Eloquent Deep Dive — Data Ko Gehrai Se Nikalna

Asking sharper questions: where, orderBy, first and findOrFail, sums and averages, groupBy, chunk for big tables, and reusable query scopes.

Introduction

`Student::all()` was the first sentence you learned. Real apps rarely want *all* — they want the toppers, the students of one city, the average of the class, the third page of a thousand rows. Eloquent's query builder is the full language for those questions: chainable methods that read like the question itself — `where`, `orderBy`, `limit` — and answer with exactly the slice you asked for.

This lesson goes deep on purpose: filtering and ordering, the difference between `first()` and `find()`, aggregates that let the DATABASE do the maths (sum, average, group totals), `chunk()` for tables too big to swallow at once, and **scopes** — questions you ask so often that you give them a name and keep them on the model.

(Roman Urdu: Ab tak table se sab kuch uthaya; ab table se sawal karna seekho — jawab utna hi mile ga jitna manga.)

Key points

- **Chains read like questions:** `Student::where('marks', '>=', 90)->orderBy('marks', 'desc')->get()` — filter, sort, fetch. `get()` ends a chain with a collection; the order of the middle links is yours.
- **`first()` vs `find()`:** `find(4)` hunts by id. `first()` takes the first row matching your chain — "the topper" is `orderBy('marks', 'desc')->first()`. Both return ONE model — and `firstOrFail()` / `findOrFail()` swap a dangerous null for an honest 404.
- **`limit()` and `offset()` page the firehose:** never pour 50,000 rows onto one screen. `limit(10)` (and Laravel's `paginate()` in real apps) keeps pages human-sized.
- **Aggregates let the database do maths:** `Student::count()`, `->sum('marks')`, `->avg('marks')`, `->max('marks')`, `->min('marks')` — one tiny answer returns, instead of hauling every row home to count it yourself. Class check: our five students sum to 415, average exactly 83.
  (Roman Urdu: Gintee database se karwao — saari rows ghar la kar khud ginna sab se mehnga tareeqa hai.)
- **`groupBy` answers "per group" questions:** one query returns totals per city, per class, per month — the shape of every report and dashboard chart.
- **`chunk()` eats big tables politely:** process 1,000 students in batches of 100 — memory stays flat, the server stays calm. Reports, emails and exports are chunk-shaped jobs (queues return at LV.16).
- **Scopes are named questions:** `scopeHighScorers()` on the model lets every future query say `Student::highScorers()` instead of repeating `where('marks', '>=', 90)` in twelve files — and when the definition of "high" changes, it changes in one place.

Code examples

Example 1 — Filter, sort, take one: the merit questions

```
$toppers = Student::where('marks', '>=', 90)
    ->orderBy('marks', 'desc')
    ->get();

foreach ($toppers as $student) {
    echo $student->name . ' — ' . $student->marks . '<br>';
}

$topper = Student::orderBy('marks', 'desc')->first();
echo 'Class topper: ' . $topper->name;
```

What it does: The chain filters to marks 90 and above, sorts highest first, and fetches. `first()` then answers a different shape of question — not a list, a single champion.

What you will see:

```
Hira Shah — 95
Sara Khan — 92
Class topper: Hira Shah
```

Two rows survived the filter, in merit order, and the single-model answer names the topper.

Example 2 — Aggregates + groupBy: the class report in two queries

```
echo 'Students: ' . Student::count();
echo 'Total marks: ' . Student::sum('marks');
echo 'Average: ' . Student::avg('marks');
echo 'Highest: ' . Student::max('marks');
echo 'Lowest: ' . Student::min('marks');

$byCity = Student::query()
    ->selectRaw('city, COUNT(*) as total, AVG(marks) as avg_marks')
    ->groupBy('city')
    ->get();
```

What it does: Five one-word questions, five one-value answers — the database counted; your PHP received only results. Then `groupBy` folds the class by city: one row per city with its student count and its average.

What you will see: the class vitals — `Students: 5 · Total marks: 415 · Average: 83 · Highest: 95 · Lowest: 64` — and the city table:

```
city    | total | avg_marks
Sahiwal | 3     | 76
Lahore  | 1     | 92
Karachi | 1     | 95
```

Sahiwal's 76 is honest maths: (88 + 76 + 64) / 3. Check every number by hand — they all come from the same five students.

Example 3 — chunk() for big tables + a named scope

```
// app/Models/Student.php — the named question lives on the model
public function scopeHighScorers($query)
{
    return $query->where('marks', '>=', 90);
}
```

```
// anywhere: the chain reads like English
$stars = Student::highScorers()->orderBy('marks', 'desc')->get();

// and the big-table pattern: report cards in polite batches
Student::orderBy('id')->chunk(2, function ($students) {
    echo 'Preparing cards for: ';
    foreach ($students as $student) {
        echo $student->name . ' ';
    }
    echo '<br>';
});
```

What it does: The scope moves the "90 or above" rule onto the model — one definition, reused everywhere, chained like a built-in method. `chunk(2, ...)` walks the whole table two students at a time: same total work, a fraction of the memory.

What you will see: the scope returns Sara and Hira; the chunker prints three polite batches:

```
Preparing cards for: Ali Raza Sara Khan
Preparing cards for: Bilal Ahmed Hira Shah
Preparing cards for: Usman Tariq
```

Quick recap

- Chains are questions: `where` → `orderBy` → `get()`; `first()` / `find()` fetch one, the `OrFail` twins refuse nulls.
- Aggregates (count/sum/avg/max/min) and `groupBy` let the database do the maths and grouping.

- `chunk()` processes big tables in memory-safe batches; scopes turn repeated questions into named model methods.

Common mistakes

1. **`all()` then filtering in PHP** — hauling 10,000 rows home to keep 3 is the most common Eloquent waste. Fix: the filter belongs in the chain (`where`) so the database returns only the survivors.
2. **Trusting `first()` without an order** — "first" without `orderBy` means "whichever the database felt like" — often id order, never guaranteed. Fix: any `first()` that means "top" or "latest" must say `orderBy` out loud.
3. **Averaging by hand after `get()`** — a full collection fetched just to compute one number. Fix: `avg()`, `sum()`, `count()` — one value travels, not the whole table.
4. **Copy-pasting the same where-chain everywhere** — twelve copies of the "high scorer" rule drift apart silently. Fix: the third time you type a chain, it becomes a scope on the model.

Practice

1. Write the "Sahiwal merit list": students of Sahiwal only, orderBy marks desc — expect Ali 88, Bilal 76, Usman 64. Then add `limit(2)` and name who disappears and why.
2. Add a second scope, `scopeFromCity($query, $city)`, that takes a parameter. Use it to print Karachi's average (95) — then explain in one line how a parameterised scope differs from the fixed highScorers rule.

Practice quiz

1. What does Student::where('marks', '>=', 90)->orderBy('marks', 'desc')->get() return?

- One model
- A collection of the matching students, filtered then sorted; get() ends a chain with a collection
- A single number
- An error, chains must end with first()

Answer: A collection of the matching students, filtered then sorted; get() ends a chain with a collection

2. What is the difference between first() and find()?

- There is none
- find(4) hunts by id; first() takes the first row matching your chain, so the topper is orderBy('marks', 'desc')->first()
- first() hunts by id, find() takes the first match
- first() returns many, find() returns one

Answer: find(4) hunts by id; first() takes the first row matching your chain, so the topper is orderBy('marks', 'desc')->first()

3. Why use Student::avg('marks') instead of fetching all rows and averaging in PHP?

- There is no reason, both are equal
- Aggregates let the database do the maths: one tiny answer returns, instead of hauling every row home to count it yourself
- avg() is only for integers
- PHP cannot average

Answer: Aggregates let the database do the maths: one tiny answer returns, instead of hauling every row home to count it yourself

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/laravel-eloquent-deep-dive · More free tools and notes at https://scopetreasure.com