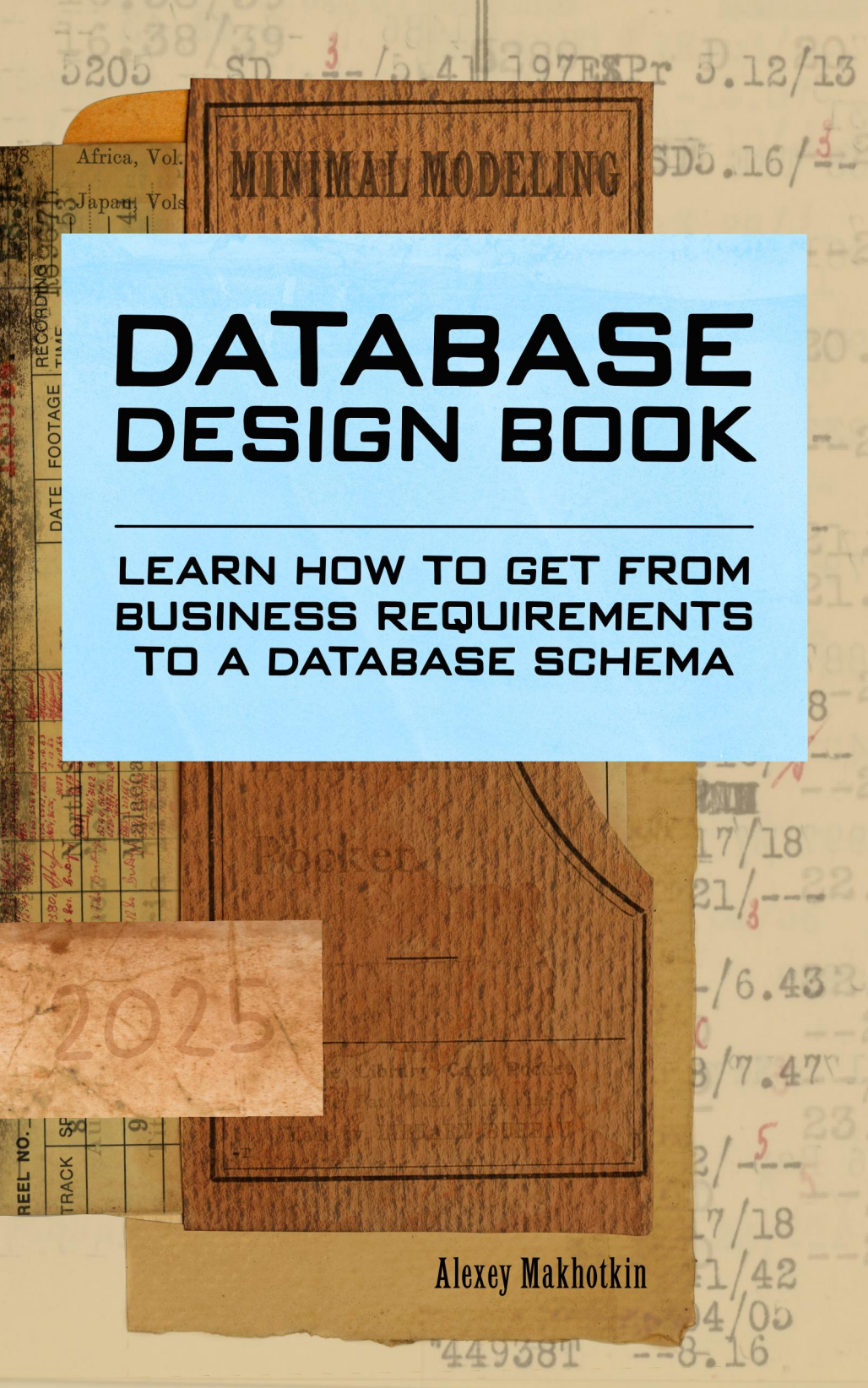

DATABASE DESIGN BOOK

LEARN HOW TO GET FROM BUSINESS REQUIREMENTS TO A DATABASE SCHEMA

Alexey Makhotkin

DATABASE DESIGN BOOK

learn how to get from business requirements to a database schema

ALEXEY MAKHOTKIN

2025

First Edition (revision: 2025-05-17)

Database Design Book: https://databasedesignbook.com/

Minimal Modeling Substack: https://minimalmodeling.substack.com/

Twitter: https://twitter.com/alexeymakhotkin

Cover design by Anna Golde
(https://www.linkedin.com/in/annagolde/).

CONTENTS

1

INTRODUCTION

This is a book on database design. You can find book updates and extra material at https://kb.databasedesignbook.com/.

1.1 WHO IS THIS BOOK FOR?

The goal of this book is to help you get from a vague idea of what you need to implement (e.g., *"I need to build a website to manage schedule and instructor appointments for our gym"*) **to the comprehensive definition of database tables.**

If you're a beginner, this book is for you. If you know that you need a database for your project, but are not sure how to begin, this is for you, too. If you've learned something about database design theory, but not sure if your design would work: you'll find help here. And if you struggle with some parts of the business domain, while the rest is clear, the approach laid out in this book will give clarity.

1.2 WHAT LEVEL OF UNDERSTANDING IS THIS BOOK AIMED AT?

The first part of the book is completely database-agnostic. All you need to understand is how your business works. This does not have to be a business, either: just some problem that necessitates a database.

The second part of the book requires some level of familiarity with common databases: how the tables are created, what physical data types exist, what are primary key and index, how the tables would be queried, and how to insert and update data.

We begin by assuming that you will use one of the traditional relational database servers, such as MySQL or PostgreSQL.

You do not need to understand database theory, such as normal forms.

1.3 WHAT YOU WOULD GET FROM READING THIS BOOK

The idea of the book is to make every step of table design actionable.

You will learn **how to extract the logical model** from a free-form description of the problem. This will also help to clarify your understanding of the problem, because the process forces you to be more precise and reduces hand waving.

You will learn **how the logical model is translated into a physical table design**. There are a dozen possible ways to represent logical models in physical space. First we discuss one such design strategy (a table per anchor); later we'll see why one might want to occasionally use a different one .

The book will help you answer a common question: **how can I prove, even to myself, that my design is complete**? Also, you will be able to defend this design, be it at a job interview or in a database exam.

A logical model is **a very good way to explain your design to other people**, for example to your team members or business stakeholders. This way of presenting your design immediately answers many questions that people may have. Because of that they can give better feedback and confirm that you're on the same page.

You will learn to **think separately about logical and physical aspects of your database**. This brings a lot of clarity: your problem lies either on the one or the other side, and you can save a lot of complexity if the other side is fixed.

Our approach is strongly rooted in relational modeling. However, it is **designed to accommodate non-relational NoSQL systems**, and even more specialized technologies such as Amazon S3. As mentioned earlier, the logical model is not dependent on any specific database server system: you just have to choose a suitable table design strategy.

1.4 WHO I AM

My name is Alexey Makhotkin. I have been working with databases for more than 25 years in various roles: as a software engineer, database administrator, team lead, head of software engineering. I've built and helped to build dozens of schemas over the years.

In 2021 I started the "Minimal Modeling" substack: https://minimalmodeling.substack.com/, trying to summarize what I have learned. This book is another step. You can contact me at squadette@gmail.com.

2

BUILDING A LOGICAL MODEL

Suppose that you want to **build a software system that stores some data in a database**. It may be one of hundreds possible kinds of software systems, such as:

- ecommerce: placing and fulfilling orders;
- money-tracking software, such as personal budget tracker or a business accounting system;
- social network with users, posts, and comments;
- tracking your progress and achievements in a role-playing game;
- etc., etc.

In each of those systems there is some sort of underlying database, with tables, columns, rows and datatypes. Your task is to design and build this database. Then the system will use it to store and retrieve the data.

How do you design and build this database? This chapter, as well as the entire book, aims to answer exactly this question.

2.1 SYSTEM REQUIREMENTS AS INPUT

What exactly are we building? In this book, it is your responsibility to specify the system that you are building. You need to gather and provide answers about what the system is supposed to do.

The point of this chapter is that your understanding of the requirements will gradually evolve.

In the beginning you have just **an informal text that roughly specifies what the system does**. Imagine that you sit down with somebody and explain what the system is supposed to do. They ask clarifying questions, you respond, and so it goes back and

3

"HELLO WORLD" USE CASE: PODCAST CATALOG

Let's design our first toy but realistic use case: a catalog of podcasts.

We're going to design an MVP: a minimum viable product. More complex use cases would be found later in the book.

We discuss how to extend this scope with more interesting features (e.g., categories) later in this section.

We're only going to build a logical model here. In the second part of the book we'll cover the physical design directly based on this model.

3.1 BUSINESS REQUIREMENTS

Let's use a screenshot of an episode from Apple Podcasts and see what information we have here (see the next page).

Here is a list of elements that we can see here:

- cover image;
- air date (*16 January 2025*);
- episode number (26);
- length (1 hr 3 min);
- episode title (*"Debanking explained"*);
- name of the show (*"Complex Systems with Patrick McKenzie"*);
- episode description (*"In this episode, . . . "*).

If you scroll down, there will be a bit more information, but let's begin with this. Oh wait, the actual audio file should prob-

4

BUILDING A PHYSICAL SCHEMA

So, now that we have a catalog of anchors, attributes and links, we can build a physical schema: the actual database tables. When you have a catalog, 75% of work is done. Constructing database tables is pretty straightforward at that point.

4.1 MANY TABLE DESIGN STRATEGIES ARE POSSIBLE

Imagine that you want to build a relatively simple, but non-trivial application, for example, a to-do list. You can write down a set of functional requirements for the app and you can define a logical model (exactly as described in the previous chapters). You do all that without any actual software development first.

The next step is to design the physical table schema. But let's do a thought experiment: suppose that we hire a dozen teams of software developers, give them the requirements document, and ask them to implement it as they see fit.

They are free to choose their favorite database server and to design tables any way they like. The only restriction is that the requirements must be implemented fully, without changes, or additional features, and so on.

There would most probably be twelve different physical schemas. Some of them may look very similar to each other, but a couple would probably look really exotic.

Physical schemas would be different because different teams would use different **table design strategies**. A table design strategy is a set of rules that prescribes how to store anchors, how to store attributes, and how to store links.

5

TABLE-PER-ANCHOR TABLE DESIGN STRATEGY

Table-per-anchor strategy is probably the **most straightforward way** of building tables:

- one table per anchor;
- one table per many-to-many links;
- each attribute is a column in its anchor's table;
- each only-one-to-many link is a column of a table for the "many" anchor.

This approach corresponds to the third and fourth normal forms as defined in the relational theory. It is roughly what they would teach you in the beginner level university courses on database design.

The resulting tables are clean, understandable, logical and error-proof. From the physical point of view, this design is neutral, it is not optimized for any specific access pattern, except for understandability.

5.1 ACTION PLAN

Creating tables from the logical model is straightforward. Here are the next actions:

Step 1. Insert table names in the list of anchors;
Step 2. For each attribute, add the column name and choose the data type;
Step 3. For each many-to-many link, choose the name of the table;
Step 4. For each only-one-to-many link, choose the name of the column;

6

OTHER TABLE DESIGN STRATEGIES

6.1 TABLE DESIGN CONCERNS

There are four main concerns that underlie every table design strategy.

First, the result tables must be able **to store all the data defined in the logical schema**. This requirement may sound trivial, but sometimes people cut corners or get confused. As a result, database schema may just be incomplete. One way to make an incomplete database schema is to skip defining a logical schema.

Second, you need to understand **the effort needed to add a new attribute, link or anchor** to the database. Note that here we only talk about adding a new element, and not about other ways of database evolution (changing, deleting, etc.). Adding a new element is probably the most common operation, and it is expected to be roughly the easiest.

Third, we need to consider how the database would **handle the commonly expected read-only queries, performance-wise**. Queries could be classified in three groups:

- **Naturally performant.** For example, retrieving some attributes by a set of IDs is virtually always the cheapest possible operation, because it is naturally optimized in almost all databases.
- **Easily optimizable.** For example, if you need to query by a value of some attribute, you can easily add an index on the corresponding table column.
- **Specifically optimized.** Some queries are so important for the system performance that you need to optimize the database specifically for them. You need to be careful with

7

DEALING WITH ABSENT DATA

Let's begin with attribute values as the most common case. As usual, we start with the business requirements.

Suppose that you run an e-commerce system selling dresses, and one of the most important properties of a dress is its color. Sooner or later you will find that dress color is not always available. One common case is that the data is imported from a third-party provider which just does not give you this information. You can't make them fix their data. How to deal with this?

Here is another example: in many social networks, there is usually a text area called "bio" where the user can enter any information about themselves. Of course, many users don't bother writing anything there, how do we deal with that?

Finally, here is another common important scenario. Imagine a serious online movies forum. Some day as a matter of policy all users are required to disclose what their favorite movie is. Without this information their access to the forum is restricted. However, in some jurisdictions they actually have a legal protection of their right to not disclose their favorite movie. So the users should be able to state that they refuse to provide this information, and our forum must honor it. How do we handle this?

7.1 USE CASE: USER'S BIO

Let's begin with the simplest case: "bio". Here is the attribute definition:

8

SECONDARY DATA

Anchors, links and attributes together are considered primary data. In other words, they are the source of truth.

For example, when we store the information about an order made by a customer, it is stored in anchor tables, attribute columns and link tables. If we lose this information, for example a table row is accidentally deleted, the system would think that this is how it is. If we delete one item from the order, we'll send a package without this item. If we delete information about the order, the system won't recognize the order number. This is primary data. If it gets deleted accidentally, you need to restore from backups.

There is another sort of data: secondary data. This is the data that was copied from the primary data, duplicated, reorganized, flattened, post-processed and so on. There are many ways to introduce and use the secondary data. If it gets deleted accidentally, it can be recalculated from the primary data (theoretically, at least).

Here is an incomplete list of things that we consider secondary data:

- cached (pre-computed) columns;
- pre-aggregated tables;
- denormalized columns and tables;
- ML models generated and updated based on data;
- materialized feeds;
- flat tables;
- full-text indexes;
- copies of data in different databases;
- data caches such as Memcached or in-memory Redis;

9

EVOLVING YOUR DATABASE

Inevitably, sooner or later in your project's lifecycle, you will have to change the structure of your database.

There are several practical reasons for the database to evolve:

- normal development of **new features**;
- changing the definition of existing database elements: anchors, attributes and links. This happens when the **business requirements change** in such a way that the original implementation is incompatible with them.
- improving database performance or scalability by **reorganizing physical schema**;
- introducing secondary representations, such as **derived columns and pre-aggregated tables**;
- **improving organizational scalability**: for example, making it easier to add new attributes by introducing a JSON column;
- **removing data** that is no longer needed, or no longer wanted;
- **migrating to a different database** for performance reasons, compliance reasons or for organizational scalability;
- **building data warehouses**: derived representations of primary data for better reporting, data analytics and so on.

Any database evolution requires changing three things:

- logical schema;
- application code;
- physical database itself.

Logical schema is just a document, so changing it is extremely cheap: you just edit the document. The effort needed to change the application and the database schema may vary significantly.

10

MOVIE TICKETS: REPEATED SALES PATTERN

Let's discuss how to model selling movie tickets. This use case illustrates a very common pattern that I call repeated sales pattern. We will only focus on the logical model. We're only going to talk about essential parts of this pattern, omitting some details.

"Repeated sales" pattern is obviously important to model many businesses: not only movie tickets, but also flight and train tickets, hotel reservations, cargo transportation and other examples. Outside of sales, this pattern arises in a well-known textbook case of "course assignments" that is often used in teaching database modeling.

10.1 BUSINESS REQUIREMENTS

Suppose that our business is a movie theater, with multiple screens. Inside every auditorium there are a number of seats, physical chairs. We want to sell tickets to people willing to occupy those chairs for a few hours at certain times of a specific day.

Our movie theater also buys rights to show various movies, as they are released.

Every day there are multiple showtimes at each auditorium. Anyone can buy a ticket for a certain time of the day at a certain date, when a certain movie will be shown.

For simplicity, we assume that every movie ticket costs exactly the same, no matter what time slot, movie, or a screen. It's not hard to add this improvement on top of the basic logical model we'll build.

Moreover, we won't even bother with payments. Generally speaking, there are several ways how you can get a ticket:

11

BOOKS AND WASHING MACHINES: POLYMORPHIC DATA PATTERN

11.1 BUSINESS REQUIREMENTS

Imagine an e-commerce website that sells a lot of different things. For example, books and washing machines, but also clothes, bicycles, LED lamps, vinyl records, groceries, gift cards, etc., etc.

For each type of item we want to help users find the best item that we offer. For example, people shopping for a washing machine may want to search by the vendor name, capacity, physical dimensions, and for some unique features that the washing machine industry has invented.

People buying books want to search by the author name, cover type, publication date, etc. This is an entirely different set of properties. (Note that we do not say "different set of attributes", because author information requires links too. This is something that is often omitted in discussions of this pattern.)

Same for the bicycles, groceries and so on.

But there are some things that are common between books, washing machines and so on: they can all be ordered and they have a price. You can place a single order for two items: a book and a washing machine, and this order will be delivered to you, maybe in two shipments.

Let's apply our approach to this sort of business and see how it works. Here, in addition to the logical model, we will also extensively discuss the physical table design.

12

PRACTICALITIES

You can experiment with the approach presented in this book using any tool that you're familiar with. You need to maintain three tables: list of anchors, list of attributes and list of links. Also, you need a place to keep a textual description of your project.

You also need to maintain the physical database schema somewhere. It depends on the way you develop your system. It may be an SQL file in your source repository, or an ORM definition, or maybe you directly design the tables in your database using some sort of UI, graphical or text-based. It's possible that you're only interested in the logical schema so you skip this part.

12.1 DOCUMENT-BASED CATALOG

It seems that the most natural choice here is some word processor, such as Google Docs (that's what I use). Tables in this book are formatted for readability, due to demands of e-book readers. For the actual table design process you need something simple.

On the following page you can see some example Google Docs screenshots (before typesetting).

12.2 SPREADSHEET-BASED CATALOG

Another type of tool that you could use is a spreadsheet, for example Google Sheets.

Create three tabs: anchors, attributes and links. Input the column names, and make the header "frozen", so that it stays on the screen when you scroll the rest of the document. Enable text wrapping on all the columns, and set text alignment to top.