Handling Large Data Sets in Laravel with Chunking & Cursors

Handling Large Data Sets in Laravel with Chunking & Cursors

Loading tens or hundreds of thousands of rows into memory will crash your app or time out your requests. Laravel provides `chunk()`, `chunkById()`, `cursor()` / `lazy()`, and their `*ById()` variants to process large data sets in small, memory-friendly pieces. In this guide you'll learn when to use each, how to avoid classic pitfalls (like missing/duplicated rows when ordering), and how to wire them into real features like CSV exports and background jobs.

1 – When to Use chunk(), chunkById(), cursor(), lazy()

Rule of thumb: `chunk()` pages using LIMIT/OFFSET (simple but can skip/duplicate if rows are inserted/deleted mid-scan). `chunkById()` uses the primary key for stable windows (best for large mutable tables). `cursor()`/`lazy()` stream one row at a time using a generator (lowest memory, slowest per-row). Use the `*ById()` variants when ordering by the primary key for safety.

2 – Using chunk() for Medium Data (Static Windows)

Process records in fixed pages. Works well for reporting tables that don't mutate during the scan.

```
use App\Models\Order;

// Process 2,000 records at a time
Order::where('status', 'paid')
    ->orderBy('id') // always order deterministically
    ->chunk(2000, function ($orders) {
        foreach ($orders as $order) {
            // do work (aggregate, export, etc.)
        }
    });Code language: PHP (php)
```

`chunk(2000, ...)` runs your callback for each page. Always add a deterministic `orderBy` (typically the PK). Avoid using this on hot tables that change rapidly; OFFSET can drift if rows are inserted/deleted between pages.

3 – Using chunkById() for Large, Mutable Tables

Prefer `chunkById()` on big tables receiving writes while you scan. It uses the last seen ID instead of OFFSET, avoiding gaps/duplication.

```
use App\Models\User;

// Process active users safely even if the table changes mid-scan
User::where('is_active', true)
    ->chunkById(3000, function ($users) {
        foreach ($users as $user) {
            // email sync, billing, etc.
```

```
        }
    }, $column = 'id');Code language: PHP (php)
```

`chunkById(3000,...)` remembers the highest `id` from the last page and continues from there, so inserts/deletes won't shift previously scanned windows. If your PK is not monotonic (e.g., UUIDs), use a sortable indexed column like an auto-increment surrogate.

4 – Using cursor() / lazy() for Streaming (Ultra Low Memory)

`cursor()` (alias: `lazy()`) yields models one-by-one using generators—minimal memory, at the cost of longer total runtime and sustained DB connection.

```
use App\Models\LogEntry;

foreach (LogEntry::where('level','error')->orderBy('id')->cursor() as $row) {
    // Stream process each $row; memory footprint stays tiny
}Code language: PHP (php)
```

`cursor()` is ideal for exports or ETL tasks. Because it keeps a DB cursor open, run it in CLI/queue workers, not behind slow web requests. Combine with `set_time_limit` or queue timeouts appropriately.

5 – Memory & N+1: Eager Load Carefully

When chunking or streaming, eager load only what you need. Avoid loading heavy relations per item; consider precomputing or joining.

```
// Keep the payload small when scanning
Order::with(['user:id,name', 'items:id,order_id,price'])
    ->whereYear('created_at', now()->year)
    ->chunkById(2000, function ($orders) {
        foreach ($orders as $order) {
            // $order->relation calls won't trigger extra queries
        }
    });Code language: PHP (php)
```

Eager load only essential columns (select lists on relations) to control memory and query volume. Heavy relations can still cause memory spikes if the chunk size is too large—tune chunk size down if needed (e.g., 500–1000).

6 – UI Feature: CSV Export with Streaming

Let's build a safe CSV export that streams rows instead of loading them all. We'll use `cursor()` inside a streamed response so memory stays flat.

```
// app/Http/Controllers/ExportController.php
namespace App\Http\Controllers;

use App\Models\Order;
use Symfony\Component\HttpFoundation\StreamedResponse;

class ExportController extends Controller
{
    public function ordersCsv(): StreamedResponse
```

```
    {
        $response = new StreamedResponse(function () {
            $handle = fopen('php://output', 'w');
            fputcsv($handle, ['ID','User','Total','Created At']);

            foreach (
                Order::with('user:id,name')
                    ->orderBy('id')
                    ->cursor() as $o
            ) {
                fputcsv($handle, [
                    $o->id,
                    optional($o->user)->name,
                    number_format($o->total / 100, 2),
                    $o->created_at,
                ]);
            }

            fclose($handle);
        });

        $response->headers->set('Content-Type', 'text/csv');
        $response->headers->set('Content-Disposition', 'attachment;
filename="orders.csv"');

        return $response;
    }
}Code language: PHP (php)
```

This streams CSV lines as they are read from the database with `cursor()`. The browser starts downloading immediately and memory usage remains near-constant regardless of table size.

```
// routes/web.php (snippet)
use App\Http\Controllers\ExportController;

Route::get('/exports/orders.csv', [ExportController::class,
'ordersCsv'])
```

```
    ->middleware(['auth']);Code language: PHP (php)
```

Protect exports behind auth (and permissions if needed). For very long streams, prefer running the query in a queue and emailing a signed URL to the file once ready.

7 – Background Jobs: Chunk Work into Batches

Break the work into manageable pieces and dispatch jobs per chunk to parallelize safely without memory bloat.

```
// app/Console/Commands/DispatchUserEmails.php
namespace App\Console\Commands;

use App\Jobs\SendMonthlyEmail;
use App\Models\User;
use Illuminate\Console\Command;

class DispatchUserEmails extends Command
{
    protected $signature = 'users:email-monthly';
    protected $description = 'Dispatch monthly emails in chunks';

    public function handle(): int
    {
        User::where('is_active', true)
            ->select('id','email','name')
            ->chunkById(500, function ($users) {
                dispatch(new
SendMonthlyEmail($users->pluck('id')->all()));
            });

        $this->info('Dispatched email jobs.');
        return self::SUCCESS;
```

```
    }
}Code language: PHP (php)
```

This command slices the user base into 500-row chunks and dispatches one job per slice. Each job can query those IDs again in isolation to send emails, avoiding giant payloads on the queue bus.

```
// app/Jobs/SendMonthlyEmail.php
namespace App\Jobs;

use App\Models\User;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendMonthlyEmail implements ShouldQueue
{
    use Queueable;

    public function __construct(public array $userIds) {}

    public function handle(): void
    {
        User::whereIn('id', $this->userIds)
            ->select('id','email','name')
            ->chunkById(100, function ($users) {
                foreach ($users as $u) {
                    // Mail::to($u->email)->queue(new
MonthlyDigest($u));
                }
            });
    }
}Code language: PHP (php)
```

Jobs receive lightweight ID lists, then process sub-chunks (100) within the job to keep memory flat and respect queue time limits. Use retry/backoff for transient failures.

8 – Avoiding Pitfalls (Ordering, Mutations, Transactions)

Large scans can behave badly if ordering is unstable or if you wrap everything in one giant transaction.

```
// Bad: OFFSET can skip/duplicate if rows change
// Good: use chunkById() with a stable increasing key
Invoice::where('status','unpaid')
    ->chunkById(2000, function ($invoices) {
        foreach ($invoices as $inv) {
            // safely process
        }
    });

// Avoid one huge transaction around a whole scan
// Instead, commit per item or per small group
DB::transaction(function () use ($orders) {
    foreach ($orders as $order) {
        // small, fast writes here
    }
});Code language: PHP (php)
```

Use `chunkById()` for mutable tables. Keep transactions small to prevent lock contention and long-running locks. If you must guarantee a consistent snapshot, consider DB-level snapshot isolation or doing the scan from a read replica.

Wrapping Up

You learned how to pick the right tool for big reads: `chunk()` for simple paging on static data, `chunkById()` for safety on hot tables, and `cursor()`/`lazy()` for true streaming. You also built a memory-safe CSV export and batched background jobs. With careful ordering, eager loading, and transaction strategy, you can process millions of rows reliably.

What's Next

- How to Use Eloquent API Resources for Clean APIs
- How to Use Eloquent Events for Auditing User Actions
- Eager Loading vs Lazy Loading in Laravel: Best Practices