Pemrograman Web Lanjut

SIB245007 | D-IV Sistem Informasi Bisnis

Pertemuan 4: **Desain Basis Data, Migrasi, dan Seeding**

Skema, Riwayat Perubahan Skema, dan Kecepatan Query

Yang Akan Kamu Pelajari

1. Merancang skema basis data dan relasi **foreign key**, beserta prinsip desain yang menjaga data konsisten dan efisien
2. Menguasai siklus hidup **migration** (`make:migration`, `up()` / `down()`, `migrate`, `rollback`) sebagai riwayat evolusi skema
3. Mengisi data lewat **seeder** dan **factory**, termasuk bulk insert berskala besar
4. Menyusun strategi **index** dan membuktikan dampaknya dengan `EXPLAIN QUERY PLAN`
5. Memposisikan pendekatan ini di peta yang lebih besar: SQL vs NoSQL, dan schema builder Laravel dibanding ekosistem lain

> Slide ini membahas konsep. Merancang skema, menulis migration, dan seeder untuk Simple POS dikerjakan di jobsheet praktikum.

Bagian 1

Gudang Berkatalog vs Tanpa Katalog

Gudang dengan katalog kartu

- Setiap barang ada di rak berlabel
- Katalog mencatat rak mana menyimpan barang apa
- Cari barang: baca katalog, langsung ke rak yang tepat

Gudang tanpa katalog

- Barang tetap ada di suatu tempat
- Mencari berarti menyusuri rak satu per satu
- Pada gudang besar, ini menyita banyak waktu

> Skema tabel basis data adalah tata letak rak itu sendiri; index adalah katalog kartunya. Tanpa index yang tepat, mesin basis data tetap bisa menemukan datanya, hanya saja dengan menyusuri seluruh tabel satu per satu.

Skema: Rancangan Struktur Tabel

> **Skema:** rancangan struktur tabel, yaitu nama kolom, tipe data, dan batasan (nullable, default, unique) yang berlaku pada setiap barisnya.

- Skema digambar di atas kertas sebelum satu baris migration pun ditulis
- Setiap kolom punya tipe data (angka, teks, tanggal) dan batasan yang menjaga data tetap valid

Merancang Skema yang Baik

- **Satu fakta, satu tempat**: kalau nama pelanggan disimpan di lima tabel berbeda, memperbaiki satu typo berarti mengejar lima baris sekaligus, dan kelimanya bisa saling bertentangan
- Pilih tipe data sespesifik mungkin (angka untuk angka, tanggal untuk tanggal), bukan `string` untuk segalanya, supaya basis data sendiri yang menolak data tidak valid
- Batasan (`nullable`, `default`, `unique`) adalah pagar di level basis data: berlaku untuk semua baris yang masuk dari jalur mana pun, bukan hanya lewat satu formulir yang kebetulan memvalidasinya

> Pengecualian sadar: kolom **snapshot** (nilai yang dibekukan pada satu momen, dibahas lengkap setelah contoh skema berikut) sengaja menyimpan salinan nilai, bukan demi "satu tempat", melainkan karena nilai itu harus membeku pada momen tertentu.

Foreign Key: Menjaga Relasi Tetap Konsisten

> **Foreign Key:** kolom pada satu tabel yang menunjuk ke `id` baris pada tabel lain, menjaga agar sebuah baris tidak pernah menunjuk ke baris induk yang tidak ada.

authors (satu) → **articles (banyak)**

- Tanpa foreign key, sebuah `articles` bisa saja menunjuk ke `author_id` yang sudah dihapus, baris "yatim" (orphan) yang bikin aplikasi gagal saat mencoba menampilkan nama penulisnya
- Satu `authors` punya banyak `articles`; setiap baris `articles` menyimpan `author_id` yang menunjuk balik ke penulisnya

Contoh Skema: Relasi Satu ke Banyak

Tabel	Kolom Utama
`authors`	`id`, `name`
`articles`	`id`, `author_id` (fk), `title`, `published_at`
`comments`	`id`, `article_id` (fk), `body`

- Relasi `authors` → `articles` sudah dibahas; tabel `comments` menambah satu tingkat lagi: satu `articles` punya banyak `comments`
- Diagram tabel seperti ini disebut **ERD** (Entity-Relationship Diagram) sederhana, dipakai memvalidasi desain sebelum menulis migration

Kolom Snapshot: Nilai yang Dibekukan

- **Masalahnya**: bayangkan subtotal struk dihitung dari harga produk **saat ini**, bukan disimpan. Kalau besok harga produk naik, subtotal struk minggu lalu ikut berubah, padahal pelanggan sudah bayar sesuai harga saat itu
- **Solusinya**: kolom **snapshot** menghitung nilainya sekali, saat transaksi terjadi, tidak dihitung ulang lagi meski harga produk di katalog berubah kemudian
- Nilai ini disebut **snapshot** karena ia potret kondisi masa lalu, bukan kondisi sekarang

> Menghitung snapshot tidak berarti server percaya begitu saja angka kiriman client. Pertemuan validasi & keamanan nanti membahas kenapa server harus menghitung sendiri nilai itu sebelum menyimpannya.

SQL vs NoSQL: Kapan Skema Tetap Menang

Basis data relasional (SQL)

- Skema tegas: kolom, tipe, dan relasi FK ditentukan di depan
- Transaksi ACID menjaga banyak tabel tetap konsisten sekaligus
- Pilihan default untuk aplikasi bisnis dengan data terstruktur dan saling terhubung, seperti Simple POS

Basis data NoSQL (dokumen/key-value)

- Bersifat schemaless: tiap dokumen boleh punya bentuk berbeda
- Lebih mudah diskalakan secara horizontal ke banyak server
- Unggul saat struktur data cair (berubah-ubah) atau volumenya masif

> Keduanya saling melengkapi, bukan bersaing: relasional untuk transaksi, NoSQL untuk keperluan lain. Simple POS pakai relasional karena butuh transaksi konsisten.

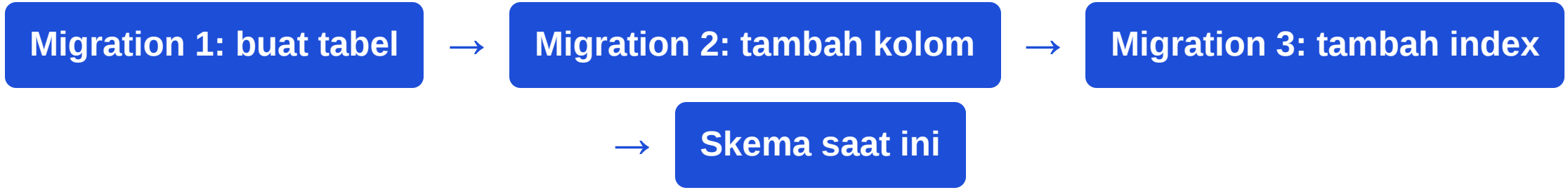
Bagian 2

Riwayat perubahan skema, dan mengisi data awal

Migration: Riwayat Skema yang Bisa Dijalankan Ulang

Tanpa migration, mengubah skema berarti klik manual lewat phpMyAdmin atau DBeaver, perubahan itu cuma ada di database lokalmu, sulit dipastikan sama persis di laptop rekan tim atau di server produksi.

> **Migration:** berkas PHP yang mendeskripsikan perubahan struktur tabel (membuat, mengubah, menghapus kolom) secara terprogram, sehingga skema basis data punya riwayat yang bisa dijalankan ulang di komputer lain.



> Mirip riwayat commit git untuk kode: migration adalah riwayat commit untuk skema basis data, dijalankan berurutan sesuai stempel waktu nama berkasnya.

Membuat Migration Baru

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

- Perintah ini membuat berkas baru berstempel waktu di `database/migrations/`, misalnya `2026_08_20_122440_create_articles_table.php`
- Berkas itu berisi dua metode: `up()` (perubahan yang diterapkan) dan `down()` (cara membatalkannya)
- Kode skema ditulis di dalam `up()`; contoh lengkapnya ada di slide `up()` dan `down()` setelah ini

Menjalankan Migration

```
php artisan migrate
```

- Menjalankan semua migration yang belum pernah dijalankan, berurutan sesuai stempel waktu nama berkasnya
- `php artisan migrate:status` menampilkan migration mana yang sudah dan belum dijalankan, berguna sebelum `migrate` di server baru

> Migration hanya dijalankan sekali. Migration yang sudah tercatat berhasil dijalankan tidak akan dijalankan ulang meski `migrate` dipanggil lagi, kecuali migration itu di-rollback dulu.

`up()` dan `down()`: Maju dan Mundur

```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()` adalah kebalikan tepat dari `up()`: kalau `up()` membuat tabel, `down()` menghapusnya
- `php artisan migrate:rollback` menjalankan `down()` dari batch migration terakhir, berguna saat migration baru ternyata keliru

Kenapa `down()` Layak Ditulis dengan Benar

- `down()` yang jujur, yang benar-benar membalikkan `up()`, membuat eksperimen skema aman untuk dicoba dan dibatalkan tanpa bekas
- Tanpa `down()` yang benar, satu-satunya jalan mundur adalah mengedit skema manual, persis hal yang ingin dihindari migration

> `php artisan migrate:fresh` menjalankan `down()` semua migration lalu `up()` lagi dari nol, jalan pintas yang praktis saat pengembangan. Jangan pernah dijalankan di basis data produksi: seluruh data hilang, bukan cuma skemanya.

Menulis Migration untuk Sebuah Tabel

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

- `foreignId('author_id')` membuat kolom `author_id` bertipe unsigned big integer
- `->constrained()` menambahkan foreign key yang menunjuk ke `authors.id` berdasarkan konvensi penamaan Laravel
- Constraint ini menjaga integritas data, tapi belum tentu berarti kolomnya sudah punya index (lihat Bagian 3)

> Nama tabel jamak snake_case (`articles`) bukan kebetulan: pada pertemuan ORM & Relasi Data berikutnya, Model Eloquent `Article` (tunggal) otomatis terpetakan ke tabel ini lewat konvensi penamaan yang sama, konvensi yang juga dipakai `constrained()` di atas.

Skema Berevolusi Lewat Migration Baru

Kebutuhan bertambah setelah tabel `articles` dipakai: sekarang perlu kolom subjudul. Jangan edit migration lama yang sudah dijalankan, buat migration baru:

```
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()` (bukan `Schema::create()`) mengubah tabel yang sudah ada, di sini menambah kolom

Aturan Emas: Migration Baru, Bukan Edit yang Lama

- Desain basis data jarang sekali jadi di percobaan pertama: skema tumbuh bersama fitur, dan migration adalah mekanisme evolusinya yang tercatat rapi
- Pola ini berulang terus: butuh kolom baru, buat migration baru; butuh hapus kolom, buat migration baru lagi yang menghapusnya

> Migration lama yang diedit setelah dijalankan membuat riwayat skema di komputer lain jadi tidak sinkron dengan riwayat di komputermu, komputer itu tidak tahu ada perubahan karena migration itu sudah tercatat "pernah dijalankan". Kalau skema perlu berubah, migration baru selalu jawabannya, bukan mengedit yang lama.

Seeder dan Factory: Mengisi Data Awal

> **Seeder:** kelas PHP yang mengisi tabel dengan data awal atau data uji, dijalankan lewat `db:seed` atau sebagai bagian dari `migrate:fresh --seed`.

> **Factory:** cetakan untuk menghasilkan satu baris data palsu yang realistis untuk sebuah Model, misalnya `User::factory()->create()` membuat satu baris `users` baru dengan nilai acak yang masuk akal.

- Mengisi data lewat formulir aplikasi satu per satu tidak praktis untuk menguji paginasi, performa, atau tampilan tabel, di sinilah seeder dan factory mengisi ratusan atau ribuan baris otomatis sekali jalan
- Seeder biasanya mengorkestrasi: memanggil factory untuk data yang butuh keunikan per-Model (mis. pengguna), dan bulk insert langsung untuk volume besar (lihat slide berikutnya)

Seeding Skala Besar: Satu-per-Satu vs Bulk Insert

`Model::create()` di dalam loop

- Satu query `INSERT` per baris
- Overhead: membentuk objek Eloquent, memicu event model
- Lambat untuk ratusan atau ribuan baris

`DB::table()->insert()` dalam batch

- Satu pernyataan `INSERT` untuk banyak baris sekaligus
- Jauh lebih sedikit round-trip ke basis data
- `array_chunk()` membagi batch karena sebagian mesin basis data membatasi jumlah baris per `INSERT`

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

Konsistensi Antar Tabel: `DB::transaction()`

> Saat satu operasi menyentuh lebih dari satu tabel yang harus konsisten satu sama lain (mis. menyimpan sebuah pesanan beserta baris-baris itemnya), bungkus dalam satu `DB::transaction(function () { ... })`. Kalau prosesnya gagal di tengah jalan, seluruh perubahan di dalam blok itu dibatalkan bersama, sehingga data tidak pernah berakhir setengah tersimpan.

Bagian 3

Membuktikan dampaknya dengan EXPLAIN QUERY PLAN

Index: Struktur Data untuk Pencarian Cepat

> **Index:** struktur data tambahan yang dibangun mesin basis data di atas satu atau beberapa kolom, mempercepat pencarian baris pada kolom itu tanpa harus memindai seluruh tabel.

Tanpa index (SCAN)

- Mesin basis data memeriksa tiap baris satu per satu
- Waktu bertambah linear seiring tabel bertumbuh

Dengan index (SEARCH)

- Mesin basis data melompat langsung ke baris yang relevan
- Hampir tidak terpengaruh ukuran tabel

Strategi Index: Kolom Mana yang Pantas?

- Kandidat kuat: kolom yang sering muncul di `WHERE`, `JOIN`, atau `ORDER BY`, foreign key hampir selalu masuk kategori ini
- Index bukan gratis: tiap `INSERT`/`UPDATE` pada tabel itu ikut menulis ulang struktur index-nya, dan index memakan ruang penyimpanan tambahan
- Karena itu jangan index semua kolom "siapa tahu berguna": ukur dulu lewat `EXPLAIN QUERY PLAN` (dibahas lengkap setelah slide berikutnya), baru tambahkan index yang terbukti diperlukan
- Kalau dua kolom selalu difilter bersamaan (mis. `category_id` dan `is_active`), composite index pada keduanya sekaligus bisa lebih efektif daripada dua index terpisah

`constrained()` Tidak Selalu Berarti Ada Index

- Di beberapa mesin basis data (mis. MySQL), `foreignId()->constrained()` otomatis membuat index sebagai efek samping
- Di SQLite, `constrained()` hanya menambahkan foreign key constraint, bukan index
- Constraint menjaga integritas data (menolak insert yang menunjuk ke baris tak ada), tapi tidak mempercepat pencarian

Membuktikan dengan `EXPLAIN QUERY PLAN`

- `EXPLAIN QUERY PLAN`: perintah SQLite yang menampilkan strategi mesin basis data untuk menjalankan sebuah query, tanpa benar-benar menjalankannya

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

- **Sebelum index**: hasilnya memuat `SCAN articles`
- **Sesudah index ditambahkan**: hasilnya berubah menjadi `SEARCH articles USING INDEX articles_author_id_index`

> Kalau perintah `sqlite3` tidak dikenali, jalur alternatifnya lewat `php artisan tinker` memakai `DB::select("EXPLAIN QUERY PLAN ...")`.

Index Bukan Optimasi Prematur

> Pada tabel berisi puluhan baris, perbedaan SCAN dan SEARCH nyaris tidak terasa. Pada tabel berisi ribuan baris, SCAN berarti setiap baris ikut diperiksa pada setiap query, dan waktu itu bertambah linear seiring tabel bertumbuh. Index bukan optimasi prematur; ia perbaikan atas bug performa yang sudah nyata begitu data mendekati skala produksi.

Schema Builder Lintas Ekosistem

Ekosistem	Cara Mendefinisikan Skema	Migration
Laravel (Eloquent)	Schema builder PHP imperatif (`Schema::create()`)	Berkas migration ditulis manual, dijalankan lewat `artisan migrate`
Prisma (Node.js)	Skema deklaratif di satu berkas `schema.prisma`	Migration di-generate otomatis dari selisih skema
SQLAlchemy + Alembic (Python)	Model Python (class per tabel)	Migration semi-otomatis lewat Alembic, membandingkan model dengan skema saat ini

> Sintaksnya berbeda-beda, tapi konsepnya sama: skema basis data adalah kode yang punya riwayat dan bisa dijalankan ulang di komputer lain. Begitu terbiasa dengan migration Laravel, berpindah ke ekosistem lain tinggal soal sintaks.

Menerapkan pada Simple POS

- Konsep skema, migration, dan index ini akan kamu terapkan langsung pada studi kasus Simple POS di jobsheet praktikum
- Merancang tabel kategori-produk-transaksi, menulis seeder berskala ratusan-ribuan baris, dan membuktikan index lewat `EXPLAIN QUERY PLAN`

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

Rangkuman (1/2)

- Skema yang baik menyimpan tiap fakta satu kali dengan tipe data dan batasan yang tepat; basis data relasional cocok untuk data transaksional seperti Simple POS, NoSQL untuk struktur data yang schemaless atau skala masif
- Migration adalah riwayat skema yang bisa dijalankan ulang: `up()` menerapkan perubahan, `down()` membalikkannya, dan skema berevolusi lewat migration baru, bukan mengedit yang lama

Rangkuman (2/2)

- Seeder mengisi data awal, factory mencetak data palsu per-Model; seeding skala besar memakai `DB::table()->insert()` dalam batch, jauh lebih efisien daripada `Model::create()` satu per satu
- Index mempercepat pencarian (`SCAN` ke `SEARCH`, dibuktikan lewat `EXPLAIN QUERY PLAN`), tapi bukan gratis: pilih kandidat dari kolom `WHERE` / `JOIN` / `ORDER BY`, jangan index semua kolom

Referensi & Diskusi

Dokumentasi resmi Laravel (Migrations, Query Builder, Seeding)

Kode lengkap: `github.com/se-polinema/simple-pos`

Pertemuan berikutnya: ORM & Relasi Data