DATABASE: QUERY BUILDER

- Introduction
- Running Database Queries
  - Chunking Results
  - Streaming Results Lazily
  - Aggregates
- Select Statements
- Raw Expressions
- Joins
- Unions
- Basic Where Clauses
  - Where Clauses
  - Or Where Clauses
  - JSON Where Clauses
  - Additional Where Clauses
  - Logical Grouping
- Advanced Where Clauses
  - Where Exists Clauses
  - Subquery Where Clauses
- Ordering, Grouping, Limit & Offset
  - Ordering
  - Grouping
  - Limit & Offset
- Conditional Clauses
- Insert Statements
  - Upserts
- Update Statements
  - Updating JSON Columns
  - Increment & Decrement
- Delete Statements
- Pessimistic Locking
- Debugging

1.1. Introduction

Trình tạo truy vấn (Query Builder) cơ sở dữ liệu của Laravel cung cấp một giao diện thuận tiện để tạo và chạy các truy vấn cơ sở dữ liệu. Nó có thể được sử dụng để thực hiện hầu hết các hoạt động cơ sở dữ liệu trong ứng dụng và hoạt động hoàn hảo với tất cả các hệ thống cơ sở dữ liệu được hỗ trợ của Laravel.

Trình tạo truy vấn Laravel sử dụng PDO parameter binding để bảo vệ ứng dụng chống lại các cuộc tấn công SQL injection.

PDO không hỗ trợ binding column names. Do đó, không được phép cho phép người dùng nhập tên cột mà truy vấn tham chiếu, bao gồm cả các cột " order by ".

1.2. Running Database Queries

1.2.1. Retrieving All Rows From A Table

Có thể sử dụng phương thức `table` được cung cấp bởi `DB` facade để bắt đầu truy vấn. Phương thức `table` trả về một fluent query builder instance cho bảng đã cho, cho phép xâu chuỗi nhiều ràng buộc hơn vào truy vấn và sau đó cuối cùng truy xuất kết quả của truy vấn bằng phương thức `get`:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
```

```
    }
}
```

Phương thức `get` trả về một instance `Illuminate\Support\Collection` chứa các kết quả của truy vấn trong đó mỗi kết quả là một thể hiện của đối tượng PHP `stdClass`. Có thể truy cập giá trị của từng cột bằng cách truy cập column như một thuộc tính của đối tượng:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->get();

foreach ($users as $user) {
    echo $user->name;
}
```

Laravel collections cung cấp nhiều phương thức cực kỳ mạnh mẽ để ánh xạ và thu nhỏ dữ liệu. Để biết thêm thông tin về Laravel collections, hãy xem tài liệu collection documentation.

Retrieving A Single Row / Column From A Table

Nếu chỉ cần truy xuất một dòng đơn lẻ từ bảng cơ sở dữ liệu, có thể sử dụng phương thức `first` của `DB` facade. Phương thức này sẽ trả về một đối tượng `stdClass`:

```
$user = DB::table('users')->where('name', 'John')->first();

return $user->email;
```

Nếu không cần toàn bộ một dòng, có thể trích xuất một giá trị từ một bản ghi bằng phương thức `value`. Phương thức này sẽ trả về giá trị của cột trực tiếp:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

Để truy xuất một dòng theo giá trị cột `id`, hãy sử dụng phương thức `find`:

```
$user = DB::table('users')->find(3);
```

Retrieving A List Of Column Values

Nếu muốn truy xuất một instance của `Illuminate\Support\Collection` chứa các giá trị của một cột, có

thể sử dụng phương pháp `pluck`.

Ví dụ, truy xuất một collection các `titles` của `user`:

```
use Illuminate\Support\Facades\DB;

$titles = DB::table('users')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

Có thể chỉ định cột mà tập hợp kết quả sẽ sử dụng làm khóa của nó bằng cách cung cấp đối số thứ hai cho phương thức `pluck`:

```
$titles = DB::table('users')->pluck('title', 'name');

foreach ($titles as $name => $title) {
    echo $title;
}
```

1.2.2. Chunking Results

Nếu cần làm việc với số lượng lớn bản ghi cơ sở dữ liệu (database records), hãy sử dụng phương thức `chunk` được cung cấp bởi `DB` facade. Phương thức này truy xuất một phần nhỏ kết quả tại một thời điểm và đưa mỗi phần vào một phần đóng để xử lý.

Ví dụ, hãy truy xuất toàn bộ bảng `users` trong chunks của 100 records cùng lúc:

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    foreach ($users as $user) {
        //
    }
});
```

Có thể dừng xử lý các phần tiếp theo bằng cách trả về `false`:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    // Process the records...

    return false;
```

Nếu đang cập nhật bản ghi cơ sở dữ liệu trong khi chunking, kết quả chunk có thể thay đổi theo những cách không mong muốn. Nếu dự định cập nhật các bản ghi đã truy xuất trong khi chunking, tốt nhất nên sử dụng phương thức `chunkById` để thay thế. Phương thức này sẽ tự động phân trang các kết quả dựa trên khóa chính của bản ghi:

```
DB::table('users')->where('active', false)
    ->chunkById(100, function ($users) {
        foreach ($users as $user) {
            DB::table('users')
                ->where('id', $user->id)
                ->update(['active' => true]);
        }
    });
```

Khi cập nhật hoặc xóa các bản ghi bên trong lệnh chunk callback, bất kỳ thay đổi nào đối với khóa chính hoặc khóa ngoại có thể ảnh hưởng đến chunk query. Điều này có thể dẫn đến việc các bản ghi không được đưa vào các kết quả chunked.

1.2.3. Streaming Results Lazily

Phương thức `lazy` hoạt động tương tự như the chunk method theo nghĩa là nó thực thi truy vấn theo từng phần. Tuy nhiên, thay vì chuyển từng chunk vào callback, phương thức `lazy()` trả về một `LazyCollection`, cho phép tương tác với các kết quả dưới dạng một single stream:

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->lazy()->each(function ($user) {
    //
});
```

Một lần nữa, nếu định cập nhật các bản ghi đã truy xuất trong khi lặp lại chúng, tốt nhất là nên sử dụng phương thức `lazyById` để thay thế. Phương thức này sẽ tự động phân trang các kết quả dựa trên khóa

chính của bản ghi:

```
DB::table('users')->where('active', false)
    ->lazyById()->each(function ($user) {
        DB::table('users')
            ->where('id', $user->id)
            ->update(['active' => true]);
    });
```

Khi cập nhật hoặc xóa bản ghi trong khi lặp lại chúng, bất kỳ thay đổi nào đối với khóa chính hoặc khóa ngoại đều có thể ảnh hưởng đến chunk query. Điều này có thể dẫn đến việc các bản ghi không được đưa vào kết quả.

1.2.4. Aggregates

Trình tạo truy vấn (Query Builder) cũng cung cấp nhiều phương thức để truy xuất các giá trị tổng hợp như `count`, `max`, `min`, `avg`, và `sum`.

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Có thể kết hợp các phương thức này với các mệnh đề khác để tinh chỉnh cách tính giá trị tổng hợp:

```
$price = DB::table('orders')
                ->where('finalized', 1)
                ->avg('price');
```

Determining If Records Exist

Thay vì sử dụng phương thức `count` để xác định xem có bản ghi nào phù hợp với các ràng buộc của truy vấn hay không, có thể sử dụng phương thức `exists` và phương thức `doesntExist`:

```
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // ...
}
```

```
if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
    // ...
}
```

1.3. Select Statements

Specifying A Select Clause

Không phải lúc nào cũng muốn chọn tất cả các cột từ bảng cơ sở dữ liệu. Sử dụng phương thức `select`, có thể chỉ định mệnh đề " select " tùy chỉnh cho truy vấn:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
            ->select('name', 'email as user_email')
            ->get();
```

Phương thức `distinct` cho phép buộc truy vấn trả về các kết quả riêng biệt:

```
$users = DB::table('users')->distinct()->get();
```

Nếu đã có một Query Builder instance và muốn thêm một cột vào mệnh đề select hiện có, có thể sử dụng phương thức `addSelect`:

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

1.4. Raw Expressions

Đôi khi cần chèn một chuỗi tùy ý vào một truy vấn. Để tạo một biểu thức chuỗi, có thể sử dụng phương thức `raw` được cung cấp bởi `DB` facade:

```
$users = DB::table('users')
             ->select(DB::raw('count(*) as user_count, status'))
             ->where('status', '<>', 1)
             ->groupBy('status')
```

```
            ->get();
```

Các raw statements sẽ được đưa vào truy vấn dưới dạng chuỗi, vì vậy phải cẩn thận để tránh tạo lỗ hổng SQL injection.

Raw Methods

Thay vì sử dụng phương thức `DB::raw`, có thể sử dụng các phương thức sau để chèn một biểu thức vào các phần khác nhau của truy vấn. Laravel không thể đảm bảo rằng bất kỳ truy vấn nào sử dụng biểu thức đều được bảo vệ khỏi các lỗ hổng SQL injection.

selectRaw

Phương thức `selectRaw` được sử dụng thay cho `addSelect(DB::raw(...))`. Phương thức này chấp nhận một mảng ràng buộc tùy chọn làm đối số thứ hai của nó:

```
$orders = DB::table('orders')
                ->selectRaw('price * ? as price_with_tax', [1.0825])
                ->get();
```

whereRaw / orWhereRaw

Phương thức `whereRaw` và `orWhereRaw` được sử dụng để đưa mệnh đề "where" vào truy vấn. Các phương thức này chấp nhận một mảng ràng buộc tùy chọn làm đối số thứ hai của chúng:

```
$orders = DB::table('orders')
                ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
                ->get();
```

havingRaw / orHavingRaw

Các phương thức `havingRaw` và `orHavingRaw` được sử dụng để cung cấp một raw string làm giá trị của mệnh đề " having ". Các phương thức này chấp nhận một mảng ràng buộc tùy chọn làm đối số thứ hai:

```
$orders = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > ?', [2500])
                ->get();
```

orderByRaw

Phương thức `orderByRaw` được sử dụng để cung cấp một raw string làm giá trị của mệnh đề "order by":

```
$orders = DB::table('orders')
                ->orderByRaw('updated_at - created_at DESC')
                ->get();
```

groupByRaw

Phương thức `groupByRaw` được sử dụng để cung cấp một raw string làm giá trị của mệnh đề `group by`:

```
$orders = DB::table('orders')
                ->select('city', 'state')
                ->groupByRaw('city, state')
                ->get();
```

1.5. Joins

Inner Join Clause

Query Builder cũng có thể được sử dụng để thêm các mệnh đề nối vào các truy vấn. Để thực hiện một " inner join" cơ bản, sử dụng phương thức `join` trên một query builder instance. Đối số đầu tiên được truyền cho phương thức `join` là tên của bảng cần tham gia, trong khi các đối số còn lại chỉ định các ràng buộc cột cho phép join. Thậm chí có thể kết hợp nhiều bảng trong một truy vấn:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

Left Join / Right Join Clause

Nếu muốn thực hiện " left join" hoặc " right join " thay vì " inner join", hãy sử dụng các phương thức `leftJoin` hoặc `rightJoin`. Các phương thức này có cùng chữ ký với phương thức `join`:

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
```

```
            ->get();


$users = DB::table('users')
            ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

Cross Join Clause

Có thể sử dụng phương thức `crossJoin` để thực hiện " cross join". Các phép nối chéo tạo ra một tích cartesian giữa bảng đầu tiên và bảng đã nối:

```
$sizes = DB::table('sizes')
            ->crossJoin('colors')
            ->get();
```

Advanced Join Clauses

Có thể chỉ định các mệnh đề join nâng cao. Để bắt đầu, hãy truyền một closure làm đối số thứ hai cho phương thức `join`. Closure sẽ nhận được một instance `Illuminate\Database\Query\JoinClause` cho phép chỉ định các ràng buộc đối với mệnh đề "join":

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
        })
        ->get();
```

Nếu muốn sử dụng mệnh đề "where" trên các phép join, sử dụng các phương thức `where` và `orWhere` được cung cấp bởi `JoinClause` instance. Thay vì so sánh hai columns, các phương thức này sẽ so sánh column với một giá trị:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')
                 ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

Subquery Joins

Có thể sử dụng các phương thức `joinSub`, `leftJoinSub` và `rightJoinSub` để join một truy vấn với một truy vấn con (subquery). Mỗi phương thức này nhận được ba đối số: truy vấn con, bí danh bảng và một closure xác định các columns liên quan.

Ví dụ, truy xuất một tập hợp users trong đó mỗi bản ghi người dùng chứa `created_at` timestamp của bài đăng blog được xuất bản gần đây nhất của người dùng:

```
$latestPosts = DB::table('posts')
                   ->select('user_id', DB::raw('MAX(created_at) as last_post_created_at'))
                   ->where('is_published', true)
                   ->groupBy('user_id');

$users = DB::table('users')
        ->joinSub($latestPosts, 'latest_posts', function ($join) {
            $join->on('users.id', '=', 'latest_posts.user_id');
        })->get();
```

1.6. Unions

Query Builder cũng cung cấp một phương thức `union` để "union" hai hoặc nhiều truy vấn lại với nhau.

Ví dụ, tạo một truy vấn ban đầu và sử dụng phương thức `union` để kết hợp nó với các truy vấn khác:

```
use Illuminate\Support\Facades\DB;

$first = DB::table('users')
            ->whereNull('first_name');

$users = DB::table('users')
            ->whereNull('last_name')
            ->union($first)
            ->get();
```

Ngoài phương thức `union`, query builder cung cấp phương thức `unionAll`. Các truy vấn được kết hợp bằng phương thức `unionAll` sẽ không bị xóa các kết quả trùng lặp. Phương thức `unionAll` có cùng chữ ký

phương thức với phương thức `union`.

1.7. Basic Where Clauses

1.7.1. Where Clauses

Có thể sử dụng phương thức `where` của query builder để thêm mệnh đề "where" vào truy vấn. Lệnh gọi cơ bản nhất đến phương thức `where` yêu cầu ba đối số. Đối số đầu tiên là tên của column. Đối số thứ hai là một toán tử, có thể là bất kỳ toán tử nào được hỗ trợ của cơ sở dữ liệu. Đối số thứ ba là giá trị để so sánh với giá trị của cột.

Ví dụ, truy vấn sau đây truy xuất người dùng trong đó giá trị của cột `votes` bằng 100 và giá trị của cột `age` lớn hơn 35:

```
$users = DB::table('users')
                ->where('votes', '=', 100)
                ->where('age', '>', 35)
                ->get();
```

Để thuận tiện, nếu muốn xác minh rằng một cột `=` với một giá trị nhất định, có thể truyền giá trị làm đối số thứ hai cho phương thức `where`. Laravel sẽ giả sử sử dụng toán tử `=`:

```
$users = DB::table('users')->where('votes', 100)->get();
```

Như đã trình bày, có thể sử dụng bất kỳ toán tử nào được hỗ trợ bởi hệ quản trị cơ sở dữ liệu:

```
$users = DB::table('users')
                ->where('votes', '>=', 100)
                ->get();

$users = DB::table('users')
                ->where('votes', '<>', 100)
                ->get();

$users = DB::table('users')
                ->where('name', 'like', 'T%')
                ->get();
```

Cũng có thể truyền một mảng các điều kiện cho hàm `where`. Mỗi phần tử của mảng phải là một mảng chứa ba đối số thường được truyền cho phương thức `where`:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

1.7.2. Or Where Clauses

Khi chuỗi cùng nhau gọi phương thức `where` của query builder, các mệnh đề "where" sẽ được joined với nhau bằng cách sử dụng toán tử `and`. Tuy nhiên, có thể sử dụng phương thức `orWhere` để join một mệnh đề với truy vấn bằng toán tử `or`. Phương thức `orWhere` chấp nhận các đối số giống như phương thức `where`:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

Nếu cần nhóm một điều kiện "or" trong dấu ngoặc đơn, có thể truyền một closure làm đối số đầu tiên cho phương thức `orWhere`:

```
$users = DB::table('users')
            ->where('votes', '>', 100)
            ->orWhere(function($query) {
                $query->where('name', 'Abigail')
                      ->where('votes', '>', 50);
            })
            ->get();
```

Ví dụ trên sẽ tạo ra SQL sau:

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

1.7.3. JSON Where Clauses

Laravel cũng hỗ trợ truy vấn các loại JSON column trên cơ sở dữ liệu hỗ trợ cho các loại JSON column. Hiện

tại, bao gồm MySQL 5.7+, PostgreSQL, SQL Server 2016 và SQLite 3.9.0 (with the JSON1 extension). Để truy vấn một JSON column, hãy sử dụng toán tử `->`:

```
$users = DB::table('users')
                ->where('preferences->dining->meal', 'salad')
                ->get();
```

Có thể sử dụng `whereJsonContains` để truy vấn JSON arrays. Tính năng này không được cơ sở dữ liệu SQLite hỗ trợ:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', 'en')
                ->get();
```

Nếu ứng dụng sử dụng cơ sở dữ liệu MySQL hoặc PostgreSQL, có thể truyền một mảng giá trị cho phương thức `whereJsonContains`:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', ['en', 'de'])
                ->get();
```

Có thể sử dụng phương thức `whereJsonLength` để truy vấn các JSON arrays theo độ dài của chúng:

```
$users = DB::table('users')
                ->whereJsonLength('options->languages', 0)
                ->get();

$users = DB::table('users')
                ->whereJsonLength('options->languages', '>', 1)
                ->get();
```

1.7.4. Additional Where Clauses

whereBetween / orWhereBetween

Phương thức `whereBetween` kiểm tra giá trị của cột nằm giữa hai giá trị:

```
$users = DB::table('users')
```

```
            ->whereBetween('votes', [1, 100])
            ->get();
```

whereNotBetween / orWhereNotBetween

Phương thức `whereNotBetween` kiểm tra giá trị của cột nằm ngoài hai giá trị:

```
$users = DB::table('users')
                    ->whereNotBetween('votes', [1, 100])
                    ->get();
```

whereIn / whereNotIn / orWhereIn / orWhereNotIn

Phương thức `whereIn` kiểm tra giá trị của một cột được chứa trong mảng đã cho:

```
$users = DB::table('users')
                    ->whereIn('id', [1, 2, 3])
                    ->get();
```

Phương thức `whereNotIn` kiểm tra giá trị của cột đã cho không được chứa trong mảng đã cho:

```
$users = DB::table('users')
                    ->whereNotIn('id', [1, 2, 3])
                    ->get();
```

whereNull / whereNotNull / orWhereNull / orWhereNotNull

Phương thức `whereNull` kiểm tra giá trị của cột đã cho là `NULL`:

```
$users = DB::table('users')
                ->whereNull('updated_at')
                ->get();
```

Phương thức `whereNotNull` kiểm tra giá trị của cột không phải là `NULL`:

```
$users = DB::table('users')
                ->whereNotNull('updated_at')
                ->get();
```

whereDate / whereMonth / whereDay / whereYear / whereTime

Phương thức `whereDate` có thể được sử dụng để so sánh giá trị của cột với một ngày:

```
$users = DB::table('users')
                ->whereDate('created_at', '2016-12-31')
                ->get();
```

Phương thức `whereMonth` có thể được sử dụng để so sánh giá trị của cột với một tháng cụ thể:

```
$users = DB::table('users')
                ->whereMonth('created_at', '12')
                ->get();
```

Phương thức `whereDay` có thể được sử dụng để so sánh giá trị của cột với một ngày cụ thể trong tháng:

```
$users = DB::table('users')
                ->whereDay('created_at', '31')
                ->get();
```

Phương pháp `whereYear` có thể được sử dụng để so sánh giá trị của cột với một năm cụ thể:

```
$users = DB::table('users')
                ->whereYear('created_at', '2016')
                ->get();
```

Phương thức `whereTime` có thể được sử dụng để so sánh giá trị của cột với một thời gian cụ thể:

```
$users = DB::table('users')
                ->whereTime('created_at', '=', '11:20:45')
                ->get();
```

whereColumn / orWhereColumn

Phương thức `whereColumn` có thể được sử dụng để kiểm tra hai cột bằng nhau:

```
$users = DB::table('users')
                ->whereColumn('first_name', 'last_name')
```

```
                ->get();
```

Cũng có thể truyền một toán tử so sánh cho phương thức `whereColumn`:

```
$users = DB::table('users')
                ->whereColumn('updated_at', '>', 'created_at')
                ->get();
```

Có thể truyền một mảng so sánh cột vào phương thức `whereColumn`. Các điều kiện này sẽ được kết hợp bằng cách sử dụng toán tử `and`:

```
$users = DB::table('users')
                ->whereColumn([
                    ['first_name', '=', 'last_name'],
                    ['updated_at', '>', 'created_at'],
                ])->get();
```

1.7.5. Logical Grouping

Đôi khi cần nhóm một số mệnh đề "where" trong dấu ngoặc đơn để được nhóm logic mong muốn của truy vấn. Trong thực tế, thường nhóm các lệnh gọi đến phương thức `orWhere` trong dấu ngoặc đơn để tránh hành vi truy vấn không mong muốn. Để thực hiện điều này, có thể truyền một closure vào phương thức `where`:

```
$users = DB::table('users')
           ->where('name', '=', 'John')
           ->where(function ($query) {
               $query->where('votes', '>', 100)
                     ->orWhere('title', '=', 'Admin');
           })
           ->get();
```

Việc truyền một closure vào phương thức `where` hướng dẫn query builder bắt đầu một nhóm ràng buộc. Closure sẽ nhận được một thể hiện của query builder mà có thể sử dụng để đặt các ràng buộc nên được chứa trong nhóm dấu ngoặc đơn. Ví dụ trên sẽ tạo ra SQL sau:

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

1.8. Advanced Where Clauses

1.8.1. Where Exists Clauses

Phương thức `whereExists` cho phép viết các mệnh đề SQL "where exists". Phương thức `whereExists` chấp nhận một closure sẽ nhận một query builder instance, cho phép xác định truy vấn sẽ được đặt bên trong mệnh đề "exists":

```
$users = DB::table('users')
           ->whereExists(function ($query) {
               $query->select(DB::raw(1))
                     ->from('orders')
                     ->whereColumn('orders.user_id', 'users.id');
           })
           ->get();
```

Truy vấn trên sẽ tạo ra SQL sau:

```
select * from users
where exists (
    select 1
    from orders
    where orders.user_id = users.id
)
```

1.8.2. Subquery Where Clauses

Đôi khi cần phải xây dựng một mệnh đề "where" để so sánh kết quả của một truy vấn con với một giá trị nhất định. Thực hiện điều này bằng cách truyền một closure và một giá trị cho phương thức `where`. Ví dụ: truy vấn sau sẽ truy xuất tất cả người dùng có "membership" gần đây của một loại nhất định;

```
use App\Models\User;

$users = User::where(function ($query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
```

```
        ->orderByDesc('membership.start_date')
        ->limit(1);
}, 'Pro')->get();
```

Hoặc, có thể xây dựng một mệnh đề "where" so sánh một cột với kết quả của một truy vấn con. Có thể thực hiện điều này bằng cách truyền một cột, toán tử và closure vào phương thức `where`.

Ví dụ, truy vấn sau sẽ truy xuất tất cả các bản ghi thu nhập trong đó số tiền nhỏ hơn mức trung bình;

```
use App\Models\Income;

$incomes = Income::where('amount', '<', function ($query) {
    $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

1.9. Ordering, Grouping, Limit & Offset

1.9.1. Ordering

The orderBy Method

Phương thức `orderBy` cho phép sắp xếp các kết quả của truy vấn theo một cột nhất định. Đối số đầu tiên được phương thức `orderBy` chấp nhận phải là cột muốn sắp xếp, trong khi đối số thứ hai xác định hướng sắp xếp và có thể là `asc` hoặc `desc`:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->get();
```

Để sắp xếp theo nhiều cột, có thể chỉ cần gọi `orderBy` nhiều lần nếu cần:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->orderBy('email', 'asc')
                ->get();
```

The latest & oldest Methods

Các phương thức `latest` và `oldest` cho phép dễ dàng sắp xếp kết quả theo ngày. Theo mặc định, kết quả sẽ được sắp xếp theo cột `created_at` của bảng. Hoặc, có thể truyền tên cột muốn sắp xếp theo:

```
$user = DB::table('users')
                ->latest()
                ->first();
```

Random Ordering

Phương thức `inRandomOrder` được sử dụng để sắp xếp các kết quả truy vấn một cách ngẫu nhiên.

Ví dụ, có thể sử dụng phương pháp này để tìm nạp một người dùng ngẫu nhiên:

```
$randomUser = DB::table('users')
                ->inRandomOrder()
                ->first();
```

Removing Existing Orderings

Phương thức `reorder` loại bỏ tất cả các mệnh đề "order by" đã được áp dụng trước đó cho truy vấn:

```
$query = DB::table('users')->orderBy('name');

$unorderedUsers = $query->reorder()->get();
```

Có thể truyền một column và direction khi gọi phương thức `reorder` để loại bỏ tất cả các mệnh đề "order by" hiện có và áp dụng một thứ tự hoàn toàn mới cho truy vấn:

```
$query = DB::table('users')->orderBy('name');

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

1.9.2. Grouping

The groupBy & having Methods

Các phương thức `groupBy` và `having` có thể được sử dụng để nhóm các kết quả truy vấn. Chữ ký của phương thức `having` tương tự như chữ ký của phương thức `where`:

```
$users = DB::table('users')
                ->groupBy('account_id')
                ->having('account_id', '>', 100)
```

```
                ->get();
```

Có thể sử dụng phương thức `havingBetween` để lọc kết quả trong một phạm vi nhất định:

```
$report = DB::table('orders')
                ->selectRaw('count(id) as number_of_orders, customer_id')
                ->groupBy('customer_id')
                ->havingBetween('number_of_orders', [5, 15])
                ->get();
```

Có thể truyền nhiều đối số cho phương thức `groupBy` để gom nhóm theo nhiều cột:

```
$users = DB::table('users')
                ->groupBy('first_name', 'status')
                ->having('account_id', '>', 100)
                ->get();
```

Để xây dựng các câu lệnh `having` nâng cao hơn, hãy xem phương thức `havingRaw`.

1.9.3. Limit & Offset

The `skip` & `take` Methods

Có thể sử dụng phương thức `skip` và `take` để giới hạn số lượng kết quả trả về từ truy vấn hoặc bỏ qua một số kết quả nhất định trong truy vấn:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Ngoài ra, có thể sử dụng các phương thức `limit` và `offset`. Các phương thức này tương đương về mặt chức năng với các phương thức `take` và `skip`, tương ứng:

```
$users = DB::table('users')
                ->offset(10)
                ->limit(5)
                ->get();
```

1.10. Conditional Clauses

Khi muốn các mệnh đề truy vấn nhất định áp dụng cho một truy vấn dựa trên một điều kiện khác.

Ví dụ, có thể chỉ muốn áp dụng câu lệnh `where` nếu một giá trị đầu vào nhất định có trong yêu cầu HTTP request. Có thể thực hiện điều này bằng cách sử dụng phương pháp `when`:

```
$role = $request->input('role');

$users = DB::table('users')
                ->when($role, function ($query, $role) {
                    return $query->where('role_id', $role);
                })
                ->get();
```

Phương thức `when` chỉ thực hiện closure đã cho khi đối số đầu tiên là `true`. Nếu đối số đầu tiên là `false`, closure sẽ không được thực hiện. Vì vậy, trong ví dụ trên, closure được cung cấp cho phương thức `when` sẽ chỉ được gọi nếu trường `role` hiện diện trong yêu cầu gửi đến và đánh giá là `true`.

Có thể truyền một closure khác làm đối số thứ ba cho phương thức `when`. Closure này sẽ chỉ thực thi nếu đối số đầu tiên đánh giá là `false`. Để minh họa cách sử dụng tính năng này, sử dụng nó để định cấu hình thứ tự mặc định của một truy vấn:

```
$sortByVotes = $request->input('sort_by_votes');

$users = DB::table('users')
                ->when($sortByVotes, function ($query, $sortByVotes) {
                    return $query->orderBy('votes');
                }, function ($query) {
                    return $query->orderBy('name');
                })
                ->get();
```

1.11. Insert Statements

Query Builder cũng cung cấp một phương thức `insert` có thể được sử dụng để chèn các bản ghi vào bảng cơ sở dữ liệu. Phương thức `insert` chấp nhận một mảng tên và giá trị cột:

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
```

```
]);
```

Có thể chèn nhiều bản ghi cùng một lúc bằng cách truyền một mảng của mảng. Mỗi mảng đại diện cho một bản ghi cần được chèn vào bảng:

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

Phương thức `insertOrIgnore` sẽ bỏ qua lỗi khi chèn bản ghi vào cơ sở dữ liệu:

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

`insertOrIgnore` sẽ bỏ qua các bản ghi trùng lặp và cũng có thể bỏ qua các loại lỗi khác tùy thuộc vào công cụ cơ sở dữ liệu. Ví dụ, `insertOrIgnore` sẽ bỏ qua bypass MySQL's strict mode.

Auto-Incrementing IDs

Nếu bảng có id tăng tự động, hãy sử dụng phương thức `insertGetId` để chèn bản ghi và sau đó truy xuất ID:

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

Khi sử dụng PostgreSQL, phương thức `insertGetId` yêu cầu cột tự động tăng dần được đặt tên là id. Nếu muốn truy xuất ID từ một "sequence" khác, có thể chuyển tên cột làm tham số thứ hai cho phương thức `insertGetId`.

1.11.1. Upserts

Phương thức `upsert` sẽ chèn các bản ghi không tồn tại và cập nhật các bản ghi đã tồn tại với các giá trị mới có thể chỉ định. Đối số đầu tiên của phương thức bao gồm các giá trị để chèn hoặc cập nhật, trong khi đối số thứ hai liệt kê (các) cột xác định duy nhất các bản ghi trong bảng được liên kết. Đối số thứ ba và cuối cùng của phương thức là một mảng cột cần được cập nhật nếu bản ghi phù hợp đã tồn tại trong cơ sở dữ

liệu:

```
DB::table('flights')->upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

Trong ví dụ trên, Laravel sẽ cố gắng chèn hai bản ghi. Nếu một bản ghi đã tồn tại với cùng giá trị cột `departure` và `destination`, Laravel sẽ cập nhật cột `price` của bản ghi đó.

Cơ sở dữ liệu ngoại trừ SQL Server yêu cầu các cột trong đối số thứ hai của phương thức `upsert` phải có chỉ mục "primary" hoặc "unique". Ngoài ra, trình điều khiển cơ sở dữ liệu MySQL bỏ qua đối số thứ hai của phương thức `upsert` và luôn sử dụng chỉ mục "primary" và "uniqu " của bảng để phát hiện các bản ghi hiện có.

1.12. Update Statements

Ngoài việc chèn các bản ghi vào cơ sở dữ liệu, query builder cũng có thể cập nhật các bản ghi hiện có bằng phương thức `update`. Phương thức `update`, giống như phương thức `insert`, chấp nhận một mảng các cặp cột và giá trị cho biết các cột sẽ được cập nhật. Có thể hạn chế truy vấn `update` bằng mệnh đề `where`:

```
$affected = DB::table('users')
              ->where('id', 1)
              ->update(['votes' => 1]);
```

Update Or Insert

Đôi khi muốn cập nhật một bản ghi hiện có trong cơ sở dữ liệu hoặc tạo nó nếu không có bản ghi phù hợp nào tồn tại. Trong trường hợp này, phương thức `updateOrInsert` có thể được sử dụng. Phương thức `updateOrInsert` chấp nhận hai đối số: một mảng các điều kiện để tìm bản ghi và một mảng các cặp giá trị và cột cho biết các cột sẽ được cập nhật.

Phương thức `updateOrInsert` sẽ cố gắng xác định một bản ghi cơ sở dữ liệu phù hợp bằng cách sử dụng các cặp giá trị và cột của đối số đầu tiên. Nếu bản ghi tồn tại, nó sẽ được cập nhật các giá trị trong đối số thứ hai. Nếu không thể tìm thấy bản ghi, bản ghi mới sẽ được chèn với các thuộc tính đã hợp nhất của cả hai đối số:

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
```

```
        ['votes' => '2']
    );
```

Updating JSON Columns

Khi cập nhật một JSON column, nên sử dụng cú pháp `->` để cập nhật khóa thích hợp trong đối tượng JSON. Thao tác này được hỗ trợ trên MySQL 5.7+ và PostgreSQL 9.5+:

```
$affected = DB::table('users')
              ->where('id', 1)
              ->update(['options->enabled' => true]);
```

Increment & Decrement

Query Builder cung cấp các phương thức thuận tiện để tăng hoặc giảm giá trị của một cột nhất định. Cả hai phương thức này đều chấp nhận ít nhất một đối số: cột cần sửa đổi. Đối số thứ hai có thể được cung cấp để chỉ định số lượng cột sẽ được tăng hoặc giảm:

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

Cũng có thể chỉ định các cột bổ sung để cập nhật trong quá trình hoạt động:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

1.13. Delete Statements

Phương thức `delete` của Query Builder có thể được sử dụng để xóa các bản ghi khỏi bảng. Có thể hạn chế các câu lệnh `delete` bằng cách thêm mệnh đề "where" trước khi gọi phương thức `delete`:

```
DB::table('users')->delete();

DB::table('users')->where('votes', '>', 100)->delete();
```

Nếu muốn truncate toàn bộ bảng, thao tác này sẽ xóa tất cả các bản ghi khỏi bảng và đặt lại ID tự động tăng dần về 0, có thể sử dụng phương thức `truncate`:

```
DB::table('users')->truncate();
```

Table Truncation & PostgreSQL

Khi truncating một cơ sở dữ liệu PostgreSQL, hành vi `CASCADE` sẽ được áp dụng. Điều này có nghĩa là tất cả các bản ghi liên quan đến khóa ngoại trong các bảng khác cũng sẽ bị xóa.

1.14. Pessimistic Locking

Query Builder cũng bao gồm một số chức năng để giúp đạt được "pessimistic" khi thực hiện các câu lệnh `select` của mình. Để thực hiện một câu lệnh với "shared lock", có thể gọi phương thức `sharedLock`. Khóa dùng chung ngăn các dòng đã chọn không được sửa đổi cho đến khi giao dịch được committed:

```
DB::table('users')
        ->where('votes', '>', 100)
        ->sharedLock()
        ->get();
```

Ngoài ra, có thể sử dụng phương thức `lockForUpdate`. Khóa "for update" ngăn không cho sửa đổi các bản ghi đã chọn hoặc không được chọn bằng một khóa được chia sẻ khác:

```
DB::table('users')
        ->where('votes', '>', 100)
        ->lockForUpdate()
        ->get();
```

1.15. Debugging

Có thể sử dụng các phương thức `dd` và `dd` để xây dựng một truy vấn để kết xuất các ràng buộc truy vấn hiện tại và SQL. Phương thức `dd` sẽ hiển thị thông tin gỡ lỗi và sau đó dừng thực hiện yêu cầu. Phương thức `dump` sẽ hiển thị thông tin gỡ lỗi nhưng cho phép yêu cầu tiếp tục thực hiện:

```
DB::table('users')->where('votes', '>', 100)->dd();


DB::table('users')->where('votes', '>', 100)->dump();
```