Optimizing Laravel Database Queries with Redis Caching

Redis is an excellent solution for caching frequently accessed data in your Laravel application. Here's a comprehensive guide to implementing Redis caching for database optimization.

1. Setup and Configuration

Install Required Packages

```
composer require predis/predis
```

Configure Environment Variables (.env)

```
CACHE_DRIVER=redis
REDIS_HOST=redis
REDIS_PASSWORD=null
REDIS_PORT=6379
```

Update Docker Compose

Add Redis service to your `docker-compose.yml`:

```
services:
  redis:
    image: redis:alpine
    container_name: laravel_redis
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    networks:
      - laravel_network

volumes:
  redis_data:
    driver: local
```

PROF

2. Basic Caching Implementation

Cache Database Results

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

// Get users with caching
$users = Cache::remember('users.all', now()->addHours(24), function () {
    return User::with('posts')->get();
});

// Get single user with caching
$user = Cache::remember('user.'.$id, now()->addHours(12), function ()
use ($id) {
    return User::with('posts')->findOrFail($id);
});
```

Cache GraphQL Resolvers

```php
// In your GraphQL resolver
public function resolve($root, array $args, $context, $resolveInfo)
{
    return Cache::remember('custom_query.'.md5(json_encode($args)),
now()->addHour(), function () use ($args) {
        return User::where('name', 'LIKE', "%{$args['search']}%")
            ->with('posts')
            ->get();
    });
}
```

3. Advanced Caching Strategies

Model Caching with Observers

Create an observer:

PROF

```bash
php artisan make:observer UserObserver --model=User
```

Implement caching logic:

```php
namespace App\Observers;

use App\Models\User;
use Illuminate\Support\Facades\Cache;

class UserObserver
{
    public function saved(User $user)
    {
```

```
        Cache::forget('users.all');
        Cache::forget('user.'.$user->id);
    }

    public function deleted(User $user)
    {
        Cache::forget('users.all');
        Cache::forget('user.'.$user->id);
    }
}
```

Register the observer in `AppServiceProvider`:

```
public function boot()
{
    User::observe(UserObserver::class);
}
```

Cache Tags for Complex Relationships

```
// Cache users with their posts using tags
$users = Cache::tags(['users', 'posts'])->remember('users.with_posts',
now()->addDay(), function () {
    return User::with('posts')->get();
});

// Clear all cache related to users
Cache::tags('users')->flush();
```

PROF

Database Query Cache Middleware

Create middleware:

```
php artisan make:middleware CacheResponses
```

Implement caching:

```
public function handle($request, Closure $next, $ttl = 60)
{
    $key = 'route.'.md5($request->url().serialize($request->all()));

    return Cache::remember($key, now()->addMinutes($ttl), function ()
use ($request, $next) {
        return $next($request);
```

```
    });
}
```

Register middleware in `Kernel.php`:

```
protected $routeMiddleware = [
    'cache.response' => \App\Http\Middleware\CacheResponses::class,
];
```

Use in routes:

```
Route::get('/users', 'UserController@index')-
>middleware('cache.response:1440'); // Cache for 24h
```

4. Real-Time Cache Invalidation

Event-Based Cache Clearing

```
// In your EventServiceProvider
protected $listen = [
    'App\Events\UserUpdated' => [
        'App\Listeners\ClearUserCache',
    ],
];

// Create listener
php artisan make:listener ClearUserCache
```

PROF

```
public function handle(UserUpdated $event)
{
    Cache::forget('user.'.$event->user->id);
    Cache::tags(['users'])->flush();
}
```

Automatic Cache Invalidation with Model Events

```
// In your model
protected static function boot()
{
    parent::boot();
```

```
    static::updated(function ($model) {
        Cache::tags([$model->getTable()])->flush();
    });

    static::deleted(function ($model) {
        Cache::tags([$model->getTable()])->flush();
    });
}
```

5. Performance Monitoring

Cache Hit/Miss Tracking

```
// Wrap your cache calls to track performance
$start = microtime(true);
$result = Cache::remember('key', $ttl, function () {
    // expensive operation
});
$elapsed = microtime(true) - $start;

Log::debug('Cache operation', [
    'key' => 'key',
    'time' => $elapsed,
    'hit' => Cache::has('key') // Before the call would be more accurate
]);
```

Redis-Specific Optimizations

```
// Pipeline multiple cache operations
Cache::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i, now()->addHour());
    }
});

// Use Redis hashes for related data
Cache::hmset('user:1:profile', [
    'name' => 'John',
    'email' => 'john@example.com'
]);
```

PROF

6. Testing Cache Implementation

Unit Test for Cached Queries

```php
public function testUserIndexUsesCache()
{
    Cache::shouldReceive('remember')
        ->once()
        ->with('users.all', \Mockery::type(\DateTimeInterface::class),
\Mockery::type('Closure'))
        ->andReturn(collect([factory(User::class)->make()]));

    $response = $this->get('/users');
    $response->assertStatus(200);
}

public function testCacheInvalidationOnUserUpdate()
{
    $user = User::factory()->create();

    Cache::shouldReceive('forget')
        ->once()
        ->with('user.'.$user->id);

    Cache::shouldReceive('forget')
        ->once()
        ->with('users.all');

    $user->update(['name' => 'New Name']);
}
```

7. Production Considerations

Cache Stampede Protection

```php
// Using atomic locks to prevent cache stampede
$value = Cache::lock('expensive_operation')->block(5, function () {
    return Cache::remember('expensive_data', now()->addHour(), function
() {
        // Expensive operation
    });
});
```

Multi-Layered Caching

```php
// Check in-memory cache first, then Redis, then database
$users = $this->memoryCache->get('users.all', function () {
    return Cache::remember('users.all', now()->addHour(), function () {
        return User::all();
    });
});
```

Monitoring with Horizon

Configure Horizon to monitor Redis:

```
composer require laravel/horizon
php artisan horizon:install
```

Example Implementation for GraphQL

```
// Optimized GraphQL resolver with caching
public function resolve($root, array $args, $context, $resolveInfo)
{
    $cacheKey = 'graphql.'.md5($resolveInfo->fieldName.json_encode($args));

    return Cache::remember($cacheKey, now()->addHour(), function () use ($args) {
        return User::query()
            ->when(isset($args['search']), function ($query) use ($args) {
                return $query->where('name', 'like', "%{$args['search']}%");
            })
            ->with(['posts' => function ($query) {
                $query->select('id', 'title', 'user_id');
            }])
            ->get();
    });
}
```

PROF

Cache Configuration Tips

1. **Tiered Caching**: Use different TTLs for different data types
    - User profiles: 24 hours
    - Product listings: 1 hour
    - Shopping cart: 5 minutes
2. **Size Management**: Monitor Redis memory usage

```
redis-cli info memory
```

3. **Compression**: For large datasets, consider compression

```
Cache::put('large_data', gzcompress($data), $seconds);
```

By implementing these Redis caching strategies, you'll significantly reduce database load while improving response times for frequently accessed data. Remember to balance cache freshness with performance benefits based on your application's specific requirements.