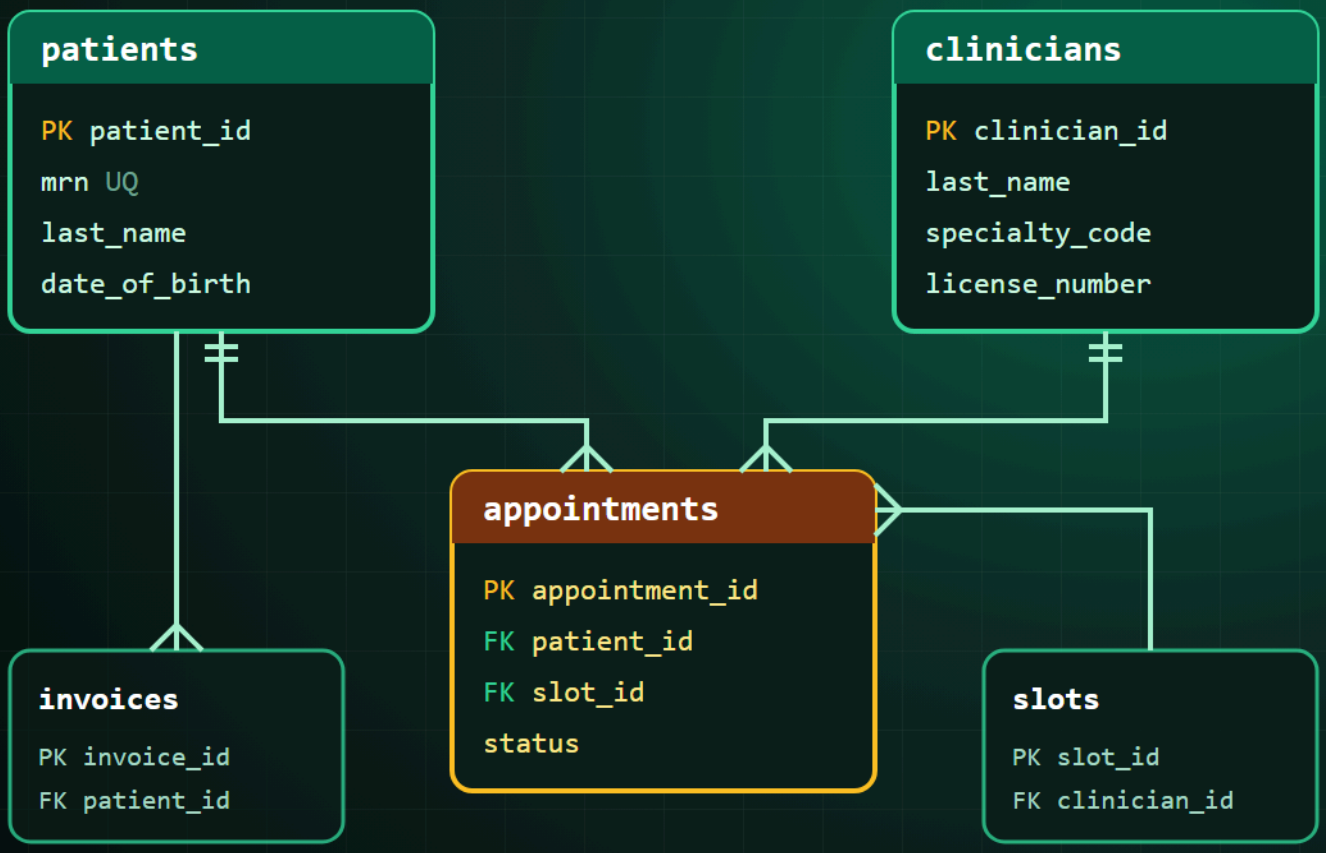


||--o{ · PK · FK · 3NF · grain · SCD2 · JSONB

A PRACTICAL GUIDE

Data Modeling with Real Examples

From requirements to relational, analytical, and NoSQL models

ER diagrams · normalization · keys · time and history · star schemas · documents · graphs · migrations

SQL · DIMENSIONAL · NOSQL · GOVERNANCE

THENEXBOOKS

Data Modeling with Real Examples

A Practical Guide from Requirements to Relational, Analytical, and NoSQL Models

First Edition

THENEXBOOKS

Data Modeling with Real Examples
A Practical Guide from Requirements to Relational, Analytical, and NoSQL Models

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Nothing in this book is medical, legal, or compliance advice; the healthcare scenario is used only to illustrate data modeling. Database products evolve quickly; always consult the official documentation for the versions you use.

PostgreSQL, MySQL, SQL Server, Oracle, MariaDB, MongoDB, Amazon DynamoDB, Apache Cassandra, Neo4j, dbt, and other product names are trademarks of their respective owners. HL7 and FHIR are registered trademarks of Health Level Seven International. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by any of these companies or organizations.

Clearbrook Health, the running example in this book, is fictional, as are all patients, clinicians, clinics, payers, partners, and data shown in its examples. Any resemblance to real people, organizations, or records is coincidental.

Preface

Every application is, underneath, a data model. Screens come and go, frameworks are replaced, services are split and merged — but the tables, documents, and keys that hold the business's facts tend to outlive all of them. A good model makes features easy to add, queries easy to write, and bugs hard to introduce. A poor one leaks into every line of code: duplicated data that drifts out of sync, reports that disagree with each other, migrations nobody dares to run.

Yet data modeling is often learned by accident. Many developers pick up a few rules — "normalize to third normal form", "add an `id` column", "use JSON when in doubt" — and discover the exceptions in production. Textbooks explain the theory with toy examples; real projects are messier. Appointments overlap, prices change over time, a person is both a patient and an employee, invoices must never be edited, and the analytics team needs the same data shaped completely differently.

This book teaches data modeling the way you will actually use it: through real examples. We start from requirements and build up the fundamentals — entities, relationships, keys, normalization, and time — then apply them to complete business domains, move into analytical and dimensional models, cover document, key-value, wide-column, graph, and time-series models, and finish with evolving and governing models in production. Every idea is shown on concrete tables with sample rows and queries, including the common mistakes and how to fix them.

Who This Book Is For

- **Application developers** who design tables for their features and want models that stay correct as the product grows.
- **Data and analytics engineers** who build warehouses and need a solid grounding in dimensional and modern analytics modeling.
- **Analysts** who write SQL every day and want to understand why data is shaped the way it is — and how to shape it better.
- **Tech leads and architects** who review schemas and choose between relational, document, and other data stores.

What You Will Learn

- How to turn requirements into conceptual, logical, and physical models, and how to read and draw ER diagrams.
- Keys and identity, normalization from first normal form to BCNF, and when denormalization is the right call.
- Patterns for many-to-many relationships, hierarchies, polymorphism, constraints, and business rules in the schema.
- How to model time: effective dates, history tables, temporal data, and audit trails.

- Complete real examples: scheduling, orders and inventory, invoices and insurance claims, multi-tenant permissions, and flexible attributes.
- Dimensional modeling: facts, dimensions, grain, star schemas, slowly changing dimensions, and a full warehouse example — plus Data Vault and lakehouse approaches.
- Document, key-value, wide-column, graph, time-series, and event models, and modeling for APIs and integration.
- Zero-downtime schema evolution, data quality, governance, privacy, and security by design.

How This Book Is Organized

Part I — Data Modeling Foundations (Chapters 1–5) covers the modeling process, levels of models, ER diagrams, and keys. **Part II — Relational Modeling in Practice** (Chapters 6–10) covers normalization, denormalization, relationship patterns, constraints, and time. **Part III — Real Examples: Modeling Business Domains** (Chapters 11–15) applies everything to five complete domains. **Part IV — Analytical and Dimensional Modeling** (Chapters 16–20) builds models for reporting and analytics. **Part V — NoSQL, Events, and Integration** (Chapters 21–25) models the same business with other data models. **Part VI — Evolving and Governing Models** (Chapters 26–28) keeps models healthy in production and ends with an end-to-end case study. The appendices provide a cheat sheet, templates and a pattern reference, a model review checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Clearbrook Health

Throughout the book we model the data of **Clearbrook Health**, a fictional network of outpatient clinics. Patients book appointments online and see clinicians who work at several clinics. Visits produce diagnoses and prescriptions, which Clearbrook's in-house pharmacy fills — the same pharmacy also sells everyday health products online. Patients receive invoices, pay by card, and have insurance claims submitted on their behalf. Clearbrook licenses its scheduling software to partner clinics, runs a data warehouse for operations reporting, collects readings from home blood-pressure monitors, and tracks referrals between clinicians.

That single, realistic business contains nearly every modeling problem you will meet elsewhere: people with many roles, time slots that must never overlap, many-to-many relationships, hierarchies, money that must be exact and auditable, flexible documents, multi-tenant data, analytics, time series, networks, and strict privacy requirements. The healthcare setting is purely illustrative — the patterns apply equally to retail, finance, logistics, SaaS, and any other domain.

Conventions

SQL examples use PostgreSQL syntax, with notes where MySQL, SQL Server, or Oracle differ in ways that matter. Tables use plural snake_case names and named constraints. Diagrams are drawn as plain text so you can reproduce them anywhere. NoSQL chapters use MongoDB-style documents, DynamoDB-style items, Cassandra CQL, and Cypher for graphs.

> **TIP**
> A recommended practice or a technique that makes your models easier to work with.

> **NOTE**
> Background information, product differences, or a useful aside.

> **WARNING**
> A common modeling mistake that leads to wrong data, painful migrations, or slow queries.

Contents

Preface

PART I — DATA MODELING FOUNDATIONS

1 What Data Modeling Is and Why It Still Matters

2 From Requirements to Model: Finding Entities, Attributes, and Relationships

3 Conceptual, Logical, and Physical Models

4 Reading and Drawing ER Diagrams: Crow's Foot, Cardinality, and Optionality

5 Keys and Identity: Natural Keys, Surrogate Keys, UUIDs, and Constraints

PART II — RELATIONAL MODELING IN PRACTICE

6 Normalization Step by Step: From Messy Spreadsheets to BCNF

7 When and How to Denormalize

8 Modeling Relationships: Many-to-Many, Self-References, Hierarchies, and Polymorphism

9 Data Types, Constraints, and Business Rules in the Schema

10 Modeling Time: Effective Dates, History Tables, Temporal Data, and Audit Trails

PART III — REAL EXAMPLES: MODELING BUSINESS DOMAINS

11 Real Example: Patients, Clinicians, and Appointment Scheduling

12 Real Example: Orders, Inventory, and the Pharmacy Storefront

13 Real Example: Invoices, Payments, and Insurance Claims

14 Real Example: Users, Roles, Permissions, and Multi-Tenant SaaS

15 Real Example: Documents, Tags, and Flexible Attributes (EAV vs. JSON Columns)

PART IV — ANALYTICAL AND DIMENSIONAL MODELING

16 OLTP vs. OLAP: Why Analytics Needs a Different Model

17 Dimensional Modeling: Facts, Dimensions, Grain, and Star Schemas

18 Slowly Changing Dimensions, Conformed Dimensions, and Bridge Tables

19 Real Example: The Clearbrook Clinic Operations Warehouse

20 Modern Analytics Modeling: Data Vault, One Big Table, and the Lakehouse

PART V — NOSQL, EVENTS, AND INTEGRATION

21 Document Modeling: Embedding vs. Referencing

22 Key-Value and Wide-Column Modeling: Designing from Access Patterns

23 Graph Modeling: Referrals, Networks, and Recommendations

24 Time-Series and Event Data: Device Readings, Logs, and Event Sourcing

25 Modeling for APIs and Integration: JSON Schema, Contracts, and Canonical Models

PART VI — EVOLVING AND GOVERNING MODELS

26 Schema Evolution and Zero-Downtime Migrations

27 Data Quality, Governance, Privacy, and Security by Design

28 Case Study: Modeling the Clearbrook Health Platform End to End

APPENDICES

A Data Modeling Cheat Sheet

B Templates and Pattern Reference

C Model Review Checklist

D Glossary

PART I

Data Modeling Foundations

What data modeling is for, how to turn requirements into entities and relationships, the three levels of models, how to read and draw ER diagrams, and how to choose keys that identify things correctly.

1 · Why Data Modeling Matters
2 · From Requirements to Model
3 · Conceptual, Logical, and Physical Models
4 · ER Diagrams and Notation
5 · Keys and Identity

CHAPTER 1 · PART I — DATA MODELING FOUNDATIONS

What Data Modeling Is and Why It Still Matters

Every application has a data model, whether someone designed it on purpose or it grew one column at a time. In this chapter we define what a data model really is, show what goes wrong when nobody designs one, and introduce Clearbrook Health, the clinic network whose data we will model throughout the rest of the book.

A Spreadsheet That Became a System

Let us start with a story that almost every developer has lived through in some form. Clearbrook Health began as a single outpatient clinic. Its front desk tracked appointments in a shared spreadsheet: one row per appointment, with columns for the patient's name, phone number, the clinician, the date and time, and a free-text "notes" column. It worked well for one clinic with four clinicians. Everyone could see the whole day at a glance, and nobody had to learn a new tool.

Then Clearbrook opened a second clinic, and a third. Online booking came next, so a developer exported the spreadsheet into a database table with exactly the same columns. Here is roughly what that first table looked like:

appointments_v0.sql

```
CREATE TABLE appointments_v0 (
    id             SERIAL PRIMARY KEY,
    patient_name   TEXT,
    patient_phone  TEXT,
    clinician      TEXT,
    clinic         TEXT,
    appt_date      TEXT,
    appt_time      TEXT,
    status         TEXT,
    notes          TEXT
);
```

And here are a few rows from it, exactly as the front desk and the booking form had entered them:

id	patient_name	patient_phone	clinician	clinic	appt_date	appt_time	status
101	Dana Whitfield	555-0142	Dr. Osei	Northside	2026-03-04	09:00	booked
102	dana whitfield	(555) 0142	Osei, K.	North Side	04/03/2026	9am	Booked
103	Marcus Lindqvist		Dr. Osei	Northside	2026-03-04	09:00	confirmed
104	D. Whitfield	555-0142	Dr. Patel	Lakeview	2026-03-31	14:30	cancelled?

Nothing about this table is syntactically wrong. The database happily accepts every row. Yet look at what it cannot tell you. Is "Dana Whitfield", "dana whitfield", and "D. Whitfield" one person or three? Is "04/03/2026" the fourth of March or the third of April? Did Dr. Osei really have two patients at 09:00 at Northside, or is row 103 a mistake? What does "confirmed" mean when the other rows say "booked", and what on earth is "cancelled?"?

Every one of those questions is a **modeling** question, not a coding question. The table stores data, but it does not describe the business. It does not say that a patient is a distinct thing with an identity, that a clinician can only be in one place at a time, that a date is a date, or that an appointment moves through a known set of states. Because the table does not say those things, every program that reads it has to guess, and every person who writes to it can quietly break the rules. Monthly reports disagree with each other. A patient receives two reminder texts. A clinician gets double-booked. Nobody can answer "how many distinct patients did we see last quarter?" with confidence.

This book is about avoiding that situation deliberately — and about repairing it when you inherit it.

What a Data Model Actually Is

A **data model** is an explicit description of the things a system needs to remember, the facts it records about them, how those things relate to each other, and the rules the data must obey. It is a statement about the business expressed in a form a database can enforce. Four ingredients show up in every model, regardless of the technology underneath it:

- **Entities** — the things we keep data about: patients, clinicians, clinics, appointments, invoices. An entity has an identity that persists even as its details change. Dana Whitfield is still the same patient after she changes her phone number.
- **Attributes** — the facts we record about each entity: a patient's date of birth, an appointment's start time, an invoice's total. Each attribute has a type and a meaning.
- **Relationships** — how entities connect: a patient *books* appointments, a clinician *works at* one or more clinics, an invoice *contains* invoice lines. Relationships have cardinality (one or many) and optionality (required or not).
- **Constraints** — the rules that valid data must satisfy: every appointment belongs to exactly one patient, a slot's end time is after its start time, status is one of a fixed list of values, a medical record number is unique.

Notice that none of those four ingredients mention tables, columns, or SQL. A data model can be drawn on a whiteboard, written as a list of sentences, or expressed as JSON documents in a document database. Tables are one very common way of *implementing* a model. That distinction — between what the business needs and how a particular database stores it — is the subject of Chapter 3, where we separate conceptual, logical, and physical models.

Here is the same appointment information from our spreadsheet expressed as a small, deliberately designed relational model. Do not worry about every detail yet; we will build up to designs like this step by step over the next several chapters.

appointments_v1.sql

```
CREATE TABLE patients (
    patient_id     BIGINT GENERATED ALWAYS AS IDENTITY,
    mrn            TEXT        NOT NULL,
    first_name     TEXT        NOT NULL,
    last_name      TEXT        NOT NULL,
    date_of_birth  DATE        NOT NULL,
    phone          TEXT,
    created_at     TIMESTAMPTZ NOT NULL DEFAULT now(),
    CONSTRAINT pk_patients PRIMARY KEY (patient_id),
    CONSTRAINT uq_patients_mrn UNIQUE (mrn)
);

CREATE TABLE clinics (
    clinic_id  BIGINT GENERATED ALWAYS AS IDENTITY,
    name       TEXT NOT NULL,
    city       TEXT NOT NULL,
    timezone   TEXT NOT NULL,
    CONSTRAINT pk_clinics PRIMARY KEY (clinic_id),
    CONSTRAINT uq_clinics_name UNIQUE (name)
);

CREATE TABLE clinicians (
    clinician_id    BIGINT GENERATED ALWAYS AS IDENTITY,
    first_name      TEXT NOT NULL,
    last_name       TEXT NOT NULL,
    specialty_code  TEXT NOT NULL,
    CONSTRAINT pk_clinicians PRIMARY KEY (clinician_id)
);

CREATE TABLE appointments (
    appointment_id  BIGINT GENERATED ALWAYS AS IDENTITY,
    patient_id      BIGINT      NOT NULL,
    clinician_id    BIGINT      NOT NULL,
    clinic_id       BIGINT      NOT NULL,
    starts_at       TIMESTAMPTZ NOT NULL,
    status          TEXT        NOT NULL DEFAULT 'BOOKED',
    reason          TEXT,
    booked_at       TIMESTAMPTZ NOT NULL DEFAULT now(),
    CONSTRAINT pk_appointments PRIMARY KEY (appointment_id),
    CONSTRAINT fk_appointments_patient
        FOREIGN KEY (patient_id) REFERENCES patients (patient_id),
    CONSTRAINT fk_appointments_clinician
        FOREIGN KEY (clinician_id) REFERENCES clinicians (clinician_id),
    CONSTRAINT fk_appointments_clinic
        FOREIGN KEY (clinic_id) REFERENCES clinics (clinic_id),
    CONSTRAINT ck_appointments_status CHECK (status IN
        ('BOOKED', 'CHECKED_IN', 'COMPLETED', 'CANCELLED', 'NO_SHOW'))
);
```

Compare this with `appointments_v0` and count the questions the database can now answer on its own. A patient is a row in `patients` with a unique medical record number, so "Dana Whitfield" exists exactly once and every appointment points at her by `patient_id`. The appointment time is a `TIMESTAMPTZ`, an unambiguous instant, so the day-month confusion is gone. The status is restricted to five values, so "cancelled?" is rejected at insert time. A foreign key guarantees that an appointment cannot reference a clinician who does not exist.

The v1 design still has gaps. Nothing yet stops Dr. Osei from being booked twice at 09:00; we will fix that in Chapter 11 with appointment slots and an exclusion constraint. But the important shift has already happened: the rules of the business now live *in the model*, not scattered across every program and every person that touches the data.

> **NOTE**
>
> We use PostgreSQL-flavored SQL throughout the book because it is widely available, standards-oriented, and supports the features we need — identity columns, check constraints, range types, `JSONB`, and more. Where MySQL, SQL Server, or Oracle differ in a way that matters for modeling, we will say so. The modeling ideas themselves are portable.

The Model Is the Business Rules, Written Down

A useful way to think about a data model is as a set of **sentences about the business** that the database is able to check. Each constraint in the v1 design corresponds to a sentence someone at Clearbrook could say out loud:

Business sentence	How the model enforces it
Every patient has exactly one medical record number.	`mrn NOT NULL` plus `uq_patients_mrn`
An appointment is always for a known patient.	`patient_id NOT NULL` plus a foreign key
An appointment happens at a specific instant.	`starts_at TIMESTAMPTZ NOT NULL`
Appointments are booked, checked in, completed, cancelled, or no-shows.	`ck_appointments_status`
A patient may have many appointments.	Many rows can share one `patient_id`

When you design a model, you are really doing two jobs at once. First, you are **discovering** those sentences — interviewing people, reading forms and reports, and noticing the rules nobody bothered to write down. Chapter 2 is entirely about that discovery process. Second, you are **encoding** the sentences in a structure that makes the right things easy and the wrong things impossible, or at least hard. The rest of the book is largely about the encoding: which structures exist, what each one costs, and when to choose which.

This framing also explains why modeling mistakes are so expensive. If an application function has a bug, you fix the function and redeploy. If the model encodes the wrong sentence — say, "a patient has one phone number" when patients actually have mobile, home, and work numbers — then every row already stored reflects that wrong belief, every query was written around it, and every downstream report depends on it. Fixing the model means migrating data, rewriting queries, and coordinating changes across teams. Chapter 26 covers how to do that safely, but it is always cheaper to get the sentence right the first time.

Why Data Modeling Still Matters

Some developers assume modern tools have made modeling unnecessary. Each common version of that argument contains a grain of truth and a larger misunderstanding.

"Our database is schemaless"

Document databases such as MongoDB do not require you to declare a schema before inserting data. That is genuinely useful for flexible or rapidly evolving data. But "schemaless" describes where the schema is enforced, not whether it exists. If your application reads `order.customer.email`, then your data has a schema: it is written in your application code instead of in the database. When two services disagree about whether the field is called `email` or `emailAddress`, you have a modeling problem that the database cannot catch for you. Chapter 21 shows how document models are designed — around access patterns, embedding, and referencing — and why that design needs as much care as a relational schema, just different care.

"The ORM generates the tables"

Object-relational mappers can create tables from class definitions, and that is convenient. But an ORM generates tables from whatever classes you wrote. If the `Appointment` class has a `patientName` string instead of a reference to a `Patient`, the generated table will faithfully reproduce the spreadsheet problem. The ORM is a translation tool; the model is still your decision — and generated schemas often lack check and unique constraints because the defaults are permissive.

"We can fix it in the application"

Validation in application code is necessary — it produces friendly error messages and catches problems early. But application validation alone is not sufficient, because data outlives code and arrives through more than one door. At Clearbrook, appointments are created by the patient booking site, by the front-desk application, by a nightly import from partner clinics, by support engineers running one-off scripts, and eventually by integrations nobody has imagined yet. Each of those paths would need to implement every rule identically, forever. A constraint in the database is implemented once and applies to every path.

"Storage is cheap, so duplication does not matter"

The problem with duplication was never primarily disk space; it is **consistency**. If a patient's phone number is stored in forty appointment rows, changing it means updating forty rows, and missing one means two different "truths" exist. Normalization (Chapter 6) is fundamentally about storing each fact once. Deliberate duplication has its place (Chapter 7), but it should be a choice with a plan for keeping copies in sync, not an accident.

> **TIP**
>
> When someone argues that modeling is unnecessary, ask a concrete question the business will need answered, such as "How many distinct patients missed more than one appointment last quarter?" Then ask how the current design would answer it. The gap between the question and the design is usually the best argument for modeling.

What Goes Wrong Without a Model: A Closer Look

Let us make the costs concrete by trying to answer a simple question against `appointments_v0`: how many appointments did each clinic have in March 2026?

```
-- Attempt against the spreadsheet-style table
SELECT clinic, count(*) AS appointments
FROM appointments_v0
WHERE appt_date LIKE '2026-03-%'
GROUP BY clinic
ORDER BY clinic;
```

With our four sample rows, this query returns a result that looks plausible but is wrong in three separate ways:

clinic	appointments
Lakeview	1
Northside	2

- Row 102 is missing because its date was stored as `04/03/2026`, which does not match the pattern.
- Even if we matched it, "North Side" would be counted as a separate clinic from "Northside".
- Row 104, "cancelled?", is counted as an appointment, although it probably should not be.

Every attempt to fix the query adds more guesses: try several date formats, normalize clinic names with `lower()` and `replace()`, exclude statuses that look like cancellations. Each guess is a small piece of undocumented business logic, and each report writer will guess slightly differently. That is how organizations end up with three dashboards that show three different appointment counts, and a meeting to decide which one is "right".

Against the v1 design, the same question has exactly one sensible answer:

```
-- Against the modeled schema: one correct way to ask
SELECT c.name AS clinic,
       count(*) AS appointments
FROM appointments a
JOIN clinics c ON c.clinic_id = a.clinic_id
WHERE a.starts_at >= TIMESTAMPTZ '2026-03-01 00:00:00+00'
  AND a.starts_at <  TIMESTAMPTZ '2026-04-01 00:00:00+00'
  AND a.status <> 'CANCELLED'
GROUP BY c.name
ORDER BY c.name;
```

The query is not shorter, but it is *unambiguous*. Clinics are identified by key, so spelling variations cannot split them. Dates are real timestamps, so a half-open range (`>=` the start, `<` the next month) captures the month precisely. Status values are known, so excluding cancellations is a clear decision rather than a pattern-matching hack. (Whether "March" should be measured in UTC or in each clinic's local timezone is a genuine business question, and the `clinics.timezone` column gives us what we need to answer it either way. Chapter 10 returns to time zones in depth.)

> **WARNING**
>
> Storing dates, times, and amounts as text is one of the most common and most damaging modeling shortcuts. Text columns accept any value, sort incorrectly (`'10:00'` sorts before `'9:00'`), cannot be range-indexed meaningfully, and push parsing logic into every query. Use `DATE`, `TIMESTAMPTZ`, and `NUMERIC` from the first day.

The Duplicate Patient Problem

The hardest damage from the v0 design is not the dates — those can be cleaned up with some effort. It is identity. Once appointments exist without a stable reference to a patient, you have to **reconstruct** identity after the fact, a process usually called record matching or deduplication. A typical first attempt looks like this:

```
-- Guessing which rows belong to the same person
SELECT lower(regexp_replace(patient_name, '[^A-Za-z]', '', 'g')) AS name_key,
       regexp_replace(patient_phone, '[^0-9]', '', 'g')         AS phone_key,
       count(*) AS appointment_rows
FROM appointments_v0
GROUP BY 1, 2
ORDER BY appointment_rows DESC;
```

This groups "Dana Whitfield" and "dana whitfield" together because they normalize to the same key and share a phone number. But "D. Whitfield" becomes `dwhitfield`, a different key, even though the phone matches. Match on phone alone and you will merge family members who share a home number. Match on name alone and you will merge two unrelated people who happen to share a

common name. In healthcare, merging two different patients' records is a serious error, and failing to merge one patient's records means a clinician may not see her full history. There is no purely technical answer; real systems use careful matching rules and human review.

The modeled design avoids most of this problem by making identity a first-class concept. A patient gets a `patient_id` and a medical record number when she is registered. The booking form looks her up rather than retyping her name. Duplicates can still happen — someone registers twice with different emails — but they are exceptions to manage, not the default state of the data. Chapter 5 is devoted to identity: natural keys, surrogate keys, UUIDs, and how to choose.

What a Good Model Gives You

It is worth being explicit about what we are optimizing for, because "good model" is otherwise too vague to guide decisions. Throughout this book we will evaluate designs against five qualities:

1. **Correctness.** The model can represent every valid state of the business and, as far as practical, rejects invalid states. A patient cannot have two medical record numbers; an invoice line cannot belong to no invoice.
2. **Clarity.** Someone new to the system can read the schema and understand the business. Names mean what they say; each table represents one kind of thing; relationships are explicit rather than implied by matching strings.
3. **Queryability.** The questions the business asks can be answered with straightforward queries. If every report needs a dozen `CASE` expressions to reinterpret the data, the model is fighting its users.
4. **Performance.** The workload the system actually runs — its reads, writes, and their volumes — can be served efficiently with reasonable indexes, without heroic tuning. Performance is a property of a model *and* a workload together, so we always ask which workload.
5. **Evolvability.** When the business changes — a new clinic type, a new insurance rule, a new country — the model can absorb the change with additive migrations rather than a rewrite.

These qualities sometimes pull against each other. A fully normalized model maximizes correctness and clarity but may require many joins for a hot read path; a denormalized one reads faster but needs extra machinery to stay consistent. A flexible JSON column eases evolution but weakens the database's ability to check correctness. Good modeling is not about maximizing one quality; it is about making the trade-offs **consciously**, documenting them, and choosing differently for different parts of the system. You will see this pattern again and again: first a clean design, then an honest look at where it hurts, then a targeted adjustment.

One Business, Many Models

A single organization rarely has only one data model. Clearbrook, like most real businesses, needs different shapes of data for different jobs:

Need at Clearbrook	Model style	Covered in
Booking, billing, pharmacy orders	Normalized relational (OLTP)	Parts II and III
Operations dashboards and trends	Dimensional star schema	Part IV
Clinical documents with varying fields	Document or JSON columns	Chapters 15, 21
High-volume session and cache lookups	Key-value / wide-column	Chapter 22
Referral networks between clinicians	Graph	Chapter 23
Home blood-pressure readings	Time series / events	Chapter 24
Data exchanged with partners	API contracts, JSON Schema	Chapter 25

These are not competing religions; they are tools shaped by different workloads. The same underlying business concepts — patient, clinician, appointment — appear in all of them, but shaped differently. An appointment in the transactional model is a row with foreign keys. In the warehouse it becomes a fact row with keys to date, clinic, and clinician dimensions. In the referral graph, the clinician becomes a node and the referral becomes an edge. Understanding the concepts once, at the conceptual level, is what lets you translate them faithfully between these shapes. That is why Part I starts with concepts and requirements rather than with syntax.

Meet Clearbrook Health

From here on, nearly every example in this book uses one fictional organization, so you can watch a single model grow and stay consistent rather than meeting disconnected toy schemas in each chapter. Clearbrook Health is a network of outpatient clinics in several cities. To keep scale discussions concrete, let us assume Clearbrook operates about 40 clinics, employs several hundred clinicians, and handles on the order of two million appointments a year. These figures are hypothetical assumptions for the examples, not a claim about any real organization.

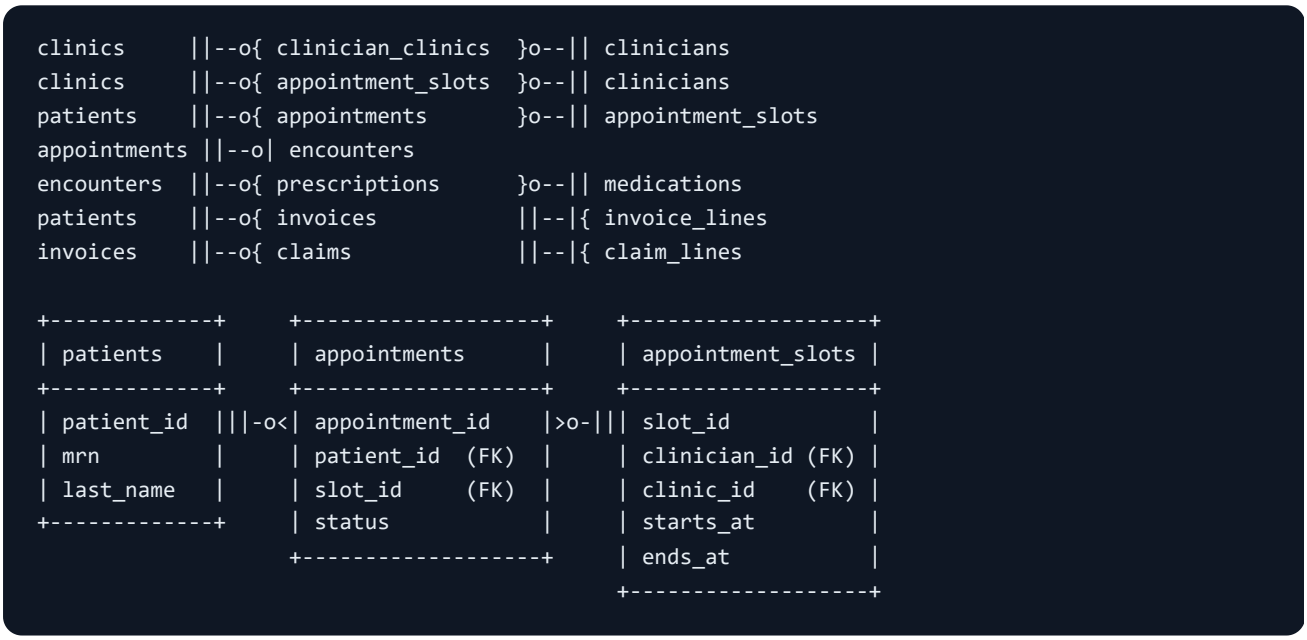
Here is what Clearbrook does, and therefore what its data must support:

- **Scheduling.** Clinicians publish appointment slots at one or more clinics. Patients book slots online or through the front desk. Appointments move through the states `BOOKED`, `CHECKED_IN`, `COMPLETED`, `CANCELLED`, and `NO_SHOW`.
- **Clinical records.** A completed appointment produces an encounter, which records notes, diagnoses (coded — real systems use standards such as ICD-10; our examples use obviously fake codes like `DX-0001`), and prescriptions.
- **Pharmacy.** Clearbrook's in-house pharmacy fills prescriptions and also sells over-the-counter products online, so it needs products, inventory, orders, and order lines.
- **Billing.** Patients receive invoices with lines; they pay by card or other methods; many visits are billed to insurance, which involves policies, payers such as the fictional "Northgate Mutual", claims, and claim lines.

- **Software as a service.** Clearbrook licenses its scheduling software to partner clinics such as the fictional "Riverbend Family Practice". Each partner is a tenant with its own users, roles, and permissions, and one tenant must never see another's data.
- **Analytics.** A data warehouse answers operational questions: no-show rates, clinician utilization, revenue by payer, pharmacy stock turnover.
- **Devices.** Some patients use home blood-pressure monitors that upload readings, a classic time-series workload.
- **Referrals.** Clinicians refer patients to specialists, forming a network that is naturally modeled as a graph.
- **Privacy.** Health data is sensitive and regulated. Laws such as HIPAA in the United States and GDPR in the European Union may apply to systems like this; the exact obligations are a question for a compliance team, not for a data model alone, but the model must make privacy controls possible.

The following diagram sketches the core of the transactional model at a very high level. It uses crow's-foot notation, which Chapter 4 explains in detail; for now, read `||--o{` as "one to zero-or-many".

clearbrook_core_overview.txt



Do not try to memorize this diagram. It is deliberately incomplete, and several relationships in it will be questioned as the book goes on: can one appointment produce several encounters? Should an invoice link to an encounter, an appointment, or neither? Chapter 2 teaches you to ask such questions, and Part III answers them.

We use consistent naming for Clearbrook tables: plural snake_case table names (`patients`), primary key columns named after the entity (`patient_id`), and constraint prefixes `pk_`, `fk_`, `uq_`, and `ck_`. Consistent naming is itself a modeling decision: it makes the schema readable and mistakes easier to spot.

How Modeling Fits into Building Software

Modeling is not a phase you finish before coding starts; it is an activity that runs alongside development. Still, it has a recognizable rhythm, and this book follows it. We will use the following workflow repeatedly:

1. **Collect requirements as questions and rules.** What must the system remember? What will people ask of it? What must never happen? (Chapter 2)
2. **Sketch a conceptual model.** Entities and relationships in business language, reviewed with the people who know the business. (Chapters 2–4)
3. **Refine a logical model.** Attributes, keys, normalized structures, and precise cardinalities, still largely independent of a particular database. (Chapters 3, 5, 6, 8)
4. **Produce a physical model.** Tables, types, constraints, indexes, and partitioning for a specific database and workload. (Chapters 3, 7, 9, 10)
5. **Prove it with queries and sample data.** Load realistic rows and write the queries the business needs. If a query is awkward or ambiguous, the model is telling you something.
6. **Evolve it safely.** Change the model as the business changes, with migrations that do not break running systems. (Chapter 26)

Step 5 is the one most often skipped. A diagram can look elegant and still fail the first real question, so we test models like code: insert awkward sample rows and run the queries that matter. When a design cannot express a valid situation, such as a patient with two insurance policies, that is a failing test.

A Small Proof in Practice

Here is what step 5 looks like for the v1 design. We insert a few rows, including one that should be rejected, and confirm the database behaves the way the business sentences say it should:

```
INSERT INTO patients (mrn, first_name, last_name, date_of_birth, phone)
VALUES ('MRN-000101', 'Dana', 'Whitfield', DATE '1987-06-12', '555-0142'),
       ('MRN-000102', 'Marcus', 'Lindqvist', DATE '1975-11-30', NULL);

INSERT INTO clinics (name, city, timezone)
VALUES ('Northside', 'Harbor City', 'America/Chicago'),
       ('Lakeview',  'Harbor City', 'America/Chicago');

INSERT INTO clinicians (first_name, last_name, specialty_code)
VALUES ('Kofi', 'Osei', 'FAMILY_MED'),
       ('Anjali', 'Patel', 'CARDIOLOGY');

-- Valid: a booked appointment for a known patient
INSERT INTO appointments (patient_id, clinician_id, clinic_id, starts_at)
VALUES (1, 1, 1, TIMESTAMPTZ '2026-03-04 09:00:00-06');

-- Rejected: status outside the allowed list
INSERT INTO appointments (patient_id, clinician_id, clinic_id, starts_at, status)
VALUES (1, 2, 2, TIMESTAMPTZ '2026-03-31 14:30:00-05', 'cancelled?');
-- ERROR: new row for relation "appointments" violates check constraint
```