DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Multiple Databases with Active Record

This guide covers using multiple databases with your Rails application.

After reading this guide you will know:

- How to set up your application for multiple databases.
- How automatic connection switching works.
- How to use horizontal sharding for multiple databases.
- What features are supported and what's still a work in progress.

As an application grows in popularity and usage you'll need to scale the application to support your new users and their data. One way in which your application may need to scale is on the database level. Rails now has support for multiple databases so you don't have to store your data all in one place.

At this time the following features are supported:

- Multiple writer databases and a replica for each
- Automatic connection switching for the model you're working with
- Automatic swapping between the writer and replica depending on the HTTP verb and recent writes
- Rails tasks for creating, dropping, migrating, and interacting with the multiple databases

The following features are not (yet) supported:

- Load balancing replicas

Setting up your application

While Rails tries to do most of the work for you there are still some steps you'll need to do to get your application ready for multiple databases.

Let's say we have an application with a single writer database and we need to add a new database for some new tables we're adding. The name of the new database will be "animals".

The `database.yml` looks like this:

```
production:
  database: my_primary_database
  adapter: mysql2
  username: root
  password: <%= ENV['ROOT_PASSWORD'] %>
```

Let's add a replica for the first configuration, and a second database called animals and a replica for that as well. To do this we need to change our `database.yml` from a 2-tier to a 3-tier config.

If a primary configuration is provided, it will be used as the "default" configuration. If there is no configuration named `"primary"`, Rails will use the first configuration as default for each environment. The default configurations will use the default Rails filenames. For example, primary configurations will use `schema.rb` for the schema file, whereas all the other entries will use `[CONFIGURATION_NAMESPACE]_schema.rb` for the filename.

```
production:
  primary:
    database: my_primary_database
    username: root
    password: <%= ENV['ROOT_PASSWORD'] %>
    adapter: mysql2
  primary_replica:
    database: my_primary_database
    username: root_readonly
    password: <%= ENV['ROOT_READONLY_PASSWORD'] %>
    adapter: mysql2
    replica: true
  animals:
    database: my_animals_database
    username: animals_root
    password: <%= ENV['ANIMALS_ROOT_PASSWORD'] %>
    adapter: mysql2
    migrations_paths: db/animals_migrate
  animals_replica:
    database: my_animals_database
    username: animals_readonly
    password: <%= ENV['ANIMALS_READONLY_PASSWORD'] %>
    adapter: mysql2
    replica: true
```

When using multiple databases, there are a few important settings.

First, the database name for the `primary` and `primary_replica` should be the same because they contain the same data. This is also the case for `animals` and `animals_replica`.

Second, the username for the writers and replicas should be different, and the replica user's database permissions should be set to only read and not write.

When using a replica database, you need to add a `replica: true` entry to the replica in the `database.yml`. This is because Rails otherwise has no way of knowing which one is a replica and which one is the writer. Rails will not run certain tasks, such as migrations, against replicas.

Lastly, for new writer databases, you need to set the `migrations_paths` to the directory where you will store migrations for that database. We'll look more at `migrations_paths` later on in this guide.

Now that we have a new database, let's set up the connection model. In order to use the new database we need to create a new abstract class and connect to the animals databases.

```
class AnimalsRecord < ApplicationRecord
  self.abstract_class = true

  connects_to database: { writing: :animals, reading: :animals_replica }
end
```

Then we need to update `ApplicationRecord` to be aware of our new replica.

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

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

If you use a differently named class for your application record you need to set `primary_abstract_class` instead, so that Rails knows which class `ActiveRecord::Base` should share a connection with.

```
class PrimaryApplicationRecord < ActiveRecord::Base
  self.primary_abstract_class = true
end
```

Classes that connect to primary/primary_replica can inherit from your primary abstract class like standard Rails applications:

```
class Person < ApplicationRecord
end
```

By default Rails expects the database roles to be `writing` and `reading` for the primary and replica respectively. If you have a legacy system you may already have roles set up that you don't want to change. In that case you can set a new role name in your application config.

```
config.active_record.writing_role = :default
config.active_record.reading_role = :readonly
```

It's important to connect to your database in a single model and then inherit from that model for the tables rather than connect multiple individual models to the same database. Database clients have a limit to the number of open connections there can be and if you do this it will multiply the number of connections you have since Rails uses the model class name for the connection specification name.

Now that we have the `database.yml` and the new model set up, it's time to create the databases. Rails 6.0 ships with all the rails tasks you need to use multiple databases in Rails.

You can run `bin/rails -T` to see all the commands you're able to run. You should see the following:

```
$ bin/rails -T
rails db:create                          # Creates the database from DATABASE_URL or config/
rails db:create:animals                  # Create animals database for current environment
rails db:create:primary                  # Create primary database for current environment
rails db:drop                            # Drops the database from DATABASE_URL or config/da
rails db:drop:animals                    # Drop animals database for current environment
rails db:drop:primary                    # Drop primary database for current environment
rails db:migrate                         # Migrate the database (options: VERSION=x, VERBOSI
rails db:migrate:animals                 # Migrate animals database for current environment
rails db:migrate:primary                 # Migrate primary database for current environment
rails db:migrate:status                  # Display status of migrations
rails db:migrate:status:animals          # Display status of migrations for animals databas
rails db:migrate:status:primary          # Display status of migrations for primary databas
rails db:reset                           # Drops and recreates all databases from their sch
rails db:reset:animals                   # Drops and recreates the animals database from its
rails db:reset:primary                   # Drops and recreates the primary database from its
rails db:rollback                        # Rolls the schema back to the previous version (sp
rails db:rollback:animals                # Rollback animals database for current environmen
rails db:rollback:primary                # Rollback primary database for current environmen
rails db:schema:dump                     # Creates a database schema file (either db/schema.
rails db:schema:dump:animals             # Creates a database schema file (either db/schema.
rails db:schema:dump:primary             # Creates a db/schema.rb file that is portable aga
rails db:schema:load                     # Loads a database schema file (either db/schema.rb
rails db:schema:load:animals             # Loads a database schema file (either db/schema.rb
rails db:schema:load:primary             # Loads a database schema file (either db/schema.rb
rails db:setup                           # Creates all databases, loads all schemas, and ini
rails db:setup:animals                   # Creates the animals database, loads the schema, a
rails db:setup:primary                   # Creates the primary database, loads the schema, a
```

Running a command like `bin/rails db:create` will create both the primary and animals databases. Note that there is no command for creating the database users, and you'll need to do that manually to support the readonly users for your replicas. If you want to create just the animals database you can run `bin/rails db:create:animals`.

Connecting to Databases without Managing Schema and Migrations

If you would like to connect to an external database without any database management tasks such as schema management, migrations, seeds, etc., you can

set the per database config option `database_tasks: false`. By default it is set to true.

```
production:
  primary:
    database: my_database
    adapter: mysql2
  animals:
    database: my_animals_database
    adapter: mysql2
    database_tasks: false
```

Generators and Migrations

Migrations for multiple databases should live in their own folders prefixed with the name of the database key in the configuration.

You also need to set the `migrations_paths` in the database configurations to tell Rails where to find the migrations.

For example the `animals` database would look for migrations in the `db/animals_migrate` directory and `primary` would look in `db/migrate`. Rails generators now take a `--database` option so that the file is generated in the correct directory. The command can be run like so:

```
$ bin/rails generate migration CreateDogs name:string --database animals
```

If you are using Rails generators, the scaffold and model generators will create the abstract class for you. Simply pass the database key to the command line.

```
$ bin/rails generate scaffold Dog name:string --database animals
```

A class with the database name and `Record` will be created. In this example the database is `Animals` so we end up with `AnimalsRecord`:

```
class AnimalsRecord < ApplicationRecord
  self.abstract_class = true

  connects_to database: { writing: :animals }
end
```

The generated model will automatically inherit from `AnimalsRecord`.

```
class Dog < AnimalsRecord
end
```

Note: Since Rails doesn't know which database is the replica for your writer you will need to add this to the abstract class after you're done.

Rails will only generate the new class once. It will not be overwritten by new scaffolds or deleted if the scaffold is deleted.

If you already have an abstract class and its name differs from `AnimalsRecord`, you can pass the `--parent` option to indicate you want a different abstract class:

```
$ bin/rails generate scaffold Dog name:string --database animals --parent Animals::Record
```

This will skip generating `AnimalsRecord` since you've indicated to Rails that you want to use a different parent class.

Activating automatic role switching

Finally, in order to use the read-only replica in your application, you'll need to activate the middleware for automatic switching.

Automatic switching allows the application to switch from the writer to replica or replica to writer based on the HTTP verb and whether there was a recent write by the requesting user.

If the application is receiving a POST, PUT, DELETE, or PATCH request the application will automatically write to the writer database. For the specified time after the write, the application will read from the primary. For a GET or HEAD request the application will read from the replica unless there was a recent write.

To activate the automatic connection switching middleware you can run the automatic swapping generator:

```
$ bin/rails g active_record:multi_db
```

And then uncomment the following lines:

```
Rails.application.configure do
  config.active_record.database_selector = { delay: 2.seconds }
  config.active_record.database_resolver = ActiveRecord::Middleware::DatabaseSelector::Resol
  config.active_record.database_resolver_context = ActiveRecord::Middleware::DatabaseSelecto
end
```

Rails guarantees "read your own write" and will send your GET or HEAD request to the writer if it's within the `delay` window. By default the delay is set to 2 seconds. You should change this based on your database infrastructure. Rails doesn't guarantee "read a recent write" for other users within the delay window and will send GET and HEAD requests to the replicas unless they wrote recently.

The automatic connection switching in Rails is relatively primitive and deliberately doesn't do a whole lot. The goal is a system that demonstrates how to do automatic connection switching that was flexible enough to be customizable by app developers.

The setup in Rails allows you to easily change how the switching is done and what parameters it's based on. Let's say you want to use a cookie instead of a session to decide when to swap connections. You can write your own class:

```ruby
class MyCookieResolver
  # code for your cookie class
end
```

And then pass it to the middleware:

```ruby
config.active_record.database_selector = { delay: 2.seconds }
config.active_record.database_resolver = ActiveRecord::Middleware::DatabaseSelector::Resolve
config.active_record.database_resolver_context = MyCookieResolver
```

Using manual connection switching

There are some cases where you may want your application to connect to a writer or a replica and the automatic connection switching isn't adequate. For example, you may know that for a particular request you always want to send the request to a replica, even when you are in a POST request path.

To do this Rails provides a `connected_to` method that will switch to the connection you need.

```ruby
ActiveRecord::Base.connected_to(role: :reading) do
  # all code in this block will be connected to the reading role
end
```

The "role" in the `connected_to` call looks up the connections that are connected on that connection handler (or role). The `reading` connection handler will hold all the connections that were connected via `connects_to` with the role name of `reading`.

Note that `connected_to` with a role will look up an existing connection and switch using the connection specification name. This means that if you pass an unknown role like `connected_to(role: :nonexistent)` you will get an error that says `ActiveRecord::ConnectionNotEstablished (No connection pool for 'ActiveRecord::Base' found for the 'nonexistent' role.)`

If you want Rails to ensure any queries performed are read only, pass `prevent_writes: true`. This just prevents queries that look like writes from being sent to the database. You should also configure your replica database to run in readonly mode.

```ruby
ActiveRecord::Base.connected_to(role: :reading, prevent_writes: true) do
  # Rails will check each query to ensure it's a read query
end
```

Horizontal sharding

Horizontal sharding is when you split up your database to reduce the number of rows on each database server, but maintain the same schema across "shards". This is commonly called "multi-tenant" sharding.

The API for supporting horizontal sharding in Rails is similar to the multiple database / vertical sharding API that's existed since Rails 6.0.

Shards are declared in the three-tier config like this:

```
production:
  primary:
    database: my_primary_database
    adapter: mysql2
  primary_replica:
    database: my_primary_database
    adapter: mysql2
    replica: true
  primary_shard_one:
    database: my_primary_shard_one
    adapter: mysql2
  primary_shard_one_replica:
    database: my_primary_shard_one
    adapter: mysql2
    replica: true
```

Models are then connected with the `connects_to` API via the `shards` key:

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

  connects_to shards: {
    default: { writing: :primary, reading: :primary_replica },
    shard_one: { writing: :primary_shard_one, reading: :primary_shard_one_replica }
  }
end
```

Then models can swap connections manually via the `connected_to` API. If using sharding, both a `role` and a `shard` must be passed:

```
ActiveRecord::Base.connected_to(role: :writing, shard: :default) do
  @id = Person.create! # Creates a record in shard default
end

ActiveRecord::Base.connected_to(role: :writing, shard: :shard_one) do
  Person.find(@id) # Can't find record, doesn't exist because it was created
                   # in the default shard
end
```

The horizontal sharding API also supports read replicas. You can swap the role and the shard with the `connected_to` API.

```
ActiveRecord::Base.connected_to(role: :reading, shard: :shard_one) do
  Person.first # Lookup record from read replica of shard one
end
```

Activating automatic shard switching

Applications are able to automatically switch shards per request using the provided middleware.

The `ShardSelector` Middleware provides a framework for automatically swapping shards. Rails provides a basic framework to determine which shard to switch to and allows for applications to write custom strategies for swapping if needed.

The `ShardSelector` takes a set of options (currently only `lock` is supported) that can be used by the middleware to alter behavior. `lock` is true by default and will prohibit the request from switching shards once inside the block. If `lock` is false, then shard swapping will be allowed. For tenant based sharding, `lock` should always be true to prevent application code from mistakenly switching between tenants.

The same generator as the database selector can be used to generate the file for automatic shard swapping:

```
$ bin/rails g active_record:multi_db
```

Then in the file uncomment the following:

```
Rails.application.configure do
  config.active_record.shard_selector = { lock: true }
  config.active_record.shard_resolver = ->(request) { Tenant.find_by!(host: request.host).sh
end
```

Applications must provide the code for the resolver as it depends on application specific models. An example resolver would look like this:

```
config.active_record.shard_resolver = ->(request) {
  subdomain = request.subdomain
  tenant = Tenant.find_by_subdomain!(subdomain)
  tenant.shard
}
```

Granular Database Connection Switching

In Rails 6.1 it's possible to switch connections for one database instead of all databases globally.

With granular database connection switching, any abstract connection class will be able to switch connections without affecting other connections. This is useful for switching your `AnimalsRecord` queries to read from the replica while ensuring your `ApplicationRecord` queries go to the primary.

```
AnimalsRecord.connected_to(role: :reading) do
  Dog.first # Reads from animals_replica
```

```
  Person.first  # Reads from primary
end
```

It's also possible to swap connections granularly for shards.

```
AnimalsRecord.connected_to(role: :reading, shard: :shard_one) do
  Dog.first # Will read from shard_one_replica. If no connection exists for shard_one_replic
  # a ConnectionNotEstablished error will be raised
  Person.first # Will read from primary writer
end
```

To switch only the primary database cluster use `ApplicationRecord`:

```
ApplicationRecord.connected_to(role: :reading, shard: :shard_one) do
  Person.first # Reads from primary_shard_one_replica
  Dog.first # Reads from animals_primary
end
```

`ActiveRecord::Base.connected_to` maintains the ability to switch connections globally.

Handling associations with joins across databases

As of Rails 7.0+, Active Record has an option for handling associations that would perform a join across multiple databases. If you have a has many through or a has one through association that you want to disable joining and perform 2 or more queries, pass the `disable_joins: true` option.

For example:

```
class Dog < AnimalsRecord
  has_many :treats, through: :humans, disable_joins: true
  has_many :humans

  has_one :home
  has_one :yard, through: :home, disable_joins: true
end

class Home
  belongs_to :dog
  has_one :yard
end

class Yard
  belongs_to :home
end
```

Previously calling `@dog.treats` without `disable_joins` or `@dog.yard` without `disable_joins` would raise an error because databases are unable to handle joins across clusters. With the `disable_joins` option, Rails will generate

multiple select queries to avoid attempting joining across clusters. For the above association, `@dog.treats` would generate the following SQL:

```
SELECT "humans"."id" FROM "humans" WHERE "humans"."dog_id" = ?  [["dog_id", 1]]
SELECT "treats".* FROM "treats" WHERE "treats"."human_id" IN (?, ?, ?)  [["human_id", 1], ["
```

While `@dog.yard` would generate the following SQL:

```
SELECT "home"."id" FROM "homes" WHERE "homes"."dog_id" = ? [["dog_id", 1]]
SELECT "yards".* FROM "yards" WHERE "yards"."home_id" = ? [["home_id", 1]]
```

There are some important things to be aware of with this option:

1) There may be performance implications since now two or more queries will be performed (depending on the association) rather than a join. If the select for `humans` returned a high number of IDs the select for `treats` may send too many IDs.
2) Since we are no longer performing joins, a query with an order or limit is now sorted in-memory since order from one table cannot be applied to another table.
3) This setting must be added to all associations where you want joining to be disabled. Rails can't guess this for you because association loading is lazy, to load `treats` in `@dog.treats` Rails already needs to know what SQL should be generated.

Schema Caching

If you want to load a schema cache for each database you must set a `schema_cache_path` in each database configuration and set `config.active_record.lazily_load_schema_cach = true` in your application configuration. Note that this will lazily load the cache when the database connections are established.

Caveats

Load Balancing Replicas

Rails also doesn't support automatic load balancing of replicas. This is very dependent on your infrastructure. We may implement basic, primitive load balancing in the future, but for an application at scale this should be something your application handles outside of Rails.