Query Performance Tuning in Laravel + MySQL

Query Performance Tuning in Laravel + MySQL

Even well-built Laravel apps can slow down when queries become inefficient. Performance tuning involves analyzing how queries run, indexing properly, caching results, and monitoring live traffic. In this guide, you'll learn practical techniques to tune MySQL queries in Laravel, from `EXPLAIN` plans to eager loading, indexes, and caching. We'll also add a simple UI profiler to visualize queries during development.

1 – Find Slow Queries with Laravel Debug Tools

Enable query logging in Laravel to see what queries run and how long they take. In dev, you can log all queries easily:

```
// app/Providers/AppServiceProvider.php (boot)
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Log;

public function boot(): void
{
    DB::listen(function ($query) {
        Log::info(
            $query->sql,
            ['bindings' => $query->bindings, 'time_ms' =>
$query->time]
        );
    });
}Code language: PHP (php)
```

This hooks into Laravel's DB layer and logs every query's SQL, bindings, and runtime to

`storage/logs/laravel.log`. Use this in development or staging only—it can be noisy in production.

2 – Analyze Queries with EXPLAIN

Use MySQL's `EXPLAIN` to see how a query executes. This reveals whether indexes are used and where full scans happen.

```
EXPLAIN SELECT * FROM orders WHERE status = 'paid' ORDER BY created_at DESC;Code language: SQL (Structured Query Language) (sql)
```

If the result shows "Using where; Using index" you're good. If it says "Using filesort" or "Using temporary", you may need to add or adjust indexes.

3 – Add Proper Indexes in Migrations

Indexes speed up WHERE, ORDER BY, and JOIN clauses. Add them in migrations for frequently queried columns.

```
// database/migrations/xxxx_add_indexes_to_orders.php
Schema::table('orders', function (Blueprint $table) {
    $table->index('status');
    $table->index(['user_id','created_at']); // compound index
});Code language: PHP (php)
```

`status` gets a single index for filters like `WHERE status = 'paid'`. A compound index on `(user_id, created_at)` speeds up queries filtering by user and sorting by creation date.

Don’t over-index—each index costs extra storage and slower writes.

4 – Reduce N+1 Queries with Eager Loading

N+1 queries often cause slowdowns. Fix them with `with()` or `load()` to prefetch related data in fewer queries.

```
// Slow (lazy loading)
$users = User::all();
foreach ($users as $user) {
    echo $user->posts->count(); // query per user
}

// Fast (eager loading)
$users = User::withCount('posts')->get();
foreach ($users as $user) {
    echo $user->posts_count; // no extra queries
}Code language: PHP (php)
```

`withCount()` lets you grab relation counts directly with one query, instead of running a query for every loop iteration. This can save hundreds of queries on list pages.

5 – Optimize Pagination Queries

Pagination with `OFFSET` gets slower on large tables because MySQL still scans skipped rows. Use `where('id','>',...)` or `chunkById()` for “keyset pagination.”

```
// Classic pagination (slow with big OFFSET)
$posts = Post::orderBy('id')->offset(50000)->limit(20)->get();

// Keyset pagination (fast)
$posts = Post::where('id','>', $lastSeenId)
             ->orderBy('id')
             ->limit(20)
             ->get();Code language: PHP (php)
```

Keyset pagination avoids scanning all skipped rows, making it dramatically faster on large datasets. Store the last seen ID in your pagination links.

6 – Cache Heavy Queries

When a query result doesn't change often, cache it in Redis or the file cache. Use `remember()` to wrap queries.

```
use Illuminate\Support\Facades\Cache;

$stats = Cache::remember('dashboard_stats', 600, function () {
    return Order::selectRaw('status, COUNT(*) as total')
        ->groupBy('status')
        ->pluck('total','status');
});Code language: PHP (php)
```

This caches the grouped order counts for 10 minutes (`600` seconds). Subsequent requests return from cache instantly instead of rerunning the query.

7 – UI Example: Query Profiler Panel

Let’s add a simple UI snippet to show executed queries on a page during development. This is a lightweight alternative to installing Telescope or Debugbar.

```
<!-- resources/views/layouts/partials/query-profiler.blade.php -->
@php
  $queries = DB::getQueryLog();
@endphp

@if(app()->environment('local'))
  <div class="container mt-5">
    <h5>Executed Queries ({{ count($queries) }})</h5>
    <ul class="list-group">
      @foreach($queries as $q)
        <li class="list-group-item">
          {{ $q['query'] }}
          [{{ implode(',', $q['bindings']) }}]
          ({{ $q['time'] }} ms)
        </li>
      @endforeach
    </ul>
  </div>
@endifCode language: PHP (php)
```

This partial prints all queries on the page with execution times. To enable logging, call `DB::enableQueryLog()` in a service provider for your local environment.

Wrapping Up

Performance tuning in Laravel is a mix of good schema design, query inspection, caching, and avoiding N+1 traps. You saw how to log and analyze queries, add indexes, use eager

loading, optimize pagination, and cache results. With these tools, your Laravel + MySQL app can handle far more traffic smoothly.

What’s Next

- 10 Proven Ways to Optimize Laravel for High Traffic
- Handling Large Data Sets with Chunking & Cursors
- Laravel and Docker: Setting Up a Scalable Dev Environment