Acts As List

Build Status

gem version 1.2.1 gem version 1.2.1

ANNOUNCING: Positioning, the gem

As maintainer of both Acts As List and the Ranked Model gems, I've become intimately aquainted with the strengths and weaknesses of each. I ended up writing a small scale Rails Concern for positioning database rows for a recent project and it worked really well so I've decided to release it as a gem: Positioning

Positioning works similarly to Acts As List in that it maintains a sequential list of integer values as positions. It differs in that it encourages a unique constraints on the position column and supports multiple lists per database table. It borrows Ranked Model's concept of relative positioning. I encourage you to check it out and give it a whirl on your project!

Description

This `acts_as` extension provides the capabilities for sorting and reordering a number of objects in a list. The class that has this specified needs to have a `position` column defined as an integer on the mapped database table.

0.8.0 Upgrade Notes

There are a couple of changes of behaviour from `0.8.0` onwards:

- If you specify `add_new_at: :top`, new items will be added to the top of the list like always. But now, if you specify a position at insert time: `.create(position: 3)`, the position will be respected. In this example, the item will end up at position 3 and will move other items further down the list. Before `0.8.0` the position would be ignored and the item would still be added to the top of the list. #220
- `acts_as_list` now copes with disparate position integers (i.e. gaps between the numbers). There has been a change in behaviour for the `higher_items` method. It now returns items with the first item in the collection being the closest item to the reference item, and the last item in the collection being the furthest from the reference item (a.k.a. the first item in the list). #223

Installation

In your Gemfile:

```
gem 'acts_as_list'
```

Or, from the command line:

```
gem install acts_as_list
```

Example

At first, you need to add a `position` column to desired table:

```
rails g migration AddPositionToTodoItem position:integer
rake db:migrate
```

After that you can use `acts_as_list` method in the model:

```ruby
class TodoList < ActiveRecord::Base
  has_many :todo_items, -> { order(position: :asc) }
end

class TodoItem < ActiveRecord::Base
  belongs_to :todo_list
  acts_as_list scope: :todo_list
end

todo_list = TodoList.find(...)
todo_list.todo_items.first.move_to_bottom
todo_list.todo_items.last.move_higher
```

Instance Methods Added To ActiveRecord Models

You'll have a number of methods added to each instance of the ActiveRecord model that to which `acts_as_list` is added.

In `acts_as_list`, "higher" means further up the list (a lower `position`), and "lower" means further down the list (a higher `position`). That can be confusing, so it might make sense to add tests that validate that you're using the right method given your context.

Methods That Change Position and Reorder List

- `list_item.insert_at(2)`

- `list_item.move_lower` will do nothing if the item is the lowest item
- `list_item.move_higher` will do nothing if the item is the highest item
- `list_item.move_to_bottom`
- `list_item.move_to_top`
- `list_item.remove_from_list`

Methods That Change Position Without Reordering List

- `list_item.increment_position`
- `list_item.decrement_position`
- `list_item.set_list_position(3)`

Methods That Return Attributes of the Item's List Position

- `list_item.first?`
- `list_item.last?`
- `list_item.in_list?`
- `list_item.not_in_list?`
- `list_item.default_position?`
- `list_item.higher_item`
- `list_item.higher_items` will return all the items above `list_item` in the list (ordered by the position, ascending)
- `list_item.lower_item`
- `list_item.lower_items` will return all the items below `list_item` in the list (ordered by the position, ascending)

Adding `acts_as_list` To An Existing Model

As it stands `acts_as_list` requires position values to be set on the model before the instance methods above will work. Adding something like the below to your migration will set the default position. Change the parameters to order if you want a different initial ordering.

```ruby
class AddPositionToTodoItem < ActiveRecord::Migration
  def change
    add_column :todo_items, :position, :integer
    TodoItem.order(:updated_at).each.with_index(1) do |todo_item, index
        |
      todo_item.update_column :position, index
    end
  end
end
```

If you are using the scope option things can get a bit more complicated. Let's say you have `acts_as_list scope: :todo_list`, you might instead need something like this:

```ruby
TodoList.all.each do |todo_list|
  todo_list.todo_items.order(:updated_at).each.with_index(1) do |
      todo_item, index|
    todo_item.update_column :position, index
  end
end
```

When using PostgreSQL, it is also possible to leave this migration up to the database layer. Inside of the `change` block you could write:

```ruby
 execute <<~SQL.squish
   UPDATE todo_items
   SET position = mapping.new_position
   FROM (
     SELECT
       id,
       ROW_NUMBER() OVER (
         PARTITION BY todo_list_id
         ORDER BY updated_at
       ) AS new_position
     FROM todo_items
   ) AS mapping
   WHERE todo_items.id = mapping.id;
 SQL
```

Notes

All `position` queries (select, update, etc.) inside gem methods are executed without the default scope (i.e. `Model.unscoped`), this will prevent nasty issues when the default scope is different from `acts_as_list` scope.

The `position` column is set after validations are called, so you should not put a `presence` validation on the `position` column.

If you need a scope by a non-association field you should pass an array, containing field name, to a scope:

```ruby
class TodoItem < ActiveRecord::Base
  # `task_category` is a plain text field (e.g. 'work', 'shopping', '
      meeting'), not an association
  acts_as_list scope: [:task_category]
```

```ruby
end
```

You can also add multiple scopes in this fashion:

```ruby
class TodoItem < ActiveRecord::Base
  belongs_to :todo_list
  acts_as_list scope: [:task_category, :todo_list_id]
end
```

Furthermore, you can optionally include a hash of fixed parameters that will be included in all queries:

```ruby
class TodoItem < ActiveRecord::Base
  belongs_to :todo_list
  # or `discarded_at` if using discard
  acts_as_list scope: [:task_category, :todo_list_id, deleted_at: nil]
end
```

This is useful when using this gem in conjunction with the popular acts_as_paranoid or discard gems.

More Options

- `column` default: `position`. Use this option if the column name in your database is different from position.
- `top_of_list` default: `1`. Use this option to define the top of the list. Use 0 to make the collection act more like an array in its indexing.
- `add_new_at` default: `:bottom`. Use this option to specify whether objects get added to the `:top` or `:bottom` of the list. `nil` will result in new items not being added to the list on create, i.e, position will be kept nil after create.
- `touch_on_update` default: **`true`**. Use `touch_on_update: false` if you don't want to update the timestamps of the associated records.
- `sequential_updates` Specifies whether insert_at should update objects positions during shuffling one by one to respect position column unique not null constraint. Defaults to true if position column has unique index, otherwise false. If constraint is `deferrable initially deferred` (PostgreSQL), overriding it with false will speed up insert_at.

Disabling temporarily

If you need to temporarily disable `acts_as_list` during specific operations such as mass-update or imports:

```ruby
TodoItem.acts_as_list_no_update do
  perform_mass_update
end
```

In an `acts_as_list_no_update` block, all callbacks are disabled, and positions are not updated. New records will be created with the default value from the database. It is your responsibility to correctly manage `positions` values.

You can also pass an array of classes as an argument to disable database updates on just those classes. It can be any ActiveRecord class that has acts_as_list enabled.

```ruby
class TodoList < ActiveRecord::Base
  has_many :todo_items, -> { order(position: :asc) }
  acts_as_list
end

class TodoItem < ActiveRecord::Base
  belongs_to :todo_list
  has_many :todo_attachments, -> { order(position: :asc) }

  acts_as_list scope: :todo_list
end

class TodoAttachment < ActiveRecord::Base
  belongs_to :todo_item
  acts_as_list scope: :todo_item
end

TodoItem.acts_as_list_no_update([TodoAttachment]) do
  TodoItem.find(10).update(position: 2)
  TodoAttachment.find(10).update(position: 1)
  TodoAttachment.find(11).update(position: 2)
  TodoList.find(2).update(position: 3) # For this instance the
      callbacks will be called because we haven't passed the class as an
       argument
end
```

Troubleshooting Database Deadlock Errors

When using this gem in an app with a high amount of concurrency, you may see “deadlock” errors raised by your database server. It’s difficult for the gem to provide a solution that fits every app. Here are some steps you can take to mitigate and handle these kinds of errors.

1) Use the Most Concise API

One easy way to reduce deadlocks is to use the most concise gem API available for what you want to accomplish. In this specific example, the more concise API for creating a list item at a position results in one transaction instead of two, and it issues fewer SQL statements. Issuing fewer statements tends to lead to faster transactions. Faster transactions are less likely to deadlock.

Example:

```ruby
# Good
TodoItem.create(todo_list: todo_list, position: 1)

# Bad
item = TodoItem.create(todo_list: todo_list)
item.insert_at(1)
```

2) Rescue then Retry

Deadlocks are always a possibility when updating tables rows concurrently. The general advice from MySQL documentation is to catch these errors and simply retry the transaction; it will probably succeed on another attempt. (see How to Minimize and Handle Deadlocks) Retrying transactions sounds simple, but there are many details that need to be chosen on a per-app basis: How many retry attempts should be made? Should there be a wait time between attempts? What *other* statements were in the transaction that got rolled back?

Here a simple example of rescuing from deadlock and retrying the operation: * `ActiveRecord::Deadlocked` is available in Rails >= 5.1.0. * If you have Rails < 5.1.0, you will need to rescue `ActiveRecord::StatementInvalid` and check `#cause`.

```ruby
  attempts_left = 2
  while attempts_left > 0
    attempts_left -= 1
    begin
      TodoItem.transaction do
        TodoItem.create(todo_list: todo_list, position: 1)
      end
      attempts_left = 0
    rescue ActiveRecord::Deadlocked
      raise unless attempts_left > 0
    end
  end
```

You can also use the approach suggested in this StackOverflow post: https://stackoverflow.com/questions/4027659/ac
deadlock-retry

3) Lock Parent Record

In addition to reacting to deadlocks, it is possible to reduce their frequency with more pessimistic locking. This approach uses the parent record as a mutex for the entire list. This kind of locking is very effective at reducing the frequency of deadlocks while updating list items. However, there are some things to keep in mind: * This locking pattern needs to be used around *every* call that modifies the list; even if it does not reorder list items. * This locking pattern effectively serializes operations on the list. The throughput of operations on the list will decrease. * Locking the parent record may lead to deadlock elsewhere if some other code also locks the parent table.

Example:

```ruby
todo_list = TodoList.create(name: "The List")
todo_list.with_lock do
  item = TodoItem.create(description: "Buy Groceries", todo_list:
      todo_list, position: 1)
end
```

Versions

Version `0.9.0` adds `acts_as_list_no_update` (https://github.com/brendon/acts_as_list/pull/244) and compatibility with not-null and uniqueness constraints on the database (https://github.com/brendon/acts_as_lis
These additions shouldn't break compatibility with existing implementations.

As of version `0.7.5` Rails 5 is supported.

All versions `0.1.5` onwards require Rails 3.0.x and higher.

A note about data integrity

We often hear complaints that `position` values are repeated, incorrect etc. For example, #254. To ensure data integrity, you should rely on your database. There are two things you can do:

1. Use constraints. If you model `Item` that `belongs_to` an `Order`, and it has a `position` column, then add a unique constraint on `items` with `[:order_id, :position]`. Think of it as a list invariant. What are the properties of your list that don't change no matter how many items you have in it? One such propery is that each item has a distinct position. Another *could be* that position is always greater than 0. It is strongly recommended that you rely on your database to enforce these invariants or constraints. Here are the docs for PostgreSQL and MySQL.

2. Use mutexes or row level locks. At its heart the duplicate problem is that of handling concurrency. Adding a contention resolution mechanism like locks will solve it to some extent. But it is not a solution or replacement for constraints. Locks are also prone to deadlocks.

As a library, `acts_as_list` may not always have all the context needed to apply these tools. They are much better suited at the application level.

Roadmap

1. Sort based feature

Contributing to `acts_as_list`

- Check out the latest master to make sure the feature hasn't been implemented or the bug hasn't been fixed yet
- Check out the issue tracker to make sure someone already hasn't requested it and/or contributed it
- Fork the project
- Start a feature/bugfix branch
- Commit and push until you are happy with your contribution
- Make sure to add tests for it. This is important so I don't break it in a future version unintentionally.
- Please try not to mess with the Rakefile, version, or history. If you want to have your own version, or is otherwise necessary, that is fine, but please isolate to its own commit so I can cherry-pick around it.
- I would recommend using Rails 3.1.x and higher for testing the build before a pull request. The current test harness does not quite work with 3.0.x. The plugin itself works, but the issue lies with testing infrastructure.

Copyright