Advanced Active Record

Unlock advanced Active Record techniques with our concise cheatsheet for expert Rails developers. Covering complex queries, associations, eager loading, validations, and performance optimizations, this guide boosts your Rails app efficiency with practical examples and best practices.

Advanced Querying and Relation Chaining

Basic Querying

`where` - Adds conditions to the query.

Examples:

```
User.where(name: 'John')
```

```
User.where('age > ?', 20)
```

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

`order` - Specifies the order of the result set.

Examples:

`User.order(:name)` # Ascending order

`User.order(age: :desc)` # Descending order

```
User.order('name DESC, age ASC')
```

`limit` - Limits the number of records returned.

Example:

```
User.limit(10)
```

`offset` - Specifies the number of records to skip.

Example:

```
User.offset(30)
```

`select` - Specifies which columns to retrieve.

Examples:

```
User.select(:id, :name)
```

```
User.select('id, name, email')
```

Advanced `where` Conditions

`where.not` - Excludes records matching the condition.

Example:

```
User.where.not(name: 'John')
```

`where(id: [1, 2, 3])` - Using an array for `IN` queries.

Example:

```
User.where(id: [1, 2, 3])
```

`where('name LIKE ?', '%John%')` - Using LIKE for pattern matching.

Example:

```
User.where('name LIKE ?', '%John%')
```

`where(age: nil)` - Finding records with NULL values.

Example:

```
User.where(age: nil)
```

```
User.where.not(age: nil)
```

`where('age > ? AND city = ?', 25, 'New York')` - Complex AND conditions.

Example:

```
User.where('age > ? AND city = ?', 25, 'New York')
```

`where('age > ? OR city = ?', 25, 'New York')` - Complex OR conditions.

Example:

```
User.where('age > ? OR city = ?', 25, 'New York')
```

Aggregation Methods

`count` - Returns the number of records.

Examples:

```
User.count
```

```
User.where(age: 25).count
```

`average` - Calculates the average value of a column.

Example:

```
User.average(:age)
```

`minimum` - Returns the minimum value of a column.

Example:

```
User.minimum(:age)
```

`maximum` - Returns the maximum value of a column.

Example:

```
User.maximum(:age)
```

`sum` - Calculates the sum of a column.

Example:

```
User.sum(:age)
```

Grouping and Having

`group` - Groups records based on a column.

Example:

```
User.group(:city).count
```

`having` - Adds conditions to the grouped results.

Example:

```
User.group(:city).having('COUNT(*) > 10')
```

Example: Grouping users by city and finding cities with more than 5 users.

```
User.select(:city, 'COUNT(*) AS user_count').group(:city).having('COUNT(*) > 5')
```

Pluck, Pick and Distinct

`pluck` - Retrieves specific columns as an array.

Examples:

`User.pluck(:name)` # Returns an array of names

`User.pluck(:id, :name)` # Returns an array of [id, name] pairs

`distinct` - Ensures only unique values are returned.

Example:

`User.distinct.pluck(:city)`

Combining `distinct` and `pluck` to get a unique list of cities.

```
User.distinct.pluck(:city)
```

`pick` - Retrieves a single attribute value from the database.

Examples:

`User.where(name: 'John').pick(:id)` # Returns the ID of the first user named John

`User.where(active: true).pick(:email)` # Returns the email of the first active user

Using `pick` with `order` to retrieve the email of the newest user.

```
User.order(created_at: :desc).pick(:email)
```

Combining `pick` with conditions.

```
Product.where('price > ?', 100).pick(:name)
```

Eager Loading (N+1 Problem)

`includes` - Eager loads associated records.

Example:

`User.includes(:posts)` # Loads users and their posts in one query

`preload` - Similar to includes but uses separate queries.

Example:

`User.preload(:posts)`

`eager_load` - Uses a LEFT OUTER JOIN to eager load.

Example:

`User.eager_load(:posts)`

Using `includes` with multiple associations.

Example:

`User.includes(:posts, :comments)`

Specifying conditions when eager loading.

Example:

`User.includes(posts: { comments: :author }).where('posts.published = ?', true)`

Scopes

Defining a simple scope.

```
class User < ApplicationRecord
  scope :active, -> { where(active: true) }
end
```

Using a scope with parameters.

```
class User < ApplicationRecord
  scope :older_than, ->(age) { where('age > ?', age) }
end
```

Calling a scope.

```
User.active # Returns active users
User.older_than(25) # Returns users older than 25
```

Default scopes.

```
class User < ApplicationRecord
  default_scope { order(created_at: :desc) }
end
```

Removing default scope with `unscoped`.

```
User.unscoped.all # Ignores the default scope
```

Combining scopes.

```
User.active.older_than(20)
```

Existence and Association

`where.missing(association)`

Find records where the specified association does **not** exist.

Example:
Find all users without any posts:

```
User.where.missing(:posts)
```

`where.associated(association)`

Find records where the specified association **does** exist.

Example:
Find all users who have at least one post:

```
User.where.associated(:posts)
```

Combining `where.missing` with conditions

Filter records where an association is missing and apply other conditions.

Example:
Find users without posts and whose status is 'active':

```
User.where(status: 'active').where.missing(:posts)
```

Using `where.associated` with conditions

Filter records where an association exists and apply additional criteria.

Example:
Find users with posts that were created in the last week:

```
User.where.associated(:posts).where(posts: { created_at: 1.week.ago..Time.now })
```

`where.missing` with nested associations

Find records where a nested association does not exist.

Example:
Find categories that have no products with active reviews:

```
Category.where.missing(products: :reviews).where(reviews: { status: 'active' })
```

`where.associated` with nested associations

Find records where a nested association exists.

Example:
Find categories that have products with at least one review:

```
Category.where.associated(products: :reviews)
```

Chaining `where.associated` for multiple associations

Ensure multiple associations exist for a given record.

Example:
Find articles that have both comments and tags:

```
Article.where.associated(:comments).where.associated(:tags)
```

Chaining `where.missing` for multiple associations

Ensure multiple associations are absent for a given record.

Example:
Find users who don't have profiles and don't have orders:

```
User.where.missing(:profile).where.missing(:orders)
```

Optimizing `where.missing` with indexes

Ensure appropriate indexes are in place on foreign key columns for performance.

Best Practice:
Index the `user_id` column in the `posts` table.

Optimizing `where.associated` with indexes

Ensure appropriate indexes are in place on foreign key columns for performance.

Best Practice:
Index the `category_id` column in the `products` table.

Using `exists?` with `where.associated`

Check if any associated records exist without loading them.

Example:
Check if any users have associated posts:

```
User.where.associated(:posts).exists?
```

Advanced Active Record with Arel

Introduction to Arel

Arel is a SQL AST (Abstract Syntax Tree) manager for Ruby. It simplifies the generation of complex SQL queries, offering a more Ruby-like syntax.

It's especially useful when Active Record's query interface becomes insufficient for your needs.

Arel provides a way to build SQL queries programmatically using Ruby objects that represent SQL components (tables, columns, predicates, etc.).

Arel is typically used behind the scenes by Active Record, but you can also use it directly to construct more intricate queries.

By using Arel, you bypass the Active Record query interface and directly manipulate the SQL query that will be executed against the database.

Arel is particularly useful when you need to perform complex joins, subqueries, or conditional queries that are difficult to express using Active Record's standard methods.

Basic Arel Usage

Create a table object	`table = Arel::Table.new(:users)`
Access a column	`column = table[:id]`
Build a select query	`query = table.project(table[:name])`
Add a where clause	`query = query.where(table[:age].gt(18))`
Compile the query to SQL	`sql = query.to_sql # => SELECT "users"."name" FROM "users" WHERE ("users"."age" > 18)`
Execute the query using Active Record connection	`results = ActiveRecord::Base.connection.execute(sql)`

Advanced Query Building

Combining Predicates:

You can combine predicates using `and` and `or` to create more complex conditions.

```
users = Arel::Table.new(:users)
query =
users.where(users[:active].eq(true).and(user
s[:age].gt(18)))
```

Using Joins:

Arel simplifies creating joins between tables. Use `join` method and specify the join type using `on`.

```
users = Arel::Table.new(:users)
orders = Arel::Table.new(:orders)

join =
users.join(orders).on(users[:id].eq(orders[:
user_id]))
query = users.project(users[:name],
orders[:order_date]).join(join)
```

Subqueries:

Arel allows embedding subqueries into your main queries using the `as` method to alias the subquery.

```
subquery =
Arel::SelectManager.new(Arel::Table.engine)
subquery.from(Arel::Table.new(:orders)).proj
ect(Arel.star.count.as('order_count')).where
(Arel::Table.new(:orders)[:user_id].eq(1))

users = Arel::Table.new(:users)
query = users.project(users[:name],
subquery.as('user_orders'))
```

Common Arel Predicates

`eq`	Equal to.
`not_eq`	Not equal to.
`gt`	Greater than.
`gteq`	Greater than or equal to.
`lt`	Less than.
`lteq`	Less than or equal to.
`in`	Value is in a set.
`not_in`	Value is not in a set.
`matches`	Pattern matching (LIKE).
`does_not_match`	Negated pattern matching (NOT LIKE).
`cont`	For ransack gem. Contains value.

Arel with Active Record

Integrating Arel with Active Record allows you to use complex Arel queries within your Rails models.

```
class User < ApplicationRecord
  def self.complex_query
    users = Arel::Table.new(:users)
    query =
users.where(users[:active].eq(true).and(user
s[:age].gt(18)))
    User.from(users.where(query.where_sql))
  end
end
```

You can then call this method like any other scope or class method on your model.

```
users = User.complex_query
```

This approach provides a clean and maintainable way to incorporate raw SQL or Arel-based queries into your Active Record models.

Advanced Associations

Has Many Through Association

The `has_many :through` association is used to create a many-to-many relationship with another model through a join model. This allows you to easily query across multiple tables.

Defining the Association:

```
class Doctor < ApplicationRecord
  has_many :appointments
  has_many :patients, through: :appointments
end

class Patient < ApplicationRecord
  has_many :appointments
  has_many :doctors, through: :appointments
end

class Appointment < ApplicationRecord
  belongs_to :doctor
  belongs_to :patient
end
```

Explanation:

- Doctors have many patients through appointments.
- Patients have many doctors through appointments.
- Appointments belong to both doctors and patients.

This setup allows you to easily query doctors for their patients and vice versa.

Example Usage:

```
doctor = Doctor.find(1)
patients = doctor.patients  # Returns all patients associated with the doctor
```

Self-Referential Associations

A self-referential association is where a model has a relationship with itself. This is commonly used for hierarchical data, such as categories or employee hierarchies.

Example - Employee Hierarchy:

```
class Employee < ApplicationRecord
  belongs_to :manager, class_name: 'Employee', optional: true
  has_many :employees, foreign_key: :manager_id
end
```

Explanation:

- `belongs_to :manager` establishes that an employee belongs to a manager, which is another employee.
- `has_many :employees` establishes that an employee can have many employees reporting to them.
- `class_name: 'Employee'` specifies that the association is with the Employee model itself.
- `foreign_key: :manager_id` specifies the column in the employees table that stores the manager's ID.

Example Usage:

```
employee = Employee.find(1)
manager = employee.manager  # Returns the employee's manager
subordinates = employee.employees # Returns all employees who report to this employee
```

Polymorphic Associations

Polymorphic associations allow a model to belong to multiple other models, on a single association. A common use case is for comments that can belong to either articles or events.

Defining the Association:

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic: true
end

class Article < ApplicationRecord
  has_many :comments, as: :commentable
end

class Event < ApplicationRecord
  has_many :comments, as: :commentable
end
```

Explanation:

- `belongs_to :commentable, polymorphic: true` indicates that the comment can belong to any model, specified by the `commentable_type` and `commentable_id` columns.
- `has_many :comments, as: :commentable` defines the other end of the polymorphic association in the Article and Event models.

Database Migration:
When using polymorphic associations, ensure your database migration includes the necessary columns:

```
create_table :comments do |t|
  t.text :content
  t.references :commentable, polymorphic: true, index: true
  t.timestamps
end
```

Example Usage:

```
article = Article.find(1)
comment = article.comments.create(content: 'Great article!')

event = Event.find(1)
comment = event.comments.create(content: 'Exciting event!')
```

Association Scopes

Association scopes allow you to customize the data retrieved through an association using a block or a lambda. This is useful for filtering or ordering associated records.

Using a Block:

```
class User < ApplicationRecord
  has_many :active_orders, -> { where(status: 'active') }, class_name: 'Order'
end
```

Explanation:

- This defines an association `active_orders` that only retrieves orders with a status of 'active'.
- `class_name: 'Order'` specifies that the association is with the Order model.

Using a Lambda:

```
class User < ApplicationRecord
  has_many :recent_orders, -> { order(created_at: :desc).limit(5) }, class_name: 'Order'
end
```

Explanation:

- This defines an association `recent_orders` that retrieves the 5 most recently created orders, ordered by `created_at` in descending order.

Example Usage:

```
user = User.find(1)
active_orders = user.active_orders # Returns only active orders for the user
recent_orders = user.recent_orders # Returns the 5 most recent orders for the user
```

Inverse Of

The `inverse_of` option in associations informs Active Record about the inverse association, allowing it to use cached objects and avoid unnecessary database queries. This can significantly improve performance.

Example:

```
class Post < ApplicationRecord
  belongs_to :author, inverse_of: :posts
  has_many :comments, inverse_of: :post
end

class Author < ApplicationRecord
  has_many :posts, inverse_of: :author
end

class Comment < ApplicationRecord
  belongs_to :post, inverse_of: :comments
end
```

Explanation:

- `inverse_of: :posts` in the `belongs_to :author` association tells Active Record that the `author` association in the `Post` model is the inverse of the `posts` association in the `Author` model.
- Similarly, `inverse_of: :author` in the `has_many :posts` association informs Active Record about the inverse relationship.

Benefits:

- **Improved Performance:** Active Record can use cached objects instead of querying the database when the inverse association is already loaded.
- **Data Consistency:** Changes made to one side of the association are automatically reflected on the other side.

Usage Notes:

- `inverse_of` should be used in both sides of the association.
- It works best with `belongs_to` and `has_many` associations.

Association Callbacks

Association callbacks are methods that are triggered when adding or removing associated objects. These are useful for maintaining data integrity or performing actions related to the association.

Available Callbacks:

- `before_add`
- `after_add`
- `before_remove`
- `after_remove`

Example:

```
class Author < ApplicationRecord
  has_many :books, before_add: :check_book_count, after_add: :log_book_addition, before_remove: :check_book_removal

  private

  def check_book_count(book)
    if self.books.count >= 10
      raise 'Author cannot have more than 10 books'
    end
  end

  def log_book_addition(book)
    Rails.logger.info "Book #{book.title} added to author #{self.name}"
  end

  def check_book_removal(book)
    puts "Removing #{book.title} from #{self.name}"
  end
end
```

Explanation:

- `before_add: :check_book_count` is called before a book is added to the author's books collection. If the author already has 10 books, it raises an error.
- `after_add: :log_book_addition` is called after a book is added to the author's books collection. It logs the addition of the book.
- `before_remove :check_book_removal` is called before a book is removed from the author's book collection.

Usage Notes:

- Callbacks receive the associated object as an argument.
- `before_add` and `before_remove` can halt the addition or removal by raising an exception.

Eager Loading Strategies in Rails

Understanding the N+1 Query Problem

The N+1 query problem occurs when Active Record executes one query to fetch a collection of records (the '1' query), and then executes N additional queries to fetch associated records for each of the initial records. This can significantly degrade performance.

Example:

```
# Without eager loading
posts = Post.all
posts.each do |post|
  puts post.user.name # Triggers a new query for each post to fetch the user
end
```

In the above example, if there are 100 posts, it will result in 1 (Post.all) + 100 (post.user) queries. This is highly inefficient.

Eager loading is a technique to reduce the number of queries by pre-loading the associated records, thus mitigating the N+1 problem.

Eager Loading with `includes`

`includes` is the most common and recommended way to perform eager loading in Rails. It tells Active Record to fetch the associated records in as few queries as possible.

Example:

```
posts = Post.includes(:user)
posts.each do |post|
  puts post.user.name # Accessing user does not trigger a new query
end
```

`includes` intelligently decides whether to use `LEFT OUTER JOIN` or separate queries based on the associations. Generally, it uses `LEFT OUTER JOIN` for simple associations and separate queries for more complex associations or when preloading multiple associations.

You can specify multiple associations to be eager loaded:

```
posts = Post.includes(:user, :comments)
```

You can also eager load nested associations:

```
posts = Post.includes(user: :profile)
```

Using `where` clause with includes:

```
posts = Post.includes(:user).where(users: { active: true })
```

Eager Loading with `preload`

`preload` is another method for eager loading that always uses separate queries for each association. It is less intelligent than `includes` but can be useful in specific scenarios.

Example:

```
posts = Post.preload(:user)
posts.each do |post|
  puts post.user.name # Accessing user does not trigger a new query
end
```

Unlike `includes`, `preload` doesn't use joins. It always performs separate queries to load the associated records.

When to use `preload`:

- When you explicitly want separate queries.
- When dealing with complex associations where `includes` might not be optimal.

Multiple associations with preload:

```
posts = Post.preload(:user, :comments)
```

Nested associations with preload:

```
posts = Post.preload(user: :profile)
```

Eager Loading with `eager_load`

`eager_load` forces Active Record to use a `LEFT OUTER JOIN` to fetch associated records. It is more restrictive than `includes` and `preload`.

Example:

```
posts = Post.eager_load(:user)
posts.each do |post|
  puts post.user.name # Accessing user does not trigger a new query
end
```

`eager_load` always uses `LEFT OUTER JOIN`, regardless of the complexity of the association.

When to use `eager_load`:

- When you specifically want to use `LEFT OUTER JOIN`.
- When you need to filter based on the associated records in the same query.

Using `where` clause with eager_load:

```
posts = Post.eager_load(:user).where(users: { active: true })
```

Multiple associations with eager_load:

```
posts = Post.eager_load(:user, :comments)
```

Comparison of Eager Loading Methods

Method	Behavior
`includes`	Chooses between `LEFT OUTER JOIN` or separate queries based on the association.
`preload`	Always uses separate queries.
`eager_load`	Always uses `LEFT OUTER JOIN`.
Recommendation	`includes` is generally preferred due to its flexibility and intelligence.

Practical Tips and Considerations

Always profile your queries to identify N+1 issues. Tools like `bullet` gem can help detect these problems in development.

Use eager loading judiciously. Over-eager loading can also impact performance by fetching unnecessary data.

Consider using `pluck` when you only need specific attributes from associated records instead of loading the entire object.

When dealing with large datasets, be mindful of memory consumption when eager loading. You might need to batch your queries or use more advanced techniques like custom SQL.

Always check the generated SQL queries to understand how Active Record is fetching the data. You can use `to_sql` method to inspect the query.

```
posts = Post.includes(:user).where(users: {active: true })
puts posts.to_sql #print generated SQL
```

Advanced Scopes

Lambda Scopes

Lambda scopes allow you to define reusable query logic.

Syntax:

```
scope :scope_name, -> { where(condition: true) }
```

Example:

```
class User < ApplicationRecord
  scope :active, -> { where(active: true) }
  scope :inactive, -> { where(active: false) }
end
```

Using lambda scopes with arguments:

Syntax:

```
scope :scope_name, ->(argument) { where(column: argument) }
```

Example:

```
class Product < ApplicationRecord
  scope :expensive_than, ->(price) { where('price > ?', price) }
end
```

Calling lambda scopes:

```
User.active # Returns all active users
Product.expensive_than(100) # Returns all products with price > 100
```

Lambda scopes are lazy loaded; the query is not executed until you call it.

This allows for further chaining and composition.

Chaining Scopes

Scopes can be chained together to create more complex queries.

Example:

```
class User < ApplicationRecord
  scope :active, -> { where(active: true) }
  scope :admin, -> { where(role: 'admin') }
end
```

Chaining scopes:

```
User.active.admin # Returns all active admin users
```

Chaining with conditions:

```
User.active.where(age: 18..65) # Returns active users between 18 and 65
```

Combining scope and class methods:

```
class User < ApplicationRecord
  scope :active, -> { where(active: true) }

  def self.search(query)
    where('name LIKE ?', "%#{query}%")
  end
end


User.active.search('john') # Returns active users with 'john' in their name
```

Careful with ordering; the order of chained scopes can affect the final query.

Example:

```
User.order(:age).active # Order by age first, then filter active users
```

Dynamic Scopes

Dynamic scopes are method-based finders that allow you to create queries based on method names.

Example:

```
User.find_by_name('John') # Finds a user with the name 'John'
```

Using dynamic scopes with multiple attributes:

```
User.find_by_name_and_active('John', true)
# Finds a user with name 'John' and active status true
```

Dynamic scopes also work for `find_or_create_by` and `find_or_initialize_by`:

```
User.find_or_create_by_name('John')
# Finds a user with name 'John' or creates one if it doesn't exist
```

Be cautious with dynamic scopes as they can lead to security vulnerabilities if user input is directly used in the method name.

It is recommended to use strong parameters to sanitize inputs.

Extending Active Record with custom methods

You can extend ActiveRecord::Base to add custom methods to all your models. This is typically done in an initializer.

```
# config/initializers/active_record_extensions.rb
ActiveSupport.on_load(:active_record) do
  module ActiveRecord
    module Extensions
      def awesome_method
        puts "This is awesome!"
      end
    end
    include Extensions
  end
end
```

After defining the extension, it will be available in all your models:

```
class User < ApplicationRecord
end

User.new.awesome_method # => "This is awesome!"
```

Use this approach sparingly to keep models focused and avoid polluting the base class with too many unrelated methods.

Eager Loading

Eager loading helps prevent N+1 queries by loading associated records in a single query.

Syntax:

```
Model.includes(:association)
```

Example:

```
User.includes(:posts).where(active: true)
# Loads all users and their posts in two queries instead of N+1 queries
```

Eager loading multiple associations:

```
User.includes(:posts, :comments)
# Loads users, posts, and comments in a minimal number of queries
```

Nested eager loading:

```
User.includes(posts: [:comments])
# Loads users, their posts, and the comments for each post
```

Using `preload` instead of `includes` forces separate queries for each association. Useful when `includes` generates complex queries.

```
User.preload(:posts)
```

Using `eager_load` performs a LEFT OUTER JOIN, which can be more efficient but may lead to data duplication if not used carefully.

```
User.eager_load(:posts)
```

Advanced querying with joins

Using `joins` to create more specific queries.

```
Article.joins(:comments).where('comments.approved = ?', true)
```

This will return all articles that have approved comments.

You can also specify `LEFT OUTER JOINS` for including records even when the association is not present.

```
Article.joins('LEFT OUTER JOIN comments ON comments.article_id = articles.id').group('articles.id')
```

Complex example using `joins` with custom SQL:

```
User.joins("INNER JOIN user_groups ON users.id = user_groups.user_id INNER JOIN groups ON groups.id = user_groups.group_id").where("groups.name = 'admins'")
```

This returns all users that are members of the 'admins' group.

Advanced Active Record Callbacks

Conditional Callbacks

`if` option	Specifies a symbol, string, or Proc. The callback will only be executed if this evaluates to true. `before_save :normalize_name, if: :name_changed?`
`unless` option	Similar to `if`, but the callback will only be executed if the condition evaluates to false. `after_create :send_welcome_email, unless: :welcome_email_sent?`

Using symbols	Referencing a method defined in the model.
	`private`
	`def name_changed?`
	`  name_previously_changed?`
	`end`

Using strings	A string that will be evaluated in the context of the model. `before_validation :ensure_name_has_value, if: "name.blank?"`
Using Procs	A Proc object that is called. Useful for more complex conditions. `before_create :set_creation_date, if: Proc.new { |record| record.created_at.nil? }`

Combining `if` and `unless`	It's generally best to avoid using both `if` and `unless` for the same callback, as it can become confusing.
	`# Avoid this:`
	`before_save :do_something, if: :condition1, unless: :condition2`

Ordered Callbacks

Callback execution order	Callbacks are generally executed in the order they are defined.
	`before_validation :callback_one`
	`before_validation :callback_two`
	In this case, `callback_one` will be executed before `callback_two`.

Impact of `halted_callback_hook`	If a `before_*` callback returns `false`, it halts the execution of subsequent callbacks and the action. This can affect the order in which validations or other logic is applied.
Explicit Ordering (gem)	Gems like `active_record-orderable` can provide more explicit control over the order of callback execution if the default order is insufficient. (Not a standard Rails feature.)

Testing Callback Order	Write tests to ensure callbacks are firing in the expected order, especially when the order is critical for data integrity or application logic.
	`it 'executes callbacks in the correct order' do`
	`  expect(instance).to receive(:callback_one).ordered`
	`  expect(instance).to receive(:callback_two).ordered`
	`  instance.run_callbacks :validation`
	`end`

Dependencies Between Callbacks	If one callback depends on the result of another, ensure the dependency is clear and the order is correct. Refactor if the dependencies become too complex.
Debugging Callback Order	Use `Rails.logger.debug` statements within the callbacks to trace their execution order during development. Alternatively, use a debugger.

Custom Callback Patterns

Creating Custom Callback Methods	Define methods that encapsulate specific logic to be executed during a particular lifecycle event.
	`class User < ApplicationRecord`
	`  before_create :generate_token`
	
	`  private`
	
	`  def generate_token`
	`  self.token = SecureRandom.hex(10)`
	`  end`
	`end`

Using Observers	Observers are a way to extract callback logic into separate classes, promoting separation of concerns. However, observers are deprecated in Rails 5.1 and removed in Rails 6.
	`# Deprecated in Rails 5.1, removed in Rails 6`
	`class UserObserver < ActiveRecord::Observer`
	`  def after_create(user)`
	`  #...`
	`  end`
	`end`

Service Objects	Move complex logic out of the model and into service objects. Callbacks can then trigger these service objects.
	`class CreateUser`
	`  def self.call(user_params)`
	`  user = User.new(user_params)`
	`  if user.save`
	`  WelcomeEmailService.new(user).send`
	`  end`
	`  end`
	`end`
	
	`class User < ApplicationRecord`
	`  after_create :call_welcome_email_service`
	
	`  private`
	
	`  def call_welcome_email_service`
	`  WelcomeEmailService.new(self).send`
	`  end`
	`end`

Asynchronous Callbacks	Use `after_commit` with `on: :create`, `on: :update`, or `on: :destroy` to perform actions *after* the database transaction is complete. Useful for sending emails or triggering other external processes. `after_commit :send_welcome_email, on: :create`

Callback Chains	Create methods that trigger other methods, allowing for a sequence of actions during a callback.
	`before_save :process_data`
	
	`private`
	
	`def process_data`
	`  step_one`
	`  step_two`
	`  step_three`
	`end`

State Machines	Use state machine gems (like `aasm` or `statesman`) to manage complex state transitions and trigger callbacks based on those transitions.
	`include AASM`
	
	`aasm do`
	`  state :idle, initial: true`
	`  state :running`
	
	`  event :start do`
	`  transitions from: :idle, to: :running, after: :do_something`
	`  end`
	`end`

Auditing changes	Implement callbacks to track changes to model attributes, logging the changes for auditing purposes. Gems like `paper_trail` simplify this.
	`# using PaperTrail`
	`has_paper_trail`

Validations

Custom Validators

Create custom validators to encapsulate complex validation logic.

Example:

```
class EmailValidator <
ActiveModel::EachValidator
  def validate_each(record, attribute,
value)
    unless value =~ /\A[^@\s]+@([^@\s]+\.)+
[^@\s]+\z/
      record.errors.add attribute,
(options[:message] || "is not an email")
    end
  end
end

class Person < ApplicationRecord
  validates :email, presence: true, email:
true
end
```

Using `validates_with`:

```
class GoodnessValidator <
ActiveModel::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors.add :base, "This person
is evil"
    end
  end
end

class Person < ApplicationRecord
  validates_with GoodnessValidator
end
```

Custom validators can accept options:

```
class ExclusionValidator <
ActiveModel::EachValidator
  def validate_each(record, attribute,
value)
    if options[:in].include?(value)
      record.errors.add attribute,
(options[:message] || "is reserved")
    end
  end
end

class Person < ApplicationRecord
  validates :username, exclusion: { in:
%w(admin superuser) }
  validates :age, exclusion: { in: 30..60,
message: 'is not allowed' }
end
```

Conditional Validations

Execute validations only under certain conditions using `:if` and `:unless`.

Example: Validate `postal_code` only if `country` is 'USA'.

```
class Address < ApplicationRecord
  validates :postal_code, presence: true,
if: :is_usa?

  def is_usa?
    country == 'USA'
  end
end
```

Using `:unless`:

```
class Article < ApplicationRecord
  validates :body, presence: true, unless:
:is_published?

  def is_published?
    published
  end
end
```

Using `:if` with a Proc:

```
class Person < ApplicationRecord
  validates :email, presence: true, if:
Proc.new { |p| p.age > 18 }
end
```

Using `:unless` with a Proc:

```
class Event < ApplicationRecord
  validates :description, presence: true,
unless: Proc.new { |e| e.name.blank? }
end
```

Validating Associations

Validate associated records using `validates_associated`.

Example: Validate associated `Address` when saving a `Person`.

```
class Person < ApplicationRecord
  has_one :address
  validates_associated :address
end

class Address < ApplicationRecord
  belongs_to :person
  validates :street, presence: true
end
```

Customize the validation process with `:on` option:

```
class Project < ApplicationRecord
  has_many :tasks
  validates_associated :tasks, on: :create
end
```

Use with custom validation methods:

```
class Order < ApplicationRecord
  has_many :line_items
  validate :validate_line_items

  private

  def validate_line_items
    line_items.each do |item|
      errors.add(:base, "Invalid line item")
unless item.valid?
    end
  end
end
```

Custom Validation Methods

Define custom validation methods for more complex logic.

Example: A method that checks if the discount is valid based on the order total.

```
class Order < ApplicationRecord
  validate :discount_is_valid

  private

  def discount_is_valid
    if discount > total
      errors.add(:discount, "cannot be
greater than total")
    end
  end
end
```

Using multiple attributes:

```
class Booking < ApplicationRecord
  validate :check_availability

  private

  def check_availability
    if start_date >= end_date
      errors.add(:start_date, "must be
before end date")
      errors.add(:end_date, "must be after
start date")
    end
  end
end
```

Adding errors to specific attributes:

```
class Product < ApplicationRecord
  validate :check_price

  private

  def check_price
    if price <= 0
      errors.add(:price, "must be greater
than zero")
    end
  end
end
```

Conditional Validation Groups

Group validations and conditionally apply them.

Example: Validate fields required for admin users only.

```
class User < ApplicationRecord
  with_options if: :is_admin? do
    validates :employee_id, presence: true
    validates :department, presence: true
  end

  def is_admin?
    role == 'admin'
  end
end
```

Using `with_options` with multiple conditions:

```
class Article < ApplicationRecord
  with_options if: Proc.new { |a|
a.published? && a.premium? } do
    validates :premium_content, presence:
true
    validates :access_code, presence: true
  end
end
```

Combining with custom validators:

```
class Event < ApplicationRecord
  with_options if: :is_special_event? do
    validates :special_requirements,
presence: true
    validates_with SpecialEventValidator
  end
end
```

Transactions and Nested Transactions for Data Integrity

Transactions

Transactions are used to ensure data integrity by grouping multiple operations into a single atomic unit. If any operation fails, the entire transaction is rolled back, preventing partial updates.

Basic Transaction:

```
ActiveRecord::Base.transaction do
  account.update!(balance: account.balance - 100)
  log.create!(message: 'Withdrawal of $100')
end
```

If `account.update!` or `log.create!` raises an exception, the entire transaction is rolled back, and no changes are persisted.

Handling Exceptions:

```
ActiveRecord::Base.transaction do
  begin
    account.update!(balance: account.balance - 100)
    log.create!(message: 'Withdrawal of $100')
  rescue => e
    puts "Transaction failed: #{e.message}"
    raise ActiveRecord::Rollback # Explicitly rollback
  end
end
```

Raising `ActiveRecord::Rollback` within a transaction block will cause the transaction to be rolled back without raising an error.

Transaction Options:

```
ActiveRecord::Base.transaction(isolation: :serializable, requires_new: true) do
  # Transaction logic here
end
```

`isolation: :serializable` - Sets the transaction isolation level to serializable, preventing certain concurrency issues.
`requires_new: true` - Forces the transaction to create a new transaction, even if one already exists.

Best Practices:
Keep transactions short and focused to minimize lock contention.
Handle exceptions within the transaction block to ensure proper rollback.
Use specific exception handling to avoid masking unexpected errors.

Transactions are crucial for maintaining data integrity in concurrent environments.

Ensure that all operations within a transaction are logically related.

Nested Transactions

Nested transactions allow you to create transactions within transactions, providing more granular control over data consistency. However, ActiveRecord only supports emulated nested transactions using savepoints.

Emulated Nested Transactions:

```
ActiveRecord::Base.transaction do
  account.update!(balance: account.balance - 50)


ActiveRecord::Base.transaction(requires_new: true) do
    log.create!(message: 'Inner transaction log')
  end
end
```

When `requires_new: true` is used, ActiveRecord creates a savepoint before the inner transaction and rolls back to the savepoint if the inner transaction fails.

Savepoints:

```
ActiveRecord::Base.transaction do
  account.update!(balance: account.balance - 25)
  savepoint 'before_log'
  begin
    log.create!(message: 'Savepoint log')
  rescue => e
    puts "Inner transaction failed: #{e.message}"
    rollback_to_savepoint 'before_log'
  end
end
```

The `savepoint` method creates a savepoint, and `rollback_to_savepoint` rolls back to that savepoint if an error occurs.

Caveats:
Emulated nested transactions using savepoints have limitations, such as not being true nested transactions at the database level.
Not all databases support savepoints; check your database documentation.

Best Practices:
Use nested transactions sparingly, as they can add complexity.
Ensure proper error handling in each nested transaction block.
Consider alternatives like smaller, independent transactions if possible.

Nested transactions should be used carefully and with a clear understanding of their limitations.

Savepoints can be very helpful in complex scenarios where partial rollbacks are needed.

Always verify that your database supports savepoints before relying on them.

Isolation Levels

Transaction isolation levels define the degree to which transactions are isolated from each other's modifications. Higher isolation levels provide more data consistency but can reduce concurrency.

Read Uncommitted:
Allows transactions to read uncommitted changes from other transactions. This is the lowest isolation level and can lead to dirty reads.

```
ActiveRecord::Base.transaction(isolation: :read_uncommitted) do
  # Transaction logic here
end
```

Read Committed:
Ensures that transactions only read committed changes from other transactions, preventing dirty reads.

```
ActiveRecord::Base.transaction(isolation: :read_committed) do
  # Transaction logic here
end
```

Repeatable Read:
Guarantees that if a transaction reads a row, subsequent reads of the same row within the same transaction will return the same value, preventing non-repeatable reads.

```
ActiveRecord::Base.transaction(isolation: :repeatable_read) do
  # Transaction logic here
end
```

Serializable:
The highest isolation level, ensuring that transactions are executed as if they were executed serially, preventing phantom reads and all other concurrency issues.

```
ActiveRecord::Base.transaction(isolation: :serializable) do
  # Transaction logic here
end
```

Choosing the Right Isolation Level:
The choice of isolation level depends on the application's requirements. Serializable provides the highest level of data consistency but can reduce concurrency. Read Committed is often a good balance between consistency and concurrency.

Best Practices:
Understand the trade-offs between isolation levels and concurrency.
Use the appropriate isolation level for each transaction based on its specific requirements.
Avoid using Read Uncommitted in most cases due to the risk of dirty reads.

Carefully select isolation levels to optimize for both data integrity and application performance.

Be aware of the default isolation level of your database system.

Connection Management

Proper connection management is crucial for efficient and reliable database interactions. ActiveRecord provides tools for managing database connections, including connection pooling and connection sharing.

Connection Pooling:
ActiveRecord uses connection pooling to maintain a pool of database connections that can be reused by different threads or processes, reducing the overhead of establishing new connections for each request.

```
# Configuration in database.yml
pool: 5 # Maximum number of connections in the pool
```

Connection Sharing:
In threaded environments, ActiveRecord can share database connections between threads, further reducing the number of connections required.

```
ActiveRecord::Base.connection_pool.with_connection do |connection|
  # Use the connection here
end
```

Connection Timeout:
Set a connection timeout to prevent long-running operations from holding connections indefinitely.

```
# Configuration in database.yml
reaping_frequency: 10 # Check for idle connections every 10 seconds
```

Connection Disconnection:
Explicitly disconnect connections when they are no longer needed to free up resources.

```
ActiveRecord::Base.connection.disconnect!
```

Best Practices:
Configure the connection pool size based on the application's concurrency and database server capacity.
Use connection sharing in threaded environments to reduce connection overhead.
Set appropriate connection timeouts to prevent resource exhaustion.
Monitor connection usage to identify and resolve connection leaks.

Efficient connection management is key to maintaining application performance and stability.

Regularly review and adjust connection pool settings based on application load and performance metrics.

Avoid holding connections open for extended periods to minimize resource consumption.

Idempotency

Idempotency ensures that an operation can be applied multiple times without changing the result beyond the initial application. This is crucial for handling retries and ensuring data consistency in distributed systems.

Ensuring Idempotency:
Use unique constraints to prevent duplicate records.

```
class CreateOrders < ActiveRecord::Migration[7.0]
  def change
    create_table :orders do |t|
      t.string :order_id, null: false
      t.timestamps
    end
    add_index :orders, :order_id, unique: true
  end
end
```

Optimistic Locking:
Use optimistic locking to prevent concurrent updates from overwriting each other.

```
class Order < ApplicationRecord
  validates :order_id, uniqueness: true
end
```

Idempotent Operations:
Design operations to be idempotent by checking if the operation has already been performed before applying it.

```
def process_payment(payment_id)
  payment = Payment.find_by(payment_id: payment_id)
  return if payment&.processed?

  # Process the payment here
  payment.update!(processed: true)
end
```

Best Practices:
Use unique constraints to prevent duplicate records.
Implement optimistic locking to handle concurrent updates.
Design operations to be idempotent by checking their current state before applying them.
Use a combination of techniques to ensure idempotency in different scenarios.

Idempotency is essential for building resilient and reliable applications.

Implement idempotent operations to handle retries and ensure data consistency.

Combine unique constraints, optimistic locking, and idempotent operations for comprehensive protection.

Locking Mechanisms (Optimistic and Pessimistic Locking)

Optimistic Locking

Optimistic locking assumes that conflicts are rare. It checks for modifications made by another process before saving changes.
Uses a `lock_version` attribute to track updates. If `lock_version` has changed, an `ActiveRecord::StaleObjectError` is raised.

Rails automatically adds `lock_version` to your model if you create a migration like this:

```
add_column :your_models, :lock_version, :integer, default: 0
```

When a record is fetched, its `lock_version` is stored. Before updating, Active Record verifies that the `lock_version` in the database matches the `lock_version` of the fetched record.

Example usage:

```
user = User.find(1)
user.email = 'new_email@example.com'
user.save! # Raises ActiveRecord::StaleObjectError if lock_version is stale
```

Handling `ActiveRecord::StaleObjectError`:

```
begin
  user = User.find(1)
  user.email = 'new_email@example.com'
  user.save!
rescue ActiveRecord::StaleObjectError
  # Handle the conflict, e.g., retry or merge changes
  puts 'Record has been updated by another user.'
end
```

Optimistic locking is suitable for applications where conflicts are infrequent. It avoids holding locks for extended periods, improving concurrency.

Pessimistic Locking

Pessimistic locking explicitly locks a database row to prevent concurrent updates.
It's suitable when conflicts are likely and data integrity is critical.
Uses database-level locking mechanisms.

Rails provides the `lock` method to acquire a pessimistic lock:

```
user = User.find(1).lock!
user.email = 'new_email@example.com'
user.save!
```

The `lock!` method adds a `FOR UPDATE` clause to the SQL query, which locks the row until the transaction is committed or rolled back.

```
SELECT * FROM users WHERE id = 1 LIMIT 1 FOR UPDATE
```

Pessimistic locking should be used within a transaction to ensure atomicity:

```
User.transaction do
  user = User.find(1).lock!
  user.email = 'new_email@example.com'
  user.save!
end
```

Locking specific records:

```
users = User.where(active: true).lock!
```

Considerations: Pessimistic locking can reduce concurrency if locks are held for too long. Use it judiciously.

Comparison

Optimistic Locking	Pessimistic Locking
Assumes conflicts are rare.	Assumes conflicts are likely.
Uses `lock_version` column.	Uses database-level locks.
Raises `ActiveRecord::StaleObjectError` on conflict.	Blocks other transactions until the lock is released.
Better concurrency in low-conflict scenarios.	Guarantees data integrity in high-conflict scenarios.
Requires conflict resolution logic.	Can lead to deadlocks if not managed carefully.

When to use Optimistic Locking

Use optimistic locking when:

- Conflicts are rare and the overhead of pessimistic locking is not justified.
- Concurrency is a priority, and you're willing to handle `StaleObjectError` exceptions.
- You want to avoid holding database locks for extended periods.

Examples:

- Updating user profiles where concurrent updates are unlikely.
- Modifying infrequently accessed settings.

When to use Pessimistic Locking

Use pessimistic locking when:

- Conflicts are likely and data integrity is paramount.
- You need to ensure that a series of operations are performed atomically without interference.
- You can tolerate reduced concurrency in exchange for data consistency.

Examples:

- Processing financial transactions where concurrent updates could lead to incorrect balances.
- Managing inventory levels where precise counts are essential.

Locking and Transactions

It's crucial to use locking mechanisms within transactions to ensure atomicity and consistency.

Example (Pessimistic Locking within a Transaction):

```
ActiveRecord::Base.transaction do
  account = Account.find(account_id, lock: true) #Explicitly lock the record
  account.balance -= amount
  account.save!
  other_account = Account.find(other_account_id, lock: true)
  other_account.balance += amount
  other_account.save!
end
```

Example (Optimistic Locking and Retries):

```
def update_record(record)
  begin
    record.update!(attributes)
  rescue ActiveRecord::StaleObjectError
    record.reload #Reload the record
    # Resolve conflicts or retry
    update_record(record) #Recursive call until success or max retries
  end
end
```

Transactions ensure that all operations within the block are treated as a single atomic unit. If any operation fails, the entire transaction is rolled back, maintaining data integrity.

Advanced Active Record Migrations

Reversible Migrations

Reversible migrations allow you to define both the `up` and `down` operations, making it easy to rollback changes.

```
class CreateProducts <
ActiveRecord::Migration[7.1]
  def up
    create_table :products do |t|
      t.string :name
      t.text :description
      t.decimal :price

      t.timestamps
    end
  end

  def down
    drop_table :products
  end
end
```

Alternatively, use `change` method for reversible migrations.

```
class CreateProducts <
ActiveRecord::Migration[7.1]
  def change
    create_table :products do |t|
      t.string :name
      t.text :description
      t.decimal :price

      t.timestamps
    end
  end
end
```

For operations that can't be automatically reversed, raise `IrreversibleMigration`.

```
class AddAdminFlagToUsers <
ActiveRecord::Migration[7.1]
  def up
    add_column :users, :admin, :boolean,
default: false
  end

  def down
    raise
ActiveRecord::IrreversibleMigration
  end
end
```

Changing Existing Tables

`add_column :table_name, :column_name, :column_type, options`	Adds a new column to the table. **Example:** `add_column :users, :email, :string, null: false, default: ''`
`remove_column :table_name, :column_name`	Removes an existing column from the table. **Example:** `remove_column :users, :email`
`rename_column :table_name, :old_column_name, :new_column_name`	Renames an existing column. **Example:** `rename_column :users, :username, :name`
`change_column :table_name, :column_name, :column_type, options`	Changes the data type or options of an existing column. **Example:** `change_column :products, :price, :decimal, precision: 8, scale: 2`

Adding and Removing Indexes

`add_index :table_name, :column_name(s), options`	Adds an index to a column or a set of columns. **Example:** `add_index :users, :email, unique: true`
`remove_index :table_name, :column_name(s), options`	Removes an index. **Example:** `remove_index :users, :email`
`add_index :table_name, [:col1, :col2], unique: true`	Creates a composite index for multiple columns. **Example:** `add_index :orders, [:customer_id, :order_date]`

Using SQL Directly

Sometimes, you need to execute raw SQL queries within migrations.

```
class AddSomeData <
ActiveRecord::Migration[7.1]
  def up
    execute "INSERT INTO products (name,
description, price) VALUES ('Example
Product', 'A sample product', 9.99)"
  end

  def down
    execute "DELETE FROM products WHERE name
= 'Example Product'"
  end
end
```

Use `execute` method with caution, especially when the operation is not easily reversible. Consider using `ActiveRecord::IrreversibleMigration`.

Data Migrations

Data migrations involve modifying existing data as part of the schema change. This is often combined with schema changes.

```
class UpdateProductPrices <
ActiveRecord::Migration[7.1]
  def up
    Product.all.each do |product|
      product.update_attribute(:price,
product.price * 1.1) # Increase price by 10%
    end
  end

  def down
    Product.all.each do |product|
      product.update_attribute(:price,
product.price / 1.1) # Revert price change
    end
  end
end
```

Ensure your data migrations are idempotent and reversible for safety.
Consider using `say_with_time` helper to measure execution time.

Using Transactions

Wrap your migrations in a transaction to ensure that all changes are applied or rolled back together, maintaining data consistency.

```
class ComplexMigration <
ActiveRecord::Migration[7.1]
  def change
    transaction do
      add_column :users, :temp_email,
:string
      # Some data manipulation here
      rename_column :users, :temp_email,
:email
    end
  end
end
```

If any part of the migration fails, the entire transaction will be rolled back.

Advanced Active Record: Executing Raw SQL Safely

Executing Raw SQL Queries

Active Record provides a way to execute raw SQL queries when the framework's built-in methods are insufficient. However, it's crucial to sanitize inputs to prevent SQL injection vulnerabilities.

Use `ActiveRecord::Base.connection.execute(sql)` to execute raw SQL.

Example:

```
sql = "SELECT * FROM users WHERE name =
'John Doe'"
results =
ActiveRecord::Base.connection.execute(sql)
```

Warning: This example is vulnerable to SQL injection if the name is taken from user input.

Using `sanitize_sql_array`

Alternatively, `sanitize_sql_array` can be used to sanitize SQL queries.

This method constructs a SQL query with proper escaping.

Example:

```
name = params[:name]
sql_array = ["SELECT * FROM users WHERE name
= ?", name]
safe_sql =
ActiveRecord::Base.sanitize_sql_array(sql_ar
ray)
results =
ActiveRecord::Base.connection.execute(safe_s
ql)
```

This method is especially useful when constructing more complex queries dynamically.

SQL Injection Prevention

To prevent SQL injection, use parameterized queries. Active Record will automatically escape and sanitize the inputs.

Use placeholders (`?` for positional, or named placeholders like `:name`) and pass the values as arguments.

Using Positional Placeholders

```
name = params[:name]
sql = "SELECT * FROM users WHERE name = ?"
results =
ActiveRecord::Base.connection.execute(sql,
[name])
```

In this example, the `?` placeholder is replaced by the value of `name`. Active Record ensures that `name` is properly escaped to prevent SQL injection.

```
sql = "SELECT * FROM products WHERE price >
? AND category = ?"
results =
ActiveRecord::Base.connection.execute(sql,
[min_price, category])
```

Multiple placeholders can be used. Ensure the order of values in the array matches the order of placeholders in the SQL query.

Considerations

- Always sanitize user inputs when using raw SQL.
- Parameterized queries are the preferred method to prevent SQL injection.
- Avoid concatenating strings directly into the SQL query.
- Review raw SQL queries carefully to ensure they are secure.
- Use named placeholders for better readability and maintainability.

Using Named Placeholders

```
name = params[:name]
sql = "SELECT * FROM users WHERE name =
:name"
results =
ActiveRecord::Base.connection.execute(sql, {
name: name })
```

Here, `:name` is a named placeholder that is replaced by the value associated with the `name` key in the hash.

```
sql = "SELECT * FROM products WHERE price >
:min_price AND category = :category"
results =
ActiveRecord::Base.connection.execute(sql, {
min_price: min_price, category: category })
```

Named placeholders improve readability, especially with multiple parameters. The order in the hash does not matter.

Single Table Inheritance (STI) and Polymorphic Associations

Single Table Inheritance (STI) Basics

Single Table Inheritance (STI) allows you to store multiple subclasses of a model in a single database table.

Key Concepts:

- A single table stores all subclasses.
- A `type` column distinguishes between subclasses.

Defining STI:

Create a base class and subclasses that inherit from it. Add a `type` column to the database table for the base class.

```
class Payment < ApplicationRecord
end

class CreditCardPayment < Payment
end

class BankTransferPayment < Payment
end
```

Migration:

The migration should create the `payments` table with a `type` column (string).

```
class CreatePayments <
ActiveRecord::Migration[7.1]
  def change
    create_table :payments do |t|
      t.string :type
      t.decimal :amount
      t.timestamps
    end
  end
end
```

Creating Records:

When creating records, the `type` column is automatically set.

```
credit_card_payment =
CreditCardPayment.create(amount: 50.00)
bank_transfer_payment =
BankTransferPayment.create(amount: 100.00)

CreditCardPayment.all # => Returns only
CreditCardPayment instances
Payment.all # => Returns all Payment
instances (including subclasses)
```

Querying:

You can query based on the `type` column.

```
Payment.where(type: 'CreditCardPayment') #
=> Returns CreditCardPayment instances
```

STI Gotchas and Considerations

Null `type` values:

If a record has a null `type`, it will be instantiated as the base class.

Table bloat:

All attributes for all subclasses are in one table, which can lead to many null columns and larger table sizes if the subclasses have very different fields.

Database indexes:

Add indexes to columns frequently used in queries to improve performance.

```
class AddIndexToPayments <
ActiveRecord::Migration[7.1]
  def change
    add_index :payments, :type
  end
end
```

When to avoid STI:

Avoid STI if subclasses have significantly different attributes, as this can lead to a sparse table with many null values. Consider using separate tables with a shared interface or polymorphic associations instead.

Testing STI:

Ensure that you thoroughly test each subclass to verify that they behave as expected within the STI structure.

Potential performance issues:

Can occur when the table grows very large, especially if there are many columns and subclasses. Monitor query performance and consider denormalization strategies if needed.

Polymorphic Associations Basics

Polymorphic associations allow a model to belong to different types of other models using a single association.

Key Concepts:

- A single association can connect to multiple models.
- Uses `*_id` and `*_type` columns in the database.

Defining Polymorphic Associations:
Add a polymorphic association to the model that will belong to different types of models.

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic: true
end

class Article < ApplicationRecord
  has_many :comments, as: :commentable
end

class Event < ApplicationRecord
  has_many :comments, as: :commentable
end
```

Migration:
The migration should create the `comments` table with `commentable_id` (integer) and `commentable_type` (string) columns.

```
class CreateComments <
ActiveRecord::Migration[7.1]
  def change
    create_table :comments do |t|
      t.text :body
      t.references :commentable,
polymorphic: true, index: true
      t.timestamps
    end
  end
end
```

Creating Records:
When creating records, both `*_id` and `*_type` columns are set.

```
article = Article.create(title: 'Polymorphic
Associations')
event = Event.create(name: 'Tech
Conference')

comment_for_article =
article.comments.create(body: 'Great
article!')
comment_for_event =
event.comments.create(body: 'Excited to
attend!')
```

Accessing Associations:
You can access the associated object through the polymorphic association.

```
comment_for_article.commentable # => Returns
the Article instance
comment_for_event.commentable # => Returns
the Event instance
```

Polymorphic Associations Querying and Usage

Querying:
You can query based on the `*_type` and `*_id` columns.

```
Comment.where(commentable_type: 'Article',
commentable_id: article.id)
Comment.where(commentable: article)
```

Eager Loading:
Use eager loading to avoid N+1 queries when accessing polymorphic associations.

```
articles = Article.includes(:comments)
articles.each { |article|
article.comments.each { |comment|
comment.body } }
```

Benefits:

- Flexibility in associating models.
- Reduced code duplication.
- Simplified data model for certain relationships.

Considerations:

- Can complicate queries if not properly indexed.
- Requires careful planning to ensure data integrity.

Inverse Associations:
If you need to update the `commentable` association from the `Comment` model, you can use the `inverse_of` option.

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic:
true, inverse_of: :comments
end
```

Advanced Polymorphic Association Techniques

Customizing `*_type` values:
You can customize the values stored in the `*_type` column using a before_validation callback.

```
class Image < ApplicationRecord
  belongs_to :imageable, polymorphic: true
  before_validation :set_imageable_type

  private

  def set_imageable_type
    self.imageable_type =
imageable.class.name
  end
end
```

Validations:
Add validations to ensure that the associated object exists and is of the correct type.

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic: true
  validates :commentable, presence: true
end
```

Scopes:
Define scopes to easily query comments for specific commentable types.

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic: true
  scope :for_articles, -> {
where(commentable_type: 'Article') }
end

Comment.for_articles # => Returns comments
for articles
```

Polymorphic Joins:
When querying across multiple tables, use polymorphic joins to efficiently retrieve associated records.

Testing Polymorphic Associations:
Ensure comprehensive testing of polymorphic associations, covering different associated models and edge cases.

STI with Polymorphism:
You can combine STI and polymorphic associations, for example, having different types of comments (STI) associated with different models (polymorphism).

Query Optimization and Performance Tuning Techniques

Eager Loading (N+1 Problem)

The N+1 query problem occurs when Active Record executes one query to fetch a collection of records, and then performs additional queries for each record in the collection to fetch associated data.

Example (without eager loading):

```
@posts = Post.all
@posts.each do |post|
  puts post.user.name  # Triggers N+1 queries
end
```

Solution: Eager Loading with `includes`

```
@posts = Post.includes(:user).all
@posts.each do |post|
  puts post.user.name  # No additional queries
end
```

`includes` uses LEFT OUTER JOIN or separate queries to load associations, optimizing the query count.

Eager Loading with Multiple Associations

```
@posts = Post.includes(:user, :comments).all
```

Nested Eager Loading

```
@posts = Post.includes(user: :profile).all
```

Conditional Eager Loading

```
@posts = Post.includes(:user).where(users: { active: true })
```

`preload` vs `eager_load` vs `includes`

- `includes`: Chooses the most efficient loading strategy (usually LEFT OUTER JOIN or separate queries).
- `preload`: Loads associations in separate queries.
- `eager_load`: Forces the use of a LEFT OUTER JOIN.

Using `pluck` and `select`

`pluck` is used to retrieve specific columns directly from the database as an array, avoiding the instantiation of Active Record objects.

Example:

```
User.pluck(:id, :email) # => [[1, 'user1@example.com'], [2, 'user2@example.com']]
```

`select` is used to specify which columns to retrieve, useful for reducing the amount of data transferred from the database.

Example:

```
User.select(:id, :email) # Returns Active Record objects with only id and email attributes
```

When to use `pluck` vs `select`

- Use `pluck` when you only need specific column values and don't need Active Record object functionality.
- Use `select` when you need Active Record objects but want to limit the columns retrieved.

Chaining with `pluck` and `select`

```
User.where(active: true).pluck(:email) # => ['user1@example.com', 'user2@example.com']
```

Using `distinct` with `pluck`

```
User.pluck(:email).uniq # => ['user1@example.com', 'user2@example.com']
Post.distinct.pluck(:category) # => ['news', 'tutorial']
```

Batch Processing

Batch processing is essential for handling large datasets efficiently, avoiding memory issues and improving performance.

`find_each`

Iterates over a large number of records in batches, loading each batch into memory.

```
User.find_each(batch_size: 1000) do |user|
  # Process each user
end
```

`find_in_batches`

Similar to `find_each`, but yields an array of records for each batch.

```
User.find_in_batches(batch_size: 1000) do |users|
  # Process each batch of users
  users.each { |user| ... }
end
```

`in_batches`

Returns an Enumerable that can be chained with other methods.

```
User.where(active: true).in_batches(of: 500).each_record do |user|
  # Process each user
end
```

Updating in Batches

```
User.find_in_batches(batch_size: 1000) do |users|
  User.transaction do
    users.each { |user| user.update(status: 'processed') }
  end
end
```

Important Considerations

- Always use transactions when performing batch updates to ensure data consistency.
- Adjust `batch_size` based on available memory and processing capacity.

Optimistic and Pessimistic Locking

Optimistic Locking

Assumes that multiple users are unlikely to edit the same record simultaneously. Uses a `lock_version` column to detect conflicting updates.

Add `lock_version` column to table

```
rails generate migration
AddLockVersionToPosts lock_version:integer
```

Usage

```
post = Post.find(1)
post.update(title: 'New Title') # Raises
ActiveRecord::StaleObjectError if
lock_version has changed
```

Pessimistic Locking

Locks a record for exclusive access until the transaction is complete, preventing other users from modifying it.

Usage

```
Post.transaction do
  post = Post.lock.find(1)
  post.update(title: 'New Title')
end
```

When to use Optimistic vs Pessimistic Locking

- Use Optimistic Locking when conflicts are rare and you want to minimize database locking overhead.
- Use Pessimistic Locking when conflicts are frequent and data integrity is critical.

Customizing Pessimistic Locking

```
Post.transaction do
  post = Post.lock('FOR UPDATE
NOWAIT').find(1)
  post.update(title: 'New Title')
end
```

Handling `StaleObjectError`

```
begin
  post = Post.find(1)
  post.update(title: 'New Title')
rescue ActiveRecord::StaleObjectError
  # Handle conflict (e.g., reload record and
retry)
end
```

Using Counter Caches

Counter caches store the number of associated records directly in the parent table, avoiding the need to query the associated table for the count.

Example:

Add a `comments_count` column to the `posts` table.

```
rails generate migration
AddCommentsCountToPosts
comments_count:integer
```

Update the `belongs_to` association

```
class Comment < ApplicationRecord
  belongs_to :post, counter_cache: true
end
```

Accessing the counter cache

```
post = Post.find(1)
puts post.comments_count # No additional
query needed
```

Resetting the counter cache

If you add the counter cache to an existing application, you'll need to initialize the counter.

```
Post.find_each do |post|
  Post.reset_counters(post.id, :comments)
end
```

Custom Counter Cache Column

```
class Comment < ApplicationRecord
  belongs_to :post, counter_cache:
:approved_comments_count
end
```

Advanced Active Record: Lazy Loading, Caching, and Memoization

Lazy Loading (N+1 Problem)

Lazy loading is the default behavior in Active Record where associated data is only loaded when it's accessed. This can lead to the N+1 problem.

Explanation: When you iterate through a collection of records and access an associated record for each, Active Record might execute one query to fetch the initial records (1 query) and then one query for each associated record (N queries).

Example (N+1 Problem):

```
users = User.all
users.each do |user|
  puts user.posts.count # Triggers a new query for each user
end
```

Consequences: This results in many database queries, significantly slowing down the application.

Eager Loading (Solution to N+1)

Eager loading is a technique to load associated records in a single query, mitigating the N+1 problem.

Methods:

- `includes`
- `preload`
- `eager_load`

Using `includes`:

```
users = User.includes(:posts).all
users.each do |user|
  puts user.posts.count # No additional queries
end
```

`includes` is smart and will use `LEFT OUTER JOIN` or separate queries based on the situation.

Using `preload`:

```
users = User.preload(:posts).all
users.each do |user|
  puts user.posts.count # No additional queries
end
```

`preload` always uses separate queries.

Using `eager_load`:

```
users = User.eager_load(:posts).all
users.each do |user|
  puts user.posts.count # No additional queries
end
```

`eager_load` forces a `LEFT OUTER JOIN`.

Caching

Fragment Caching:

Cache portions of a view.

```
<% cache @user do %>
  <%= render @user %>
<% end %>
```

Action Caching:

Cache the entire result of an action. Less common now.

```
class ProductsController < ApplicationController
  caches_action :index, expires_in: 1.hour
end
```

Low-Level Caching:

Directly interact with the cache store.

```
Rails.cache.fetch("user-#{user.id}", expires_in: 12.hours) do
  user.posts.to_a
end
```

Cache Stores:

- `memory_store` (Not for production)
- `file_store` (Good for single server)
- `mem_cache_store` (Popular, requires memcached)
- `redis_cache_store` (Requires Redis)

Memoization

Definition:

Memoization is a technique to store the result of an expensive function call and return the cached result when the same inputs occur again.

Implementation:

```
def expensive_operation
  @expensive_operation ||= begin
    # Perform expensive calculation here
    result = some_expensive_calculation
    result
  end
end
```

Usage with Associations:

```
class User < ApplicationRecord
  def visible_posts
    @visible_posts ||= posts.where(visible: true).to_a
  end
end
```

Benefits:

Reduces redundant calculations and database queries, improving performance.

Caveats:

Be careful with mutable objects. The cached value might become outdated if the object is modified.

Counter Caching

Counter caching is a feature where a column is added to a parent model to cache the count of associated records, reducing the need to query the associated table for a count every time.

How it Works: Active Record automatically increments or decrements the counter cache column when associated records are created or destroyed.

Example:

Assume a `User` has many `posts`. Add a `posts_count` column to the `users` table.

```
class AddPostsCountToUsers < ActiveRecord::Migration[6.0]
  def change
    add_column :users, :posts_count, :integer, default: 0
  end
end
```

Configuration in Model:

```
class Post < ApplicationRecord
  belongs_to :user, counter_cache: true
end
```

Accessing the Count:

```
user = User.find(1)
puts user.posts_count # Access the cached count
```

Benefits: Greatly reduces the number of queries when displaying counts of associated records.

Note:

For existing data, you may need to manually update the counter cache:

```
User.find_each { |user|
User.reset_counters(user.id, :posts) }
```

Advanced Active Record: Error Handling, Debugging, and Logging

Validation Errors

Active Record provides built-in validation features to ensure data integrity. When validations fail, the `errors` object is populated.

`object.valid?` - Runs validations and returns true if no errors are found, false otherwise.

`object.errors` - Returns an `ActiveModel::Errors` object containing all validation errors.

`object.errors.full_messages` - Returns an array of human-readable error messages.

`object.errors[:attribute]` - Returns an array of errors for a specific attribute.

Example:

```
user = User.new(name: nil, email: 'invalid_email')
user.valid? # => false
user.errors.full_messages # => ["Name can't be blank", "Email is invalid"]
user.errors[:name] # => ["can't be blank"]
```

To display validation errors in a Rails view:

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@user.errors.count, "error") %> prohibited this user from being saved:</h2>

    <ul>
    <% @user.errors.full_messages.each do |message| %>
      <li><%= message %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

Debugging with Rails Logger

Rails provides a built-in logger to output debugging information. You can access it via `Rails.logger`.

`Rails.logger.debug("message")` - Logs a debug message.

`Rails.logger.info("message")` - Logs an informational message.

`Rails.logger.warn("message")` - Logs a warning message.

`Rails.logger.error("message")` - Logs an error message.

`Rails.logger.fatal("message")` - Logs a fatal error message.

Example:

```
Rails.logger.debug("Processing the request...")
user = User.find_by(id: params[:id])
if user.nil?
  Rails.logger.warn("User not found with id: #{params[:id]}")
else
  Rails.logger.info("User found: #{user.name}")
end
```

Query Debugging

Debugging Active Record queries is crucial for optimizing performance and identifying issues.

`ActiveRecord::Base.logger` - Configures the logger for Active Record queries. By default it uses Rails logger.

Enable logging in `config/database.yml` by setting `logger: logger: <%= Logger.new(STDOUT) %>`.

`puts` queries in console: `ActiveRecord::Base.logger = Logger.new(STDOUT)` in rails console.

Use `explain` to analyze query performance:

```
user = User.find(1)
puts user.posts.where(published: true).explain
```

The `explain` output shows how the database executes the query, helping you identify potential bottlenecks (e.g., missing indexes).

Transaction Handling and Error Rollback

Active Record transactions ensure data consistency by grouping multiple operations into a single atomic unit. If any operation fails, the entire transaction is rolled back.

`ActiveRecord::Base.transaction do ... end` - Wraps a block of code in a transaction.

Example:

```
ActiveRecord::Base.transaction do
  account.update!(balance: account.balance - 100)
  order.update!(status: 'paid')
end
```

If any exception is raised within the transaction block (e.g., due to a validation failure), the transaction is automatically rolled back.

You can manually trigger a rollback using `raise ActiveRecord::Rollback`.

Example of manual rollback:

```
ActiveRecord::Base.transaction do
  account.update!(balance: account.balance - 100)
  if order.total > 1000
    raise ActiveRecord::Rollback, "Order total exceeds limit"
  end
  order.update!(status: 'paid')
end
```

Modularizing Code with Concerns and Service Objects

Concerns: Introduction

Concerns are modules that encapsulate reusable code, promoting the DRY (Don't Repeat Yourself) principle.

They help organize large models by extracting specific functionalities into separate files.

Concerns are typically placed in the `app/models/concerns` directory.

To include a concern in a model, use the `include` keyword:

```
class ModelName < ApplicationRecord
  include ConcernName
end
```

When naming concern files, use snake_case (e.g., `searchable.rb`).

The corresponding module name should be in CamelCase (e.g., `Searchable`).

Concerns can define methods, scopes, validations, and callbacks that become part of the including model.

Example:

```
# app/models/concerns/searchable.rb
module Searchable
  extend ActiveSupport::Concern

  included do
    scope :search, -> (query) { where('name
LIKE ?', "%#{query}%") }
  end
end

# app/models/product.rb
class Product < ApplicationRecord
  include Searchable
end

Product.search('example') # => Returns
products matching the search query
```

Concerns: Best Practices

Ensure concerns have a single, well-defined responsibility to maintain clarity and reusability.

Use the `included` block to inject code into the model class when the concern is included.

This is where you define scopes, validations, and callbacks that should be added to the model.

```
module Commentable
  extend ActiveSupport::Concern

  included do
    has_many :comments, as: :commentable
  end
end
```

Avoid concerns that are too specific to a single model. Aim for generic, reusable functionality.

Test concerns independently to ensure they function correctly before including them in models.

Use `class_methods` block to define class-level methods in concerns.

```
module Votable
  extend ActiveSupport::Concern

  class_methods do
    def popular
      where('votes > 10')
    end
  end
end
```

Service Objects: Introduction

Service objects encapsulate complex business logic that doesn't naturally belong in models, controllers, or views.

They promote separation of concerns and improve code testability and maintainability.

Service objects are plain Ruby objects (POROs) that typically perform a single, well-defined operation.

Service objects are often placed in the `app/services` directory, but this is just a convention.

A typical service object has a public method (often called `call`) that executes the business logic.

Example:

```
# app/services/create_user.rb
class CreateUser
  def initialize(params)
    @params = params
  end

  def call
    User.create!(@params)
  rescue ActiveRecord::RecordInvalid => e
    OpenStruct.new(success?: false, error:
e.message)
  else
    OpenStruct.new(success?: true, user:
user)
  end
end

# Usage in controller
result = CreateUser.new(params).call

if result.success?
  # Handle success
else
  # Handle failure
end
```

Service Objects: Benefits

Improved code organization: Service objects keep controllers and models lean by extracting complex logic.

Increased testability: Service objects are easier to test in isolation compared to controller actions or model methods.

Enhanced reusability: Service objects can be reused across multiple controllers or even different parts of the application.

Reduced complexity: Breaking down complex operations into smaller service objects makes the code more readable and maintainable.

Clear separation of concerns: Service objects enforce a clear separation between the presentation layer (controllers) and the business logic.

Transaction Management: Service objects are excellent places to wrap operations in database transactions to ensure data consistency.

Service Objects: Best Practices

Each service object should perform a single, well-defined operation. Avoid creating large, monolithic service objects.
Keep service objects stateless whenever possible. Pass all necessary data as arguments to the `call` method.
Use meaningful names for service objects that clearly indicate their purpose (e.g., `CreateUser`, `SendEmail`, `ProcessPayment`).
Handle exceptions and errors gracefully within the service object. Return a consistent response format (e.g., using `OpenStruct`) to indicate success or failure.
Consider using dependency injection to pass dependencies (e.g., other service objects, repositories) into the service object.
Test service objects thoroughly with unit tests to ensure they function correctly under various conditions.

Custom Attribute Types and Serialization Strategies

Custom Attribute Types

Active Record allows you to define custom attribute types to handle specific data formats or validation logic. This can simplify your models and encapsulate complex behavior.

Define a custom type by creating a class that inherits from `ActiveRecord::Type::Value` or `ActiveRecord::Type::Serialized`.

The `cast` method transforms raw input (from the database or user) into the appropriate Ruby object. The `serialize` method converts the Ruby object back into a database-friendly value.

Example: A custom type for handling encrypted strings.

```
class EncryptedString <
ActiveRecord::Type::Value
  def cast(value)
    return nil if value.nil?
    # Assuming you have an
encryption/decryption mechanism
    decrypt(value)
  end

  def serialize(value)
    return nil if value.nil?
    encrypt(value)
  end

  private

  def encrypt(value)
    # Encryption logic here
    "encrypted_#{value}"
  end

  def decrypt(value)
    # Decryption logic here
    value.gsub("encrypted_", "")
  end
end
```

Register the custom type:

```
ActiveRecord::Type.register(:encrypted_strin
g, EncryptedString)
```

Use the custom type in your model:

```
class User < ApplicationRecord
  attribute :secret, :encrypted_string
end
```

Serialization Strategies

Serialization is the process of converting Ruby objects into a format that can be stored in the database (e.g., JSON, YAML). Active Record provides built-in serialization capabilities.

The `serialize` method in Active Record allows you to store complex Ruby objects (e.g., arrays, hashes) in a single database column. The column type should be `text` or `string`.

Example: Serializing a hash to YAML:

```
class Preferences < ApplicationRecord
  serialize :settings, Hash
end
```

You can specify a different coder (e.g., JSON) if needed:

```
class Preferences < ApplicationRecord
  serialize :settings, JSON
end
```

When the `settings` attribute is accessed, Active Record automatically deserializes the YAML or JSON data into a Ruby hash. When the `settings` attribute is modified, Active Record serializes the hash back into YAML or JSON before saving it to the database.

Use serialization for simple data structures. For more complex or frequently queried data, consider using separate columns or a dedicated data store.

Overwriting Accessors

You can overwrite the default getter and setter methods (accessors) for Active Record attributes to add custom behavior.

This allows you to perform actions before or after getting or setting the attribute value (e.g., formatting, validation, logging).

Example: Custom getter and setter for a `name` attribute:

```
class User < ApplicationRecord
  def name
    # Custom getter logic
    read_attribute(:name).upcase
  end

  def name=(new_name)
    # Custom setter logic
    write_attribute(:name, new_name.strip)
  end
end
```

`read_attribute(:attribute_name)` is used to read the raw value from the database.

`write_attribute(:attribute_name, value)` is used to set the value to be saved to the database.

By overwriting the accessors, you change the attribute behavior, so use it with caution and make sure it aligns with model logic.

Attribute API

Active Record provides a powerful Attribute API that allows you to define attributes on your models without corresponding database columns. These are often referred to as virtual attributes.

The `attribute` method allows you to define these attributes, along with their type. This enables type casting and other attribute-related features.

Example: Defining a virtual attribute `full_name`:

```
class User < ApplicationRecord
  attribute :full_name, :string

  def full_name
    "#{first_name} #{last_name}"
  end

  def full_name=(name)
    parts = name.split(' ')
    self.first_name = parts.first
    self.last_name = parts.last
  end
end
```

The `attribute` method also accepts a default value:

```
class Product < ApplicationRecord
  attribute :available, :boolean, default:
true
end
```

Virtual attributes are useful for form handling, calculations, and other data manipulations that don't require database storage.

Dirty Tracking

Active Record provides dirty tracking, which allows you to track changes made to an object's attributes. This is useful for auditing, conditional updates, and other change-related logic.

The `changed?` method returns `true` if any attribute has been changed since the object was last loaded or saved.

The `changes` method returns a hash of changed attributes, with the original and new values:

```
user = User.find(1)
user.name = 'New Name'
user.changes # => { 'name' => ['Old Name',
'New Name'] }
```

You can check if a specific attribute has changed using `attribute_changed?`:

```
user.name_changed? # => true
```

You can access the previous value of an attribute using `attribute_was`:

```
user.name_was # => 'Old Name'
```

Dirty tracking helps you optimize updates by only saving changed attributes, triggering callbacks only when relevant attributes change, and providing an audit trail of changes.

Multi-Database and Replica Support

Connecting to Multiple Databases

Rails 6.0 and later versions provide built-in support for connecting to multiple databases. This feature is useful for sharding, read replicas, and separating data concerns.

Define database connections in your `config/database.yml` file.

Example `config/database.yml` setup:

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  username: your_username
  password: your_password

development:
  <<: *default
  database: your_app_development

primary:
  <<: *default
  database: your_app_primary_development

secondary:
  <<: *default
  database: your_app_secondary_development
```

Specify which models should use which database connection by using the `connects_to` method in your model.

```
class ApplicationRecord < ActiveRecord::Base
  self.abstract_class = true
end

class User < ApplicationRecord
  connects_to database: { writing: :primary, reading: :primary }
end

class AuditLog < ApplicationRecord
  connects_to database: { writing: :secondary, reading: :secondary }
end
```

Write/Read from Specific DB

`connects_to database: { writing: :primary, reading: :primary }`

Specifies that both writing and reading operations should use the `primary` database connection.

`connects_to database: { writing: :primary, reading: :secondary }`

Specifies that writing operations should use the `primary` database, while reading operations should use the `secondary` database. Useful for read replicas.

Using `connected_to` block

You can use `connected_to` to execute code blocks within a specific database connection.

```
User.connected_to(database: :primary) do
  User.create(name: 'Primary User')
end

User.connected_to(database: :secondary) do
  # Perform read operations on the secondary database
  users = User.all
end
```

Configuring Read Replicas

To configure read replicas, define multiple database connections in `database.yml`, one for the primary and one or more for the replicas.

```
primary:
  <<: *default
  database: your_app_primary_development

replica1:
  <<: *default
  database: your_app_replica1_development
  host: replica1.example.com

replica2:
  <<: *default
  database: your_app_replica2_development
  host: replica2.example.com
```

Specify the writing and reading connections in your model:

```
class User < ApplicationRecord
  connects_to database: { writing: :primary, reading: :replica1 }
end
```

Rails will automatically route read queries to the replica database and write queries to the primary database.

Switching Connections Dynamically

`connected_to` with `shard:`

You can dynamically switch connections using the `connected_to` method with the `shard` option to target different databases at runtime.

```
User.connected_to(shard: :primary) do
  # Operations on the primary database
end

User.connected_to(shard: :secondary) do
  # Operations on the secondary database
end
```

Using a block with `writing:` and `reading:`

Specify writing and reading connections within the block.

```
User.connected_to(database: { writing: :primary, reading: :replica1 }) do
  # Operations using primary for writing and replica1 for reading
end
```

Considerations and Best Practices

- Ensure your database schema is consistent across all databases involved in multi-database setups.
- Use database migrations to manage schema changes across all databases.
- Monitor replication lag when using read replicas to ensure data consistency.
- Handle connection errors and failover scenarios gracefully.
- Test your multi-database configurations thoroughly to prevent data corruption or inconsistencies.

Advanced Active Record Testing

Unit Testing Models

Focus on testing model logic in isolation, without involving the database directly as much as possible.

- **Use Mocks & Stubs:** Replace database interactions with mocks or stubs to control return values and avoid slow, unpredictable database access.
- **Testing Validations:** Ensure your validations work as expected by testing valid and invalid attribute combinations.
- **Testing Callbacks:** Verify that callbacks are triggered and perform their intended actions.

Example using `rspec-mocks`:

```
describe User do
  describe '#valid?' do
    it 'is invalid with a short password' do
      user = User.new(password: 'short')
      expect(user.valid?).to be_falsey
    end

    it 'is valid with a long password' do
      user = User.new(password:
'long_enough')
      expect(user.valid?).to be_truthy
    end
  end
end
```

Testing Associations

Verifying association behavior, such as ensuring correct data retrieval through associations.

Use factories to create associated records and test the relationships.

```
describe User do
  it 'has many articles' do
    user = create(:user_with_articles)
    expect(user.articles.count).to be > 0
  end
end
```

Testing dependent options (`:destroy`, `:nullify`, `:restrict_with_error`, `:restrict_with_exception`).

Ensure that dependent records are handled correctly when the parent record is destroyed.

```
describe 'dependent destroy' do
  it 'destroys associated articles' do
    user = create(:user_with_articles)
    expect { user.destroy }.to change {
Article.count }.by(-3)
  end
end
```

Integration Testing

Involves testing the interaction between different parts of the application, including models and the database.

- **Database State Verification:** Ensure that database records are created, updated, and deleted correctly.
- **Transaction Testing:** Confirm that transactions are handled properly, especially in complex operations.
- **Testing Complex Queries:** Validate that complex Active Record queries return the expected results.

Example of an integration test:

```
describe 'User creates article' do
  it 'creates a new article in the database'
do
    user = create(:user)
    expect {
      user.articles.create(title: 'New
Article', content: 'Content')
    }.to change { Article.count }.by(1)
  end
end
```

System Tests

Simulate user interactions to test features end-to-end.

Using Capybara to simulate user actions and verify the results.

```
describe 'Create article' do
  it 'allows a user to create a new article'
do
    sign_in_as(create(:user))
    visit '/articles/new'
    fill_in 'Title', with: 'My Article'
    fill_in 'Content', with: 'Article
Content'
    click_button 'Create Article'
    expect(page).to have_content('Article
was successfully created.')
  end
end
```

Focus on critical paths and user workflows.

Write tests that cover the most important user scenarios to ensure core functionality.

Testing Database Interactions

Strategies for testing direct database interactions, including complex queries and data migrations.

- **Query Object Testing:** Test query objects in isolation to ensure they generate the correct SQL queries.
- **Data Migration Testing:** Verify that data migrations correctly transform data.
- **Raw SQL Queries:** Ensure raw SQL queries are tested for correctness and security (e.g., preventing SQL injection).

Example:

```
describe 'SqlQuery' do
  it 'returns correct result' do
    result =
ActiveRecord::Base.connection.exec_query("SE
LECT * FROM users WHERE name = 'test'")
    expect(result.count).to eq(1)
  end
end
```

Performance Testing

Measuring and improving the performance of Active Record queries and database operations.

Use tools like `benchmark` to measure the execution time of critical queries.

```
require 'benchmark'

n = 100
Benchmark.bm do |x|
  x.report { n.times { User.where(name:
'test').first } }
end
```

Identifying and optimizing slow queries.

Use `Bullet` gem to detect N+1 queries and other performance issues.