Active Record Query Interface

January 13, 2015

This guide covers different ways to retrieve data from the database using Active Record.
After reading this guide, you will know:

- How to find records using a variety of methods and conditions.
- How to specify the order, retrieved attributes, grouping, and other properties of the found records.
- How to use eager loading to reduce the number of database queries needed for data retrieval.
- How to use dynamic finders methods.
- How to check for the existence of particular records.
- How to perform various calculations on Active Record models.
- How to run EXPLAIN on relations.

If you're used to using raw SQL to find database records, then you will generally find that there are better ways to carry out the same operations in Rails. Active Record insulates you from the need to use SQL in most cases.

Code examples throughout this guide will refer to one or more of the following models:
All of the following models use `id` as the primary key, unless specified otherwise.

```
class Client < ActiveRecord::Base
  has_one :address
  has_many :orders
  has_and_belongs_to_many :roles
end

class Address < ActiveRecord::Base
  belongs_to :client
end

class Order < ActiveRecord::Base
  belongs_to :client, counter_cache: true
end

class Role < ActiveRecord::Base
  has_and_belongs_to_many :clients
end
```

Active Record will perform queries on the database for you and is compatible with most database systems (MySQL, PostgreSQL and SQLite to name a few). Regardless of which database system you're using, the Active Record method format will always be the same.

1 Retrieving Objects from the Database

To retrieve objects from the database, Active Record provides several finder methods. Each finder method allows you to pass arguments into it to perform certain queries on your database without writing raw SQL.

The methods are:

- `bind`
- `create_with`
- `distinct`
- `eager_load`
- `extending`
- `from`
- `group`
- `having`
- `includes`
- `joins`
- `limit`
- `lock`
- `none`
- `offset`
- `order`
- `preload`
- `readonly`
- `references`
- `reorder`
- `reverse_order`
- `select`
- `uniq`
- `where`

All of the above methods return an instance of `ActiveRecord::Relation`.

The primary operation of `Model.find(options)` can be summarized as:

- Convert the supplied options to an equivalent SQL query.
- Fire the SQL query and retrieve the corresponding results from the database.
- Instantiate the equivalent Ruby object of the appropriate model for every resulting row.
- Run `after_find` callbacks, if any.

1.1 Retrieving a Single Object

Active Record provides several different ways of retrieving a single object.

1.1.1 `find` Using the `find` method, you can retrieve the object corresponding to the specified *primary key* that matches any supplied options. For example:

```
# Find the client with primary key (id) 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients WHERE (clients.id = 10) LIMIT 1
```

The `find` method will raise an `ActiveRecord::RecordNotFound` exception if no matching record is found.

You can also use this method to query for multiple objects. Call the `find` method and pass in an array of primary keys. The return will be an array containing all of the matching records for the supplied *primary keys*. For example:

```
# Find the clients with primary keys 1 and 10.
client = Client.find([1, 10]) # Or even Client.find(1, 10)
# => [#<Client id: 1, first_name: "Lifo">, #<Client id: 10, first_name:
"Ryan">]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients WHERE (clients.id IN (1,10))
```

The `find` method will raise an `ActiveRecord::RecordNotFound` exception unless a matching record is found for **all** of the supplied primary keys.

1.1.2 `take` The `take` method retrieves a record without any implicit ordering. For example:

```
client = Client.take
# => #<Client id: 1, first_name: "Lifo">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients LIMIT 1
```

The `take` method returns `nil` if no record is found and no exception will be raised.

You can pass in a numerical argument to the `take` method to return up to that number of results. For example

```
client = Client.take(2)
# => [
  #<Client id: 1, first_name: "Lifo">,
  #<Client id: 220, first_name: "Sara">
]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients LIMIT 2
```

The `take!` method behaves exactly like `take`, except that it will raise `ActiveRecord::RecordNotFound` if no matching record is found.

The retrieved record may vary depending on the database engine.

1.1.3 `first` The `first` method finds the first record ordered by the primary key. For example:

```
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 1
```

The `first` method returns `nil` if no matching record is found and no exception will be raised.

You can pass in a numerical argument to the `first` method to return up to that number of results. For example

```
client = Client.first(3)
# => [
  #<Client id: 1, first_name: "Lifo">,
  #<Client id: 2, first_name: "Fifo">,
  #<Client id: 3, first_name: "Filo">
]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 3
```

The `first!` method behaves exactly like `first`, except that it will raise `ActiveRecord::RecordNotFound` if no matching record is found.

1.1.4 `last` The `last` method finds the last record ordered by the primary key. For example:

```
client = Client.last
# => #<Client id: 221, first_name: "Russel">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

The `last` method returns `nil` if no matching record is found and no exception will be raised.

You can pass in a numerical argument to the `last` method to return up to that number of results. For example

```
client = Client.last(3)
# => [
  #<Client id: 219, first_name: "James">,
  #<Client id: 220, first_name: "Sara">,
  #<Client id: 221, first_name: "Russel">
]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 3
```

The `last!` method behaves exactly like `last`, except that it will raise `ActiveRecord::RecordNotFound` if no matching record is found.

1.1.5 find_by

The `find_by` method finds the first record matching some conditions. For example:

```
Client.find_by first_name: 'Lifo'
# => #<Client id: 1, first_name: "Lifo">

Client.find_by first_name: 'Jon'
# => nil
```

It is equivalent to writing:

```
Client.where(first_name: 'Lifo').take
```

The `find_by!` method behaves exactly like `find_by`, except that it will raise `ActiveRecord::RecordNotFound` if no matching record is found. For example:

```
Client.find_by! first_name: 'does not exist'
# => ActiveRecord::RecordNotFound
```

This is equivalent to writing:

```
Client.where(first_name: 'does not exist').take!
```

1.2 Retrieving Multiple Objects in Batches

We often need to iterate over a large set of records, as when we send a newsletter to a large set of users, or when we export data.

This may appear straightforward:

```
# This is very inefficient when the users table has thousands of rows.
User.all.each do |user|
  NewsMailer.weekly(user).deliver_now
end
```

But this approach becomes increasingly impractical as the table size increases, since `User.all.each` instructs Active Record to fetch *the entire table* in a single pass, build a model object per row, and then keep the entire array of model objects in memory. Indeed, if we have a large number of records, the entire collection may exceed the amount of memory available.

Rails provides two methods that address this problem by dividing records into memory-friendly batches for processing. The first method, `find_each`, retrieves a batch of records and then yields *each* record to the block individually as a model. The second method, `find_in_batches`, retrieves a batch of records and then yields *the entire batch* to the block as an array of models.

The `find_each` and `find_in_batches` methods are intended for use in the batch processing of a large number of records that wouldn't fit in memory all at once. If you just need to loop over a thousand records the regular find methods are the preferred option.

1.2.1 `find_each` The `find_each` method retrieves a batch of records and then yields *each* record to the block individually as a model. In the following example, `find_each` will retrieve 1000 records (the current default for both `find_each` and `find_in_batches`) and then yield each record individually to the block as a model. This process is repeated until all of the records have been processed:

```
User.find_each do |user|
  NewsMailer.weekly(user).deliver_now
end
```

To add conditions to a `find_each` operation you can chain other Active Record methods such as `where`:

```
User.where(weekly_subscriber: true).find_each do |user|
  NewsMailer.weekly(user).deliver_now
end
```

1.2.1.1 Options for `find_each`

The `find_each` method accepts most of the options allowed by the regular `find` method, except for `:order` and `:limit`, which are reserved for internal use by `find_each`.

Two additional options, `:batch_size` and `:start`, are available as well.

`:batch_size`

The `:batch_size` option allows you to specify the number of records to be retrieved in each batch, before being passed individually to the block. For example, to retrieve records in batches of 5000:

```
User.find_each(batch_size: 5000) do |user|
  NewsMailer.weekly(user).deliver_now
end
```

`:start`

By default, records are fetched in ascending order of the primary key, which must be an integer. The `:start` option allows you to configure the first ID of the sequence whenever the lowest ID is not the one you need. This would be useful, for example, if you wanted to resume an interrupted batch process, provided you saved the last processed ID as a checkpoint.

For example, to send newsletters only to users with the primary key starting from 2000, and to retrieve them in batches of 5000:

```
User.find_each(start: 2000, batch_size: 5000) do |user|
  NewsMailer.weekly(user).deliver_now
end
```

Another example would be if you wanted multiple workers handling the same processing queue. You could have each worker handle 10000 records by setting the appropriate `:start` option on each worker.

1.2.2 `find_in_batches` The `find_in_batches` method is similar to `find_each`, since both retrieve batches of records. The difference is that `find_in_batches` yields *batches* to the block as an array of models, instead of individually. The following example will yield to the supplied block an array of up to 1000 invoices at a time, with the final block containing any remaining invoices:

```
# Give add_invoices an array of 1000 invoices at a time
Invoice.find_in_batches do |invoices|
  export.add_invoices(invoices)
end
```

1.2.2.1 Options for `find_in_batches`

The `find_in_batches` method accepts the same `:batch_size` and `:start` options as `find_each`.

2 Conditions

The `where` method allows you to specify conditions to limit the records returned, representing the `WHERE`-part of the SQL statement. Conditions can either be specified as a string, array, or hash.

2.1 Pure String Conditions

If you'd like to add conditions to your find, you could just specify them in there, just like `Client.where("orders_count = '2'")`. This will find all clients where the `orders_count` field's value is 2.

Building your own conditions as pure strings can leave you vulnerable to SQL injection exploits. For example, `Client.where("first_name LIKE '%#{params[:first_name]}%'")` is not safe. See the next section for the preferred way to handle conditions using an array.

2.2 Array Conditions

Now what if that number could vary, say as an argument from somewhere? The find would then take the form:

```
Client.where("orders_count = ?", params[:orders])
```

Active Record will go through the first element in the conditions value and any additional elements will replace the question marks `(?)` in the first element.

If you want to specify multiple conditions:

```
Client.where("orders_count = ? AND locked = ?", params[:orders], false)
```

In this example, the first question mark will be replaced with the value in `params[:orders]` and the second will be replaced with the SQL representation of `false`, which depends on the adapter.

This code is highly preferable:

```
Client.where("orders_count = ?", params[:orders])
```

to this code:

```
Client.where("orders_count = #{params[:orders]}")
```

because of argument safety. Putting the variable directly into the conditions string will pass the variable to the database **as-is**. This means that it will be an unescaped variable directly from a user who may have malicious intent. If you do this, you put your entire database at risk because once a user finds out they can exploit your database they can do just about anything to it. Never ever put your arguments directly inside the conditions string.

For more information on the dangers of SQL injection, see the Ruby on Rails Security Guide.

2.2.1 Placeholder Conditions Similar to the `(?)` replacement style of params, you can also specify keys/values hash in your array conditions:

```
Client.where("created_at >= :start_date AND created_at <= :end_date",
  {start_date: params[:start_date], end_date: params[:end_date]})
```

This makes for clearer readability if you have a large number of variable conditions.

2.3 Hash Conditions

Active Record also allows you to pass in hash conditions which can increase the readability of your conditions syntax. With hash conditions, you pass in a hash with keys of the fields you want conditionalised and the values of how you want to conditionalise them:

Only equality, range and subset checking are possible with Hash conditions.

2.3.1 Equality Conditions

```
Client.where(locked: true)
```

The field name can also be a string:

```
Client.where('locked' => true)
```

In the case of a belongs_to relationship, an association key can be used to specify the model if an Active Record object is used as the value. This method works with polymorphic relationships as well.

```
Article.where(author: author)
Author.joins(:articles).where(articles: { author: author })
```

The values cannot be symbols. For example, you cannot do `Client.where(status:  :active)`.

2.3.2 Range Conditions

```
Client.where(created_at: (Time.now.midnight - 1.day)..Time.now.midnight)
```

This will find all clients created yesterday by using a `BETWEEN` SQL statement:

```
SELECT * FROM clients WHERE (clients.created_at BETWEEN '2008-12-21 00:00:00' AND
'2008-12-22 00:00:00')
```

This demonstrates a shorter syntax for the examples in Array Conditions

2.3.3 Subset Conditions If you want to find records using the `IN` expression you can pass an array to the conditions hash:

```
Client.where(orders_count: [1,3,5])
```

This code will generate SQL like this:

```
SELECT * FROM clients WHERE (clients.orders_count IN (1,3,5))
```

2.4 NOT Conditions

`NOT` SQL queries can be built by `where.not`.

```
Article.where.not(author: author)
```

In other words, this query can be generated by calling `where` with no argument, then immediately chain with `not` passing `where` conditions.

3 Ordering

To retrieve records from the database in a specific order, you can use the `order` method.

For example, if you're getting a set of records and want to order them in ascending order by the `created_at` field in your table:

```
Client.order(:created_at)
# OR
Client.order("created_at")
```

You could specify `ASC` or `DESC` as well:

```
Client.order(created_at: :desc)
# OR
Client.order(created_at: :asc)
# OR
Client.order("created_at DESC")
# OR
Client.order("created_at ASC")
```

Or ordering by multiple fields:

```
Client.order(orders_count: :asc, created_at: :desc)
# OR
Client.order(:orders_count, created_at: :desc)
# OR
Client.order("orders_count ASC, created_at DESC")
# OR
Client.order("orders_count ASC", "created_at DESC")
```

If you want to call `order` multiple times e.g. in different context, new order will append previous one

```
Client.order("orders_count ASC").order("created_at DESC")
# SELECT * FROM clients ORDER BY orders_count ASC, created_at DESC
```

4 Selecting Specific Fields

By default, `Model.find` selects all the fields from the result set using `select *`.

To select only a subset of fields from the result set, you can specify the subset via the `select` method.

For example, to select only `viewable_by` and `locked` columns:

```
Client.select("viewable_by, locked")
```

The SQL query used by this find call will be somewhat like:

```
SELECT viewable_by, locked FROM clients
```

Be careful because this also means you're initializing a model object with only the fields that you've selected. If you attempt to access a field that is not in the initialized record you'll receive:

```
ActiveModel::MissingAttributeError: missing attribute: <attribute>
```

Where `<attribute>` is the attribute you asked for. The `id` method will not raise the `ActiveRecord ::MissingAttributeError`, so just be careful when working with associations because they need the `id` method to function properly.

If you would like to only grab a single record per unique value in a certain field, you can use `distinct`:

```
Client.select(:name).distinct
```

This would generate SQL like:

```
SELECT DISTINCT name FROM clients
```

You can also remove the uniqueness constraint:

```
query = Client.select(:name).distinct
# => Returns unique names

query.distinct(false)
# => Returns all names, even if there are duplicates
```

5 Limit and Offset

To apply `LIMIT` to the SQL fired by the `Model.find`, you can specify the `LIMIT` using `limit` and `offset` methods on the relation.

You can use `limit` to specify the number of records to be retrieved, and use `offset` to specify the number of records to skip before starting to return the records. For example

```
Client.limit(5)
```

will return a maximum of 5 clients and because it specifies no offset it will return the first 5 in the table. The SQL it executes looks like this:

```
SELECT * FROM clients LIMIT 5
```

Adding `offset` to that

```
Client.limit(5).offset(30)
```

will return instead a maximum of 5 clients beginning with the 31st. The SQL looks like:

```
SELECT * FROM clients LIMIT 5 OFFSET 30
```

6 Group

To apply a `GROUP BY` clause to the SQL fired by the finder, you can specify the `group` method on the find.

For example, if you want to find a collection of the dates orders were created on:

```
Order.select("date(created_at) as ordered_date, sum(price) as
total_price").group("date(created_at)")
```

And this will give you a single `Order` object for each date where there are orders in the database.

The SQL that would be executed would be something like this:

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders
GROUP BY date(created_at)
```

6.1 Total of grouped items

To get the total of grouped items on a single query call `count` after the `group`.

```
Order.group(:status).count
# => { 'awaiting_approval' => 7, 'paid' => 12 }
```

The SQL that would be executed would be something like this:

```
SELECT COUNT (*) AS count_all, status AS status
FROM "orders"
GROUP BY status
```

7 Having

SQL uses the `HAVING` clause to specify conditions on the `GROUP BY` fields. You can add the `HAVING` clause to the SQL fired by the `Model.find` by adding the `:having` option to the find.

For example:

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").
  group("date(created_at)").having("sum(price) > ?", 100)
```

The SQL that would be executed would be something like this:

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders
GROUP BY date(created_at)
HAVING sum(price) > 100
```

This will return single order objects for each day, but only those that are ordered more than $100 in a day.

8 Overriding Conditions

8.1 unscope

You can specify certain conditions to be removed using the `unscope` method. For example:

```
Article.where('id > 10').limit(20).order('id asc').unscope(:order)
```

The SQL that would be executed:

```
SELECT * FROM articles WHERE id > 10 LIMIT 20

# Original query without `unscope`
SELECT * FROM articles WHERE id > 10 ORDER BY id asc LIMIT 20
```

You can also unscope specific `where` clauses. For example:

```
Article.where(id: 10, trashed: false).unscope(where: :id)
# SELECT "articles".* FROM "articles" WHERE trashed = 0
```

A relation which has used `unscope` will affect any relation it is merged in to:

```
Article.order('id asc').merge(Article.unscope(:order))
# SELECT "articles".* FROM "articles"
```

8.2 only

You can also override conditions using the `only` method. For example:

```
Article.where('id > 10').limit(20).order('id desc').only(:order, :where)
```

The SQL that would be executed:

```
SELECT * FROM articles WHERE id > 10 ORDER BY id DESC

# Original query without `only`
SELECT "articles".* FROM "articles" WHERE (id > 10) ORDER BY id desc LIMIT 20
```

8.3 reorder

The `reorder` method overrides the default scope order. For example:

```
class Article < ActiveRecord::Base
  has_many :comments, -> { order('posted_at DESC') }
end

Article.find(10).comments.reorder('name')
```

The SQL that would be executed:

```
SELECT * FROM articles WHERE id = 10
SELECT * FROM comments WHERE article_id = 10 ORDER BY name
```

In case the `reorder` clause is not used, the SQL executed would be:

```
SELECT * FROM articles WHERE id = 10
SELECT * FROM comments WHERE article_id = 10 ORDER BY posted_at DESC
```

8.4 reverse_order

The `reverse_order` method reverses the ordering clause if specified.

```
Client.where("orders_count > 10").order(:name).reverse_order
```

The SQL that would be executed:

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY name DESC
```

If no ordering clause is specified in the query, the `reverse_order` orders by the primary key in reverse order.

```
Client.where("orders_count > 10").reverse_order
```

The SQL that would be executed:

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY clients.id DESC
```

This method accepts **no** arguments.

8.5 rewhere

The `rewhere` method overrides an existing, named where condition. For example:

```
Article.where(trashed: true).rewhere(trashed: false)
```

The SQL that would be executed:

```
SELECT * FROM articles WHERE `trashed` = 0
```

In case the `rewhere` clause is not used,

```
Article.where(trashed: true).where(trashed: false)
```

the SQL executed would be:

```
SELECT * FROM articles WHERE `trashed` = 1 AND `trashed` = 0
```

9 Null Relation

The `none` method returns a chainable relation with no records. Any subsequent conditions chained to the returned relation will continue generating empty relations. This is useful in scenarios where you need a chainable response to a method or a scope that could return zero results.

```
Article.none # returns an empty Relation and fires no queries.

# The visible_articles method below is expected to return a Relation.
@articles = current_user.visible_articles.where(name: params[:name])

def visible_articles
  case role
  when 'Country Manager'
    Article.where(country: country)
  when 'Reviewer'
    Article.published
  when 'Bad User'
    Article.none # => returning [] or nil breaks the caller code in this case
  end
end
```

10 Readonly Objects

Active Record provides `readonly` method on a relation to explicitly disallow modification of any of the returned objects. Any attempt to alter a readonly record will not succeed, raising an `ActiveRecord::ReadOnlyRecord` exception.

```
client = Client.readonly.first
client.visits += 1
client.save
```

As `client` is explicitly set to be a readonly object, the above code will raise an `ActiveRecord::ReadOnlyRecord` exception when calling `client.save` with an updated value of *visits*.

11 Locking Records for Update

Locking is helpful for preventing race conditions when updating records in the database and ensuring atomic updates.

Active Record provides two locking mechanisms:

- Optimistic Locking
- Pessimistic Locking

11.1 Optimistic Locking

Optimistic locking allows multiple users to access the same record for edits, and assumes a minimum of conflicts with the data. It does this by checking whether another process has made changes to a record since it was opened. An `ActiveRecord::StaleObjectError` exception is thrown if that has occurred and the update is ignored.

Optimistic locking column

In order to use optimistic locking, the table needs to have a column called `lock_version` of type integer. Each time the record is updated, Active Record increments the `lock_version` column. If an update request is made with a lower value in the `lock_version` field than is currently in the `lock_version` column in the database, the update request will fail with an `ActiveRecord::StaleObjectError`. Example:

```
c1 = Client.find(1)
c2 = Client.find(1)

c1.first_name = "Michael"
c1.save

c2.name = "should fail"
c2.save # Raises an ActiveRecord::StaleObjectError
```

You're then responsible for dealing with the conflict by rescuing the exception and either rolling back, merging, or otherwise apply the business logic needed to resolve the conflict.

This behavior can be turned off by setting `ActiveRecord::Base.lock_optimistically = false`.

To override the name of the `lock_version` column, `ActiveRecord::Base` provides a class attribute called `locking_column`:

```
class Client < ActiveRecord::Base
  self.locking_column = :lock_client_column
end
```

11.2 Pessimistic Locking

Pessimistic locking uses a locking mechanism provided by the underlying database. Using `lock` when building a relation obtains an exclusive lock on the selected rows. Relations using `lock` are usually wrapped inside a transaction for preventing deadlock conditions.

For example:

```
Item.transaction do
  i = Item.lock.first
  i.name = 'Jones'
  i.save
end
```

The above session produces the following SQL for a MySQL backend:

```
SQL (0.2ms)   BEGIN
Item Load (0.3ms)   SELECT * FROM ‘items‘ LIMIT 1 FOR UPDATE
Item Update (0.4ms)   UPDATE ‘items‘ SET ‘updated_at‘ = ’2009-02-07 18:05:56’, ‘name‘ =
’Jones’ WHERE ‘id‘ = 1
SQL (0.8ms)   COMMIT
```

You can also pass raw SQL to the `lock` method for allowing different types of locks. For example, MySQL has an expression called `LOCK IN SHARE MODE` where you can lock a record but still allow other queries to read it. To specify this expression just pass it in as the lock option:

```
Item.transaction do
  i = Item.lock("LOCK IN SHARE MODE").find(1)
  i.increment!(:views)
end
```

If you already have an instance of your model, you can start a transaction and acquire the lock in one go using the following code:

```
item = Item.first
item.with_lock do
  # This block is called within a transaction,
  # item is already locked.
  item.increment!(:views)
end
```

12 Joining Tables

Active Record provides a finder method called `joins` for specifying `JOIN` clauses on the resulting SQL. There are multiple ways to use the `joins` method.

12.1 Using a String SQL Fragment

You can just supply the raw SQL specifying the `JOIN` clause to `joins`:

```
Client.joins(’LEFT OUTER JOIN addresses ON addresses.client_id = clients.id’)
```

This will result in the following SQL:

```
SELECT clients.* FROM clients LEFT OUTER JOIN addresses ON addresses.client_id =
clients.id
```

12.2 Using Array/Hash of Named Associations

This method only works with `INNER JOIN`.

Active Record lets you use the names of the associations defined on the model as a shortcut for specifying `JOIN` clauses for those associations when using the `joins` method.

For example, consider the following `Category`, `Article`, `Comment`, `Guest` and `Tag` models:

```
class Category < ActiveRecord::Base
  has_many :articles
end

class Article < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end

class Comment < ActiveRecord::Base
  belongs_to :article
  has_one :guest
end

class Guest < ActiveRecord::Base
  belongs_to :comment
end

class Tag < ActiveRecord::Base
  belongs_to :article
end
```

Now all of the following will produce the expected join queries using `INNER JOIN`:

12.2.1 Joining a Single Association

```
Category.joins(:articles)
```

This produces:

```
SELECT categories.* FROM categories
  INNER JOIN articles ON articles.category_id = categories.id
```

Or, in English: "return a Category object for all categories with articles". Note that you will see duplicate categories if more than one article has the same category. If you want unique categories, you can use `Category .joins(:articles).uniq`.

12.2.2 Joining Multiple Associations

```
Article.joins(:category, :comments)
```

This produces:

```
SELECT articles.* FROM articles
  INNER JOIN categories ON articles.category_id = categories.id
  INNER JOIN comments ON comments.article_id = articles.id
```

Or, in English: "return all articles that have a category and at least one comment". Note again that articles with multiple comments will show up multiple times.

12.2.3 Joining Nested Associations (Single Level)

```
Article.joins(comments: :guest)
```

This produces:

```
SELECT articles.* FROM articles
  INNER JOIN comments ON comments.article_id = articles.id
  INNER JOIN guests ON guests.comment_id = comments.id
```

Or, in English: "return all articles that have a comment made by a guest."

12.2.4 Joining Nested Associations (Multiple Level)

```
Category.joins(articles: [{ comments: :guest }, :tags])
```

This produces:

```
SELECT categories.* FROM categories
  INNER JOIN articles ON articles.category_id = categories.id
  INNER JOIN comments ON comments.article_id = articles.id
  INNER JOIN guests ON guests.comment_id = comments.id
  INNER JOIN tags ON tags.article_id = articles.id
```

12.3 Specifying Conditions on the Joined Tables

You can specify conditions on the joined tables using the regular Array and String conditions. Hash conditions provides a special syntax for specifying conditions for the joined tables:

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where('orders.created_at' => time_range)
```

An alternative and cleaner syntax is to nest the hash conditions:

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where(orders: { created_at: time_range })
```

This will find all clients who have orders that were created yesterday, again using a `BETWEEN` SQL expression.

13 Eager Loading Associations

Eager loading is the mechanism for loading the associated records of the objects returned by `Model.find` using as few queries as possible.

N + 1 queries problem

Consider the following code, which finds 10 clients and prints their postcodes:

```
clients = Client.limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

This code looks fine at the first sight. But the problem lies within the total number of queries executed. The above code executes 1 (to find 10 clients) + 10 (one per each client to load the address) = **11** queries in total.

Solution to N + 1 queries problem

Active Record lets you specify in advance all the associations that are going to be loaded. This is possible by specifying the `includes` method of the `Model.find` call. With `includes`, Active Record ensures that all of the specified associations are loaded using the minimum possible number of queries.

Revisiting the above case, we could rewrite `Client.limit(10)` to use eager load addresses:

```
clients = Client.includes(:address).limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

The above code will execute just **2** queries, as opposed to **11** queries in the previous case:

```
SELECT * FROM clients LIMIT 10
SELECT addresses.* FROM addresses
  WHERE (addresses.client_id IN (1,2,3,4,5,6,7,8,9,10))
```

13.1 Eager Loading Multiple Associations

Active Record lets you eager load any number of associations with a single `Model.find` call by using an array, hash, or a nested hash of array/hash with the `includes` method.

13.1.1 Array of Multiple Associations

```
Article.includes(:category, :comments)
```

This loads all the articles and the associated category and comments for each article.

13.1.2 Nested Associations Hash

```
Category.includes(articles: [{ comments: :guest }, :tags]).find(1)
```

This will find the category with id 1 and eager load all of the associated articles, the associated articles' tags and comments, and every comment's guest association.

13.2 Specifying Conditions on Eager Loaded Associations

Even though Active Record lets you specify conditions on the eager loaded associations just like `joins`, the recommended way is to use joins instead.

However if you must do this, you may use `where` as you would normally.

```
Article.includes(:comments).where(comments: { visible: true })
```

This would generate a query which contains a `LEFT OUTER JOIN` whereas the `joins` method would generate one using the `INNER JOIN` function instead.

```
  SELECT "articles"."id" AS t0_r0, ... "comments"."updated_at" AS t1_r5 FROM "articles"
LEFT OUTER JOIN "comments" ON "comments"."article_id" = "articles"."id" WHERE
(comments.visible = 1)
```

If there was no `where` condition, this would generate the normal set of two queries.

Using `where` like this will only work when you pass it a Hash. For SQL-fragments you need use `references` to force joined tables:

```
Article.includes(:comments).where("comments.visible = true").references(:comments)
```

If, in the case of this `includes` query, there were no comments for any articles, all the articles would still be loaded. By using `joins` (an INNER JOIN), the join conditions **must** match, otherwise no records will be returned.

14 Scopes

Scoping allows you to specify commonly-used queries which can be referenced as method calls on the association objects or models. With these scopes, you can use every method previously covered such as `where`, `joins` and `includes`. All scope methods will return an `ActiveRecord::Relation` object which will allow for further methods (such as other scopes) to be called on it.

To define a simple scope, we use the `scope` method inside the class, passing the query that we'd like to run when this scope is called:

```
class Article < ActiveRecord::Base
  scope :published, -> { where(published: true) }
end
```

This is exactly the same as defining a class method, and which you use is a matter of personal preference:

```
class Article < ActiveRecord::Base
  def self.published
    where(published: true)
  end
end
```

Scopes are also chainable within scopes:

```
class Article < ActiveRecord::Base
  scope :published,               -> { where(published: true) }
  scope :published_and_commented, -> { published.where("comments_count > 0") }
end
```

To call this `published` scope we can call it on either the class:

```
Article.published # => [published articles]
```

Or on an association consisting of `Article` objects:

```
category = Category.first
category.articles.published # => [published articles belonging to this category]
```

14.1 Passing in arguments

Your scope can take arguments:

```
class Article < ActiveRecord::Base
  scope :created_before, ->(time) { where("created_at < ?", time) }
end
```

Call the scope as if it were a class method:

```
Article.created_before(Time.zone.now)
```

However, this is just duplicating the functionality that would be provided to you by a class method.

```
class Article < ActiveRecord::Base
  def self.created_before(time)
    where("created_at < ?", time)
  end
end
```

Using a class method is the preferred way to accept arguments for scopes. These methods will still be accessible on the association objects:

```
category.articles.created_before(time)
```

14.2 Applying a default scope

If we wish for a scope to be applied across all queries to the model we can use the `default_scope` method within the model itself.

```
class Client < ActiveRecord::Base
  default_scope { where("removed_at IS NULL") }
end
```

When queries are executed on this model, the SQL query will now look something like this:

```
SELECT * FROM clients WHERE removed_at IS NULL
```

If you need to do more complex things with a default scope, you can alternatively define it as a class method:

```
class Client < ActiveRecord::Base
  def self.default_scope
    # Should return an ActiveRecord::Relation.
  end
end
```

14.3 Merging of scopes

Just like `where` clauses scopes are merged using `AND` conditions.

```
class User < ActiveRecord::Base
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.active.inactive
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND "users"."state" =
'inactive'
```

We can mix and match `scope` and `where` conditions and the final sql will have all conditions joined with `AND`.

```
User.active.where(state: 'finished')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND "users"."state" =
'finished'
```

If we do want the `last where clause` to win then `Relation#merge` can be used.

```
User.active.merge(User.inactive)
# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'
```

One important caveat is that `default_scope` will be prepended in `scope` and `where` conditions.

```
class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
```

```
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" =
'active'

User.where(state: 'inactive')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" =
'inactive'
```

As you can see above the `default_scope` is being merged in both `scope` and `where` conditions.

14.4 Removing All Scoping

If we wish to remove scoping for any reason we can use the `unscoped` method. This is especially useful if a `default_scope` is specified in the model and should not be applied for this particular query.

```
Client.unscoped.load
```

This method removes all scoping and will do a normal query on the table.

Note that chaining `unscoped` with a `scope` does not work. In these cases, it is recommended that you use the block form of `unscoped`:

```
Client.unscoped {
  Client.created_before(Time.zone.now)
}
```

15 Dynamic Finders

For every field (also known as an attribute) you define in your table, Active Record provides a finder method. If you have a field called `first_name` on your `Client` model for example, you get `find_by_first_name` for free from Active Record. If you have a `locked` field on the `Client` model, you also get `find_by_locked` and methods.

You can specify an exclamation point (`!`) on the end of the dynamic finders to get them to raise an `ActiveRecord::RecordNotFound` error if they do not return any records, like `Client.find_by_name!("Ryan")`

If you want to find both by name and locked, you can chain these finders together by simply typing "`and`" between the fields. For example, `Client.find_by_first_name_and_locked("Ryan", true)`.

16 Find or Build a New Object

Some dynamic finders have been deprecated in Rails 4.0 and will be removed in Rails 4.1. The best practice is to use Active Record scopes instead. You can find the deprecation gem at https://github.com/rails/activerecord-deprecated_finders

It's common that you need to find a record or create it if it doesn't exist. You can do that with the `find_or_create_by` and `find_or_create_by!` methods.

16.1 find_or_create_by

The find_or_create_by method checks whether a record with the attributes exists. If it doesn't, then create is called. Let's see an example.

Suppose you want to find a client named 'Andy', and if there's none, create one. You can do so by running:

```
Client.find_or_create_by(first_name: 'Andy')
# => #<Client id: 1, first_name: "Andy", orders_count: 0, locked: true, created_at:
"2011-08-30 06:09:27", updated_at: "2011-08-30 06:09:27">
```

The SQL generated by this method looks like this:

```
SELECT * FROM clients WHERE (clients.first_name = 'Andy') LIMIT 1
BEGIN
INSERT INTO clients (created_at, first_name, locked, orders_count, updated_at) VALUES
('2011-08-30 05:22:57', 'Andy', 1, NULL, '2011-08-30 05:22:57')
COMMIT
```

find_or_create_by returns either the record that already exists or the new record. In our case, we didn't already have a client named Andy so the record is created and returned.

The new record might not be saved to the database; that depends on whether validations passed or not (just like create).

Suppose we want to set the 'locked' attribute to false if we're creating a new record, but we don't want to include it in the query. So we want to find the client named "Andy", or if that client doesn't exist, create a client named "Andy" which is not locked.

We can achieve this in two ways. The first is to use create_with:

```
Client.create_with(locked: false).find_or_create_by(first_name: 'Andy')
```

The second way is using a block:

```
Client.find_or_create_by(first_name: 'Andy') do |c|
  c.locked = false
end
```

The block will only be executed if the client is being created. The second time we run this code, the block will be ignored.

16.2 find_or_create_by!

You can also use find_or_create_by! to raise an exception if the new record is invalid. Validations are not covered on this guide, but let's assume for a moment that you temporarily add

```
validates :orders_count, presence: true
```

to your Client model. If you try to create a new Client without passing an orders_count, the record will be invalid and an exception will be raised:

```
Client.find_or_create_by!(first_name: 'Andy')
# => ActiveRecord::RecordInvalid: Validation failed: Orders count can't be blank
```

16.3 find_or_initialize_by

The find_or_initialize_by method will work just like find_or_create_by but it will call new instead of create. This means that a new model instance will be created in memory but won't be saved to the database. Continuing with the find_or_create_by example, we now want the client named 'Nick':

```
nick = Client.find_or_initialize_by(first_name: 'Nick')
# => <Client id: nil, first_name: "Nick", orders_count: 0, locked: true, created_at:
"2011-08-30 06:09:27", updated_at: "2011-08-30 06:09:27">

nick.persisted?
# => false

nick.new_record?
# => true
```

Because the object is not yet stored in the database, the SQL generated looks like this:

```
SELECT * FROM clients WHERE (clients.first_name = 'Nick') LIMIT 1
```

When you want to save it to the database, just call save:

```
nick.save
# => true
```

17 Finding by SQL

If you'd like to use your own SQL to find records in a table you can use find_by_sql. The find_by_sql method will return an array of objects even if the underlying query returns just a single record. For example you could run this query:

```
Client.find_by_sql("SELECT * FROM clients
  INNER JOIN orders ON clients.id = orders.client_id
  ORDER BY clients.created_at desc")
# =>  [
  #<Client id: 1, first_name: "Lucas" >,
  #<Client id: 2, first_name: "Jan" >,
  # ...
]
```

find_by_sql provides you with a simple way of making custom calls to the database and retrieving instantiated objects.

17.1 select_all

find_by_sql has a close relative called connection#select_all. select_all will retrieve objects from the database using custom SQL just like find_by_sql but will not instantiate them. Instead, you will get an array of hashes where each hash indicates a record.

```
Client.connection.select_all("SELECT first_name, created_at FROM clients WHERE id = '1'")
# => [
  {"first_name"=>"Rafael", "created_at"=>"2012-11-10 23:23:45.281189"},
  {"first_name"=>"Eileen", "created_at"=>"2013-12-09 11:22:35.221282"}
]
```

17.2 pluck

`pluck` can be used to query single or multiple columns from the underlying table of a model. It accepts a list of column names as argument and returns an array of values of the specified columns with the corresponding data type.

```
Client.where(active: true).pluck(:id)
# SELECT id FROM clients WHERE active = 1
# => [1, 2, 3]

Client.distinct.pluck(:role)
# SELECT DISTINCT role FROM clients
# => ['admin', 'member', 'guest']

Client.pluck(:id, :name)
# SELECT clients.id, clients.name FROM clients
# => [[1, 'David'], [2, 'Jeremy'], [3, 'Jose']]
```

`pluck` makes it possible to replace code like:

```
Client.select(:id).map { |c| c.id }
# or
Client.select(:id).map(&:id)
# or
Client.select(:id, :name).map { |c| [c.id, c.name] }
```

with:

```
Client.pluck(:id)
# or
Client.pluck(:id, :name)
```

Unlike `select`, `pluck` directly converts a database result into a Ruby `Array`, without constructing `ActiveRecord` objects. This can mean better performance for a large or often-running query. However, any model method overrides will not be available. For example:

```
class Client < ActiveRecord::Base
  def name
    "I am #{super}"
  end
end
```

```
Client.select(:name).map &:name
# => ["I am David", "I am Jeremy", "I am Jose"]

Client.pluck(:name)
# => ["David", "Jeremy", "Jose"]
```

Furthermore, unlike `select` and other `Relation` scopes, `pluck` triggers an immediate query, and thus cannot be chained with any further scopes, although it can work with scopes already constructed earlier:

```
Client.pluck(:name).limit(1)
# => NoMethodError: undefined method ‘limit’ for #<Array:0x007ff34d3ad6d8>

Client.limit(1).pluck(:name)
# => ["David"]
```

17.3 ids

`ids` can be used to pluck all the IDs for the relation using the table's primary key.

```
Person.ids
# SELECT id FROM people
```

```
class Person < ActiveRecord::Base
  self.primary_key = "person_id"
end

Person.ids
# SELECT person_id FROM people
```

18 Existence of Objects

If you simply want to check for the existence of the object there's a method called `exists?`. This method will query the database using the same query as `find`, but instead of returning an object or collection of objects it will return either `true` or `false`.

```
Client.exists?(1)
```

The `exists?` method also takes multiple values, but the catch is that it will return `true` if any one of those records exists.

```
Client.exists?(id: [1,2,3])
# or
Client.exists?(name: ['John', 'Sergei'])
```

It's even possible to use `exists?` without any arguments on a model or a relation.

```
Client.where(first_name: 'Ryan').exists?
```

The above returns `true` if there is at least one client with the `first_name` 'Ryan' and `false` otherwise.

```
Client.exists?
```

The above returns `false` if the `clients` table is empty and `true` otherwise.
You can also use `any?` and `many?` to check for existence on a model or relation.

```
# via a model
Article.any?
Article.many?

# via a named scope
Article.recent.any?
Article.recent.many?

# via a relation
Article.where(published: true).any?
Article.where(published: true).many?

# via an association
Article.first.categories.any?
Article.first.categories.many?
```

19 Calculations

This section uses count as an example method in this preamble, but the options described apply to all subsections.

All calculation methods work directly on a model:

```
Client.count
# SELECT count(*) AS count_all FROM clients
```

Or on a relation:

```
Client.where(first_name: 'Ryan').count
# SELECT count(*) AS count_all FROM clients WHERE (first_name = 'Ryan')
```

You can also use various finder methods on a relation for performing complex calculations:

```
Client.includes("orders").where(first_name: 'Ryan', orders: { status: 'received' }).count
```

Which will execute:

```
SELECT count(DISTINCT clients.id) AS count_all FROM clients
  LEFT OUTER JOIN orders ON orders.client_id = client.id WHERE
  (clients.first_name = 'Ryan' AND orders.status = 'received')
```

19.1 Count

If you want to see how many records are in your model's table you could call `Client.count` and that will return the number. If you want to be more specific and find all the clients with their age present in the database you can use `Client.count(:age)`.

For options, please see the parent section, Calculations.

19.2 Average

If you want to see the average of a certain number in one of your tables you can call the `average` method on the class that relates to the table. This method call will look something like this:

```
Client.average("orders_count")
```

This will return a number (possibly a floating point number such as 3.14159265) representing the average value in the field.

For options, please see the parent section, Calculations.

19.3 Minimum

If you want to find the minimum value of a field in your table you can call the `minimum` method on the class that relates to the table. This method call will look something like this:

```
Client.minimum("age")
```

For options, please see the parent section, Calculations.

19.4 Maximum

If you want to find the maximum value of a field in your table you can call the `maximum` method on the class that relates to the table. This method call will look something like this:

```
Client.maximum("age")
```

For options, please see the parent section, Calculations.

19.5 Sum

If you want to find the sum of a field for all records in your table you can call the `sum` method on the class that relates to the table. This method call will look something like this:

```
Client.sum("orders_count")
```

For options, please see the parent section, Calculations.

20 Running EXPLAIN

You can run EXPLAIN on the queries triggered by relations. For example,

```
User.where(id: 1).joins(:articles).explain
```

may yield

```
EXPLAIN for: SELECT `users`.* FROM `users` INNER JOIN `articles` ON `articles`.`user_id` =
`users`.`id` WHERE `users`.`id` = 1
+----+-------------+----------+-------+---------------+
| id | select_type | table    | type  | possible_keys |
+----+-------------+----------+-------+---------------+
|  1 | SIMPLE      | users    | const | PRIMARY       |
|  1 | SIMPLE      | articles | ALL   | NULL          |
+----+-------------+----------+-------+---------------+
+---------+---------+-------+------+-------------+
| key     | key_len | ref   | rows | Extra       |
+---------+---------+-------+------+-------------+
| PRIMARY | 4       | const |    1 |             |
| NULL    | NULL    | NULL  |    1 | Using where |
+---------+---------+-------+------+-------------+

2 rows in set (0.00 sec)
```

under MySQL.

Active Record performs a pretty printing that emulates the one of the database shells. So, the same query running with the PostgreSQL adapter would yield instead

```
EXPLAIN for: SELECT "users".* FROM "users" INNER JOIN "articles" ON "articles"."user_id" =
"users"."id" WHERE "users"."id" = 1
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..37.24 rows=8 width=0)
   Join Filter: (articles.user_id = users.id)
   ->  Index Scan using users_pkey on users  (cost=0.00..8.27 rows=1 width=4)
         Index Cond: (id = 1)
   ->  Seq Scan on articles  (cost=0.00..28.88 rows=8 width=4)
         Filter: (articles.user_id = 1)
(6 rows)
```

Eager loading may trigger more than one query under the hood, and some queries may need the results of previous ones. Because of that, `explain` actually executes the query, and then asks for the query plans. For example,

```
User.where(id: 1).includes(:articles).explain
```

yields

```
EXPLAIN for: SELECT `users`.* FROM `users`  WHERE `users`.`id` = 1
+----+-------------+-------+-------+---------------+
| id | select_type | table | type  | possible_keys |
+----+-------------+-------+-------+---------------+
|  1 | SIMPLE      | users | const | PRIMARY       |
+----+-------------+-------+-------+---------------+
+---------+---------+-------+------+-------+
| key     | key_len | ref   | rows | Extra |
+---------+---------+-------+------+-------+
| PRIMARY | 4       | const |    1 |       |
+---------+---------+-------+------+-------+

1 row in set (0.00 sec)

EXPLAIN for: SELECT `articles`.* FROM `articles`  WHERE `articles`.`user_id` IN (1)
+----+-------------+----------+------+---------------+
| id | select_type | table    | type | possible_keys |
+----+-------------+----------+------+---------------+
|  1 | SIMPLE      | articles | ALL  | NULL          |
+----+-------------+----------+------+---------------+
+------+---------+------+------+-------------+
| key  | key_len | ref  | rows | Extra       |
+------+---------+------+------+-------------+
| NULL | NULL    | NULL |    1 | Using where |
+------+---------+------+------+-------------+


1 row in set (0.00 sec)
```

under MySQL.

20.1 Interpreting EXPLAIN

Interpretation of the output of EXPLAIN is beyond the scope of this guide. The following pointers may be helpful:

- SQLite3: EXPLAIN QUERY PLAN
- MySQL: EXPLAIN Output Format
- PostgreSQL: Using EXPLAIN

21 Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.
