# Rails: How to Batch Insert Records with a Single INSERT Statement

We can easily insert records one by one by supplying an array of objects to `.create` in Ruby on Rails, but each insertion results in a separate request to the database. This may not be an issue for apps that receive low traffic or where batches are small, but what if your app receives high traffic or the batches are large? In the past, we had to use SQL directly to batch insert all at once, but as of Rails 6 we now have `.insert\_all`!

## The Problem(s)

My web app allows users to create multiple records at once. The frontend sends an HTTP request containing an array of objects to the backend, which then calls `.create` on the appropriate model using the parameters from that request. However, this is slowing down our response times since each record results in a separate request to the database. For example, if we try to `.create` 12 records, we’ll see 12 separate INSERTs in the logs.

We haven’t been using SQL because we want the records to pass through the relevant model validations. We also use Active Record callbacks to add data to these objects before they are saved. But we’re now at a point where this isn’t feasible.

## Need to Know

When we call `.create` on a model, objects pass through model validations and Active Model callbacks are run (e.g. `after\_validation`, `before\_save`…). When we insert records using SQL or `.insert\_all`, callbacks and model validations are bypassed. As of right now, the only way to batch insert or upsert records in a database with a single request is through SQL and Rails 6 methods, such as `.insert\_all`. Database constraints will still apply, as always.

According to this Redgate article (<https://www.red-gate.com/simple-talk/sql/performance/comparing-multiple-rows-insert-vs-single-row-insert-with-three-data-load-methods/>), inserting five rows in a single statement reduced execution time by 40% when compared to five separate inserts. And then there’s this lovely quote from the MySQL documentation: “If you are inserting many rows from the same client at the same time, use [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statements with multiple VALUES lists to insert several rows at a time. This is considerably faster (many times faster in some cases) than using separate single-row [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statements.” (https://dev.mysql.com/doc/refman/8.0/en/insert-optimization.html)

## What We’ll Be Doing

Here is the code we’ll be working with:

```

class CreateBands < ActiveRecord::Migration[6.0]

def change

create\_table :bands do |t|

t.string :name, null: false

t.integer :members, null: false

t.timestamps

end

add\_index :bands, :name, unique: true

end

end

class Band < ApplicationRecord

validates :name, presence: true, length: { minimum: 3 }, uniqueness: true

validates :members, presence: true, numericality: { greater\_than: 0 }

end

class BandsController < ApplicationController

def index

render json: Band.all

end

def create

begin

Band.transaction do

@bands = Band.create!(bands\_params)

end

rescue ActiveRecord::RecordInvalid => exception

@bands = {

error: {

status: 422,

message: exception

}

}

end

render json: @bands

end

private

def bands\_params

params.permit(bands: [:name, :members]).require(:bands)

end

end

```

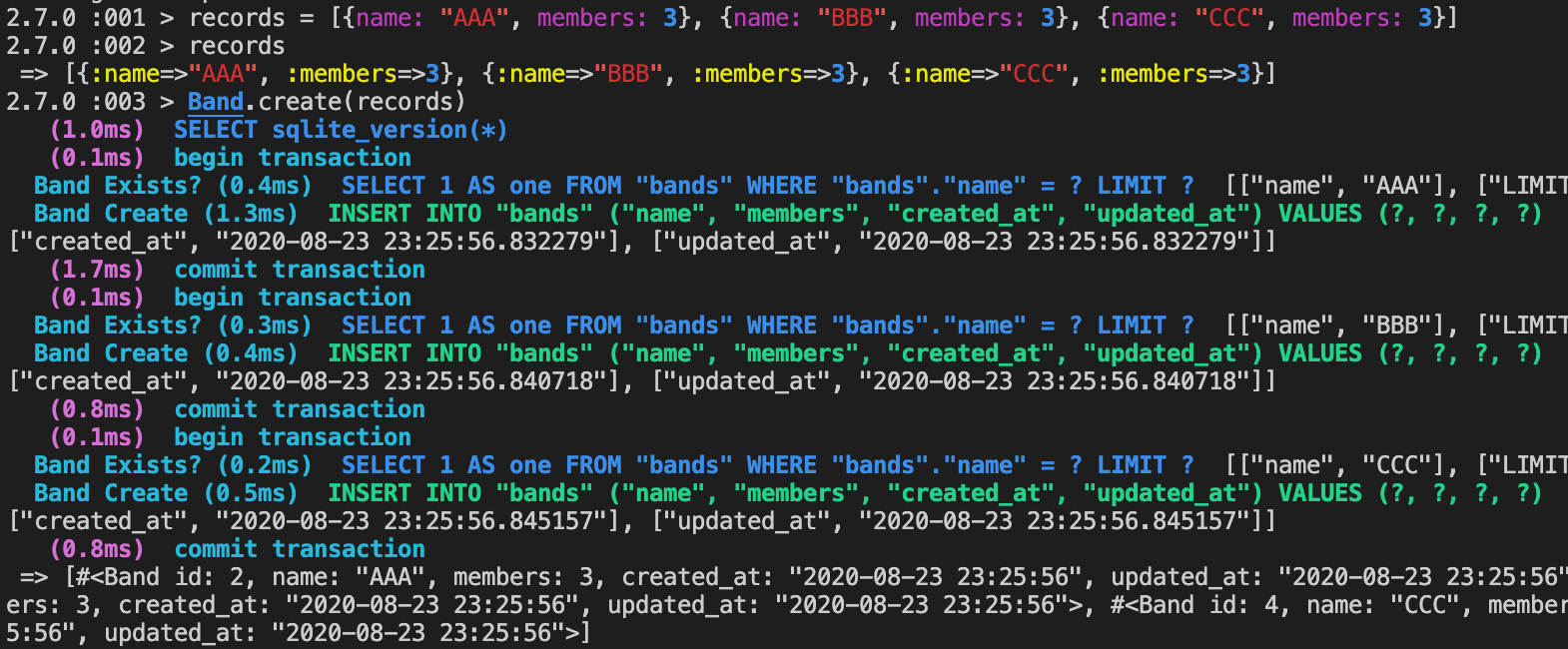
We’ll modify this to work with `.insert\_all`, and then we’ll tackle the problems as they come up. Our end goal is to have an all-or-nothing insertion that respects all of our necessary validations.

Notice how the model validations don’t currently match the database constraints, which are a little looser. We’ll tackle this issue as well, since batch insertions rely on database constraints.

Lastly, we’ll explore how we can insert records that require additional modifications before insertion, like adding more information or running validations that can’t easily be applied as database constraints.

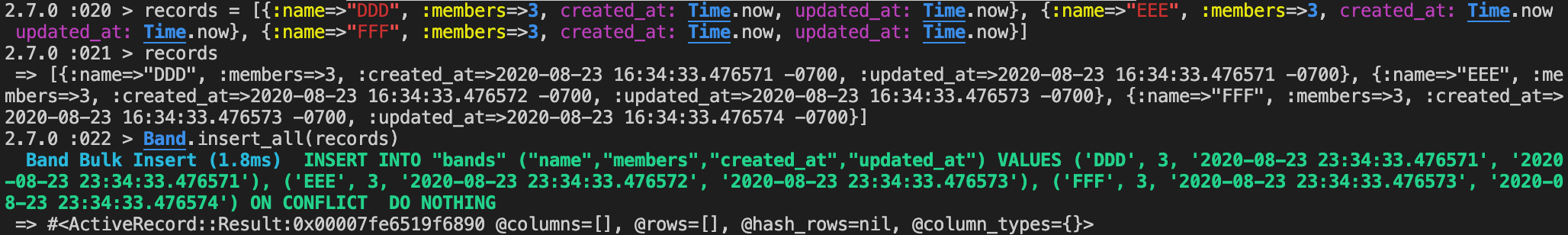
## Quick Log Check:.create vs .insert\_all

Let’s take a look at the log for creating three records via `.create` on our Band model. Notice that there are three total transactions here:



Also notice that we didn’t have to add the timestamps for updated\_at and created\_at. Those were added for us.

Now let’s take a look at using `.insert\_all`:



There’s a single bulk insert, but since our Band records require timestamps, we had to add those in for our records to pass the database constraints. Otherwise, any records missing that information would not be inserted.

## Step 1: Make it Work!

Right now, we can send an array of bands and they’ll be inserted into our database one by one. Let’s see if we can get our bands to bulk insert instead without worrying about the fine details of what’s valid or not valid. In short, just make it work, even if it’s not great.

Let’s start by making our create action as dumb as possible:

```

def create

bands = Band.insert\_all(bands\_params)

render json: bands

end

```

Now let’s make a request containing several bands and see what error we get:

```

ActiveRecord::NotNullViolation (SQLite3::ConstraintException: NOT NULL constraint failed: bands.created\_at)

```

In the migration, we added timestamps using the `timestamps` method. This automatically adds a not null constraint for both the created\_at and updated\_at columns. When we add records by instantiating objects from the model, Active Record automatically adds timestamps for us. Since we’re now bypassing that step to insert multiple rows with a single statement, we now need to add this data ourselves. Let’s add some class methods to the Band model to get this working:

```

def self.insert\_all(records)

normalized = normalize(records)

super(normalized)

end

def self.normalize(records)

records.each do |rec|

add\_timestamp(rec)

end

end

def self.add\_timestamp(record)

time = Time.now.utc

record['created\_at'] = time

record['updated\_at'] = time

end

```

Now, when I send a fetch request in my browser, the records are created, but my response is an array of object, and each object has an id:

```

[{ id: 1 }, { id: 2 }]

```

*Note: This behavior is specific to PostgreSQL. SQLite3, for example, will return an empty array by default.*

That’s not a very useful response, so let’s fix that up by using the returning option to specify which data we want to have returned to us:

```

# BandsController

def create

bands = Band.insert\_all(bands\_params, returning: [:id, :name, :members])

render json: bands

end

# Band

def self.insert\_all(records, options)

normalized = normalize(records)

super(normalized, options)

end

```

*Note: The returning option is not valid when using SQLite3. It will produce an error!*

Now we receive a more useful response containing the data we want:

```

[

{ id: 3, name: “Radiohead”, members: 5 },

{ id: 4, name: “Refused”, members: 4 }

]

```

What happens if we try to make these same records again? Our database constraints state that the band names must be unique, so the records aren’t created. Our response consists of an empty array.

What if we try to create some bands with 0 members? There’s no database constraint for that, but there is a model validation. Since `.insert\_all` bypasses model validations, those records are inserted, so we now have bands in the database that have 0 members. Let’s clear this up in the next step.

## Step 2: Better Database Constraints

Right now, our database constraints ensure that any new band rows contain a name and some number of members. They also ensure that each band has a unique name. However, the Band model has additional validations: it checks that the band name is three characters or more and that the number of members is greater than 0.

For `.insert\_all` to honor these rules, we’ll need to add a change migration and we’re going to have to bust out our old friend SQL!

```

class AddCheckConstraintToBandsForNameLength < ActiveRecord::Migration[6.0]

def change

execute <<-SQL

ALTER TABLE bands

ADD CONSTRAINT check\_minimum\_length

CHECK (LENGTH(name) > 2)

SQL

end

end

```

This new constraint doesn’t show up in schema.rb, but we can easily test it in the console. We shouldn’t be able to insert any bands with names of less than three characters.

```

bands = Band.insert\_all([{ name: ‘bb’, members: 5 }], returning: [:id, :name])

# => ActiveRecord::StatementInvalid (PG::CheckViolation: ERROR: new row for relation "bands" violates check constraint "check\_minimum\_length")

bands = Band.insert\_all([{ name: ‘bbb’, members: 5 }], returning: [:id, :name])

bands.first

# => { “id”=> 12, “name” => “bbb” }

```

So it looks like our minimum length constraint is working. Let’s also add our minimum band member constraint, since bands should have more than zero members:

```

class AddCheckConstraintToBandsForMinimumMembers < ActiveRecord::Migration[6.0]

def change

execute <<-SQL

ALTER TABLE bands

ADD CONSTRAINT check\_minimum\_number\_of\_band\_members

CHECK (members > 0)

SQL

end

end

```

This has also worked. You’re just going to have to trust me on that.

After testing this out, I can now only batch insert valid records, and my database constraints perfectly match my model validations from earlier.

But what if we needed to add some additional data during some part of the object lifecycle? How could we do that?

## Step 3: Adding Additional Data Before Insert

I ran a change migration and added a ‘gibberish’ column to our bands table. I then made the following change to the model:

```

after\_validation :add\_gibberish

def add\_gibberish

self.gibberish = "#{self.name} likes gibberish"

end

```

I wish I had come up with something more creative to show here, but it just wasn’t happening today.

Since `.insert\_all` doesn’t care about the object lifecycle, we can’t add this nonsense after validation. Instead, we’ll need to update our Band model like so:

```

def add\_gibberish

self.class.make\_gibberish(self)

end

def self.make\_gibberish(record)

record[:gibberish] = "#{record[:name]} likes gibberish"

end

def self.normalize(records)

records.each do |rec|

add\_timestamp(rec)

make\_gibberish(rec)

end

end

```

If we were running many methods, we could instead instantiate a new object from the Band class, check if the object is valid, and then use the attributes from that object as our record:

```

def self.object\_to\_record(record)

band = Band.new(record)

band.valid?

band.attributes

end

```

By doing this, we can run any lifecycle methods that occur before or right after validation. But this will do nothing to help us if we’re running methods around save.

## Step 4: Refactor

Our Band model is looking pretty messy, and I’m pretty sure the `.insert\_all` override/monkey patch is general enough to put it elsewhere. Actually, let’s not be monkeys and instead use a new method name. And we’ll factor this code out into a concern, which I’ll call Insertable:

```

# app/models/concerns/insertable.rb

module Insertable

extend ActiveSupport::Concern

class\_methods do

def add\_timestamp(record)

time = Time.now.utc

record['created\_at'] = time

record['updated\_at'] = time

end

def insert\_all\_normalized(records, options = {returning: [:id]})

normalized = normalize(records)

result = nil

begin

result = insert\_all(normalized, options)

rescue ActiveRecord::StatementInvalid => e

result = e

end

result

end

end

end

```

I also moved `add\_timestamp` into the concern because it’s not specific to the Band model. Notice that `.insert\_all\_normalized` is still relying on `.normalize`. That method will always be specific to a model, so anytime we mix Insertable into a model class, we’ll need to declare `.normalize` in that model. I also decided to handle any exceptions here, so that the controller can be as “dumb” as possible, and if this method is called in multiple controllers, it’ll reduce the amount of code we need to write.

Our Band model now looks like this:

```

class Band < ApplicationRecord

include Insertable

validates :name, presence: true, length: { minimum: 3 }, uniqueness: true

validates :members, presence: true, numericality: { greater\_than: 0 }

after\_validation :add\_gibberish

def add\_gibberish

self.class.make\_gibberish(self)

end

def self.make\_gibberish(record)

record[:gibberish] = "#{record[:name]} likes gibberish"

end

def self.normalize(records)

records.each do |rec|

add\_timestamp(rec)

make\_gibberish(rec)

end

end

end

```

And here’s our controller:

```

class BandsController < ApplicationController

def index

render json: Band.all

end

def create

bands = Band.insert\_all\_normalized(bands\_params, returning: [:id, :name, :members])

if bands.kind\_of?(ActiveRecord::Result)

render json: bands

else

render json: {

status: 422,

error: bands

}, status: 422

end

end

private

def bands\_params

params.permit(bands: [:name, :members]).require(:bands)

end

end

```