# ADVANCED ACTIVE RECORD

*A Collection of Practical Deep Dives into Active Record Topics*

TOM COPELAND

# Advanced Active Record

## A Collection of Practical Deep Dives into Active Record Topics

Tom Copeland

This book is for sale at http://leanpub.com/advancedactiverecord

This version was published on 2018-01-15


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# Contents

# Scopes

Queries let us get data out of the database. Scopes let us organize those queries so that they're accessible in various contexts. This chapter will look at some of the edge cases around defining and using scopes.

## A Case for AREL

The standard use case for scopes is to encapsulate a query so it can be reused. For example, we could add a scope on `Reviewer` that finds newly created reviewers:

```
scope :very_recent, -> { where("created_at > ?", 1.hour.ago)}
```

This scope looks a bit grotty since it has some hardcoded SQL. But there's a more serious issue. Suppose that, given a particular book, we want to find all that book's new reviewers. We can traverse the `reviews` association, but instead of a collection of reviewers we get an error:

```
>> Book.first.reviewers.very_recent
  Book Load (0.4ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" ASC LI\
MIT $1  [["LIMIT", 1]]
  Reviewer Load (0.5ms)  SELECT  "reviewers".* FROM "reviewers" INNER JOIN "revi\
ews" ON "reviewers"."id" = "reviews"."reviewer_id" WHERE "reviews"."book_id" = $\
1 AND (created_at > '2017-09-23 03:15:26.525710') LIMIT $2  [["book_id", 1], ["L\
IMIT", 11]]
ActiveRecord::StatementInvalid: PG::AmbiguousColumn: ERROR:  column reference "c\
reated_at" is ambiguous
```

```
LINE 1: ..."reviewer_id" WHERE "reviews"."book_id" = $1 AND (created_at...
                                                             ^
: SELECT  "reviewers".* FROM "reviewers" INNER JOIN "reviews" ON "reviewers"."id\
" = "reviews"."reviewer_id" WHERE "reviews"."book_id" = $1 AND (created_at > '20\
17-09-23 03:15:26.525710') LIMIT $2
```

That hardcoded SQL is biting us. It's understandable that Active Record would produce an invalid query here; in our scope definition we're giving it a raw SQL string and saying that SQL should be transplanted into any query using that scope. So it's on us. Fortunately we can dip into the Arel object model to solve this. Here's an alternate syntax:

```
scope :very_recent, -> { where(arel_table[:created_at].gt(1.hour.ago)) }
```

`arel_table` table is a class method on `ActiveRecord::Base`, so it's accessible from any model. It returns the Arel table definition and informs the scope exactly which `created_at` column is being referenced. Now our scope usage across a join works fine because the ambiguity is removed:

```
>> Book.first.reviewers.very_recent
  Book Load (0.2ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" ASC LI\
MIT $1  [["LIMIT", 1]]
  Reviewer Load (0.7ms)  SELECT  "reviewers".* FROM "reviewers" INNER JOIN "revi\
ews" ON "reviewers"."id" = "reviews"."reviewer_id" WHERE "reviews"."book_id" = $\
1 AND ("reviewers"."created_at" > '2017-09-23 03:19:44.797524') LIMIT $2  [["boo\
k_id", 1], ["LIMIT", 11]]
```

This also applies if you're merging scopes. Finding all books with reviews created in the past week could be done with a scope on `Book`:

```
scope :reviewed_in_last_week, -> { joins(:reviews).where("reviews.created_at > ?\
", 1.week.ago) }
```

But you'd want to avoid mixing SQL related to the `Review` model in `Book`, so you'd extract that query to a scope on `Review`:

```
scope :in_past_week, -> { where("created_at > ?", 1.week.ago) }
```

And then the `reviewed_in_last_week` scope could merge the scope from `Reviewer`, resulting in a more readable and less invasive scope:

```
scope :reviewed_in_last_week, -> { joins(:reviews).merge(Review.in_past_week) }
```

But that results in the same error due to the same column ambiguity. Fortunately, you can resolve it using the previous solution. We'll reference the Arel column in the `in_past_week` scope:

```
scope :in_past_week, -> { where(arel_table[:created_at].gt(1.week.ago)) }
```

I don't often recommend replacing non-trivial query expressions with their Arel equivalents. To my eye the SQL is easier to read; I find operators like `<=` more expressive than Arel methods like `lteq`. But occasionally, as in this situation, doing that translation can be the difference between success and a query that doesn't work. Having a "avoid SQL, favor Arel" policy wouldn't be unreasonable if your team is up for it.

## Single-record Behavior

Occasionally you'll see well-intentioned code defining a scope that returns an instance of an Active Record model. For example, here's a scope that returns the most recent featured review:

```
scope :most_recent_featured, -> { featured.last }
```

This appears to work; invoking it returns a single review:

```
>> Review.most_recent_featured
  Review Load (3.2ms)  SELECT  "reviews".* FROM "reviews" WHERE "reviews"."featu\
red" = $1 ORDER BY "reviews"."id" DESC LIMIT $2  [["featured", "t"], ["LIMIT", 1\
]]
=> #<Review id: 2, aasm_state: "published"
```

There's a conceptual problem around expectations here though. This is a scope, so we should be able to chain it. But since we're getting a single record back, that doesn't work; we can't chain something that's not an `ActiveRecord::Relation`. Here's an attempt to do so:

```
>> Review.most_recent_featured.order(updated_at: :desc)
  Review Load (0.6ms)  SELECT  "reviews".* FROM "reviews" WHERE "reviews"."featu\
red" = $1 ORDER BY "reviews"."id" DESC LIMIT $2  [["featured", "t"], ["LIMIT", 1\
]]
NoMethodError: undefined method `order' for #<Review:0x007fa1d7d91b80>
```

This may seem obvious, but the point of scopes is to return relations. A scope that returns only a single record breaks that contract. You could use `scope :most_recent_featured, -> { featured.limit(1) }` which would return a relation and thus be chainable. But then client code that wants the most recent record would still need to call `first`, which would be clumsy.

One way to adjust to this situation is to change tactics. Instead of a scope, you can define a class method. By defining a class method, you can return a single object without setting up the expectations that a scope implies:

```
def self.most_recent_featured
  featured.last
end
```

Defining a scope that returns something other than a relation violates all sorts of expectations around chaining and merging. Avoid it!

## Default Scopes and Unscoped

One error-prone yet remarkably popular feature of scopes is the concept of default scopes. The basics here are straightforward. Default scopes give us a way to automatically apply a set of conditions when querying a table. Sometimes this is as simple as a default `order` clause that always returns recently updated items first. A more interesting application is when a default scope is used to implement a "soft delete" capability. Typically a library will implement this with a `deleted_at` column. When that column contains a timestamp and thus is not null, the default scope ensures that record will no longer be returned when the table is queried. So it's a way to mark rows as removed without really deleting them, which also means they could be restored if needed.

The "critic" application has a default scope that's somewhere in the middle. The `Rating` model has a default scope which only allows approved ratings to be retrieved:

```
default_scope -> { where(approved: true) }
```

This means everywhere in the application we avoid the need to filter out unapproved ratings. Very handy!

Another useful feature of default scopes is that, for default scopes that use an equality check, new objects receive the default scope's value at initialization. Here's an example using the same default scope but in the context of record initialization:

```
>> Rating.new
=> #<Rating id: nil, score: nil, approved: true, book_id: nil>
```

The `approved` attribute now gets assigned `true`. This behavior of new objects automatically receiving the default scope's default value is either quite helpful or quite surprising depending on if you're expecting it.

At any rate, we'll need to bypass the default scope in some cases. For example, an administrative portion of the application would need to see ratings which hadn't yet been approved in order to approve them. We can bypass the default scope with an `unscoped` call:

```
>> Rating.count
   (0.9ms)  SELECT COUNT(*) FROM "ratings" WHERE "ratings"."approved" = $1  [["a\
pproved", "t"]]
=> 1
>> Rating.unscoped.count
   (3.1ms)  SELECT COUNT(*) FROM "ratings"
=> 2
```

The `unscoped` method has some unexpected behavior, though. Consider the case where we have a book and want all ratings whether they're approved or not. It might seem reasonable that we could chain `unscoped` as we traverse the association, but no. That would retrieve all ratings, even those for other books:

```
>> Book.last.ratings.unscoped
  Book Load (0.4ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" DESC L\
IMIT $1  [["LIMIT", 1]]
  Rating Load (2.8ms)  SELECT  "ratings".* FROM "ratings" LIMIT $1  [["LIMIT", 1\
1]]
=> #<ActiveRecord::Relation [#<Rating id: 1, score: 1, approved: false, book_id:\
 2>, #<Rating id: 2, score: 5, approved: true, book_id: 1>]>
```

In other words, `unscoped` removes all conditions, even those provided as a result of traversing associations. So the default scope condition we want to remove is gone, which is correct. But the book id restriction from the association, which is a condition we want to keep, is also removed. The `unscoped` method accepts a block, but traversing the association still applies the scope:

```
>> Rating.unscoped { Book.last.ratings }
  Book Load (0.5ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" DESC L\
IMIT $1  [["LIMIT", 1]]
  Rating Load (0.4ms)  SELECT  "ratings".* FROM "ratings" WHERE "ratings"."appro\
ved" = $1 AND "ratings"."book_id" = $2 LIMIT $3  [["approved", "t"], ["book_id",\
 2], ["LIMIT", 11]]
=> #<ActiveRecord::Associations::CollectionProxy [#<Rating id: 2, score: 5, appr\
oved: true, book_id: 2>]>
```

To fetch all ratings when using the block form of `unscoped`, we need to provide an alternate `where` clause:

```
>> Rating.unscoped { Book.last.ratings.where.not(score: nil) }
  Book Load (0.5ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" DESC L\
IMIT $1  [["LIMIT", 1]]
  Rating Load (2.8ms)  SELECT  "ratings".* FROM "ratings" WHERE "ratings"."book_\
id" = $1 AND ("ratings"."score" IS NOT NULL) LIMIT $2  [["book_id", 2], ["LIMIT"\
, 11]]
=> #<ActiveRecord::AssociationRelation [#<Rating id: 1, score: 1, approved: fals\
e, book_id: 2>, #<Rating id: 2, score: 5, approved: true, book_id: 2>]>
```

But a more readable approach is to use `unscope`. This has a similar name to `unscoped`, but different behavior. It provides a way to remove conditions:

```
>> Book.last.ratings.unscope(:where)
  Book Load (0.5ms)  SELECT  "books".* FROM "books" WHERE "books"."id" = $1 LIMI\
T $2  [["id", 2], ["LIMIT", 1]]
  Rating Load (2.3ms)  SELECT  "ratings".* FROM "ratings" LIMIT $1  [["LIMIT", 1\
1]]
=> #<ActiveRecord::AssociationRelation [#<Rating id: 1, score: 1, approved: fals\
e, book_id: 2>, #<Rating id: 2, score: 5, approved: true, book_id: 2>, #<Rating \
id: 3, score: 5, approved: true, book_id: 1>]>
```

Rails 5 provides the slightly more readable method `rewhere`, which is a thin wrapper around `unscope`. When using `rewhere` we need to supply the same `where` condition redefinition:

```
>> Book.last.ratings.rewhere(approved: [true, false])
  Book Load (0.4ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" DESC L\
IMIT $1  [["LIMIT", 1]]
  Rating Load (3.0ms)  SELECT  "ratings".* FROM "ratings" WHERE "ratings"."book_\
id" = $1 AND "ratings"."approved" IN ('t', 'f') LIMIT $2  [["book_id", 2], ["LIM\
IT", 11]]
=> #<ActiveRecord::AssociationRelation [#<Rating id: 1, score: 1, approved: fals\
e, book_id: 2>, #<Rating id: 2, score: 5, approved: true, book_id: 2>]>
```

Both `unscope` and `rewhere` require the client code to be aware of exactly what the default scope was doing, though. And more generally, these approaches are one-off fixes that would need to be duplicated every time we need to query all ratings. A more comprehensive approach is to provide a new association dedicated to fetching items that would normally be excluded by the default scope:

```
has_many :all_ratings, -> { unscope(where: :approved) }, class_name: 'Rating'
```

Now we can use `Book.last.all_ratings` to see both approved and unapproved ratings. Alternatively, we could define a scope on `Rating` that uses the `rewhere` method:

```
scope :not_approved, -> { rewhere(approved: false) }
```

This technique returns the ratings that haven't been approved even when queried via an association, e.g., `Book.last.ratings.not_approved`.

We've spent some time pointing out complexities with default scopes. A quick search will turn up plenty of eloquent blog posts that recommend never using them at all. The usual arguments include some of the issues that we've mentioned so far around complexities in bypassing them. Another point against them is that only the Rails application is aware of the soft-delete status. Other applications that connect to the same database will need to recognize this field and handle things appropriately. This is especially true for a data analytics team. All their SQL queries must include a `WHERE deleted_at IS NOT NULL` in order to be accurate.

There are two scenarios when I think they're still worth the trouble. The first is when you're using a UUID as a primary key. In that case, the default Active Record sort order is by primary key, so, calling `ActiveRecord::Base#first` on that model may not get you what you want. A default scope that orders by `created_at` may be reasonable there.

Another situation is the classic that we've discussed in this section. Sometimes you need a soft delete capability and you rarely if ever need to deal with "deleted" records. You should use an existing gem that supports this, such as paranoia[1]. That way you'll have helper scopes, a consistent deleted item field name, and various utility methods to make it all more bearable. Also, it's best to avoid giving the deleted status business logic implications. That is, for example, when cancelling an order you should change a `status` field rather than populating the soft delete column. This will help out your co-workers downstream. The analytics team will appreciate simply being able to exclude all records with a non-null `deleted_at` value rather than having more complex logic around that.

Default scopes have earned their reputation as being problematic. Use them if needed, but use them with care!

## Scopes as Builders

A common shortcut with scopes is to use them as builders. In other words, you can create a `Review` via the `featured` scope:

```
Review.featured.create!(book: Book.last, reviewer: Reviewer.first, content: "gre\
at!")
```

And that results in a `Review` with the `featured` boolean toggled to `true`. This probably isn't a big surprise given that we observed this when creating an instance of a model with a default scope. It's the same when navigating through a non-default scope.

This doesn't work for all scopes, though. For example, we might have an `oldish` scope that finds reviews from a month ago:

[1] https://rubygems.org/gems/paranoia

```
scope :oldish, -> { where(arel_table[:created_at].lt(1.month.ago)) }
```

And using that scope to create a new review results in a review with a `created_at` value of the current time rather than a date that conforms to that scope's condition. The reason for this is that when Active Record collects attributes from the current set of scopes for the purpose of creating a new object, it only looks at scopes that are comparing for equality. You can see where this happens in `ActiveRecord::Relation::WhereClause#to_h`; that method only extracts Arel nodes of the type `Equality`:

```
equalities = predicates.grep(Arel::Nodes::Equality)
```

As a short side trip, this code snippet is using a handy feature of `Enumerable#grep`. It does a triple equals (===) comparison, so if you pass a class name as an argument, it'll return only those elements that have that class:

```
>> [1,2,"3",4,"5"].grep String
=> ["3", "5"]
```

Of course, you can override === method on your own classes to get this behavior:

```
>> class Foo ; def ===(other) ; other == 42 ; end ; end
=> :===
>> [21,42].grep Foo
=> [42]
```

Back to scopes! You can get a preview of which attributes Active Record will prepopulate on a newly built object by invoking the `where_values_hash` method on a scope. Clearly there's a difference between `featured`, which is an equality comparison, and `oldish`, which isn't:

```
>> Review.oldish.where_values_hash
=> {}
>> Review.featured.where_values_hash
=> {"featured"=>true}
```

And of course this works on an Arel expression as well:

```
>> Review.where(Review.arel_table[:id].eq(2)).where_values_hash
=> {:id=>2}
```

Active Record's behavior in this scenario is understandable. In the case of `oldish`, how could Active Record know what date the client code would want populated on an open-ended scope like that? Restricting this feature to equality relations seems like the only way to usefully implement it.

# Onwards

For the most part, scopes help us manage queries primarily within a single model. But a relational database is all about relationships, and Active Record gives us associations to help manage those relationships. In the next chapter we'll look at some interesting features and behavior around associations.