Advanced Web Programming

SIB245007 | D-IV Sistem Informasi Bisnis

Meeting 4: **Database Design, Migrations, and Seeding**

Schema, Schema Change History, and Query Speed

What You'll Learn

1. Design a database schema and its **foreign key** relationships, along with design principles that keep data consistent and efficient
2. Master a **migration**'s lifecycle (`make:migration`, `up()`/`down()`, `migrate`, `rollback`) as a replayable history of schema evolution
3. Fill in data through **seeders** and **factories**, including large-scale bulk inserts
4. Build an **index** strategy and prove its impact with `EXPLAIN QUERY PLAN`
5. Place this approach on the bigger map: SQL vs. NoSQL, and Laravel's schema builder compared with other ecosystems

> This slide deck covers concepts. Designing the schema, writing migrations, and seeders for Simple POS happens in the practicum jobsheet.

Part 1

A Warehouse With a Catalog vs. Without One

A warehouse with a card catalog

- Every item sits on a labeled shelf
- The catalog records which shelf holds what
- Looking for an item: read the catalog, go straight to the right shelf

A warehouse without one

- The item is still there, somewhere
- Searching means walking every shelf one by one
- In a large warehouse, this eats a lot of time

> A database table's schema is the shelf layout itself; an index is the card catalog. Without the right index, the database engine can still find the data, just by scanning the entire table one row at a time.

Schema: A Table's Structural Design

> **Schema:** the design of a table's structure: column names, data types, and constraints (nullable, default, unique) that apply to every row in it.

- A schema gets sketched on paper before a single migration file gets written
- Every column has a data type (number, text, date) and constraints that keep the data valid

Designing a Good Schema

- **One fact, one place**: if a customer's name lives in five different tables, fixing one typo means chasing five rows at once, and all five can end up disagreeing with each other
- Pick the most specific data type available (a number for numbers, a date for dates), not `string` for everything, so the database itself rejects invalid data
- Constraints (`nullable`, `default`, `unique`) are a fence at the database level: they apply to every row that comes in, from any path, not only through one form that happens to validate it

> A deliberate exception: a **snapshot** column (a value frozen at one moment, covered in full after the example schema below) intentionally stores a copy of a value, not for the sake of "one place", but because that value has to freeze at a particular moment.

Foreign Key: Keeping Relationships Consistent

> **Foreign Key:** a column on one table pointing to the `id` of a row on another table, guaranteeing a row can never point at a parent row that doesn't exist.

authors (one) → **articles (many)**

- One `authors` row has many `articles`; every `articles` row stores an `author_id` pointing back to its author

Example Schema: A One-to-Many Relationship

Table	Main Columns
`authors`	`id`, `name`
`articles`	`id`, `author_id` (fk), `title`, `published_at`
`comments`	`id`, `article_id` (fk), `body`

- The `authors` → `articles` relationship is already covered; the `comments` table adds one more level: one `articles` row has many `comments`
- A diagram like this is called a simple **ERD** (Entity-Relationship Diagram), used to validate a design before writing a migration

Snapshot Columns: Values Frozen in Time

- A **snapshot** column computes its value **once**, at the moment the row is created (e.g. when an article is published, when a transaction happens), instead of recomputing it every time the row is **read** later
- Example: a comment count is recorded when an article is published; the per-item price on a transaction stays what it was at that moment, even if the product's catalog price changes afterward
- This value deliberately doesn't change even if its source data changes later, that's why it's called a **snapshot**: a picture of the past, not the present

> Computing a snapshot column doesn't mean the server can just trust whatever number the client (browser/app) sends. The validation & security meeting later covers why the server must compute this value itself before saving it, regardless of what the client submitted.

SQL vs. NoSQL: When a Schema Still Wins

Relational databases (SQL)

- A strict schema: columns, types, and FK relationships are decided up front
- ACID transactions keep several tables consistent at once
- The default choice for business applications with structured, interconnected data, like Simple POS

NoSQL databases (document/key-value)

- A flexible schema: every document can take a different shape
- Easier to scale out horizontally across many servers
- Shines when the data's shape shifts constantly, or the volume is massive

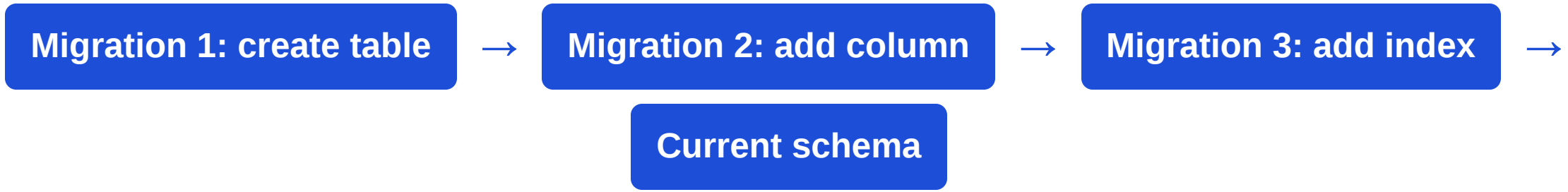
> The two aren't mutually exclusive: many systems pick relational for transactions and NoSQL elsewhere. This book stays relational, since Simple POS needs consistent transactions.

Part 2

Schema change history, and filling in initial data

Migrations: A Replayable Schema History

Migration: a PHP file describing a structural change to a table (creating, altering, or dropping columns) programmatically, so the database schema has a history that can be replayed on any other machine.



Similar to a git commit history for code: a migration is a commit history for the database schema, run in order according to its filename timestamp.

Creating a New Migration

```
php artisan make:migration create_articles_table
```

- This command creates a new, timestamped file under `database/migrations/`, e.g. `2026_08_20_122440_create_articles_table.php`
- That file holds two methods: `up()` (the change being applied) and `down()` (how to undo it)
- Schema code goes inside `up()`; the full example is on the `up()` and `down()` slide after this

Running Migrations

```
php artisan migrate
```

- Runs every migration that hasn't run yet, in order by its filename timestamp
- `php artisan migrate:status` shows which migrations have and haven't run, useful before running `migrate` on a fresh server

> A migration only ever runs once. One already recorded as run is skipped on later `migrate` calls, unless it's rolled back first.

`up()` and `down()`: Forward and Back

```php
public function up(): void
{
    Schema::create('articles', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->timestamps();
    });
}

public function down(): void
{
    Schema::dropIfExists('articles');
}
```

- `down()` is the exact inverse of `up()`: if `up()` creates a table, `down()` drops it
- `php artisan migrate:rollback` runs `down()` for the last migration batch, useful when a new migration turns out to be wrong

Why a Correct `down()` Is Worth Writing

- An honest `down()`, one that truly reverses `up()`, makes it safe to try a schema experiment and back out of it without a trace
- Without a correct `down()`, the only way back is editing the schema by hand, the exact thing migrations exist to avoid

> `php artisan migrate:fresh` runs `down()` on every migration, then `up()` again from scratch, a convenient shortcut during development. Never run it against a production database: every row of data is gone, not just the schema.

Writing a Migration for a Table

```
Schema::create('articles', function (Blueprint $table) {
    $table->id();
    $table->foreignId('author_id')->constrained();
    $table->string('title');
    $table->timestamps();
});
```

- `foreignId('author_id')` creates an `author_id` column as an unsigned big integer
- `->constrained()` adds a foreign key pointing to `authors.id`, following Laravel's naming convention
- This constraint protects data integrity, but doesn't necessarily mean the column already has an index (see Part 3)

> The plural, snake_case table name (`articles`) isn't a coincidence: the upcoming ORM & Data Relations meeting shows the singular model `Article` mapping to it through the same convention `constrained()` relies on above.

A Schema Evolves Through New Migrations

A requirement shows up after `articles` is already in use: it now needs a subtitle column. Don't edit the old migration that already ran, create a new one instead:

```
php artisan make:migration add_subtitle_to_articles_table
```

```
public function up(): void
{
    Schema::table('articles', function (Blueprint $table) {
        $table->string('subtitle')->nullable();
    });
}
```

- `Schema::table()` (not `Schema::create()`) alters a table that already exists, here adding a column

The Golden Rule: A New Migration, Never an Edit

- A database design is rarely right on the first try: the schema grows alongside the features, and migrations are the neatly recorded mechanism for that evolution
- The pattern repeats forever: need a new column, write a new migration; need to drop one, write another new migration that drops it

> An old migration edited after it has run leaves the schema history on another machine out of sync with the history on yours, that other machine has no idea anything changed, because the migration is already recorded as "already run". If the schema needs to change, a new migration is always the answer, not editing an old one.

Seeders and Factories: Filling in Initial Data

> **Seeder:** a PHP class that fills a table with initial or test data, run through `db:seed` or as part of `migrate:fresh --seed`.

> **Factory:** a blueprint for generating one realistic fake row of data for a Model, e.g. `User::factory()->create()` creates one new `users` row with sensible random values.

- A seeder usually orchestrates: it calls a factory for data that needs per-Model uniqueness (e.g. users), and does a direct bulk insert for large volumes (see the next slide)

Seeding at Scale: Row-by-Row vs. Bulk Insert

`Model::create()` in a loop

- One `INSERT` query per row
- Overhead: building an Eloquent object, firing model events
- Slow for hundreds or thousands of rows

`DB::table()->insert()` in batches

- One `INSERT` statement for many rows at once
- Far fewer round-trips to the database
- `array_chunk()` splits the batch because some database engines cap the number of rows per `INSERT`

```php
$articles = [];
foreach ($authorIds as $authorId) {
    for ($i = 0; $i < 30; $i++) {
        $articles[] = [
            'author_id' => $authorId,
            'title' => fake()->sentence(),
            'created_at' => now(),
            'updated_at' => now(),
        ];
    }
```

Consistency Across Tables: `DB::transaction()`

> When one operation touches more than one table that must stay consistent with each other (e.g. saving an order along with its line items), wrap it in a single `DB::transaction(function () { ... })`. If the process fails partway through, every change inside that block rolls back together, so the data never ends up half-saved.

Part 3

Proving the impact with EXPLAIN QUERY PLAN

Index: A Data Structure for Fast Lookups

> **Index:** an additional data structure the database engine builds over one or more columns, speeding up lookups on that column without scanning the entire table.

Without an index (SCAN)

- The database engine checks every row one by one
- Time grows linearly as the table grows

With an index (SEARCH)

- The database engine jumps straight to the relevant rows
- Barely affected by table size

Index Strategy: Which Columns Deserve One?

- Strong candidates: columns that show up often in `WHERE`, `JOIN`, or `ORDER BY`, foreign keys almost always qualify
- An index isn't free: every `INSERT` / `UPDATE` on that table also rewrites the index's structure, and an index takes up extra storage
- So don't index every column "just in case": measure first with `EXPLAIN QUERY PLAN` (covered in full after the next slide), then add only the indexes that prove necessary
- If two columns are always filtered together (e.g. `category_id` and `is_active`), a composite index on both can outperform two separate single-column indexes

`constrained()` Doesn't Always Mean There's an Index

- On some database engines (e.g. MySQL), `foreignId()->constrained()` automatically creates an index as a side effect
- On SQLite, `constrained()` only adds a foreign key constraint, not an index
- The constraint protects data integrity (rejecting inserts that point at a nonexistent row), but doesn't speed up lookups

Proving It With `EXPLAIN QUERY PLAN`

- `EXPLAIN QUERY PLAN`: a SQLite command that shows the strategy the database engine will use to run a query, without actually running it

```
sqlite3 database/database.sqlite \
  "EXPLAIN QUERY PLAN SELECT * FROM articles WHERE author_id = 3;"
```

- **Before an index**: the result includes `SCAN articles`
- **After an index is added**: the result changes to `SEARCH articles USING INDEX articles_author_id_index`

> If the `sqlite3` command isn't found, the alternative path is `php artisan tinker` with `DB::select("EXPLAIN QUERY PLAN ...")`.

An Index Isn't Premature Optimization

> On a table with a few dozen rows, the difference between SCAN and SEARCH is barely noticeable. On a table with thousands of rows, SCAN means every row gets examined on every single query, and that time grows linearly as the table grows. An index isn't premature optimization here; it's a fix for a performance bug that's already real the moment data approaches production scale.

Schema Builders Across Ecosystems

Ecosystem	How the Schema Is Defined	Migrations
Laravel (Eloquent)	Imperative PHP schema builder (`Schema::create()`)	Migration files written by hand, run via `artisan migrate`
Prisma (Node.js)	A declarative schema in one `schema.prisma` file	Migrations auto-generated from the schema diff
SQLAlchemy + Alembic (Python)	Python models (one class per table)	Semi-automatic migrations via Alembic, diffing models against the current schema

> The syntax differs, but the concept is the same: a database schema is code with a history that can be replayed on another machine. Once migrations click in Laravel, moving to another ecosystem is mostly a matter of syntax.

Applying This to Simple POS

- You'll apply the schema, migration, and index concepts directly to the Simple POS case study in the practicum jobsheet
- Designing the category-product-transaction tables, writing a seeder at hundreds-to-thousands-row scale, and proving the index with `EXPLAIN QUERY PLAN`

> Full code: `github.com/se-polinema/simple-pos` , branch `chapter-04`

Summary (1/2)

- A good schema stores each fact once, with the right data types and constraints; a relational database suits transactional data like Simple POS's, while NoSQL suits data whose shape keeps shifting or whose scale is massive
- A migration is a replayable schema history: `up()` applies a change, `down()` reverses it, and a schema evolves through new migrations, never by editing an old one

Summary (2/2)

- Seeders fill in initial data, factories mint fake data per Model; large-scale seeding uses `DB::table()->insert()` in batches, far more efficient than `Model::create()` one row at a time
- An index speeds up lookups (`SCAN` to `SEARCH`, proven with `EXPLAIN QUERY PLAN`), but it isn't free: pick candidates from `WHERE` / `JOIN` / `ORDER BY` columns, don't index everything

References & Discussion

Official Laravel documentation (Migrations, Query Builder, Seeding)

Full code: `github.com/se-polinema/simple-pos`

Next meeting: ORM & Data Relations