\_name,
cv.onhand\_qty FROM catalog\_v cv WHERE true
ORDER BY display\_name

3

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

“Store” is a customer UI for buying books online.

Other tabs represent the employee UI, which is available only to the bookstore staff (the admin panel).

“Catalog” is the storekeeper’s UI which is used for ordering books to the store and viewing arrivals and sales.

“Books” and “Authors” are the UI for librarians, where they can register 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 by the end of the course.

The source code of the application frontend will not be discussed in this course, but is available for download:

<https://pubgit.postgrespro.ru/pub/dev1app.git>

## Application demonstration

This demo shows the Bookstore app as it would look like after completing all practice assignments. The app opens in a separate browser tab in the course VM:

Opening `http://localhost...`

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

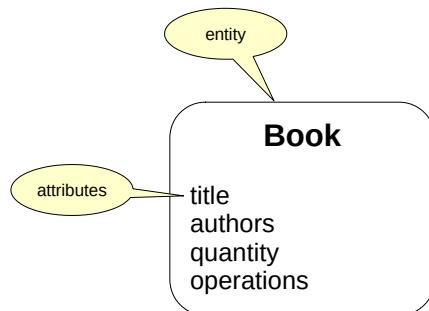


## An ER model for high-level design

entities – concepts of the application domain

relationships – connections between entities

attributes – properties of entities and relationships



After taking a look at the application's UI and functionality, we need to figure out its data schema. We will not go into details about 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 ("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 (such as 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.

# Data schema



id	title	author	qty	operation
1	The Tempest	William Shakespeare	10	+11
1	THE TEMPEST	William Shakespeare	10	-1
2	Romeo and Juliet	William Shakespeare	4	+4
3	Good Omens	Terry Pratchett	7	+7
3	Good Omens	Neil Gaiman	7	0

10 = 11 - 1

7.0  
or 0.7  
or 7.7  
?

## Some data is duplicated

- hard to maintain consistency
- hard to perform updates
- hard to write queries

6

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 ends up duplicated (as highlighted on the slide). Data duplication may lead to problems with consistency, the very thing a database system is supposed to ensure.

For example, each of the two rows related to book 3 must list the total quantity (7 items). How should purchases be recorded, then? Some rows will need to be added to record the purchase operations. And then the quantity in all the duplicated rows will need to be reduced from 7 to 6. But what if an error leads to data discrepancy between these rows? How can we define a constraint to make sure the values stay in sync?

Many queries will also become overcomplicated. How can we find the total number of books? Or get a list of distinct authors?

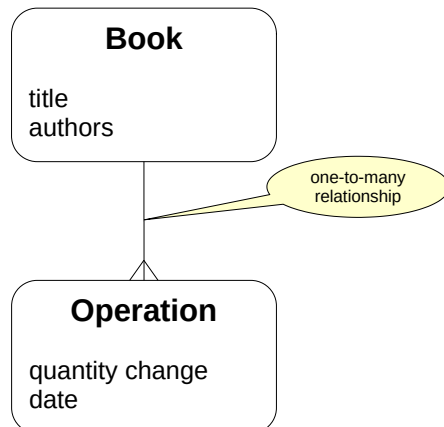
Thus, such a schema will not work well for relational databases.

# Books and operations



## Normalization reduces data redundancy

Large entities are split into smaller ones



7

To work with data in a relational database system properly, we need to eliminate redundancy. 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; in essence, it is enough to understand that all this math pursues one and the same goal: eliminating redundancy.

The way to reduce redundancy is to split a larger entity into smaller ones. How exactly to split 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 a one-to-many relationship: there can be several operations on each book, but each operation relates to only one book.

# Data 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

8

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

An operation changes the quantity of books (sell books if negative, order new books if positive). Note that the book entity has no *quantity* attribute anymore. Instead, you get the quantity by adding up all changes made by operations related to this book. Having an additional *quantity* attribute would only create data redundancy again.

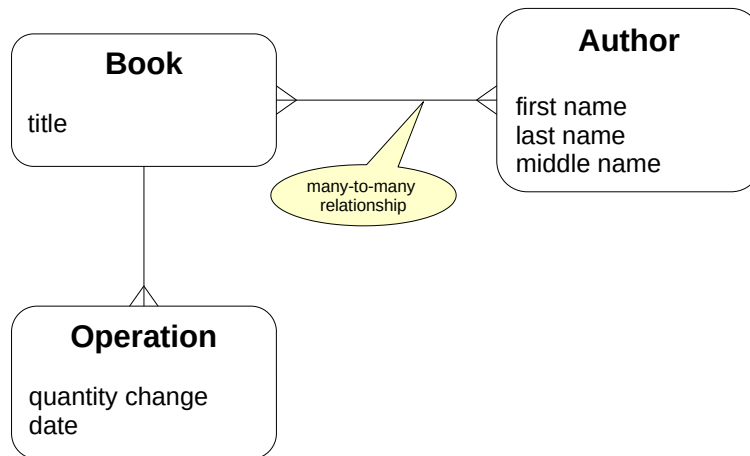
This solution might raise some eyebrows at first. Is it really a good idea to have to calculate the amount every time instead of having a separate field to query? The answer is that we can simply create a view that shows the current amount of books in store. This will not lead to redundancy, because the view is just another query.

But what about performance? If summing up all changes brings too much overhead, we can resort to denormalization: add the *quantity* field to the *books* table and ensure its consistency with the *operations* table. Whether to do this or not is beyond the scope of this course (it is discussed in the QPT "Query Performance Tuning" course). Common sense suggests that it's not required for our bare-bones app, but we will get back to denormalization when we get to the "Triggers" lecture.

Thus, 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 go deeper: separate books from authors and tie them by a many-to-many relationship: a 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 author's attributes will be their first, last and middle name. It makes sense because we may need to work with each of these attributes separately, for example, to display the author's last name and initials.

## Application schema

```
=> \c bookstore
```

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

The 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	

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)

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.

GENERATED AS IDENTITY columns take their values from special database objects called sequences. We can obtain 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 | 1
title   | The Tale of Tsar Saltan
-[ RECORD 2 ]-----
book_id | 2
title   | Romeo and Juliet
-[ RECORD 3 ]-----
book_id | 3
title   | Good Omens
-[ RECORD 4 ]-----
book_id | 4
title   | Dark Avenues
-[ RECORD 5 ]-----
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
-[ RECORD 6 ]-----
book_id | 6
title   | Three Men in a Boat (To Say Nothing of the Dog)
-[ RECORD 7 ]-----
book_id | 7
title   | 101 Famous Poems
```

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, 1_000_000);
```

```
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: 105.677 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: 0.294 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)

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	2025-04-22
2	1	10	2025-04-22
3	1	-1	2025-04-22

(3 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").

You will see some other data types in the examples. Check the handout "Basic Data Types and Functions" (datatypes.pdf) for details.

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").



## 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
- hard to maintain consistency

## Functions

- reading data via table functions
- writing data by calling functions
- the application is separated from the data model and is limited by API
- lots of wrapper functions required
- potential performance issues

11

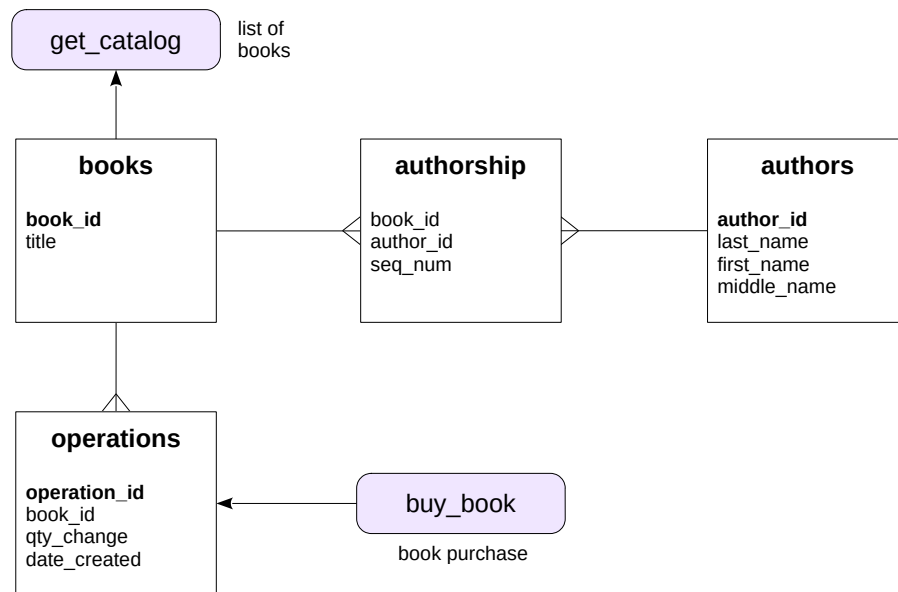
There are several ways to set up an API.

The first option is to allow the application to access and modify database tables directly. In this case, the application must have the exact 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 is the approach that provides the most flexibility.

Another option is to forbid direct table access from the application and allow only function calls. Reading data 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 functions.



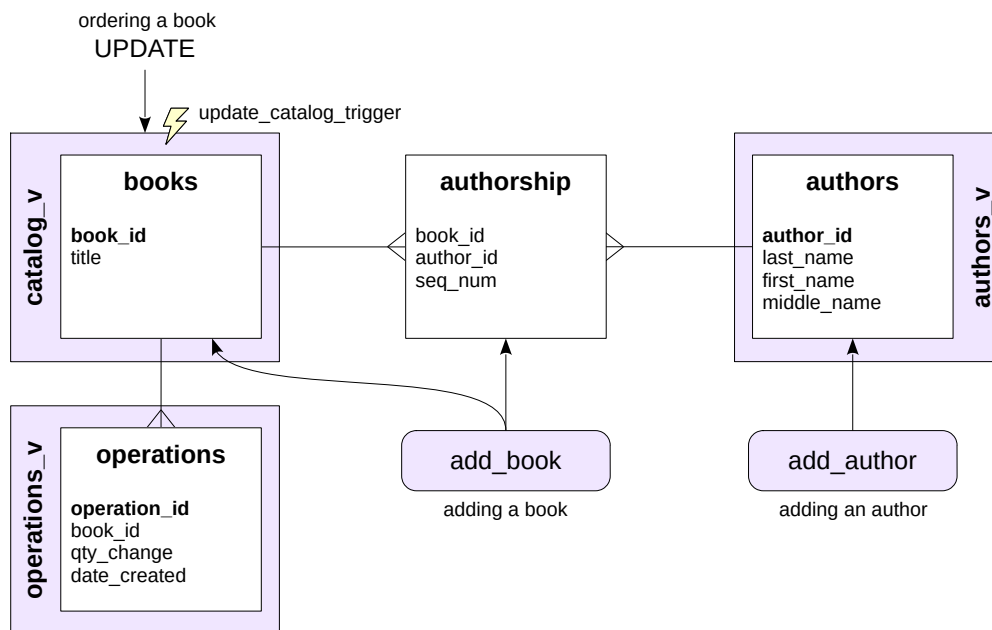
In this application, we will try different ways of setting up the interface (although it's usually better to stick to one approach when developing real applications).

The store will use API functions:

- **get\_catalog** for looking up books (see "SQL. Composite Types")
- **buy\_book** for making a purchase (see "PL/pgSQL. Query Execution")



# Employee API



The admin panel is going to retrieve data by accessing the following views (which we 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. Query Execution” 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 name, last name, and middle name (if available):

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

author_id	display_name
1	Alexander Sergeyevich Pushkin
5	Ivan Alekseyevich Bunin
7	Jerome Klapka Jerome
2	William Shakespeare
3	Terry Pratchett
4	Neil Gaiman
6	Jonathan Swift

(7 rows)

The catalog view displays only the book title for now:

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

```

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

```

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

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

```

book_id | op_type | qty_change | date_created
-----+-----+-----+-----
      1 | Arrival |         10 | 22.04.2025
      1 | Arrival |         10 | 22.04.2025
      1 | Sale   |          1 | 22.04.2025
(3 rows)

```

# Takeaways



Database design is a separate complex topic

theory is important, but it should not take precedence over common sense

Normalized data simplifies the development and facilitates consistency support

The API can use tables, views, functions, and triggers

## Practice



1. Make sure that you are connected to the *bookstore* database and the *bookstore* schema is the current one.
2. Create *books*, *authors*, *authorship*, and *operations* tables with all the necessary constraints, exactly as shown in the demo.
3. Insert 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 the data in “Books”, “Authors”, and “Catalog” tabs.

16

1. Use `current_database()` and `current_schema()` functions.
2. Use the demonstrated output of `psql`'s `\d` commands as a reference.
3. You can use the data shown in the demo, or come up with your own data.
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 necessary.

## Task 1. Database and schema

```
=> SELECT current_database(), current_schema();
```

```
current_database | current_schema
-----+-----
bookstore       | bookstore
(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

Operations:

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

INSERT 0 3

## 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 'Sale'
       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