Si está acostumbrado a usar SQL sin formato para buscar registros en la base de datos, generalmente encontrará que hay mejores formas de llevar a cabo las mismas operaciones en Rails. Active Record lo aísla de la necesidad de usar SQL en la mayoría de los casos.

Los ejemplos de código a lo largo de esta guía se referirán a uno o más de los siguientes modelos:

Todos los siguientes modelos se usan idcomo clave principal, a menos que se especifique lo contrario.

|  |
| --- |
| 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 realizará consultas en la base de datos por usted y es compatible con la mayoría de los sistemas de bases de datos (MySQL, PostgreSQL y SQLite, por nombrar algunos). Independientemente del sistema de base de datos que esté utilizando, el formato del método de registro activo siempre será el mismo.

**1 Recuperando objetos de la base de datos**

Para recuperar objetos de la base de datos, Active Record proporciona varios métodos de búsqueda. Cada método de búsqueda le permite pasar argumentos para realizar ciertas consultas en su base de datos sin escribir SQL sin procesar.

Los métodos son:

* 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

Todos los métodos anteriores devuelven una instancia de ActiveRecord::Relation.

La operación principal de Model.find(options)puede resumirse como:

* Convierta las opciones suministradas en una consulta SQL equivalente.
* Activa la consulta SQL y recupera los resultados correspondientes de la base de datos.
* Crea una instancia del objeto Ruby equivalente del modelo apropiado para cada fila resultante.
* Ejecutar after\_findy luego after\_initializedevoluciones de llamada, en su caso.

**1.1 Recuperando un solo objeto**

Active Record proporciona varias formas diferentes de recuperar un solo objeto.

**1.1.1 find**

Con el findmétodo, puede recuperar el objeto correspondiente a la *clave principal* especificada que coincide con las opciones suministradas. Por ejemplo:

|  |
| --- |
| # Find the client with primary key (id) 10.  client = Client.find(10)  # => #<Client id: 10, first\_name: "Ryan"> |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients WHERE (clients.id = 10) LIMIT 1 |

El findmétodo generará una ActiveRecord::RecordNotFoundexcepción si no se encuentra un registro coincidente.

También puede usar este método para consultar múltiples objetos. Llame al findmétodo y pase una serie de claves principales. La devolución será una matriz que contiene todos los registros coincidentes para las *claves principales* proporcionadas . Por ejemplo:

|  |
| --- |
| # 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">] |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients WHERE (clients.id IN (1,10)) |

El findmétodo generará una ActiveRecord::RecordNotFoundexcepción a menos que se encuentre un registro coincidente para **todas** las claves primarias provistas.

**1.1.2 take**

El takemétodo recupera un registro sin ningún orden implícito. Por ejemplo:

|  |
| --- |
| client = Client.take  # => #<Client id: 1, first\_name: "Lifo"> |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients LIMIT 1 |

El takemétodo regresa nilsi no se encuentra ningún registro y no se generará ninguna excepción.

Puede pasar un argumento numérico al takemétodo para regresar a esa cantidad de resultados. Por ejemplo

|  |
| --- |
| client = Client.take(2)  # => [    #<Client id: 1, first\_name: "Lifo">,    #<Client id: 220, first\_name: "Sara">  ] |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients LIMIT 2 |

El take!método se comporta exactamente igual take, excepto que aumentará ActiveRecord::RecordNotFoundsi no se encuentra ningún registro coincidente.

El registro recuperado puede variar según el motor de la base de datos.

**1.1.3 first**

El firstmétodo encuentra el primer registro ordenado por la clave principal. Por ejemplo:

|  |
| --- |
| client = Client.first  # => #<Client id: 1, first\_name: "Lifo"> |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients ORDER BY clients.id ASC LIMIT 1 |

El firstmétodo se devuelve nilsi no se encuentra ningún registro coincidente y no se generará ninguna excepción.

Puede pasar un argumento numérico al firstmétodo para regresar a esa cantidad de resultados. Por ejemplo

|  |
| --- |
| client = Client.first(3)  # => [    #<Client id: 1, first\_name: "Lifo">,    #<Client id: 2, first\_name: "Fifo">,    #<Client id: 3, first\_name: "Filo">  ] |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients ORDER BY clients.id ASC LIMIT 3 |

El first!método se comporta exactamente igual first, excepto que aumentará ActiveRecord::RecordNotFoundsi no se encuentra ningún registro coincidente.

**1.1.4 last**

El lastmétodo encuentra el último registro ordenado por la clave principal. Por ejemplo:

|  |
| --- |
| client = Client.last  # => #<Client id: 221, first\_name: "Russel"> |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients ORDER BY clients.id DESC LIMIT 1 |

El lastmétodo se devuelve nilsi no se encuentra ningún registro coincidente y no se generará ninguna excepción.

Puede pasar un argumento numérico al lastmétodo para regresar a esa cantidad de resultados. Por ejemplo

|  |
| --- |
| client = Client.last(3)  # => [    #<Client id: 219, first\_name: "James">,    #<Client id: 220, first\_name: "Sara">,    #<Client id: 221, first\_name: "Russel">  ] |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients ORDER BY clients.id DESC LIMIT 3 |

El last!método se comporta exactamente igual last, excepto que aumentará ActiveRecord::RecordNotFoundsi no se encuentra ningún registro coincidente.

**1.1.5 find\_by**

El find\_bymétodo encuentra el primer registro que coincide con algunas condiciones. Por ejemplo:

|  |
| --- |
| Client.find\_by first\_name: 'Lifo'  # => #<Client id: 1, first\_name: "Lifo">    Client.find\_by first\_name: 'Jon'  # => nil |

Es equivalente a escribir:

|  |
| --- |
| Client.where(first\_name: 'Lifo').take |

El equivalente SQL de lo anterior es:

|  |
| --- |
| SELECT \* FROM clients WHERE (clients.first\_name = 'Lifo') LIMIT 1 |

El find\_by!método se comporta exactamente igual find\_by, excepto que aumentará ActiveRecord::RecordNotFoundsi no se encuentra ningún registro coincidente. Por ejemplo:

|  |
| --- |
| Client.find\_by! first\_name: 'does not exist'  # => ActiveRecord::RecordNotFound |

Esto es equivalente a escribir:

|  |
| --- |
| Client.where(first\_name: 'does not exist').take! |

**1.2 Recuperando múltiples objetos en lotes**

A menudo necesitamos repetir un gran conjunto de registros, como cuando enviamos un boletín a un gran grupo de usuarios o cuando exportamos datos.

Esto puede parecer sencillo:

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

Pero este enfoque se vuelve cada vez más impráctico a medida que aumenta el tamaño de la tabla, ya que User.all.eachordena a Active Record obtener *toda la tabla* en una sola pasada, crear un objeto modelo por fila y luego mantener toda la matriz de objetos modelo en la memoria. De hecho, si tenemos una gran cantidad de registros, toda la colección puede exceder la cantidad de memoria disponible.

Rails proporciona dos métodos que abordan este problema dividiendo los registros en lotes fáciles de memorizar para su procesamiento. El primer método, find\_eachrecupera un lote de registros y luego entrega *cada* registro al bloque individualmente como modelo. El segundo método, find\_in\_batchesrecupera un lote de registros y luego cede *todo el lote* al bloque como una matriz de modelos.

Los métodos find\_eachy find\_in\_batchesestán destinados a ser utilizados en el procesamiento por lotes de una gran cantidad de registros que no cabrían en la memoria de una sola vez. Si solo necesita recorrer más de mil registros, los métodos de búsqueda regulares son la opción preferida.

**1.2.1 find\_each**

El find\_eachmétodo recupera un lote de registros y luego entrega *cada* registro al bloque individualmente como modelo. En el siguiente ejemplo, find\_eachrecuperará 1000 registros (el valor predeterminado actual para ambos find\_eachy find\_in\_batches) y luego rendirá cada registro individualmente al bloque como modelo. Este proceso se repite hasta que se hayan procesado todos los registros:

|  |
| --- |
| User.find\_each do |user|    NewsMailer.weekly(user).deliver\_now  end |

Para agregar condiciones a una find\_eachoperación, puede encadenar otros métodos de Active Record como where:

|  |
| --- |
| User.where(weekly\_subscriber: true).find\_each do |user|    NewsMailer.weekly(user).deliver\_now  end |

1.2.1.1 Opciones para find\_each

El find\_eachmétodo acepta la mayoría de las opciones permitidas por el findmétodo regular , excepto para :ordery :limit, que están reservadas para uso interno por find\_each.

Dos opciones adicionales, :batch\_sizey :start, están disponibles también.

**:batch\_size**

La :batch\_sizeopción le permite especificar el número de registros que se recuperarán en cada lote, antes de pasarlos individualmente al bloque. Por ejemplo, para recuperar registros en lotes de 5000:

|  |
| --- |
| User.find\_each(batch\_size: 5000) do |user|    NewsMailer.weekly(user).deliver\_now  end |

**:start**

De forma predeterminada, los registros se obtienen en orden ascendente de la clave principal, que debe ser un número entero. La :startopción le permite configurar la primera ID de la secuencia siempre que la ID más baja no sea la que necesita. Esto sería útil, por ejemplo, si desea reanudar un proceso por lotes interrumpido, siempre que haya guardado el último ID procesado como punto de control.

Por ejemplo, para enviar boletines solo a usuarios con la clave principal a partir de 2000, y para recuperarlos en lotes de 5000:

|  |
| --- |
| User.find\_each(start: 2000, batch\_size: 5000) do |user|    NewsMailer.weekly(user).deliver\_now  end |

Otro ejemplo sería si desea que varios trabajadores manejen la misma cola de procesamiento. Puede hacer que cada trabajador maneje 10000 registros configurando la :startopción adecuada en cada trabajador.

**1.2.2 find\_in\_batches**

El find\_in\_batchesmétodo es similar a find\_each, ya que ambos recuperan lotes de registros. La diferencia es que find\_in\_batchesproduce *lotes* para el bloque como una matriz de modelos, en lugar de individualmente. El siguiente ejemplo arrojará al bloque suministrado una matriz de hasta 1000 facturas a la vez, con el bloque final que contiene las facturas restantes:

|  |
| --- |
| # 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 Opciones para find\_in\_batches

El find\_in\_batchesmétodo acepta el mismo :batch\_sizey :startopciones como find\_each.

**2 Condiciones**

El wheremétodo le permite especificar condiciones para limitar los registros devueltos, que representan la parte WHEREde la declaración SQL. Las condiciones se pueden especificar como una cadena, matriz o hash.

**2.1 Condiciones de cadena pura**

Si desea agregar condiciones a su búsqueda, puede especificarlas allí, al igual que Client.where("orders\_count = '2'"). Esto encontrará todos los clientes donde el orders\_countvalor del campo es 2.

Construir sus propias condiciones como cadenas puras puede hacerlo vulnerable a exploits de inyección de SQL. Por ejemplo, Client.where("first\_name LIKE '%#{params[:first\_name]}%'")no es seguro. Consulte la siguiente sección para conocer la forma preferida de manejar las condiciones usando una matriz.

**2.2 Condiciones de matriz**

Ahora, ¿qué pasaría si ese número pudiera variar, por ejemplo, como un argumento de algún lado? El hallazgo tomaría la forma siguiente:

|  |
| --- |
| Client.where("orders\_count = ?", params[:orders]) |

Active Record pasará por el primer elemento en el valor de las condiciones y cualquier elemento adicional reemplazará los signos de interrogación (?)en el primer elemento.

Si quiere especificar múltiples condiciones:

|  |
| --- |
| Client.where("orders\_count = ? AND locked = ?", params[:orders], false) |

En este ejemplo, el primer signo de interrogación se reemplazará por el valor in params[:orders]y el segundo se reemplazará por la representación SQL de false, que depende del adaptador.

Este código es altamente preferible:

|  |
| --- |
| Client.where("orders\_count = ?", params[:orders]) |

a este código:

|  |
| --- |
| Client.where("orders\_count = #{params[:orders]}") |

debido a la seguridad del argumento Poner la variable directamente en la cadena de condiciones pasará la variable a la base de datos **tal como está** . Esto significa que será una variable sin escape directamente de un usuario que pueda tener una intención maliciosa. Si hace esto, pone toda su base de datos en riesgo porque una vez que el usuario descubre que puede explotar su base de datos, puede hacer casi cualquier cosa. Nunca coloque sus argumentos directamente dentro de la cadena de condiciones.

Para obtener más información sobre los peligros de la inyección SQL, consulte la [Guía de seguridad de Ruby on Rails](http://www.guiasrails.es/security.html#sql-injection) .

**2.2.1 Condiciones del marcador de posición**

De forma similar al (?)estilo de reemplazo de params, también puede especificar hash de claves / valores en las condiciones de su matriz:

|  |
| --- |
| Client.where("created\_at >= :start\_date AND created\_at <= :end\_date",    {start\_date: params[:start\_date], end\_date: params[:end\_date]}) |

Esto permite una legibilidad más clara si tiene una gran cantidad de condiciones variables.

**2.3 Condiciones de hash**

Active Record también le permite pasar en condiciones de hash que pueden aumentar la legibilidad de la sintaxis de sus condiciones. Con las condiciones de hash, transfiere un hash con las claves de los campos que desea condicionalizar y los valores de cómo desea condicionarlos:

Solo la igualdad, el rango y la verificación de subconjuntos son posibles con las condiciones de Hash.

**2.3.1 Condiciones de igualdad**

|  |
| --- |
| Client.where(locked: true) |

El nombre del campo también puede ser una cadena:

|  |
| --- |
| Client.where('locked' => true) |

En el caso de una relación belongs\_to, se puede usar una clave de asociación para especificar el modelo si se usa un objeto Active Record como valor. Este método también funciona con relaciones polimórficas.

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

Los valores no pueden ser símbolos. Por ejemplo, no puedes hacer Client.where(status: :active).

**2.3.2 Condiciones de rango**

|  |
| --- |
| Client.where(created\_at: (Time.now.midnight - 1.day)..Time.now.midnight) |

Esto encontrará todos los clientes creados ayer usando una BETWEENdeclaración SQL:

|  |
| --- |
| SELECT \* FROM clients WHERE (clients.created\_at BETWEEN '2008-12-21 00:00:00' AND '2008-12-22 00:00:00') |

Esto demuestra una sintaxis más corta para los ejemplos en [Array Conditions](http://www.guiasrails.es/active_record_querying.html" \l "array-conditions)

**2.3.3 Condiciones de subconjunto**

Si desea buscar registros utilizando la INexpresión, puede pasar una matriz al hash de condiciones:

|  |
| --- |
| Client.where(orders\_count: [1,3,5]) |

Este código generará SQL así:

|  |
| --- |
| SELECT \* FROM clients WHERE (clients.orders\_count IN (1,3,5)) |

**2.4 NO Condiciones**

NOTLas consultas SQL pueden ser compiladas por where.not.

|  |
| --- |
| Article.where.not(author: author) |

En otras palabras, esta consulta se puede generar llamando wheresin argumentos, luego cadena inmediatamente con condiciones de notpaso where.

**3 ordenando**

Para recuperar registros de la base de datos en un orden específico, puede usar el ordermétodo.

Por ejemplo, si obtiene un conjunto de registros y desea ordenarlos en orden ascendente por el created\_atcampo en su tabla:

|  |
| --- |
| Client.order(:created\_at)  # OR  Client.order("created\_at") |

Usted podría especificar ASCo DESCtambién:

|  |
| --- |
| Client.order(created\_at: :desc)  # OR  Client.order(created\_at: :asc)  # OR  Client.order("created\_at DESC")  # OR  Client.order("created\_at ASC") |

O ordenando por múltiples campos:

|  |
| --- |
| 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") |

Si desea llamar ordervarias veces, por ejemplo, en un contexto diferente, la nueva orden se agregará a la anterior

|  |
| --- |
| Client.order("orders\_count ASC").order("created\_at DESC")  # SELECT \* FROM clients ORDER BY orders\_count ASC, created\_at DESC |

**4 Seleccionar campos específicos**

Por defecto, Model.findselecciona todos los campos del conjunto de resultados usando select \*.

Para seleccionar solo un subconjunto de campos del conjunto de resultados, puede especificar el subconjunto a través del selectmétodo.

Por ejemplo, para seleccionar solo viewable\_byy lockedcolumnas:

|  |
| --- |
| Client.select("viewable\_by, locked") |

La consulta SQL utilizada por esta llamada de búsqueda será algo así como:

|  |
| --- |
| SELECT viewable\_by, locked FROM clients |

Tenga cuidado porque esto también significa que está inicializando un objeto modelo con solo los campos que ha seleccionado. Si intenta acceder a un campo que no está en el registro inicializado, recibirá:

|  |
| --- |
| ActiveModel::MissingAttributeError: missing attribute: <attribute> |

¿Dónde <attribute>está el atributo que pediste? El idmétodo no aumentará ActiveRecord::MissingAttributeError, así que tenga cuidado al trabajar con asociaciones porque necesitan el idmétodo para funcionar correctamente.

Si desea obtener solo un registro por valor único en un campo determinado, puede usar distinct:

|  |
| --- |
| Client.select(:name).distinct |

Esto generaría SQL como:

|  |
| --- |
| SELECT DISTINCT name FROM clients |

También puede eliminar la restricción de exclusividad:

|  |
| --- |
| query = Client.select(:name).distinct  # => Returns unique names    query.distinct(false)  # => Returns all names, even if there are duplicates |

**5 Límite y desplazamiento**

Para aplicar LIMITal SQL activado por Model.find, puede especificar el LIMITuso limity los offsetmétodos en la relación.

Puede usar limitpara especificar la cantidad de registros que se recuperarán y usar offsetpara especificar la cantidad de registros que se deben omitir antes de comenzar a devolver los registros. Por ejemplo

|  |
| --- |
| Client.limit(5) |

devolverá un máximo de 5 clientes y dado que no especifica ningún desplazamiento, devolverá los primeros 5 en la tabla. El SQL que ejecuta se ve así:

|  |
| --- |
| SELECT \* FROM clients LIMIT 5 |

Añadiendo offseta eso

|  |
| --- |
| Client.limit(5).offset(30) |

devolverá un máximo de 5 clientes a partir del día 31. El SQL se ve así:

|  |
| --- |
| SELECT \* FROM clients LIMIT 5 OFFSET 30 |

**6 Grupo**

Para aplicar una GROUP BYcláusula al SQL activado por el buscador, puede especificar el groupmétodo en el buscador .

Por ejemplo, si desea buscar una recopilación de las fechas, las órdenes se crearon en:

|  |
| --- |
| Order.select("date(created\_at) as ordered\_date, sum(price) as total\_price").group("date(created\_at)") |

Y esto le dará un único Orderobjeto para cada fecha en la que haya pedidos en la base de datos.

El SQL que se ejecutará sería algo como esto:

|  |
| --- |
| SELECT date(created\_at) as ordered\_date, sum(price) as total\_price  FROM orders  GROUP BY date(created\_at) |

**6.1 Total de elementos agrupados**

Para obtener el total de elementos agrupados en una sola llamada de consulta countdespués del group.

|  |
| --- |
| Order.group(:status).count  # => { 'awaiting\_approval' => 7, 'paid' => 12 } |

El SQL que se ejecutará sería algo como esto:

|  |
| --- |
| SELECT COUNT (\*) AS count\_all, status AS status  FROM "orders"  GROUP BY status |

**7 teniendo**

SQL usa la HAVINGcláusula para especificar condiciones en los GROUP BYcampos. Puede agregar la HAVINGcláusula al SQL activado por el Model.findagregando la :havingopción al hallazgo.

Por ejemplo:

|  |
| --- |
| Order.select("date(created\_at) as ordered\_date, sum(price) as total\_price").    group("date(created\_at)").having("sum(price) > ?", 100) |

El SQL que se ejecutará sería algo como esto:

|  |
| --- |
| SELECT date(created\_at) as ordered\_date, sum(price) as total\_price  FROM orders  GROUP BY date(created\_at)  HAVING sum(price) > 100 |

Esto devolverá objetos de orden única para cada día, pero solo aquellos que se ordenan más de $ 100 por día.

**8 Condiciones principales**

**8.1 unscope**

Puede especificar ciertas condiciones para eliminar con el unscopemétodo. Por ejemplo:

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

El SQL que se ejecutará:

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

También puede descapacitar wherecláusulas específicas . Por ejemplo:

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

Una relación que haya usado unscopeafectará cualquier relación en la que se haya fusionado:

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

**8.2 only**

También puede anular las condiciones usando el onlymétodo. Por ejemplo:

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

El SQL que se ejecutará:

|  |
| --- |
| 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**

El reordermétodo anula el orden de alcance predeterminado. Por ejemplo:

|  |
| --- |
| class Article < ActiveRecord::Base    has\_many :comments, -> { order('posted\_at DESC') }  end    Article.find(10).comments.reorder('name') |

El SQL que se ejecutará:

|  |
| --- |
| SELECT \* FROM articles WHERE id = 10  SELECT \* FROM comments WHERE article\_id = 10 ORDER BY name |

En caso de que la reordercláusula no se use, el SQL ejecutado sería:

|  |
| --- |
| SELECT \* FROM articles WHERE id = 10  SELECT \* FROM comments WHERE article\_id = 10 ORDER BY posted\_at DESC |

**8.4 reverse\_order**

El reverse\_ordermétodo invierte la cláusula de ordenamiento si se especifica.

|  |
| --- |
| Client.where("orders\_count > 10").order(:name).reverse\_order |

El SQL que se ejecutará:

|  |
| --- |
| SELECT \* FROM clients WHERE orders\_count > 10 ORDER BY name DESC |

Si no se especifica ninguna cláusula de ordenación en la consulta, las reverse\_orderórdenes se procesan con la clave principal en orden inverso.

|  |
| --- |
| Client.where("orders\_count > 10").reverse\_order |

El SQL que se ejecutará:

|  |
| --- |
| SELECT \* FROM clients WHERE orders\_count > 10 ORDER BY clients.id DESC |

Este método **no** acepta argumentos.

**8.5 rewhere**

El rewheremétodo anula una condición existente con nombre. Por ejemplo:

|  |
| --- |
| Article.where(trashed: true).rewhere(trashed: false) |

El SQL que se ejecutará:

|  |
| --- |
| SELECT \* FROM articles WHERE `trashed` = 0 |

En caso de que la rewherecláusula no se use,

|  |
| --- |
| Article.where(trashed: true).where(trashed: false) |

el SQL ejecutado sería:

|  |
| --- |
| SELECT \* FROM articles WHERE `trashed` = 1 AND `trashed` = 0 |

**9 Relación nula**

El nonemétodo devuelve una relación encadenable sin registros. Cualquier condición subsiguiente encadenada a la relación devuelta continuará generando relaciones vacías. Esto es útil en escenarios donde necesita una respuesta encadenable a un método o un alcance que podría devolver cero resultados.

|  |
| --- |
| 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 objetos de solo lectura**

Active Record proporciona el readonlymétodo en una relación para prohibir explícitamente la modificación de cualquiera de los objetos devueltos. Cualquier intento de alterar un registro de solo lectura no tendrá éxito, generando una ActiveRecord::ReadOnlyRecordexcepción.

|  |
| --- |
| client = Client.readonly.first  client.visits += 1  client.save |

Como clientestá establecido explícitamente para ser un objeto de solo lectura, el código anterior generará una ActiveRecord::ReadOnlyRecordexcepción cuando llame client.savecon un valor actualizado de *visitas* .

**11 Bloqueo de registros para la actualización**

El bloqueo es útil para evitar condiciones de carrera cuando se actualizan registros en la base de datos y se aseguran actualizaciones atómicas.

Active Record proporciona dos mecanismos de bloqueo:

* Bloqueo optimista
* Bloqueo pesimista

**11.1 Bloqueo optimista**

El bloqueo optimista permite que múltiples usuarios accedan al mismo registro para ediciones, y asume un mínimo de conflictos con los datos. Lo hace al verificar si otro proceso ha realizado cambios en un registro desde que se abrió. Se ActiveRecord::StaleObjectErrorlanza una excepción si eso ocurrió y se ignora la actualización.

**Columna de bloqueo optimista**

Para utilizar el bloqueo optimista, la tabla necesita tener una columna llamada lock\_versionde tipo entero. Cada vez que se actualiza el registro, Active Record incrementa la lock\_versioncolumna. Si se realiza una solicitud de actualización con un valor inferior en el lock\_versioncampo que el que se encuentra actualmente en la lock\_versioncolumna de la base de datos, la solicitud de actualización fallará con un ActiveRecord::StaleObjectError. Ejemplo:

|  |
| --- |
| c1 = Client.find(1)  c2 = Client.find(1)    c1.first\_name = "Michael"  c1.save    c2.name = "should fail"  c2.save # Raises an ActiveRecord::StaleObjectError |

Entonces es responsable de enfrentar el conflicto rescatando la excepción y retrocediendo, fusionándose o aplicando la lógica comercial necesaria para resolver el conflicto.

Este comportamiento se puede desactivar configurando ActiveRecord::Base.lock\_optimistically = false.

Para anular el nombre de la lock\_versioncolumna, ActiveRecord::Baseproporciona un atributo de clase llamado locking\_column:

|  |
| --- |
| class Client < ActiveRecord::Base    self.locking\_column = :lock\_client\_column  end |

**11.2 Bloqueo pesimista**

El bloqueo pesimista utiliza un mecanismo de bloqueo proporcionado por la base de datos subyacente. El uso lockal construir una relación obtiene un bloqueo exclusivo en las filas seleccionadas. Las relaciones que utilizan lockgeneralmente se envuelven dentro de una transacción para evitar condiciones de interbloqueo.

Por ejemplo:

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

La sesión anterior produce el siguiente SQL para un back-end MySQL:

|  |
| --- |
| 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 |

También puede pasar SQL sin procesar al lockmétodo para permitir diferentes tipos de bloqueos. Por ejemplo, MySQL tiene una expresión llamada LOCK IN SHARE MODEdonde puede bloquear un registro pero aún permitir que otras consultas lo lean. Para especificar esta expresión simplemente pásalo como la opción de bloqueo:

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

Si ya tiene una instancia de su modelo, puede iniciar una transacción y adquirir el bloqueo de una vez con el siguiente código:

|  |
| --- |
| item = Item.first  item.with\_lock do    # This block is called within a transaction,    # item is already locked.    item.increment!(:views)  end |

**12 Unirse a las tablas**

Active Record proporciona un método de búsqueda llamado joinspara especificar JOINcláusulas en el SQL resultante. Hay varias formas de usar el joinsmétodo.

**12.1 Usar un fragmento SQL de cadena**

Solo puede proporcionar el SQL sin formato que especifica la JOINcláusula para joins:

|  |
| --- |
| Client.joins('LEFT OUTER JOIN addresses ON addresses.client\_id = clients.id') |

Esto dará como resultado el siguiente SQL:

|  |
| --- |
| SELECT clients.\* FROM clients LEFT OUTER JOIN addresses ON addresses.client\_id = clients.id |

**12.2 Uso de Array / Hash de asociaciones nombradas**

Este método solo funciona con INNER JOIN.

Active Record le permite usar los nombres de las [asociaciones](http://www.guiasrails.es/association_basics.html) definidas en el modelo como un atajo para especificar JOINcláusulas para esas asociaciones cuando usa el joinsmétodo.

Por ejemplo, considere los siguientes Category, Article, Comment, Guesty Tagmodelos:

|  |
| --- |
| 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 |

Ahora todo lo siguiente producirá las consultas de combinación esperadas usando INNER JOIN:

**12.2.1 Unirse a una sola asociación**

|  |
| --- |
| Category.joins(:articles) |

Esto produce:

|  |
| --- |
| SELECT categories.\* FROM categories    INNER JOIN articles ON articles.category\_id = categories.id |

O, en inglés: "devuelve un objeto Category para todas las categorías con artículos". Tenga en cuenta que verá categorías duplicadas si más de un artículo tiene la misma categoría. Si quieres categorías únicas, puedes usar Category.joins(:articles).uniq.

**12.2.2 Unirse a asociaciones múltiples**

|  |
| --- |
| Article.joins(:category, :comments) |

Esto produce:

|  |
| --- |
| SELECT articles.\* FROM articles    INNER JOIN categories ON articles.category\_id = categories.id    INNER JOIN comments ON comments.article\_id = articles.id |

O, en inglés: "devuelve todos los artículos que tienen una categoría y al menos un comentario". Tenga en cuenta nuevamente que los artículos con múltiples comentarios aparecerán varias veces.

**12.2.3 Unirse a asociaciones anidadas (nivel único)**

|  |
| --- |
| Article.joins(comments: :guest) |

Esto produce:

|  |
| --- |
| SELECT articles.\* FROM articles    INNER JOIN comments ON comments.article\_id = articles.id    INNER JOIN guests ON guests.comment\_id = comments.id |

O, en inglés: "devolver todos los artículos que tienen un comentario hecho por un invitado".

**12.2.4 Unirse a asociaciones anidadas (nivel múltiple)**

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

Esto produce:

|  |
| --- |
| 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 Especificación de condiciones en las tablas unidas**

Puede especificar condiciones en las tablas combinadas utilizando las condiciones normales de [Array](http://www.guiasrails.es/active_record_querying.html" \l "array-conditions) y [String](http://www.guiasrails.es/active_record_querying.html" \l "pure-string-conditions) . [Las condiciones de hash](http://www.guiasrails.es/active_record_querying.html#hash-conditions) proporcionan una sintaxis especial para especificar condiciones para las tablas unidas:

|  |
| --- |
| time\_range = (Time.now.midnight - 1.day)..Time.now.midnight  Client.joins(:orders).where('orders.created\_at' => time\_range) |

Una sintaxis alternativa y más limpia es anidar las condiciones de hash:

|  |
| --- |
| time\_range = (Time.now.midnight - 1.day)..Time.now.midnight  Client.joins(:orders).where(orders: { created\_at: time\_range }) |

Esto encontrará todos los clientes que tienen pedidos que se crearon ayer, una vez más utilizando una BETWEENexpresión SQL.

**13 asociaciones de carga impaciente**

La carga ansiosa es el mecanismo para cargar los registros asociados de los objetos devueltos mediante el Model.finduso de la menor cantidad de consultas posible.

**Problema de consultas N + 1**

Considere el siguiente código, que encuentra 10 clientes e imprime sus códigos postales:

|  |
| --- |
| clients = Client.limit(10)    clients.each do |client|    puts client.address.postcode  end |

Este código se ve bien a primera vista. Pero el problema se encuentra dentro del número total de consultas ejecutadas. El código anterior ejecuta 1 (para encontrar 10 clientes) + 10 (uno por cada cliente para cargar la dirección) = **11** consultas en total.

**Solución al problema de consultas N + 1**

Active Record le permite especificar de antemano todas las asociaciones que se van a cargar. Esto es posible especificando el includesmétodo de la Model.findllamada. Con includes, Active Record garantiza que todas las asociaciones especificadas se carguen utilizando la mínima cantidad posible de consultas.

Revisando el caso anterior, podríamos reescribir Client.limit(10)para usar direcciones de carga ansiosas:

|  |
| --- |
| clients = Client.includes(:address).limit(10)    clients.each do |client|    puts client.address.postcode  end |

El código anterior ejecutará solo **2** consultas, a diferencia de **11** consultas en el caso anterior:

|  |
| --- |
| 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 le permite cargar cualquier número de asociaciones con una sola Model.findllamada utilizando una matriz, hash o un hash anidado de matriz / hash con el includesmétodo.

**13.1.1 Matriz de asociaciones múltiples**

|  |
| --- |
| Article.includes(:category, :comments) |

Esto carga todos los artículos y la categoría y los comentarios asociados para cada artículo.

**13.1.2 Asociaciones anidadas hash**

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

Esto encontrará la categoría con id 1 y ansioso por cargar todos los artículos asociados, las etiquetas y los comentarios de los artículos asociados, y la asociación de invitados de cada comentario.

**13.2 Especificación de condiciones en asociaciones cargadas de ganas**

A pesar de que Active Record le permite especificar condiciones en las asociaciones cargadas ansiosas joins, la forma recomendada es usar [uniones en su](http://www.guiasrails.es/active_record_querying.html#joining-tables) lugar.

Sin embargo, si debe hacer esto, puede usarlo wherecomo lo haría normalmente.

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

Esto generaría una consulta que contiene un LEFT OUTER JOINmientras que el joinsmétodo generaría uno usando la INNER JOINfunción en su lugar.

|  |
| --- |
| 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) |

Si no hubiera una wherecondición, esto generaría el conjunto normal de dos consultas.

Usar whereesto solo funcionará cuando le pase Hash. Para los fragmentos SQL que necesita usar referencespara forzar tablas unidas:

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

Si, en el caso de esta includesconsulta, no hubiera comentarios para ningún artículo, todos los artículos seguirían cargándose. Al usar joins(INNER JOIN), las condiciones de unión **deben**coincidir, de lo contrario, no se devolverán registros.

**14 ámbitos**

El alcance le permite especificar consultas de uso común a las que se puede hacer referencia como llamadas a métodos en los objetos o modelos de asociación. Con estos ámbitos, puede usar todos los métodos anteriormente cubiertos, como where, joinsy includes. Todos los métodos de alcance devolverán un ActiveRecord::Relationobjeto que permitirá que otros métodos (como otros ámbitos) sean llamados.

Para definir un alcance simple, usamos el scopemétodo dentro de la clase, pasando la consulta que nos gustaría ejecutar cuando se llama a este alcance:

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

Esto es exactamente lo mismo que definir un método de clase, y lo que usa es una cuestión de preferencia personal:

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

Los ámbitos también se pueden encadenar dentro de los ámbitos:

|  |
| --- |
| class Article < ActiveRecord::Base    scope :published,               -> { where(published: true) }    scope :published\_and\_commented, -> { published.where("comments\_count > 0") }  end |

Para llamar a este publishedalcance podemos llamarlo a cualquiera de las clases:

|  |
| --- |
| Article.published # => [published articles] |

O en una asociación que consiste en Articleobjetos:

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

**14.1 Pasar argumentos**

Su alcance puede tomar argumentos:

|  |
| --- |
| class Article < ActiveRecord::Base    scope :created\_before, ->(time) { where("created\_at < ?", time) }  end |

Llame al alcance como si fuera un método de clase:

|  |
| --- |
| Article.created\_before(Time.zone.now) |

Sin embargo, esto solo duplica la funcionalidad que le proporcionaría un método de clase.

|  |
| --- |
| class Article < ActiveRecord::Base    def self.created\_before(time)      where("created\_at < ?", time)    end  end |

Usar un método de clase es la forma preferida de aceptar argumentos para ámbitos. Estos métodos seguirán estando accesibles en los objetos de asociación:

|  |
| --- |
| category.articles.created\_before(time) |

**14.2 Aplicar un alcance predeterminado**

Si deseamos aplicar un alcance en todas las consultas al modelo, podemos usar eldefault\_scopemétodo dentro del mismo modelo.

|  |
| --- |
| class Client < ActiveRecord::Base    default\_scope { where("removed\_at IS NULL") }  end |

Cuando las consultas se ejecutan en este modelo, la consulta SQL ahora tendrá un aspecto similar a este:

|  |
| --- |
| SELECT \* FROM clients WHERE removed\_at IS NULL |

Si necesita hacer cosas más complejas con un alcance predeterminado, puede definirlo alternativamente como un método de clase:

|  |
| --- |
| class Client < ActiveRecord::Base    def self.default\_scope      # Should return an ActiveRecord::Relation.    end  end |

**14.3 Fusión de ámbitos**

Al igual que las wherecláusulas, los ámbitos se combinan mediante ANDcondiciones.

|  |
| --- |
| 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' |

Podemos mezclar y combinar scopey wherecondiciones y el sql final tendrá todas las condiciones unidas AND.

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

Si queremos last where clauseque gane, entonces Relation#mergese puede usar.

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

Una advertencia importante es que default\_scopese antepone en scopey wherecondiciones.

|  |
| --- |
| 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' |

Como se puede ver por encima de la default\_scopese está fusionando en tanto scopey wherecondiciones.

**14.4 Eliminar todo el alcance**

Si deseamos eliminar el alcance por algún motivo, podemos usar el unscopedmétodo. Esto es especialmente útil si a default\_scopese especifica en el modelo y no se debe aplicar para esta consulta en particular.

|  |
| --- |
| Client.unscoped.load |

Este método elimina todo el alcance y hará una consulta normal en la tabla.

Tenga en cuenta que el encadenamiento unscopedcon a scopeno funciona. En estos casos, se recomienda que use la forma de bloque de unscoped:

|  |
| --- |
| Client.unscoped {    Client.created\_before(Time.zone.now)  } |

**15 buscadores dinámicos**

Para cada campo (también conocido como un atributo) que define en su tabla, Active Record proporciona un método de búsqueda. Si tiene un campo llamado first\_nameen su Clientmodelo, por ejemplo, lo obtiene find\_by\_first\_namede forma gratuita desde Active Record. Si tiene un lockedcampo en el Clientmodelo, también obtiene find\_by\_lockedy métodos.

Puede especificar un signo de exclamación ( !) al final de los buscadores dinámicos para que generen un ActiveRecord::RecordNotFounderror si no devuelven ningún registro, comoClient.find\_by\_name!("Ryan")

Si desea buscar tanto por nombre como bloqueado, puede encadenar estos buscadores simplemente escribiendo " and" entre los campos. Por ejemplo Client.find\_by\_first\_name\_and\_locked("Ryan", true),.

**16 Comprender el método de encadenamiento**

El patrón Active Record implementa [Method Chaining](http://en.wikipedia.org/wiki/Method_chaining) , que nos permite usar múltiples métodos Active Record juntos de una manera simple y directa.

Puede métodos de cadena en un comunicado cuando el método anterior devuelve una llamadaActiveRecord::Relation, como all, where, y joins. Los métodos que devuelven un solo objeto (consulte [Recuperación de una sola sección de objeto](http://www.guiasrails.es/active_record_querying.html#retrieving-a-single-object) ) deben estar al final de la instrucción.

Hay algunos ejemplos a continuación. Esta guía no cubrirá todas las posibilidades, solo algunas como ejemplos. Cuando se llama a un método de registro activo, la consulta no se genera inmediatamente y se envía a la base de datos, esto sucede cuando realmente se necesitan los datos. Entonces, cada ejemplo a continuación genera una sola consulta.

**16.1 Recuperación de datos filtrados de varias tablas**

|  |
| --- |
| Person    .select('people.id, people.name, comments.text')    .joins(:comments)    .where('comments.created\_at > ?', 1.week.ago) |

El resultado debería ser algo como esto:

|  |
| --- |
| SELECT people.id, people.name, comments.text  FROM people  INNER JOIN comments    ON comments.person\_id = people.id  WHERE comments.created\_at = '2015-01-01' |

**16.2 Recuperación de datos específicos de múltiples tablas**

|  |
| --- |
| Person    .select('people.id, people.name, companies.name')    .joins(:company)    .find\_by('people.name' => 'John') # this should be the last |

Lo anterior debería generar:

|  |
| --- |
| SELECT people.id, people.name, companies.name  FROM people  INNER JOIN companies    ON companies.person\_id = people.id  WHERE people.name = 'John'  LIMIT 1 |

Recuerde que, si find\_bydevuelve más de un registro, tomará solo el primero e ignorará los demás. Tenga en cuenta la LIMIT 1declaración anterior.

**17 Encuentra o construye un nuevo objeto**

Es común que necesite encontrar un registro o crearlo si no existe. Puedes hacer eso con los métodos find\_or\_create\_byy find\_or\_create\_by!.

**17.1 find\_or\_create\_by**

El find\_or\_create\_bymétodo verifica si existe un registro con los atributos. Si no es así, entonces createse llama. Veamos un ejemplo.

Supongamos que quiere encontrar un cliente llamado 'Andy', y si no hay ninguno, cree uno. Puede hacerlo ejecutando:

|  |
| --- |
| 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"> |

El SQL generado por este método se ve así:

|  |
| --- |
| 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\_bydevuelve el registro que ya existe o el nuevo registro. En nuestro caso, aún no teníamos un cliente llamado Andy, por lo que el registro se crea y se devuelve.

El nuevo registro podría no guardarse en la base de datos; eso depende de si las validaciones pasaron o no (al igual que create).

Supongamos que queremos establecer el atributo 'bloqueado' falsesi estamos creando un nuevo registro, pero no queremos incluirlo en la consulta. Entonces queremos encontrar el cliente llamado "Andy", o si ese cliente no existe, cree un cliente llamado "Andy" que no esté bloqueado.

Podemos lograr esto de dos maneras. El primero es usar create\_with:

|  |
| --- |
| Client.create\_with(locked: false).find\_or\_create\_by(first\_name: 'Andy') |

La segunda forma es usar un bloque:

|  |
| --- |
| Client.find\_or\_create\_by(first\_name: 'Andy') do |c|    c.locked = false  end |

El bloque solo se ejecutará si el cliente se está creando. La segunda vez que ejecutemos este código, el bloque será ignorado.

**17.2 find\_or\_create\_by!**

También puede usar find\_or\_create\_by!para generar una excepción si el nuevo registro no es válido. Las validaciones no están cubiertas en esta guía, pero asumamos por un momento que agrega temporalmente

|  |
| --- |
| validates :orders\_count, presence: true |

a tu Clientmodelo. Si intenta crear una nueva Clientsin pasar un orders\_count, el registro no será válido y se generará una excepción:

|  |
| --- |
| Client.find\_or\_create\_by!(first\_name: 'Andy')  # => ActiveRecord::RecordInvalid: Validation failed: Orders count can't be blank |

**17.3 find\_or\_initialize\_by**

El find\_or\_initialize\_bymétodo funcionará igual find\_or\_create\_bypero llamará en newlugar de create. Esto significa que se creará una nueva instancia de modelo en la memoria, pero no se guardará en la base de datos. Continuando con el find\_or\_create\_byejemplo, ahora queremos que el cliente se llame '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 |

Debido a que el objeto aún no está almacenado en la base de datos, el SQL generado se ve así:

|  |
| --- |
| SELECT \* FROM clients WHERE (clients.first\_name = 'Nick') LIMIT 1 |

Cuando desee guardarlo en la base de datos, simplemente llame a save:

|  |
| --- |
| nick.save  # => true |

**18 Encontrando por SQL**

Si desea usar su propio SQL para buscar registros en una tabla, puede usar find\_by\_sql. El find\_by\_sqlmétodo devolverá una matriz de objetos, incluso si la consulta subyacente devuelve un solo registro. Por ejemplo, podría ejecutar esta consulta:

|  |
| --- |
| 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 le proporciona una forma simple de realizar llamadas personalizadas a la base de datos y recuperar objetos instanciados.

**18.1 select\_all**

find\_by\_sqltiene un pariente cercano llamado connection#select\_all. select\_allrecuperará objetos de la base de datos utilizando SQL personalizado, find\_by\_sqlpero no creará una instancia de ellos. En cambio, obtendrá una matriz de hash donde cada hash indica un registro.

|  |
| --- |
| 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"}  ] |

**18.2 pluck**

pluckse puede usar para consultar columnas únicas o múltiples desde la tabla subyacente de un modelo. Acepta una lista de nombres de columna como argumento y devuelve una matriz de valores de las columnas especificadas con el tipo de datos correspondiente.

|  |
| --- |
| 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 hace posible reemplazar código como:

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

con:

|  |
| --- |
| Client.pluck(:id)  # or  Client.pluck(:id, :name) |

A diferencia select, pluckconvierte directamente el resultado de una base de datos en un Ruby Array, sin construir ActiveRecordobjetos. Esto puede significar un mejor rendimiento para una consulta grande o que se ejecuta a menudo. Sin embargo, cualquier anulación de método modelo no estará disponible. Por ejemplo:

|  |
| --- |
| 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"] |

Además, a diferencia de selectotros Relationámbitos, pluckdesencadena una consulta inmediata y, por lo tanto, no se puede encadenar con otros ámbitos, aunque puede funcionar con ámbitos ya construidos anteriormente:

|  |
| --- |
| Client.pluck(:name).limit(1)  # => NoMethodError: undefined method `limit' for #<Array:0x007ff34d3ad6d8>    Client.limit(1).pluck(:name)  # => ["David"] |

**18.3 ids**

ids se puede usar para obtener todos los ID de la relación usando la clave principal de la tabla.

|  |
| --- |
| Person.ids  # SELECT id FROM people |
| class Person < ActiveRecord::Base    self.primary\_key = "person\_id"  end    Person.ids  # SELECT person\_id FROM people |

**19 Existencia de objetos**

Si simplemente desea verificar la existencia del objeto, se llama un método exists?. Este método consultará la base de datos utilizando la misma consulta que find, pero en lugar de devolver un objeto o conjunto de objetos, devolverá cualquiera de ellos trueo false.

|  |
| --- |
| Client.exists?(1) |

El exists?método también toma múltiples valores, pero la trampa es que volverá truesi existe alguno de esos registros.

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

Incluso es posible usar exists?sin ningún argumento sobre un modelo o una relación.

|  |
| --- |
| Client.where(first\_name: 'Ryan').exists? |

Lo anterior regresa truesi hay al menos un cliente con el first\_name'Ryan' y de lo false contrario.

|  |
| --- |
| Client.exists? |

Lo anterior vuelve falsesi la clientstabla está vacía y de lo truecontrario.

También puede usar any?y many?verificar la existencia en un modelo o relación.

|  |
| --- |
| # 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? |

**20 Cálculos**

Esta sección utiliza el recuento como un método de ejemplo en este preámbulo, pero las opciones descritas se aplican a todas las subsecciones.

Todos los métodos de cálculo trabajan directamente en un modelo:

|  |
| --- |
| Client.count  # SELECT count(\*) AS count\_all FROM clients |

O en una relación:

|  |
| --- |
| Client.where(first\_name: 'Ryan').count  # SELECT count(\*) AS count\_all FROM clients WHERE (first\_name = 'Ryan') |

También puede usar varios métodos de búsqueda en una relación para realizar cálculos complejos:

|  |
| --- |
| Client.includes("orders").where(first\_name: 'Ryan', orders: { status: 'received' }).count |

Que se ejecutará:

|  |
| --- |
| 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') |

**20.1 Cuenta**

Si desea ver cuántos registros hay en la tabla de su modelo, puede llamar Client.county devolver el número. Si desea ser más específico y encontrar todos los clientes con su edad presente en la base de datos que puede utilizar Client.count(:age).

Para ver las opciones, consulte la sección principal, [Cálculos](http://www.guiasrails.es/active_record_querying.html#calculations) .

**20.2 Promedio**

Si desea ver el promedio de un cierto número en una de sus tablas, puede llamar al averagemétodo de la clase que se relaciona con la tabla. Esta llamada al método se verá más o menos así:

|  |
| --- |
| Client.average("orders\_count") |

Esto devolverá un número (posiblemente un número de coma flotante como 3.14159265) que representa el valor promedio en el campo.

Para ver las opciones, consulte la sección principal, [Cálculos](http://www.guiasrails.es/active_record_querying.html#calculations) .

**20.3 Mínimo**

Si desea encontrar el valor mínimo de un campo en su tabla, puede llamar al minimummétodo de la clase que se relaciona con la tabla. Esta llamada al método se verá más o menos así:

|  |
| --- |
| Client.minimum("age") |

Para ver las opciones, consulte la sección principal, [Cálculos](http://www.guiasrails.es/active_record_querying.html#calculations) .

**20.4 Máximo**

Si desea encontrar el valor máximo de un campo en su tabla, puede llamar al maximummétodo en la clase que se relaciona con la tabla. Esta llamada al método se verá más o menos así:

|  |
| --- |
| Client.maximum("age") |

Para ver las opciones, consulte la sección principal, [Cálculos](http://www.guiasrails.es/active_record_querying.html#calculations) .

**20.5 Suma**

Si desea encontrar la suma de un campo para todos los registros en su tabla, puede llamar al summétodo de la clase que se relaciona con la tabla. Esta llamada al método se verá más o menos así:

|  |
| --- |
| Client.sum("orders\_count") |