Neo4j.rb Documentation

Release 8.2.5

Chris Grigg, Brian Underwood

Sep 25, 2017

Contents

Contents:

CHAPTER 1

Introduction

Neo4j.rb is an ActiveRecord-inspired OGM (Object Graph Mapping, like ORM) for Ruby supporting Neo4j 2.1+.

Terminology

Neo4j

Node An Object or Entity which has a distinct identity. Can store arbitrary properties with values

Label A means of identifying nodes. Nodes can have zero or more labels. While similar in concept to relational table names, nodes can have multiple labels (i.e. a node could have the labels `Person` and `Teacher`)

Relationship A link from one node to another. Can store arbitrary properties with values. A direction is required but relationships can be traversed bi-directionally without a performance impact.

Type Relationships always have exactly one **type** which describes how it is relating it's source and destination nodes (i.e. a relationship with a `FRIEND_OF` type might connect two `Person` nodes)

Neo4j.rb

Neo4j.rb consists of the *neo4j* and *neo4j-core* gems.

neo4j Provides `ActiveNode` and `ActiveRel` modules for object modeling. Introduces *Model* and *Association* concepts (see below). Depends on `neo4j-core` and thus both are available when `neo4j` is used

neo4j-core Provides low-level connectivity, transactions, and response object wrapping. Includes `Query` class for generating Cypher queries with Ruby method chaining.

Model A Ruby class including either the `Neo4j::ActiveNode` module (for modeling nodes) or the `Neo4j::ActiveRel` module (for modeling relationships) from the `neo4j` gem. These modules give classes the ability to define properties, associations, validations, and callbacks

Association Defined on an `ActiveNode` model. Defines either a `has_one` or `has_many` relationship to a model. A higher level abstraction of a **Relationship**

Code Examples

With Neo4j.rb, you can use either high-level abstractions for convenience or low level APIs for flexibility.

ActiveNode

ActiveNode provides an Object Graph Model (OGM) for abstracting Neo4j concepts with an `ActiveRecord`-like API:

```
# Models to create nodes
person = Person.create(name: 'James', age: 15)

# Get object by attributes
person = Person.find_by(name: 'James', age: 15)

# Associations to traverse relationships
person.houses.map(&:address)

# Method-chaining to build and execute queries
Person.where(name: 'James').order(age: :desc).first

# Query building methods can be chained with associations
# Here we get other owners for pre-2005 vehicles owned by the person in question
person.vehicles(:v).where('v.year < 2005').owners(:other).to_a
```

Setup

See the next section for instructions on *Setup*

CHAPTER 2

Setup

The neo4j.rb gems (`neo4j` and `neo4j-core`) support both Ruby and JRuby and can be used with many different frameworks and services. If you're just looking to get started you'll probably want to use the `neo4j` gem which includes `neo4j-core` as a dependency.

Below are some instructions on how to get started:

Ruby on Rails

The following contains instructions on how to setup Neo4j with Rails. If you prefer a video to follow along you can use this YouTube video

There are two ways to add neo4j to your Rails project. You can *generate a new project* with Neo4j as the default model mapper or you can *add it manually*.

Generating a new app

To create a new Rails app with Neo4j as the default model mapper use `-m` to run a script from the Neo4j project and `-O` to exclude ActiveRecord like so:

```
rails new myapp -m http://neo4jrb.io/neo4j/neo4j.rb -O
```

Note: Due to network issues sometimes you may need to run this command two or three times for the file to download correctly

An example series of setup commands:

```
rails new myapp -m http://neo4jrb.io/neo4j/neo4j.rb -O
cd myapp
rake neo4j:install[community-latest]
rake neo4j:start
```

```
rails generate scaffold User name:string email:string
rails s
open http://localhost:3000/users
```

See also:

Adding the gem to an existing project

Include in your `Gemfile`:

```
# for rubygems
gem 'neo4j', '~> 7.0.0'
```

In `application.rb`:

```
require 'neo4j/railtie'
```

Note: Neo4j does not interfere with ActiveRecord and both can be used in the same application

If you want the `rails generate` command to generate Neo4j models by default you can modify `application.rb` like so:

```
class Application < Rails::Application
  # ...

  config.generators { |g| g.orm :neo4j }
end
```

Rails configuration

For both new apps and existing apps there are multiple ways to configure how to connect to Neo4j. You can use environment variables, the `config/neo4j.yml` file, or configure via the Rails application config.

For environment variables:

```
NEO4J_URL=http://localhost:7474
NEO4J_URL=http://user:pass@localhost:7474

NEO4J_TYPE=bolt
NEO4J_URL=bolt://user:pass@localhost:7687

# jRuby only
NEO4J_TYPE=embedded
NEO4J_PATH=/path/to/graph.db
```

For the `config/neo4j.yml` file:

```
development:
  type: http
  url: http://localhost:7474

test:
```

```
  type: http
  url: http://localhost:7575

production:
  type: http
  url: http://neo4j:password@localhost:7000
```

The *railtie* provided by the *neo4j* gem will automatically look for and load this file.

You can also use your Rails configuration. The following example can be put into `config/application.rb` or any of your environment configurations (`config/environments/(development|test|production).rb`) file:

```
config.neo4j.session.type = :http
config.neo4j.session.url = 'http://localhost:7474'

# Or, for Bolt

config.neo4j.session.type = :bolt
config.neo4j.session.url = 'bolt://localhost:7687'

# Or, for embedded in jRuby

config.neo4j.session.type = :embedded
config.neo4j.session.path = '/path/to/graph.db'
```

Neo4j requires authentication by default but if you install using the built-in *rake tasks*) authentication is disabled. If you are using authentication you can configure it like this:

```
config.neo4j.session.url = 'http://neo4j:password@localhost:7474'
```

Configuring Faraday

Faraday is used under the covers to connect to Neo4j. You can use the `initialize` option to initialize the Faraday session. Example:

```
# Before 8.0.x of `neo4j` gem
config.neo4j.session.options = {initialize: { ssl: { verify: true }}}

# After 8.0.x of `neo4j` gem
# Switched to allowing a "configurator" since everything can be done there
config.neo4j.session.options = {
  faraday_configurator: proc do |faraday|
    # The default configurator uses typhoeus (it was␣
↪`Faraday::Adapter::NetHttpPersistent` for `neo4j-core` < 7.1.0), so if you override␣
↪the configurator you must specify this
    faraday.adapter :typhoeus
    # Optionally you can instead specify another adaptor
    # faraday.use Faraday::Adapter::NetHttpPersistent

    # If you need to set options which would normally be the second argument of␣
↪`Faraday.new`, you can do the following:
    faraday.options[:open_timeout] = 5
    faraday.options[:timeout] = 65
    faraday.options[:ssl] = { verify: true }
```

```
  end
}
```

If you are just using the `neo4j-core` gem, the configurator can also be set via the Neo4j HTTP adaptor. For example:

```
require 'neo4j/core/cypher_session/adaptors/http'
faraday_configurator = proc do |faraday|
  faraday.adapter :typhoeus
end
http_adaptor = Neo4j::Core::CypherSession::Adaptors::HTTP.new('http://
↪neo4j:pass@localhost:7474', faraday_configurator: faraday_configurator)
```

Any Ruby Project

Include either `neo4j` or `neo4j-core` in your `Gemfile` (`neo4j` includes `neo4j-core` as a dependency):

```
gem 'neo4j', '~> 7.0.0'
# OR
gem 'neo4j-core', '~> 7.0.0'
```

If using only `neo4j-core` you can optionally include the rake tasks (*documentation*) manually in your `Rakefile`:

```
# Both are optional

# To provide tasks to install/start/stop/configure Neo4j
require 'neo4j/rake_tasks'
# Comes from the `neo4j-rake_tasks` gem


# It was formerly requried that you load migrations via a rake task like this:
# load 'neo4j/tasks/migration.rake'
# This is NO LONGER required.  Migrations are included automatically when requiring␣
↪the `neo4j` gem.
```

If you don't already have a server you can install one with the rake tasks from `neo4j_server.rake`. See the (*rake tasks documentation*) for details on how to install, configure, and start/stop a Neo4j server in your project directory.

Connection

To open a session to the neo4j server database:

In Ruby

```
# In JRuby or MRI, using Neo4j Server mode. When the railtie is included, this␣
↪happens automatically.
Neo4j::Session.open(:http)
```

Embedded mode in JRuby

In jRuby you can access the data in server mode as above. If you want to run the database in "embedded" mode, however you can configure it like this:

```
neo4j_adaptor = Neo4j::Core::CypherSession::Adaptors::Embedded.new('/file/path/to/
↪graph.db')
neo4j_session = Neo4j::Core::CypherSession.new(neo4j_adaptor)
```

Embedded mode means that Neo4j is running inside your jRuby process. This allows for direct access to the Neo4j Java APIs for faster and more direct querying.

Using the `neo4j` gem (`ActiveNode` and `ActiveRel`) without Rails

To define your own session for the `neo4j` gem you create a `Neo4j::Core::CypherSession` object and establish it as the current session for the `neo4j` gem with the `ActiveBase` module (this is done automatically in Rails):

```
require 'neo4j/core/cypher_session/adaptors/http'
neo4j_adaptor = Neo4j::Core::CypherSession::Adaptors::HTTP.new('http://
↪user:pass@host:7474')
Neo4j::ActiveBase.on_establish_session { Neo4j::Core::CypherSession.new(neo4j_
↪adaptor) }
```

You could instead use the following, but `on_establish_session` will establish a new session for each thread for thread-safety and thus the above is recommended in general unless you know what you are doing:

```
Neo4j::ActiveBase.current_session = Neo4j::Core::CypherSession.new(neo4j_adaptor)
```

What if I'm integrating with a pre-existing Neo4j database?

When trying to get the `neo4j` gem to integrate with a pre-existing Neo4j database instance (common in cases of migrating data from a legacy SQL database into a Neo4j-powered rails app), remember that every `ActiveNode` model is required to have an ID property with a `unique` constraint upon it, and that unique ID property will default to `uuid` unless you override it to use a different ID property.

This commonly leads to getting a `Neo4j::DeprecatedSchemaDefinitionError` in Rails when attempting to access a node populated into a Neo4j database directly via Cypher (i.e. when Rails didn't create the node itself). To solve or avoid this problem, be certain to define and constrain as unique a uuid property (or whatever other property you want Rails to treat as the unique ID property) in Cypher when loading the legacy data or use the methods discussed in *Unique IDs*.

Heroku

Add a Neo4j db to your application:

```
# To use GrapheneDB:
heroku addons:create graphenedb

# To use Graph Story:
heroku addons:create graphstory
```

See also:

GrapheneDB https://devcenter.heroku.com/articles/graphenedb For plans: https://addons.heroku.com/graphenedb

Graph Story https://devcenter.heroku.com/articles/graphstory For plans: https://addons.heroku.com/graphstory

CHAPTER 3

Upgrade Guide

Version 8.0 of the `neo4j` gem and version 7.0 of the `neo4j-core` gem introduce the most significant change to the Neo4j.rb project since version 3.0 when we introduced support for the HTTP protocol. With this update comes a number of breaking changes which will be outlined on this page

What has changed

The Neo4j.rb project was origionally created just to support accessing Neo4j's embedded mode Java APIs via jRuby. In version 3.0 HTTP support was introduced, but the resulting code has been showing it's age. An entirely new API has been created in the `neo4j-core` gem. The goal of this new API is only to support making Cypher queries to Neo4j either via HTTP, Bolt (Neo4j 3.0's new binary protocol), or embedded mode in jRuby. The old code is still around to support connecting to Neo4j via it's Java APIs, but we would like to later replace it with something simpler (perhaps in another gem).

The `neo4j` gem (which provides the `ActiveNode` and `ActiveRel` modules) has been refactored to use the new API in `neo4j-core`. Because of this if you are using `ActiveNode`/`ActiveRel` not much should change.

Before upgrading, the first thing that you should do is to upgrade to the latest 7.1.x version of the `neo4j` gem and the latest `6.1.x` version of the `neo4j-core` gem. The upgrade from any previous gem > `3.0` should not be too difficult, but we are always happy to help on Gitter or Stackoverflow if you are having trouble

The `neo4j-core` gem

If you are using the `neo4j-core` gem without the `neo4j` gem, you should be able to continue using it as you have previously. It is recommended, however, that you refactor your code to use the new API. Some advantages of the new API:

- The new binary protocol ("Bolt") is supported
- You can make multiple queries at a time
- The interface is simpler

- Node and relationship objects don't change depending on the underlying query mechanism (Bolt/HTTP/embedded)
- `Path` objects are now returned

One thing to note is that Node and Relationship objects in the new API are, by design, simple objects. In the old API you could get relationships and other information by calling methods on node or relationship objects. In the new API you must create Cypher queries for all data access.

The new API

To make a new session, you must first create an adaptor object and then provide it to the session `new` method:

```
neo4j_adaptor = Neo4j::Core::CypherSession::Adaptors::Bolt.new('bolt://
↪user:pass@host:port', options)
# or
neo4j_adaptor = Neo4j::Core::CypherSession::Adaptors::HTTP.new('http://
↪user:pass@host:port', options)
# or
neo4j_adaptor = Neo4j::Core::CypherSession::Adaptors::Embedded.new('path/to/db',␣
↪options)

neo4j_session = Neo4j::Core::CypherSession.new(neo4j_adaptor)
```

With your session object, you can make queries in a number of different ways:

```
# Basic query
neo4j_session.query('MATCH (n) RETURN n LIMIT 10')

# Query with parameters
neo4j_session.query('MATCH (n) RETURN n LIMIT {limit}', limit: 10)
```

Or via the *Neo4j::Core::Query* class

```
query_obj = Neo4j::Core::Query.new.match(:n).return(:n).limit(10)

neo4j_session.query(query_obj)
```

Making multiple queries with one request is support with the HTTP Adaptor:

```
results = neo4j_session.queries do
  append 'MATCH (n:Foo) RETURN n LIMIT 10'
  append 'MATCH (n:Bar) RETURN n LIMIT 5'
end

results[0] # results of first query
results[1] # results of second query
```

When doing batched queries, there is also a shortcut for getting a new *Neo4j::Core::Query*:

```
results = neo4j_session.queries do
  append query.match(:n).return(:n).limit(10)
end

results[0] # result
```

The neo4j gem

Sessions

In 7.0 of the neo4j-core gem, the new API doesn't have the concept of a "current" session in the way that the old API did. If you are using neo4j-core, you must keep track of whatever sessions that you open yourself. In version 8.0 of the neo4j gem, however, there is a concept of a current session for your models. Previously you might have used:

```
Neo4j::Session.current
```

If you are using version 8.0 of the neo4j gem, that will be accessible, but neo4j is no longer using that old API to have a session with Neo4j. Instead you might use:

```
Neo4j::ActiveBase.current_session
```

server_db

In previous version of the neo4j gem to connect to Neo4j via HTTP you would define the value server_db in the neo4j.yml file, the NEO4J_TYPE environment variable, or a Rails configuration (config.neo4j.session.type). This should now be replaced and either bolt or http should be used depending on which connection type you need.

Also, instead of using *session_type*, *session_url*, *session_path*, and *session_options*, you should use *session.type*, *session.url*, *session.path*, and *session.options* respectively.

Some examples:

```
# config/neo4j.yml
# Before
development:
  type: server_db
  url: http://localhost:7474

# After
development:
  type: http # or bolt
  url: http://localhost:7474
```

```
# Rails config/application.rb, config/environments/development.rb, etc...

# Before
config.neo4j.session_type = :server_db
config.neo4j.session_url = 'http://localhost:7474'

# After
config.neo4j.session.type = :http # or :bolt
config.neo4j.session.url = 'http://localhost:7474'
```

Also, there was a slight change in the way that you configure the internal faraday adaptor of the neo4j-core gem:

```
# Before 8.0.x of `neo4j` gem
config.neo4j.session_options = {initialize: { ssl: { verify: true }}}
```

```
# After 8.0.x of `neo4j` gem
config.neo4j.session.options = {faraday_options: { ssl: { verify: true }}}
```

Outside of Rails

The `neo4j` gem will automatically set up a number of things with it's `railtie`. If you aren't using Rails you may need to set some things up yourself and some of the details have changed with version 8.0 of the `neo4j` gem.

Previously a connection with be established with `Neo4j::Session.open` and the default session from `neo4j-core` would be used. In version 7.0 of the `neo4j-core` gem, no such default session exists for the new API so you will need to establish a session to use the `ActiveNode` and `ActiveRel` modules like so:

```
adaptor = Neo4j::Core::CypherSession::Adaptors::HTTP.new('http://
↪username:password@localhost:7474', wrap_level: :proc)

session = Neo4j::Core::CypherSession.new(adaptor)

Neo4j::ActiveBase.current_session = session

# Or skip setting up the session yourself:

Neo4j::ActiveBase.current_adaptor = adaptor
```

If you are using multiple threads, you should use the *on_establish_session* method to define how to setup your session. The *current_session* is stored on a per-thread basis and if you spawn a new thread, this block will be used to establish the session for that thread:

```
Neo4j::ActiveBase.on_establish_session do
  adaptor = Neo4j::Core::CypherSession::Adaptors::HTTP.new('http://
↪username:password@localhost:7474', wrap_level: :proc)

  Neo4j::Core::CypherSession.new(adaptor)
end
```

Migrations:

If you would like to use the migrations provided by the `neo4j` outside of Rails you can include this in your `Rakefile`:

```
load 'neo4j/tasks/migration.rake'
```

Indexes and Constraints

In previous versions of the `neo4j` gem, `ActiveNode` models would allow you to define indexes and constraints as part of the model. While this was a convenient feature, it would often cause problems because Neo4j does not allow schema changes to happen in the same transaction as data changes. This would often happen when using `ActiveNode` because constraints and indexes would be automatically created when your model was first loaded, which may very well be in the middle of a transaction.

In version 8.0 of the `neo4j` gem, you must now create indexes and constraints separately. You can do this yourself, but version 8.0 provides fully featured migration functionality to make this easy (see the *Migrations* section).

If you still have indexes or constraints defined, the gem will check to see if those indexes or constraints exist. If they don't, an exception will be raised with command that you can run to generate the appropriate migrations. If they do exist, a warning will be given to remove the index / constraint definitions.

Also note that all `ActiveNode` models must have an `id_property` defined (which is the `uuid` property by default). These constraints will also be checked and an exception will be raised if they do not exist.

Migrations

Version 8.0 of the `neo4j` gem now includes a fully featured migration system similar to the one provided by `ActiveRecord`. See the *documentation* for details.

neo_id id_properties

In version 8.0 of the `neo4j` gem support was added to allow for defininining the internal Neo4j ID as the `id_property` for a model like so:

```
id_property :neo_id
```

Warning: Use of `neo_id` as a perminent identifier should be done with caution. Neo4j can recycle IDs from deleted nodes meaning that URLs or other external references using that ID will reference the wrong item. Neo4j may be updated in the future to support internal IDs which aren't recycled, but for now use at your own risk

Exceptions

With the new API comes some new exceptions which are raised. With the new adaptor API errors are more dependable across different ways of connecting to Neo4j.

Old Exception	New Exception
Neo4j::Server::Resource::ServerException	Neo4j::Core::CypherSession::ConnectionFailedError
Neo4j::Server::CypherResponse::ConstraintViolationError	Neo4j::Core::CypherSession::SchemaErrors::ConstraintValidationFailedError
Neo4j::Session::CypherError	Neo4j::Core::CypherSession::CypherError
?	ConstraintAlreadyExistsError
?	IndexAlreadyExistsError

CHAPTER 4

Rake Tasks

The `neo4j-core` gem (automatically included with the `neo4j` gem) includes some rake tasks which make it easy to install and manage a Neo4j server in the same directory as your Ruby project.

Warning: Setting up a Neo4j server with the rake tasks below will disable authentication and is thus most useful for development and test environments. Note that installing Neo4j on production can be as straightforward as downloading, unzipping, and starting your server.

Note: If you are using zsh, you need to prefix any rake tasks with arguments with the noglob command, e.g. `$ noglob bundle exec rake neo4j:install[community-latest]`.

neo4j:generate_schema_migration

Arguments Either the string *index* or the string *constraint*

The Neo4j label

The property

Example: rake neo4j:generate_schema_migration[constraint,Person,uuid]

Creates a migration which force creates either a constraint or an index in the database for the given label / property pair. When you create a model the gem will require that a migration be created and run and it will give you the appropriate rake task in the exception.

neo4j:install **Arguments:** `version` and `environment` (environment default is *development*)

Example: `rake neo4j:install[community-latest,development]`

Downloads and installs Neo4j into `$PROJECT_DIR/db/neo4j/<environment>/`

For the `version` argument you can specify either `community-latest`/`enterprise-latest` to get the most up-to-date stable version or you can specify a specific version with the format `community-x.x.x`/`enterprise-x.x.x`

A custom download URL can be specified using the `NEO4J_DIST` environment variable like `NEO4J_DIST=http://dist.neo4j.org/neo4j-VERSION-unix.tar.gz`

neo4j:config **Arguments:** `environment` and `port`

Example: `rake neo4j:config[development,7100]`

Configure the port which Neo4j runs on. This affects the HTTP REST interface and the web console address. This also sets the HTTPS port to the specified port minus one (so if you specify 7100 then the HTTP port will be 7099)

neo4j:start **Arguments:** `environment`

Example: `rake neo4j:start[development]`

Start the Neo4j server

Assuming everything is ok, point your browser to http://localhost:7474 and the Neo4j web console should load up.

neo4j:start **Arguments:** `environment`

Example: `rake neo4j:shell[development]`

Open a Neo4j shell console (REPL shell).

If Neo4j isn't already started this task will first start the server and shut it down after the shell is exited.

neo4j:start_no_wait **Arguments:** `environment`

Example: `rake neo4j:start_no_wait[development]`

Start the Neo4j server with the `start-no-wait` command

neo4j:stop **Arguments:** `environment`

Example: `rake neo4j:stop[development]`

Stop the Neo4j server

neo4j:restart **Arguments:** `environment`

Example: `rake neo4j:restart[development]`

Restart the Neo4j server

CHAPTER 5

ActiveNode

ActiveNode is the ActiveRecord replacement module for Rails. Its syntax should be familiar for ActiveRecord users but has some unique qualities.

To use ActiveNode, include Neo4j::ActiveNode in a class.

```
class Post
  include Neo4j::ActiveNode
end
```

Properties

All properties for Neo4j::ActiveNode objects must be declared (unlike neo4j-core nodes). Properties are declared using the property method which is the same as attribute from the active_attr gem.

Example:

```
class Post
  include Neo4j::ActiveNode
  property :title, index: :exact
  property :text, default: 'bla bla bla'
  property :score, type: Integer, default: 0

  validates :title, :presence => true
  validates :score, numericality: { only_integer: true }

  before_save do
    self.score = score * 100
  end

  has_n :friends
end
```

Properties can be indexed using the index argument on the property method, see example above.

See the Properties section for additional information.

See also:

Labels

By default `ActiveNode` takes your model class' name and uses it directly as the Neo4j label for the nodes it represents. This even includes using the module namespace of the class. That is, the class `MyClass` in the `MyModule` module will have the label `MyModule::MyClass`. To change this behavior, see the *module_handling* configuration variable.

Additionally you can change the name of a particular `ActiveNode` by using `mapped_label_name` like so:

```
class Post
  include Neo4j::ActiveNode

  self.mapped_label_name = 'BlogPost'
end
```

Indexes and Constraints

To declare a index on a constraint on a property, you should create a migration. See *Migrations*

Note: In previous versions of `ActiveNode` indexes and constraints were defined on properties directly on the models and were automatically created. This turned out to be not safe, and migrations are now required to create indexes and migrations.

Labels

The class name maps directly to the label. In the following case both the class name and label are `Post`

```
class Post
  include Neo4j::ActiveNode
end
```

If you want to specify a different label for your class you can use `mapped_label_name`:

```
class Post
  include Neo4j::ActiveNode

  self.mapped_label_name = 'BlogPost'
end
```

If you would like to use multiple labels you can use class inheritance. In the following case object created with the *Article* model would have both *Post* and *Article* labels. When querying *Article* both labels are required on the nodes as well.

```
class Post
  include Neo4j::ActiveNode
end

class Article < Post
end
```

Serialization

Pass a property name as a symbol to the serialize method if you want to save JSON serializable data (strings, numbers, hash, array, array with mixed object types*, etc.) to the database.

```
class Student
  include Neo4j::ActiveNode

  property :links

  serialize :links
end

s = Student.create(links: { neo4j: 'http://www.neo4j.org', neotech: 'http://www.
↪neotechnology.com' })
s.links
# => {"neo4j"=>"http://www.neo4j.org", "neotech"=>"http://www.neotechnology.com"}
s.links.class
# => Hash
```

Neo4j.rb serializes as JSON by default but pass it the constant Hash as a second parameter to serialize as YAML. Those coming from ActiveRecord will recognize this behavior, though Rails serializes as YAML by default.

Neo4j allows you to save Ruby arrays to undefined or String types but their contents need to all be of the same type. You can do user.stuff = [1, 2, 3] or user.stuff = ["beer, "pizza", "doritos"] but not user.stuff = [1, "beer", "pizza"]. If you wanted to do that, you could call serialize on your property in the model.

Enums

You can declare special properties that maps an integer value in the database with a set of keywords, like `ActiveRecord::Enum`

```
class Media
  include Neo4j::ActiveNode

  enum type: [:image, :video, :unknown]
end

media = Media.create(type: :video)
media.type
# => :video
media.image!
media.image?
# => true
```

For every keyword specified, a couple of methods are defined to set or check the current enum state (In the example: *image?*, *image!*, *video?*, ...).

With options `_prefix` and `_suffix`, you can define how this methods are generating, by adding a prefix or a suffix.

With `_prefix: :something`, something will be added before every method name.

```
Media.enum type: [:image, :video, :unknown], _prefix: :something
media.something_image?
media.something_image!
```

With `_suffix: true`, instead, the name of the enum is added in the bottom of all methods:

```
Media.enum type: [:image, :video, :unknown], _suffix: true
media.image_type?
media.image_type!
```

You can find elements by enum value by using a set of scope that `enum` defines:

```
Media.image
# => CYPHER: "MATCH (result_media:`Media`) WHERE (result_media.type = 0)"
Media.video
# => CYPHER: "MATCH (result_media:`Media`) WHERE (result_media.type = 1)"
```

Or by using `where`:

```
Media.where(type: :image)
# => CYPHER: "MATCH (result_media:`Media`) WHERE (result_media.type = 0)"
Media.where(type: [Media.types[:image], Media.types[:video]])
# => CYPHER: "MATCH (result_media:`StoredFile`) WHERE (result_media.type IN [0, 1])"
Media.as(:m).where('m.type <> ?', Media.types[:image])
# => CYPHER: "MATCH (result_media:`StoredFile`) WHERE (result_media.type <> 0)"
```

By default, every `enum` property will be defined as `unique`, to improve query performances. If you want to disable this, simply pass `_index: false` to `enum`:

```
class Media
  include Neo4j::ActiveNode

  enum type: [:image, :video, :unknown], _index: false
end
```

Sometimes it is desirable to have a default value for an `enum` property. To acheive this, you can simply define a property with the same name which defines a default value:

```
class Media
  include Neo4j::ActiveNode

  enum type: [:image, :video, :unknown]
  property :type, default: :video
end
```

Scopes

Scopes in `ActiveNode` are a way of defining a subset of nodes for a particular `ActiveNode` model. This could be as simple as:

```
class Person
  include Neo4j::ActiveNode

  scope :minors, -> { where(age: 0..17) }
end
```

This allows you chain a description of the defined set of nodes which can make your code easier to read such as `Person.minors` or `Car.all.owners.minors`. While scopes are very useful in encapsulating logic, this scope doesn't neccessarily save us much beyond simply using `Person.where(age: 0..17)` directly. Scopes become much more useful when they encapsulate more complicated logic:

```ruby
class Person
  include Neo4j::ActiveNode

  scope :eligible, -> { where_not(age: 0..17).where(completed_form: true) }
end
```

And because you can chain scopes together, this can make your query chains very composable and expressive like:

```ruby
# Getting all hybrid convertables owned by recently active eligible people
Person.eligible.where(recently_active: true).cars.hybrids.convertables
```

While that's useful in of itself, sometimes you want to be able to create more dynamic scopes by passing arguments. This is supported like so:

```ruby
class Person
  include Neo4j::ActiveNode

  scope :around_age_of, -> (age) { where(age: (age - 5..age + 5)) }
end

# Which can be used as:
Person.around_age_of(20)
# or
Car.all.owners.around_age_of(20)
```

All of the examples so far have used the Ruby API for automatically generating Cypher. While it is often possible to get by with this, it is sometimes not possible to create a scope without defining it with a Cypher string. For example, if you need to use `OR`:

```ruby
class Person
  include Neo4j::ActiveNode

  scope :non_teenagers, -> { where("#{identity}.age < 13 OR #{identity}.age >= 18") }
end
```

Since a Cypher query can have a number of different nodes and relationships that it is referencing, we need to be able to refer to the current node's variable. This is why we call the `identity` method, which will give the variable which is being used in the query chain on which the scope is being called.

Warning: Since the `identity` comes from whatever was specified as the cypher variable for the node on the other side of the association. If the cypher variables were generated from an untrusted source (like from a user of your app) you may leave yourself open to a Cypher injection vulnerability. It is not recommended to generate your Cypher variables based on user input!

Finally, the `scope` method just gives us a convenient way of having a method on our model class which returns another query chain object. Sometimes to make even more complex logic or even to just return a simple result which can be called on a query chain but which doesn't continue the chain, we can create a class method ourselves:

```ruby
class Person
  include Neo4j::ActiveNode

  def self.average_age
    all(:person).pluck('avg(person.age)').first
  end
end
```

So if you wanted to find the average age of all eligible people, you could call `Person.eligible.average_age` and you would be given a single number.

To implement a more complicated scope with a class method you simply need to return a query chain at the end.

Wrapping

When loading a node from the database there is a process to determine which `ActiveNode` model to choose for wrapping the node. If nothing is configured on your part then when a node is created labels will be saved representing all of the classes in the hierarchy.

That is, if you have a `Teacher` class inheriting from a `Person` model, then creating a `Person` object will create a node in the database with a `Person` label, but creating a `Teacher` object will create a node with both the `Teacher` and `Person` labels.

If there is a value for the property defined by *class_name_property* then the value of that property will be used directly to determine the class to wrap the node in.

Callbacks

Implements like Active Records the following callback hooks:

- initialize
- validation
- find
- save
- create
- update
- destroy

created_at, updated_at

```
class Blog
  include Neo4j::ActiveNode

  include Neo4j::Timestamps # will give model created_at and updated_at timestamps
  include Neo4j::Timestamps::Created # will give model created_at timestamp
  include Neo4j::Timestamps::Updated # will give model updated_at timestamp
end
```

Validation

Support the Active Model validation, such as:

validates :age, presence: true validates_uniqueness_of :name, :scope => :adult

id property (primary key)

Unique IDs are automatically created for all nodes using SecureRandom::uuid. See *UniqueIDs* for details.

Associations

`has_many` and `has_one` associations can also be defined on `ActiveNode` models to make querying and creating relationships easier.

```
class Post
  include Neo4j::ActiveNode
  has_many :in, :comments, origin: :post
  has_one :out, :author, type: :author, model_class: :Person
end

class Comment
  include Neo4j::ActiveNode
  has_one :out, :post, type: :post
  has_one :out, :author, type: :author, model_class: :Person
end

class Person
  include Neo4j::ActiveNode
  has_many :in, :posts, origin: :author
  has_many :in, :comments, origin: :author

  # Match all incoming relationship types
  has_many :in, :written_things, type: false, model_class: [:Post, :Comment]

  # or if you want to match all model classes:
  # has_many :in, :written_things, type: false, model_class: false

  # or if you watch to match Posts and Comments on all relationships (in and out)
  # has_many :both, :written_things, type: false, model_class: [:Post, :Comment]
end
```

You can query associations:

```
post.comments.to_a          # Array of comments
comment.post                # Post object
comment.post.comments       # Original comment and all of it's siblings.  Makes just␣
↪one query
post.comments.author.posts # All posts of people who have commented on the post. ␣
↪Still makes just one query
```

You can create associations

```
post.comments = [comment1, comment2]  # Removes all existing relationships
post.comments << comment3             # Creates new relationship

comment.post = post1                  # Removes all existing relationships
```

Updating Associations

You can update attributes for objects of an association like this:

```
post.comments.update_all(flagged: true)
post.comments.where(text: /.*cats.*/).update_all(flagged: true)
```

You can even update properties of the relationships for the associations like so:

```
post.comments.update_all_rels(flagged: true)
post.comments.where(text: /.*cats.*/).update_all_rels(flagged: true)
# Or to filter on the relationships
post.comments.where(flagged: nil).update_all_rels(flagged: true)
```

Dependent Associations

Similar to ActiveRecord, you can specify four `dependent` options when declaring an association.

```
class Route
  include Neo4j::ActiveNode
  has_many :out, :stops, type: :STOPPING_AT, dependent: :delete_orphans
end
```

The available options are:

- `:delete`, which will delete all associated records in Cypher. Callbacks will not be called. This is the fastest method.
- `:destroy`, which will call `each` on the association and then `destroy` on each related object. Callbacks will be called. Since this happens in Ruby, it can be a very expensive procedure, so use it carefully.
- `:delete_orphans`, which will delete only the associated records that have no other relationships of the same type.
- `:destroy_orphans`, same as above, but it takes place in Ruby.

The two orphan-destruction options are unique to Neo4j.rb. As an example of when you'd use them, imagine you are modeling tours, routes, and stops along those routes. A tour can have multiple routes, a route can have multiple stops, a stop can be in multiple routes but must have at least one. When a route is destroyed, `:delete_orphans` would delete only those related stops that have no other routes.

See also:

See also:

#has_many and #has_one

Creating Unique Relationships

By including the `unique` option in a `has_many` or `has_one` association's method call, you can change the Cypher used to create from "CREATE" to "CREATE UNIQUE."

```
has_many :out, :friends, type: 'FRIENDS_WITH', model_class: :User, unique: true
```

Instead of `true`, you can give one of three different options:

- `:none`, also used `true` is given, will not include properties to determine whether ot not to create a unique relationship. This means that no more than one relationship of the same pairing of nodes, rel type, and direction will ever be created.
- `:all`, which will include all set properties in rel creation. This means that if a new relationship will be created unless all nodes, type, direction, and rel properties are matched.
- `{on: [keys]}` will use the keys given to determine whether to create a new rel and the remaining properties will be set afterwards.

Eager Loading

ActiveNode supports eager loading of associations in two ways. The first way is transparent. When you do the following:

```
person.blog_posts.each do |post|
  puts post.title
  puts "Tags: #{post.tags.map(&:name).join(', ')}"
  post.comments.each do |comment|
    puts '  ' + comment.title
  end
end
```

Only three Cypher queries will be made:

- One to get the blog posts for the user
- One to get the tags for all of the blog posts
- One to get the comments for all of the blog posts

While three queries isn't ideal, it is better than the naive approach of one query for every call to an object's association (Thanks to DataMapper for the inspiration).

For those times when you need to load all of your data with one Cypher query, however, you can do the following to give *ActiveNode* a hint:

```
person.blog_posts.with_associations(:tags, :comments).each do |post|
  puts post.title
  puts "Tags: #{post.tags.map(&:name).join(', ')}"
  post.comments.each do |comment|
    puts '  ' + comment.title
  end
end
```

All that we did here was add `.with_associations(:tags, :comments)`. In addition to getting all of the blog posts, this will generate a Cypher query which uses the Cypher *COLLECT()* function to efficiently roll-up all of the associated objects. *ActiveNode* then automatically structures them into a nested set of *ActiveNode* objects for you.

CHAPTER 6

ActiveRel

ActiveRel is a module in the `neo4j` gem which wraps relationships. ActiveRel objects share most of their behavior with ActiveNode objects. ActiveRel is purely optional and offers advanced functionality for complex relationships.

When to Use?

It is not always necessary to use ActiveRel models but if you have the need for validation, callback, or working with properties on unpersisted relationships, it is the solution.

Note that in Neo4j it isn't possible to access relationships except by first accessing a node. Thus `ActiveRel` doesn't implement a `uuid` property like `ActiveNode`.

Setup

ActiveRel model definitions have three requirements:

- `include Neo4j::ActiveRel`
- Call `from_class` with a symbol/string referring to an `ActiveNode` model or :any
- Call `to_class` with a symbol/string referring to an `ActiveNode` model or :any

See the note on from/to at the end of this page for additional information.

```
# app/models/enrolled_in.rb
class EnrolledIn
  include Neo4j::ActiveRel
  before_save :do_this

  from_class :Student
  to_class   :Lesson
  # `type` can be specified, but it is assumed from the model name
  # In this case, without `type`, 'ENROLLED_IN' would be assumed
```

```
  # If you wanted to specify something else:
  # type 'ENROLLED'

  property :since, type: Integer
  property :grade, type: Integer
  property :notes

  validates_presence_of :since

  def do_this
    #a callback
  end
end

# Using the `ActiveRel` model in `ActiveNode` models:
# app/models/student.rb
class Student
  include Neo4j::ActiveNode

  has_many :out, :lessons, rel_class: :EnrolledIn
end

# app/models/lesson.rb
class Lesson
  include Neo4j::ActiveNode

  has_many :in, :students, rel_class: :EnrolledIn
end
```

See also:

Relationship Creation

From an ActiveRel Model

Once setup, ActiveRel models follow the same rules as ActiveNode in regard to properties. Declare them to create setter/getter methods. You can also set `created_at` or `updated_at` for automatic timestamps.

ActiveRel instances require related nodes before they can be saved. Set these using the from_node and to_node methods.

```
rel = EnrolledIn.new
rel.from_node = student
rel.to_node = lesson
```

You can pass these as parameters when calling new or create if you so choose.

```
rel = EnrolledIn.new(from_node: student, to_node: lesson)
#or
rel = EnrolledIn.create(from_node: student, to_node: lesson)
```

From a *has_many* or *has_one* association

Add the :rel_class option to an association with the name of an ActiveRel model. Association creation and querying will use this rel class, verifying classes, adding defaults, and performing callbacks.

```
class Student
  include Neo4j::ActiveNode
  has_many :out, :lessons, rel_class: :EnrolledIn
end
```

Creating Unique Relationships

The `creates_unique` class method will change the Cypher generated during rel creation from `CREATE` to `CREATE UNIQUE`. It may be called with one optional argument of the following:

- `:none`, also used when no argument is given, will not include properties to determine whether ot not to create a unique relationship. This means that no more than one relationship of the same pairing of nodes, rel type, and direction will ever be created.
- `:all`, which will include all set properties in rel creation. This means that if a new relationship will be created unless all nodes, type, direction, and rel properties are matched.
- `{on: [keys]}` will use the keys given to determine whether to create a new rel and the remaining properties will be set afterwards.

Query and Loading existing relationships

Like nodes, you can load relationships a few different ways.

:each_rel, :each_with_rel, or :pluck methods

Any of these methods can return relationship objects.

```
Student.first.lessons.each_rel { |r| }
Student.first.lessons.each_with_rel { |node, rel| }
Student.first.query_as(:s).match('(s)-[rel1:\`enrolled_in\`]->(n2)').pluck(:rel1)
```

These are available as both class or instance methods. Because both each_rel and each_with_rel return enumerables when a block is skipped, you can take advantage of the full suite of enumerable methods:

```
Lesson.first.students.each_with_rel.select{ |n, r| r.grade > 85 }
```

Be aware that select would be performed in Ruby after a Cypher query is performed. The example above performs a Cypher query that matches all students with relationships of type enrolled_in to Lesson.first, then it would call select on that.

Accessing related nodes

Once a relationship has been wrapped, you can access the related nodes using from_node and to_node instance methods. Note that these cannot be changed once a relationship has been created.

```
student = Student.first
lesson = Lesson.first
rel = EnrolledIn.create(from_node: student, to_node: lesson, since: 2014)
rel.from_node
=> #<Neo4j::ActiveRel::RelatedNode:0x00000104589d78 @node=#<Student property: 'value'>
↪>
rel.to_node
=> #<Neo4j::ActiveRel::RelatedNode:0x00000104589d50 @node=#<Lesson property: 'value'>>
```

As you can see, this returns objects of type RelatedNode which delegate to the nodes. This allows for lazy loading when a relationship is returned in the future: the nodes are not loaded until you interact with them, which is beneficial with something like each_with_rel where you already have access to the nodes and do not want superfluous calls to the server.

Advanced Usage

Separation of Relationship Logic

ActiveRel really shines when you have multiple associations that share a relationship type. You can use an ActiveRel model to separate the relationship logic and just let the node models be concerned with the labels of related objects.

```
class User
  include Neo4j::ActiveNode
  property :managed_stats, type: Integer #store the number of managed objects to␣
↪improve performance

  has_many :out, :managed_lessons,  model_class: :Lesson,  rel_class: :ManagedRel
  has_many :out, :managed_teachers, model_class: :Teacher, rel_class: :ManagedRel
  has_many :out, :managed_events,   model_class: :Event,   rel_class: :ManagedRel
  has_many :out, :managed_objects,  model_class: false,    rel_class: :ManagedRel

  def update_stats
    managed_stats += 1
    save
  end
end

class ManagedRel
  include Neo4j::ActiveRel
  after_create :update_user_stats
  validate :manageable_object
  from_class :User
  to_class :any
  type 'MANAGES'

  def update_user_stats
    from_node.update_stats
  end

  def manageable_object
    errors.add(:to_node) unless to_node.respond_to?(:managed_by)
  end
end

# elsewhere
```

```
rel = ManagedRel.new(from_node: user, to_node: any_node)
if rel.save
  # validation passed, to_node is a manageable object
else
  # something is wrong
end
```

Additional methods

`:type` instance method, `_:type` class method: return the relationship type of the model

`:_from_class` and `:_to_class` class methods: return the expected classes declared in the model

Regarding: from and to

`:from_node`, `:to_node`, `:from_class`, and `:to_class` all have aliases using `start` and `end`: `:start_class`, `:end_class`, `:start_node`, `:end_node`, `:start_node=`, `:end_node=`. This maintains consistency with elements of the Neo4j::Core API while offering what may be more natural options for Rails users.

CHAPTER 7

Properties

In classes that mixin the `Neo4j::ActiveNode` or `Neo4j::ActiveRel` modules, properties must be declared using the `property` class method. It requires a single argument, a symbol that will correspond with the getter and setter as well as the property in the database.

```
class Post
  include Neo4j::ActiveNode

  property :title
end
```

Two options are also available to both node and relationship models. They are:

- `type`, to specify the expected class of the stored value in Ruby
- `default`, a default value to set when the property is `nil`

Node models have two more options:

- `index: :exact` to declare an exact index in the database
- `constraint: :unique` to declare a unique constraint

Note that a constraint is a type of index, so there is neither need nor ability to use both.

Finally, you can serialize properties as JSON with the *serialize* class method.

In practice, you can put it all together like this:

```
class Post
  include Neo4j::ActiveNode

  property :title, type: String, default: 'This ia new post', index: :exact
  property :links

  serialize :links
end
```

You will now be able to set the `title` property through mass-assignment (`Post.new(title: 'My Title')`) or by calling the *title=* method. You can also give a hash of links (`{ homepage: 'http://neo4jrb.io', twitter: 'https://twitter.com/neo4jrb' }`) to the `links` property and it will be saved as JSON to the db.

Undeclared Properties

Neo4j, being schemaless as far as the database is concerned, does not require that property keys be defined ahead of time. As a result, it's possible (and sometimes desirable) to set properties on the node that are not also defined on the database. By including the module `Neo4j::UndeclaredProperties` no exceptions will be thrown if unknown attributes are passed to selected methods.

```
class Post
  include Neo4j::ActiveNode
  include Neo4j::UndeclaredProperties

  property :title
end

Post.create(title: 'My Post', secret_val: 123)
post = Post.first
post.secret_val #=> NoMethodError: undefined method `secret_val`
post[:secret_val] #=> 123...
```

In this case, simply adding the `secret_val` property to your model will make it available through the `secret_val` method. The module supports undeclared properties in the following methods: *new*, *create*, *[]*, *[]=*, *update_attribute*, *update_attribute!*, *update_attributes* and their corresponding aliases.

Types and Conversion

The `type` option has some interesting qualities that are worth being aware of when developing. It defines the type of object that you expect when returning the value to Ruby, _not_ the type that will be stored in the database. There are a few types available by default.

- String
- Integer
- BigDecimal
- Date
- Time
- DateTime
- Boolean (TrueClass or FalseClass)

Declaring a type is not necessary and, in some cases, is better for performance. You should omit a type declaration if you are confident in the consistency of data going to/from the database.

```
class Post
  include Neo4j::ActiveNode

  property :score, type: Integer
  property :created_at, type: DateTime
end
```

In this model, the `score` property's type will ensure that String interpretations of numbers are always converted to Integer when you return the property in Ruby. As an added bonus, it will convert before saving to the database because Neo4j is capable of storing Ints natively, so you won't have to convert every time. DateTimes, however, are a different beast, because Neo4j cannot handle Ruby's native formats. To work around this, type converter knows to change the DateTime object into an Integer before saving and then, when loading the node, it will convert the Integer back into a DateTime.

This magic comes with a cost. DateTime conversion in particular is expensive and if you are obsessed with speed, you'll find that it slows you down. A tip for those users is to set your timestamps to `type: Integer` and you will end up with Unix timestamps that you can manipulate if/when you need them in friendlier formats.

Custom Converters

It is possible to define custom converters for types not handled natively by the gem.

```
class RangeConverter
  class << self
    def primitive_type
      String
    end

    def convert_type
      Range
    end

    def to_db(value)
      value.to_s
    end

    def to_ruby(value)
      ends = value.to_s.split('..').map { |d| Integer(d) }
      ends[0]..ends[1]
    end
    alias_method :call, :to_ruby
  end

  include Neo4j::Shared::Typecaster
end
```

This would allow you to use `property :my_prop, type: Range` in a model. Each method and the `alias_method` call is required. Make sure the module inclusion happens at the end of the file.

`primitive_type` is used to fool ActiveAttr's type converters, which only recognize a few basic Ruby classes.

`convert_type` must match the constant given to the `type` option.

`to_db` provides logic required to transform your value into the class defined by `primitive_type`. It will store the object in the database as this type.

`to_ruby` provides logic to transform the DB-provided value back into the class expected by code using the property. It shuld return an object of the type set in `convert_type`.

Note the `alias_method` to make `to_ruby` respond to *call*. This is to provide compatibility with the `ActiveAttr` dependency.

An optional method, `converted?(value)` can be defined. This should return a boolean indicating whether a value is already of the expected type for Neo4j.

CHAPTER 8

Unique IDs

The database generates unique IDs and they are accessible from all nodes and relationships using the `neo_id` method. These keys are somewhat volatile and may be reused or change throughout a database's lifetime, so they are unsafe to use within an application.

Neo4j.rb requires you to define which key should act as primary key on `Neo4j::ActiveNode` classes instead of using the internal Neo4j ids. By default, ActiveNode will generate a unique ID using `SecureRandom::uuid` saving it in a `uuid` property. The instance method `id` will also point to this.

You can define a global or per-model generation methods if you do not want to use the default. Additionally, you can change the property that will be aliased to the `id` method. This can be done through *Configuration* or models themselves.

Unique IDs are **not** generated for relationships or ActiveRel models because their IDs should not be used. To query for a relationship, generate a match based from nodes. If you find yourself in situations where you need relationship IDs, you probably need to define a new ActiveNode class!

Defining your own ID

The `on` parameter tells which method is used to generate the unique id.

```
class Person
  include Neo4j::ActiveNode
  id_property :personal_id, on: :phone_and_name

  property :name
  property :phone

  def phone_and_name
    self.name + self.phone # strange example ...
  end
end
```

Using internal Neo4j IDs as id_property

Even if using internal Neo4j ids is not recommended, you can configure your model to use it:

```
class Person
  include Neo4j::ActiveNode
  id_property :neo_id
end
```

A note regarding constraints

A constraint is required for the `id_property` of an `ActiveNode` model. To create constraints, you can run the following command:

```
rake neo4j:generate_schema_migration[constraint,Model,uuid]
```

Replacing `Model` with your model name and `uuid` with another `id_property` if you have specified something else. When you are ready you can run the migrations:

```
rake neo4j:migrate
```

If you forget to do this, an exception will be raised giving you the appropriate command to generate the migration.

Adding IDs to Existing Data

If you have old or imported data in need of IDs, you can use the built-in `populate_id_property` migration helper.

Just create a new migration like this and run it:

```
rails g neo4j:migration PopulateIdProperties
```

```
class PopulateIdProperties < Neo4j::Migrations::Base
  def up
    populate_id_property :MyModel
  end

  def down
    raise IrreversibleMigration
  end
end
```

It will load the model, find its given ID property and generation method, and populate that property on all nodes of that class where an `id_property` is not already assigned. It does this in batches of up to 900 at a time by default, but this can be changed with the `MAX_PER_BATCH` environment variable (batch time taken standardized per node will be shown to help you tune batch size for your DB configuration).

Working with Legacy Schemas

If you already were using uuids, give yourself a pat on the back. Unfortunately, you may run into problems with Neo4j.rb v3. Why? By default Neo4j.rb requires a uuid index and a uuid unique constraint on every *ActiveNode*. You

can change the name of the uuid by adding `id_property` as shown above. But, either way, you're getting `uuid` as a shadow index for your nodes.

If you had a property called `uuid`, you'll have to change it or remove it since `uuid` is now a reserved word. If you want to keep it, your indexes will have to match the style of the default `id_property` (uuid index and unique).

You'll need to use the Neo4J shell or Web Interface.

Step 1: Check Indexes and Constraints

This command will provide a list of indexes and constraints

```
schema
```

Step 2: Clean up any indexes that are not unique using a migration

```
rails g neo4j:migration AddConstraintToTag
```

```
class AddConstraintToTag < Neo4j::Migrations::Base
  def up
    drop_index :Tag, :uuid
    add_constraint :Tag, :uuid
  end

  def down
    drop_constraint :Tag, :uuid
    add_index :Tag, :uuid
  end
end
```

Step 3: Add an id_property to your ActiveNode

```
id_property :uuid, auto: :uuid
```

Note: If you did not have an index or a constraint, Neo4j.rb will automatically create them for you.

CHAPTER 9

Querying

Introduction

If you are using the `neo4j-core` gem, querying is as simple as calling the `query` method on your session object and providing a query and optional parameters:

```
neo4j_session.query('MATCH (n) RETURN n LIMIT {limit}', limit: 10)
```

Using the `neo4j` gem provides a number of additional options. Firstly in the `neo4j` gem, the session is made accessible via a call to `Neo4j::ActiveBase.current_session`. So you could make the above query with:

```
Neo4j::ActiveBase.current_session.query('MATCH (n) RETURN n LIMIT {limit}', limit: 10)
```

Most of the time, though, using the `neo4j` gem involves using the `ActiveNode` and `ActiveRel` APIs as described below.

ActiveNode

Simple Query Methods

There are a number of ways to find and return nodes.

`.find`

Find an object by *id_property*

`.find_by`

`find_by` and `find_by!` behave as they do in ActiveRecord, returning the first object matching the criteria or nil (or an error in the case of `find_by!`)

```
Post.find_by(title: 'Neo4j.rb is awesome')
```

Proxy Method Chaining

Like in ActiveRecord you can build queries via method chaining. This can start in one of three ways:

- `Model.all`
- `Model.association`
- `model_object.association`

In the case of the association calls, the scope becomes a class-level representation of the association's model so far. So for example if I were to call `post.comments` I would end up with a representation of nodes from the `Comment` model, but only those which are related to the `post` object via the `comments` association.

At this point it should be mentioned that what associations return isn't an `Array` but in fact an `AssociationProxy`. `AssociationProxy` is `Enumerable` so you can still iterate over it as a collection. This allows for the method chaining to build queries, but it also enables *eager loading* of associations

If if you call a method such as `where`, you will end up with a `QueryProxy`. Similar to an `AssociationProxy`, a `QueryProxy` represents an enumerable query which hasn't yet been executed and which you can call filtering and sorting methods on as well as chaining further associations.

From an `AssociationProxy` or a `QueryProxy` you can filter, sort, and limit to modify the query that will be performed or call a further association.

Querying the proxy

Similar to ActiveRecord you can perform various operations on a proxy like so:

```
lesson.teachers.where(name: /.* smith/i, age: 34).order(:name).limit(2)
```

The arguments to these methods are translated into `Cypher` query statements. For example in the above statement the regular expression is translated into a Cypher =~ operator. Additionally all values are translated into Neo4j query parameters for the best performance and to avoid query injection attacks.

Chaining associations

As you've seen, it's possible to chain methods to build a query on one model. In addition it's possible to also call associations at any point along the chain to transition to another associated model. The simplest example would be:

```
student.lessons.teachers
```

This would returns all of the teachers for all of the lessons which the students is taking. Keep in mind that this builds only one Cypher query to be executed when the result is enumerated. Finally you can combine scoping and association chaining to create complex cypher query with simple Ruby method calls.

```
student.lessons(:l).where(level: 102).teachers(:t).where('t.age > 34').pluck(:l)
```

Here we get all of the lessons at the 102 level which have a teacher older than 34. The `pluck` method will actually perform the query and return an `Array` result with the lessons in question. There is also a `return` method which returns an `Array` of result objects which, in this case, would respond to a call to the `#l` method to return the lesson.

Note here that we're giving an argument to the associaton methods (`lessons(:l)` and `teachers(:t)`) in order to define Cypher variables which we can refer to. In the same way we can also pass in a second argument to define a variable for the relationship which the association follows:

```
student.lessons(:l, :r).where("r.start_date < {the_date} and r.end_date >= {the_date}
↪").params(the_date: '2014-11-22').pluck(:l)
```

Here we are limiting lessons by the `start_date` and `end_date` on the relationship between the student and the lessons. We can also use the `rel_where` method to filter based on this relationship:

```
student.lessons.where(subject: 'Math').rel_where(grade: 85)
```

See also:

Branching

When making association chains with `ActiveNode` you can use the `branch` method to go down one path before jumping back to continue where you started from. For example:

```
# Finds all exams for the student's lessons where there is a teacher who's age is␣
↪greater than 34
student.lessons.branch { teachers.where('t.age > 34') }.exams

# Similar to the Cypher:
# MATCH (s:Student)-[:HAS_LESSON]->(lesson:Lesson)<-[:TEACHES]-(:Teacher), (lesson)<-
↪[:FOR_LESSON]-(exam:Exam)
# RETURN exam
```

Associations and Unpersisted Nodes

There is some special behavior around association creation when nodes are new and unsaved. Below are a few scenarios and their outcomes.

When both nodes are persisted, associations changes using << or = take place immediately – no need to call save.

```
student = Student.first
Lesson = Lesson.first
student.lessons << lesson
```

In that case, the relationship would be created immediately.

When the node on which the association is called is unpersisted, no changes are made to the database until `save` is called. Once that happens, a cascading save event will occur.

```
student = Student.new
lesson = Lesson.first || Lesson.new
# This method will not save `student` or change relationships in the database:
student.lessons << lesson
```

Once we call `save` on `student`, two or three things will happen:

- Since `student` is unpersisted, it will be saved
- If `lesson` is unpersisted, it will be saved
- Once both nodes are saved, the relationship will be created

This process occurs within a transaction. If any part fails, an error will be raised, the transaction will fail, and no changes will be made to the database.

Finally, if you try to associate an unpersisted node with a persisted node, the unpersisted node will be saved and the relationship will be created immediately:

```
student = Student.first
lesson = Lesson.new
student.lessons << lesson
```

In the above example, `lesson` would be saved and the relationship would be created immediately. There is no need to call `save` on `student`.

Parameters

Neo4j supports parameters which have a number of advantages:

- You don't need to worry about injection attacks when a value is passed as a parameter
- There is no need to worry about escaping values for parameters
- If only the values that you are passing down for a query change, using parameters keeps the query string the same and allows Neo4j to cache the query execution

The Neo4j.rb project gems try as much as possible to use parameters. For example, if you call `where` with a Hash:

```
Student.all.where(age: 20)
```

A parameter will be automatically created for the value passed in.

Don't assume that all methods use parameters. Always check the resulting query!

You can also specify parameters yourself with the `params` method like so:

```
Student.all.where("s.age < {age} AND s.name = {name} AND s.home_town = {home_town}")
  .params(age: 24, name: 'James', home_town: 'Dublin')
  .pluck(:s)
```

Variable-length relationships

Introduced in version 5.1.0

It is possible to specify a variable-length qualifier to apply to relationships when calling association methods.

```
student.friends(rel_length: 2)
```

This would find the friends of friends of a student. Note that you can still name matched nodes and relationships and use those names to build your query as seen above:

```
student.friends(:f, :r, rel_length: 2).where('f.gender = {gender} AND r.since >=
↪{date}').params(gender: 'M', date: 1.month.ago)
```

Note: You can either pass a single options Hash or provide **both** the node and relationship names along with the optional Hash.

There are many ways to provide the length information to generate all the various possibilities Cypher offers:

```
# As a Integer:
## Cypher: -[:`FRIENDS`*2]->
student.friends(rel_length: 2)

# As a Range:
## Cypher: -[:`FRIENDS`*1..3]->
student.friends(rel_length: 1..3) # Get up to 3rd degree friends

# As a Hash:
## Cypher: -[:`FRIENDS`*1..3]->
student.friends(rel_length: {min: 1, max: 3})

## Cypher: -[:`FRIENDS`*0..]->
student.friends(rel_length: {min: 0})

## Cypher: -[:`FRIENDS`*..3]->
student.friends(rel_length: {max: 3})

# As the :any Symbol:
## Cypher: -[:`FRIENDS`*]->
student.friends(rel_length: :any)
```

Caution: By default, "*..3" is equivalent to "*1..3" and "*" is equivalent to "*1..", but this may change depending on your Node4j server configuration. Keep that in mind when using variable-length relationships queries without specifying a minimum value.

Note: When using variable-length relationships queries on *has_one* associations, be aware that multiple nodes could be returned!

The Query API

The `neo4j-core` gem provides a `Query` class which can be used for building very specific queries with method chaining. This can be used either by getting a fresh `Query` object from a `Session` or by building a `Query` off of a scope such as above.

```
Neo4j::ActiveBase.new_query # Get a new Query object

# Get a Query object based on a scope
Student.query_as(:s) # For a
student.lessons.query_as(:l)

# ... and based on an object:
student.query_as(:s)
```

The `Query` class has a set of methods which map directly to Cypher clauses and which return another `Query` object to allow chaining. For example:

```
student.lessons.query_as(:l) # This gives us our first Query object
  .match("l-[:has_category*]->(root_category:Category)").where("NOT(root_category-
↪[:has_category]->())")
  .pluck(:root_category)
```

Here we can make our own `MATCH` clauses unlike in model scoping. We have `where`, `pluck`, and `return` here as well in addition to all of the other clause-methods. See this page for more details.

Note that when using the `Query` API if you make multiple calls to methods it will try to combine the calls together into one clause and even to re-order them. If you want to avoid this you can use the `#break` method:

```
# Creates a query representing the cypher: MATCH (q:Person), (r:Car) MATCH (p:
↪Person)-->(q)
query_obj.match(q: Person).match('r:Car').break.match('(p: Person)-->(q)')
```

TODO Duplicate this page and link to it from here (or just duplicate it here): https://github.com/neo4jrb/neo4j-core/wiki/Queries

See also:

#proxy_as

Sometimes it makes sense to turn a `Query` object into (or back into) a proxy object like you would get from an association. In these cases you can use the *Query#proxy_as* method:

```
student.query_as(:s)
  .match("(s)-[rel:FRIENDS_WITH*1..3]->(s2:Student")
  .proxy_as(Student, :s2).lessons
```

Here we pick up the *s2* variable with the scope of the *Student* model so that we can continue calling associations on it.

`match_to` and `first_rel_to`

There are two methods, match_to and first_rel_to that both make simple patterns easier.

In the most recent release, match_to accepts nodes; in the master branch and in future releases, it will accept a node or an ID. It is essentially shorthand for association.where(neo_id: node.neo_id) and returns a QueryProxy object.

```
# starting from a student, match them to a lesson based off of submitted params, then
↪return students in their classes
student.lessons.match_to(params[:id]).students
```

first_rel_to will return the first relationship found between two nodes in a QueryProxy chain.

```
student.lessons.first_rel_to(lesson)
# or in the master branch, future releases
student.lessons.first_rel_to(lesson.id)
```

This returns a relationship object.

Finding in Batches

Finding in batches will soon be supported in the neo4j gem, but for now is provided in the neo4j-core gem (documentation)

Orm_Adapter

You can also use the orm_adapter API, by calling #to_adapter on your class. See the API, https://github.com/ianwhite/orm_adapter

Find or Create By...

QueryProxy has a `find_or_create_by` method to make the node rel creation process easier. Its usage is simple:

```
a_node.an_association(params_hash)
```

The method has branching logic that attempts to match an existing node and relationship. If the pattern is not found, it tries to find a node of the expected class and create the relationship. If *that* doesn't work, it creates the node, then creates the relationship. The process is wrapped in a transaction to prevent a failure from leaving the database in an inconsistent state.

There are some mild caveats. First, it will not work on associations of class methods. Second, you should not use it across more than one associations or you will receive an error. For instance, if you did this:

```
student.friends.lessons.find_or_create_by(subject: 'Math')
```

Assuming the `lessons` association points to a `Lesson` model, you would effectively end up with this:

```
math = Lesson.find_or_create_by(subject: 'Math')
student.friends.lessons << math
```

...which is invalid and will result in an error.

CHAPTER 10

QueryClauseMethods

The `Neo4j::Core::Query` class from the *neo4j-core* gem defines a DSL which allows for easy creation of Neo4j Cypher queries. They can be started from a session like so:

```
a_session.query
# The current session for `ActiveNode` / `ActiveRel` in the `neo4j` gem can be␣
↪retrieved with `Neo4j::ActiveBase.current_session`
```

Advantages of using the *Query* class include:

- Method chaining allows you to build a part of a query and then pass it somewhere else to be built further
- Automatic use of parameters when possible
- Ability to pass in data directly from other sources (like Hash to match keys/values)
- Ability to use native Ruby objects (such as translating *nil* values to *IS NULL*, regular expressions to Cypher-style regular expression matches, etc...)

Below is a series of Ruby code samples and the resulting Cypher that would be generated. These examples are all generated directly from the spec file and are thus all tested to work.

Neo4j::Core::Query

#match

Ruby

```
.match('n')
```

Cypher

```
MATCH n
```

Ruby

```
.match(:n)
```

Cypher

```
MATCH (n)
```

Ruby

```
.match(n: Person)
```

Cypher

```
MATCH (n:`Person`)
```

Ruby

```
.match(n: 'Person')
```

Cypher

```
MATCH (n:`Person`)
```

Ruby

```
.match(n: ':Person')
```

Cypher

```
MATCH (n:Person)
```

Ruby

```
.match(n: :Person)
```

Cypher

```
MATCH (n:`Person`)
```

Ruby

```
.match(n: [:Person, "Animal"])
```

Cypher

```
MATCH (n:`Person`:`Animal`)
```

Ruby

```
.match(n: ' :Person')
```

Cypher

```
MATCH (n:Person)
```

Ruby

```
.match(n: nil)
```

Cypher

```
MATCH (n)
```

Ruby

```
.match(n: 'Person {name: "Brian"}')
```

Cypher

```
MATCH (n:Person {name: "Brian"})
```

Ruby

```
.match(n: {name: 'Brian', age: 33})
```

Cypher

```
MATCH (n {name: {n_name}, age: {n_age}})
```

Parameters: `{:n_name=>"Brian", :n_age=>33}`

Ruby

```
.match(n: {Person: {name: 'Brian', age: 33}})
```

Cypher

```
MATCH (n:`Person` {name: {n_Person_name}, age: {n_Person_age}})
```

Parameters: `{:n_Person_name=>"Brian", :n_Person_age=>33}`

Ruby

```
.match('n--o')
```

Cypher

```
MATCH n--o
```

Ruby

```
.match('n--o', 'o--p')
```

Cypher

```
MATCH n--o, o--p
```

Ruby

```
.match('n--o').match('o--p')
```

Cypher

```
MATCH n--o, o--p
```

#optional_match

Ruby

```
.optional_match(n: Person)
```

Cypher

```
OPTIONAL MATCH (n:`Person`)
```

Ruby

```
.match('m--n').optional_match('n--o').match('o--p')
```

Cypher

```
MATCH m--n, o--p OPTIONAL MATCH n--o
```

#using

Ruby

```
.using('INDEX m:German(surname)')
```

Cypher

```
USING INDEX m:German(surname)
```

Ruby

```
.using('SCAN m:German')
```

Cypher

```
USING SCAN m:German
```

Ruby

```
.using('INDEX m:German(surname)').using('SCAN m:German')
```

Cypher

```
USING INDEX m:German(surname) USING SCAN m:German
```

#where

Ruby

```
.where()
```

Cypher

```

```

Ruby

```
.where({})
```

Cypher

```

```

Ruby

```
.where('q.age > 30')
```

Cypher

```
WHERE (q.age > 30)
```

Ruby

```
.where('q.age' => 30)
```

Cypher

```
WHERE (q.age = {q_age})
```

Parameters: `{:q_age=>30}`

Ruby

```
.where('q.age' => [30, 32, 34])
```

Cypher

```
WHERE (q.age IN {q_age})
```

Parameters: `{:q_age=>[30, 32, 34]}`

Ruby

```
.where('q.age IN {age}', age: [30, 32, 34])
```

Cypher

```
WHERE (q.age IN {age})
```

Parameters: `{:age=>[30, 32, 34]}`

Ruby

```
.where('(q.age IN {age})', age: [30, 32, 34])
```

Cypher

```
WHERE (q.age IN {age})
```

Parameters: `{:age=>[30, 32, 34]}`

Ruby

```
.where('q.name =~ ?', '.*test.*')
```

Cypher

```
WHERE (q.name =~ {question_mark_param})
```

Parameters: `{:question_mark_param=>".*test.*"}`

Ruby

```
.where('(q.name =~ ?)', '.*test.*')
```

Cypher

```
WHERE (q.name =~ {question_mark_param})
```

Parameters: `{:question_mark_param=>".*test.*"}`

Ruby

```
.where('(LOWER(str(q.name)) =~ ?)', '.*test.*')
```

Cypher

```
WHERE (LOWER(str(q.name)) =~ {question_mark_param})
```

Parameters: `{:question_mark_param=>".*test.*"}`

Ruby

```
.where('q.age IN ?', [30, 32, 34])
```

Cypher

```
WHERE (q.age IN {question_mark_param})
```

Parameters: `{:question_mark_param=>[30, 32, 34]}`

Ruby

```
.where('q.age IN ?', [30, 32, 34]).where('q.age != ?', 60)
```

Cypher

```
WHERE (q.age IN {question_mark_param}) AND (q.age != {question_mark_
↪param2})
```

Parameters: `{:question_mark_param=>[30, 32, 34], :question_mark_param2=>60}`

Ruby

```
.where(q: {age: [30, 32, 34]})
```

Cypher

```
WHERE (q.age IN {q_age})
```

Parameters: `{:q_age=>[30, 32, 34]}`

Ruby

```
.where('q.age' => nil)
```

Cypher

```
WHERE (q.age IS NULL)
```

Ruby

```
.where(q: {age: nil})
```

Cypher

```
WHERE (q.age IS NULL)
```

Ruby

```
.where(q: {neo_id: 22})
```

Cypher

```
WHERE (ID(q) = {ID_q})
```

Parameters: `{:ID_q=>22}`

Ruby

```
.where(q: {age: 30, name: 'Brian'})
```

Cypher

```
WHERE (q.age = {q_age} AND q.name = {q_name})
```

Parameters: `{:q_age=>30, :q_name=>"Brian"}`

Ruby

```
.where(q: {age: 30, name: 'Brian'}).where('r.grade = 80')
```

Cypher

```
WHERE (q.age = {q_age} AND q.name = {q_name}) AND (r.grade = 80)
```

Parameters: `{:q_age=>30, :q_name=>"Brian"}`

Ruby

```
.where(q: {name: /Brian.*/i})
```

Cypher

```
WHERE (q.name =~ {q_name})
```

Parameters: `{:q_name=>"(?i)Brian.*"}`

Ruby

```
.where(name: /Brian.*/i)
```

Cypher

```
WHERE (name =~ {name})
```

Parameters: `{:name=>"(?i)Brian.*"}`

Ruby

```
.where(name: /Brian.*/i).where(name: /Smith.*/i)
```

Cypher

```
WHERE (name =~ {name}) AND (name =~ {name2})
```

Parameters: `{:name=>"(?i)Brian.*", :name2=>"(?i)Smith.*"}`

Ruby

```
.where(q: {age: (30..40)})
```

Cypher

```
WHERE (q.age IN RANGE({q_age_range_min}, {q_age_range_max}))
```

Parameters: `{:q_age_range_min=>30, :q_age_range_max=>40}`

#where_not

Ruby

```
.where_not()
```

Cypher

```

```

Ruby

```
.where_not({})
```

Cypher

```
```

Ruby

```
.where_not('q.age > 30')
```

Cypher

```
WHERE NOT(q.age > 30)
```

Ruby

```
.where_not('q.age' => 30)
```

Cypher

```
WHERE NOT(q.age = {q_age})
```

Parameters: `{:q_age=>30}`

Ruby

```
.where_not('q.age IN ?', [30, 32, 34])
```

Cypher

```
WHERE NOT(q.age IN {question_mark_param})
```

Parameters: `{:question_mark_param=>[30, 32, 34]}`

Ruby

```
.where_not(q: {age: 30, name: 'Brian'})
```

Cypher

```
WHERE NOT(q.age = {q_age} AND q.name = {q_name})
```

Parameters: `{:q_age=>30, :q_name=>"Brian"}`

Ruby

```
.where_not(q: {name: /Brian.*/i})
```

Cypher

```
WHERE NOT(q.name =~ {q_name})
```

Parameters: `{:q_name=>"(?i)Brian.*"}`

Ruby

```
.where('q.age > 10').where_not('q.age > 30')
```

Cypher

```
WHERE (q.age > 10) AND NOT(q.age > 30)
```

Ruby

```
.where_not('q.age > 30').where('q.age > 10')
```

Cypher

```
WHERE NOT(q.age > 30) AND (q.age > 10)
```

#match_nodes

one node object

Ruby

```
.match_nodes(var: node_object)
```

Cypher

```
MATCH (var) WHERE (ID(var) = {ID_var})
```

Parameters: `{:ID_var=>246}`

Ruby

```
.optional_match_nodes(var: node_object)
```

Cypher

```
OPTIONAL MATCH (var) WHERE (ID(var) = {ID_var})
```

Parameters: `{:ID_var=>246}`

---

integer

Ruby

```
.match_nodes(var: 924)
```

Cypher

```
MATCH (var) WHERE (ID(var) = {ID_var})
```

Parameters: `{:ID_var=>924}`

---

two node objects

Ruby

```
.match_nodes(user: user, post: post)
```

Cypher

```
MATCH (user), (post) WHERE (ID(user) = {ID_user}) AND (ID(post) = {ID_
↪post})
```

Parameters: `{:ID_user=>246, :ID_post=>123}`

---

node object and integer

Ruby

```
.match_nodes(user: user, post: 652)
```

Cypher

```
MATCH (user), (post) WHERE (ID(user) = {ID_user}) AND (ID(post) = {ID_
↪post})
```

Parameters: `{:ID_user=>246, :ID_post=>652}`

#unwind

Ruby

```
.unwind('val AS x')
```

Cypher

```
UNWIND val AS x
```

Ruby

```
.unwind(x: :val)
```

Cypher

```
UNWIND val AS x
```

Ruby

```
.unwind(x: 'val')
```

Cypher

```
UNWIND val AS x
```

Ruby

```
.unwind(x: [1,3,5])
```

Cypher

```
UNWIND [1, 3, 5] AS x
```

Ruby

```
.unwind(x: [1,3,5]).unwind('val as y')
```

Cypher

```
UNWIND [1, 3, 5] AS x UNWIND val as y
```

#return

Ruby

```
.return('q')
```

Cypher

```
RETURN q
```

Ruby

```
.return(:q)
```

Cypher

```
RETURN q
```

Ruby

```
.return('q.name, q.age')
```

Cypher

```
RETURN q.name, q.age
```

Ruby

```
.return(q: [:name, :age], r: :grade)
```

Cypher

```
RETURN q.name, q.age, r.grade
```

Ruby

```
.return(q: :neo_id)
```

Cypher

```
RETURN ID(q)
```

Ruby

```
.return(q: [:neo_id, :prop])
```

Cypher

```
RETURN ID(q), q.prop
```

#order

Ruby

```
.order('q.name')
```

Cypher

```
ORDER BY q.name
```

Ruby

```
.order_by('q.name')
```

Cypher

```
ORDER BY q.name
```

Ruby

```
.order('q.age', 'q.name DESC')
```

Cypher

```
ORDER BY q.age, q.name DESC
```

Ruby

```
.order(q: :age)
```

Cypher

```
ORDER BY q.age
```

Ruby

```
.order(q: :neo_id)
```

Cypher

```
ORDER BY ID(q)
```

Ruby

```
.order(q: [:age, {name: :desc}])
```

Cypher

```
ORDER BY q.age, q.name DESC
```

Ruby

```
.order(q: [:age, {neo_id: :desc}])
```

Cypher

```
ORDER BY q.age, ID(q) DESC
```

Ruby

```
.order(q: [:age, {name: :desc, grade: :asc}])
```

Cypher

```
ORDER BY q.age, q.name DESC, q.grade ASC
```

Ruby

```
.order(q: [:age, {name: :desc, neo_id: :asc}])
```

Cypher

```
ORDER BY q.age, q.name DESC, ID(q) ASC
```

Ruby

```
.order(q: {age: :asc, name: :desc})
```

Cypher

```
ORDER BY q.age ASC, q.name DESC
```

Ruby

```
.order(q: {age: :asc, neo_id: :desc})
```

Cypher

```
ORDER BY q.age ASC, ID(q) DESC
```

Ruby

```
.order(q: [:age, 'name desc'])
```

Cypher

```
ORDER BY q.age, q.name desc
```

Ruby

```
.order(q: [:neo_id, 'name desc'])
```

Cypher

```
ORDER BY ID(q), q.name desc
```

#limit

Ruby

```
.limit(3)
```

Cypher

```
LIMIT {limit_3}
```

Parameters: `{:limit_3=>3}`

Ruby

```
.limit('3')
```

Cypher

```
LIMIT {limit_3}
```

Parameters: `{:limit_3=>3}`

Ruby

```
.limit(3).limit(5)
```

Cypher

```
LIMIT {limit_5}
```

Parameters: `{:limit_3=>3, :limit_5=>5}`

Ruby

```
.limit(nil)
```

Cypher

```

```

#skip

Ruby

```
.skip(5)
```

Cypher

```
SKIP {skip_5}
```

Parameters: `{:skip_5=>5}`

Ruby

```
.skip('5')
```

Cypher

```
SKIP {skip_5}
```

Parameters: `{:skip_5=>5}`

Ruby

```
.skip(5).skip(10)
```

Cypher

```
SKIP {skip_10}
```

Parameters: `{:skip_5=>5, :skip_10=>10}`

Ruby

```
.offset(6)
```

Cypher

```
SKIP {skip_6}
```

Parameters: `{:skip_6=>6}`

#with

Ruby

```
.with('n.age AS age')
```

Cypher

```
WITH n.age AS age
```

Ruby

```
.with('n.age AS age', 'count(n) as c')
```

Cypher

```
WITH n.age AS age, count(n) as c
```

Ruby

```
.with(['n.age AS age', 'count(n) as c'])
```

Cypher

```
WITH n.age AS age, count(n) as c
```

Ruby

```
.with(age: 'n.age')
```

Cypher

```
WITH n.age AS age
```

#create

Ruby

```
.create('(:Person)')
```

Cypher

```
CREATE (:Person)
```

Ruby

```
.create(:Person)
```

Cypher

```
CREATE (:Person)
```

Ruby

```
.create(age: 41, height: 70)
```

Cypher

```
CREATE ( {age: {age}, height: {height}})
```

Parameters: `{:age=>41, :height=>70}`

Ruby

```
.create(Person: {age: 41, height: 70})
```

Cypher

```
CREATE (:`Person` {age: {Person_age}, height: {Person_height}})
```

Parameters: `{:Person_age=>41, :Person_height=>70}`

Ruby

```
.create(q: {Person: {age: 41, height: 70}})
```

Cypher

```
CREATE (q:`Person` {age: {q_Person_age}, height: {q_Person_height}})
```

Parameters: `{:q_Person_age=>41, :q_Person_height=>70}`

Ruby

```
.create(q: {Person: {age: nil, height: 70}})
```

Cypher

```
CREATE (q:`Person` {age: {q_Person_age}, height: {q_Person_height}})
```

Parameters: `{:q_Person_age=>nil, :q_Person_height=>70}`

Ruby

```
.create(q: {:'Child:Person' => {age: 41, height: 70}})
```

Cypher

```
CREATE (q:`Child:Person` {age: {q_Child_Person_age}, height: {q_Child_
↪Person_height}})
```

Parameters: `{:q_Child_Person_age=>41, :q_Child_Person_height=>70}`

Ruby

```
.create(:'Child:Person' => {age: 41, height: 70})
```

Cypher

```
CREATE (:`Child:Person` {age: {Child_Person_age}, height: {Child_Person_
↪height}})
```

Parameters: `{:Child_Person_age=>41, :Child_Person_height=>70}`

Ruby

```
.create(q: {[:Child, :Person] => {age: 41, height: 70}})
```

Cypher

```
CREATE (q:`Child`:`Person` {age: {q_Child_Person_age}, height: {q_Child_
↪Person_height}})
```

Parameters: `{:q_Child_Person_age=>41, :q_Child_Person_height=>70}`

Ruby

```
.create([:Child, :Person] => {age: 41, height: 70})
```

Cypher

```
CREATE (:`Child`:`Person` {age: {Child_Person_age}, height: {Child_
↪Person_height}})
```

Parameters: `{:Child_Person_age=>41, :Child_Person_height=>70}`

#create_unique

Ruby

```
.create_unique('(:Person)')
```

Cypher

```
CREATE UNIQUE (:Person)
```

Ruby

```
.create_unique(:Person)
```

Cypher

```
CREATE UNIQUE (:Person)
```

Ruby

```
.create_unique(age: 41, height: 70)
```

Cypher

```
CREATE UNIQUE ( {age: {age}, height: {height}})
```

Parameters: `{:age=>41, :height=>70}`

Ruby

```
.create_unique(Person: {age: 41, height: 70})
```

Cypher

```
CREATE UNIQUE (:`Person` {age: {Person_age}, height: {Person_height}})
```

Parameters: `{:Person_age=>41, :Person_height=>70}`

Ruby

```
.create_unique(q: {Person: {age: 41, height: 70}})
```

Cypher

```
CREATE UNIQUE (q:`Person` {age: {q_Person_age}, height: {q_Person_height}
↪})
```

Parameters: `{:q_Person_age=>41, :q_Person_height=>70}`

#merge

Ruby

```
.merge('(:Person)')
```

Cypher

```
MERGE (:Person)
```

Ruby

```
.merge(:Person)
```

Cypher

```
MERGE (:Person)
```

Ruby

```
.merge(:Person).merge(:Thing)
```

Cypher

```
MERGE (:Person) MERGE (:Thing)
```

Ruby

```
.merge(age: 41, height: 70)
```

Cypher

```
MERGE ( {age: {age}, height: {height}})
```

Parameters: `{:age=>41, :height=>70}`

Ruby

```
.merge(Person: {age: 41, height: 70})
```

Cypher

```
MERGE (:`Person` {age: {Person_age}, height: {Person_height}})
```

Parameters: `{:Person_age=>41, :Person_height=>70}`

Ruby

```
.merge(q: {Person: {age: 41, height: 70}})
```

Cypher

```
MERGE (q:`Person` {age: {q_Person_age}, height: {q_Person_height}})
```

Parameters: `{:q_Person_age=>41, :q_Person_height=>70}`

#delete

Ruby

```
.delete('n')
```

Cypher

```
DELETE n
```

Ruby

```
.delete(:n)
```

Cypher

```
DELETE n
```

Ruby

```
.delete('n', :o)
```

Cypher

```
DELETE n, o
```

Ruby

```
.delete(['n', :o])
```

Cypher

```
DELETE n, o
```

Ruby

```
.detach_delete('n')
```

Cypher

```
DETACH DELETE n
```

Ruby

```
.detach_delete(:n)
```

Cypher

```
DETACH DELETE n
```

Ruby

```
.detach_delete('n', :o)
```

Cypher

```
DETACH DELETE n, o
```

Ruby

```
.detach_delete(['n', :o])
```

Cypher

```
DETACH DELETE n, o
```

#set_props

Ruby

```
.set_props('n = {name: "Brian"}')
```

Cypher

```
SET n = {name: "Brian"}
```

Ruby

```
.set_props(n: {name: 'Brian', age: 30})
```

Cypher

```
SET n = {n_set_props}
```

Parameters: `{:n_set_props=>{:name=>"Brian", :age=>30}}`

#set

Ruby

```
.set('n = {name: "Brian"}')
```

Cypher

```
SET n = {name: "Brian"}
```

Ruby

```
.set(n: {name: 'Brian', age: 30})
```

Cypher

```
SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30}`

Ruby

```
.set(n: {name: 'Brian', age: 30}, o: {age: 29})
```

Cypher

```
SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}, o.`age` =
↪{setter_o_age}
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30, :setter_o_age=>29}`

Ruby

```
.set(n: {name: 'Brian', age: 30}).set_props('o.age = 29')
```

Cypher

```
SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}, o.age = 29
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30}`

Ruby

```
.set(n: :Label)
```

Cypher

```
SET n:`Label`
```

Ruby

```
.set(n: [:Label, 'Foo'])
```

Cypher

```
SET n:`Label`, n:`Foo`
```

Ruby

```
.set(n: nil)
```

Cypher

```

```

#on_create_set

Ruby

```
.on_create_set('n = {name: "Brian"}')
```

Cypher

```
ON CREATE SET n = {name: "Brian"}
```

Ruby

```
.on_create_set(n: {})
```

Cypher

```
```

Ruby

```
.on_create_set(n: {name: 'Brian', age: 30})
```

Cypher

```
ON CREATE SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30}`

Ruby

```
.on_create_set(n: {name: 'Brian', age: 30}, o: {age: 29})
```

Cypher

```
ON CREATE SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}, o.
↪`age` = {setter_o_age}
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30, :setter_o_age=>29}`

Ruby

```
.on_create_set(n: {name: 'Brian', age: 30}).on_create_set('o.age = 29')
```

Cypher

```
ON CREATE SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}, o.
↪age = 29
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30}`

#on_match_set

Ruby

```
.on_match_set('n = {name: "Brian"}')
```

Cypher

```
ON MATCH SET n = {name: "Brian"}
```

Ruby

```
.on_match_set(n: {})
```

Cypher

```
```

Ruby

```
.on_match_set(n: {name: 'Brian', age: 30})
```

Cypher

```
ON MATCH SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30}`

Ruby

```
.on_match_set(n: {name: 'Brian', age: 30}, o: {age: 29})
```

Cypher

```
ON MATCH SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}, o.
↪`age` = {setter_o_age}
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30, :setter_o_age=>29}`

Ruby

```
.on_match_set(n: {name: 'Brian', age: 30}).on_match_set('o.age = 29')
```

Cypher

```
ON MATCH SET n.`name` = {setter_n_name}, n.`age` = {setter_n_age}, o.age␣
↪= 29
```

Parameters: `{:setter_n_name=>"Brian", :setter_n_age=>30}`

#remove

Ruby

```
.remove('n.prop')
```

Cypher

```
REMOVE n.prop
```

Ruby

```
.remove('n:American')
```

Cypher

```
REMOVE n:American
```

Ruby

```
.remove(n: 'prop')
```

Cypher

```
REMOVE n.prop
```

Ruby

```
.remove(n: :American)
```

Cypher

```
REMOVE n:`American`
```

Ruby

```
.remove(n: [:American, "prop"])
```

Cypher

```
REMOVE n:`American`, n.prop
```

Ruby

```
.remove(n: :American, o: 'prop')
```

Cypher

```
REMOVE n:`American`, o.prop
```

Ruby

```
.remove(n: ':prop')
```

Cypher

```
REMOVE n:`prop`
```

#start

Ruby

```
.start('r=node:nodes(name = "Brian")')
```

Cypher

```
START r=node:nodes(name = "Brian")
```

Ruby

```
.start(r: 'node:nodes(name = "Brian")')
```

Cypher

```
START r = node:nodes(name = "Brian")
```

---

clause combinations

Ruby

```
.match(q: Person).where('q.age > 30')
```

Cypher

```
MATCH (q:`Person`) WHERE (q.age > 30)
```

Ruby

```
.where('q.age > 30').match(q: Person)
```

Cypher

```
MATCH (q:`Person`) WHERE (q.age > 30)
```

Ruby

```
.where('q.age > 30').start('n').match(q: Person)
```

Cypher

```
START n MATCH (q:`Person`) WHERE (q.age > 30)
```

Ruby

```
.match(q: {age: 30}).set_props(q: {age: 31})
```

Cypher

```
MATCH (q {age: {q_age}}) SET q = {q_set_props}
```

Parameters: `{:q_age=>30, :q_set_props=>{:age=>31}}`

Ruby

```
.match(q: Person).with('count(q) AS count')
```

Cypher

```
MATCH (q:`Person`) WITH count(q) AS count
```

Ruby

```
.match(q: Person).with('count(q) AS count').where('count > 2')
```

Cypher

```
MATCH (q:`Person`) WITH count(q) AS count WHERE (count > 2)
```

Ruby

```
.match(q: Person).with(count: 'count(q)').where('count > 2').with(new_
↪count: 'count + 5')
```

Cypher

```
MATCH (q:`Person`) WITH count(q) AS count WHERE (count > 2) WITH count +␣
↪5 AS new_count
```

Ruby

```
.match(q: Person).match('r:Car').break.match('(p: Person)-->q')
```

Cypher

```
MATCH (q:`Person`), r:Car MATCH (p: Person)-->q
```

Ruby

```
.match(q: Person).break.match('r:Car').break.match('(p: Person)-->q')
```

Cypher

```
MATCH (q:`Person`) MATCH r:Car MATCH (p: Person)-->q
```

Ruby

```
.match(q: Person).match('r:Car').break.break.match('(p: Person)-->q')
```

Cypher

```
MATCH (q:`Person`), r:Car MATCH (p: Person)-->q
```

Ruby

```
.with(:a).order(a: {name: :desc}).where(a: {name: 'Foo'})
```

Cypher

```
WITH a ORDER BY a.name DESC WHERE (a.name = {a_name})
```

Parameters: `{:a_name=>"Foo"}`

Ruby

```
.with(:a).limit(2).where(a: {name: 'Foo'})
```

Cypher

```
WITH a LIMIT {limit_2} WHERE (a.name = {a_name})
```

Parameters: `{:a_name=>"Foo", :limit_2=>2}`

Ruby

```
.with(:a).order(a: {name: :desc}).limit(2).where(a: {name: 'Foo'})
```

Cypher

```
WITH a ORDER BY a.name DESC LIMIT {limit_2} WHERE (a.name = {a_name})
```

Parameters: `{:a_name=>"Foo", :limit_2=>2}`

Ruby

```
.order(a: {name: :desc}).with(:a).where(a: {name: 'Foo'})
```

Cypher

```
WITH a ORDER BY a.name DESC WHERE (a.name = {a_name})
```

Parameters: `{:a_name=>"Foo"}`

Ruby

```
.limit(2).with(:a).where(a: {name: 'Foo'})
```

Cypher

```
WITH a LIMIT {limit_2} WHERE (a.name = {a_name})
```

Parameters: `{:a_name=>"Foo", :limit_2=>2}`

Ruby

```
.order(a: {name: :desc}).limit(2).with(:a).where(a: {name: 'Foo'})
```

Cypher

```
WITH a ORDER BY a.name DESC LIMIT {limit_2} WHERE (a.name = {a_name})
```

Parameters: `{:a_name=>"Foo", :limit_2=>2}`

Ruby

```
.with('1 AS a').where(a: 1).limit(2)
```

Cypher

```
WITH 1 AS a WHERE (a = {a}) LIMIT {limit_2}
```

Parameters: `{:a=>1, :limit_2=>2}`

Ruby

```
.match(q: Person).where('q.age = {age}').params(age: 15)
```

Cypher

```
MATCH (q:`Person`) WHERE (q.age = {age})
```

Parameters: `{:age=>15}`

CHAPTER 11

Configuration

To configure any of these variables you can do the following:

In Rails

In either `config/application.rb` or one of the environment configurations (e.g. `config/environments/development.rb`) you can set `config.neo4j.variable_name = value` where **variable_name** and **value** are as described below.

Other Ruby apps

You can set configuration variables directly in the Neo4j configuration class like so: `Neo4j::Config[:variable_name] = value` where **variable_name** and **value** are as described below.

Variables

skip_migration_check **Default:** `false`

Prevents the `neo4j` gem from raising `Neo4j::PendingMigrationError` in web requests when migrations haven't been run. For environments (like testing) where you need to use the `neo4j:schema:load` rake task to build the database instead of migrations. Automatically set to `true` in Rails test environments by default

class_name_property **Default:** `:_classname`

Which property should be used to determine the *ActiveNode* class to wrap the node in

If there is no value for this property on a node the node's labels will be used to determine the *ActiveNode* class

See also:

Wrapping

include_root_in_json **Default:** `true`

When serializing `ActiveNode` and `ActiveRel` objects, should there be a root in the JSON of the model name.

See also:

http://api.rubyonrails.org/classes/ActiveModel/Serializers/JSON.html

transform_rel_type **Default:** `:upcase`

Available values: `:upcase`, `:downcase`, `:legacy`, `:none`

Determines how relationship types as specified in associations are transformed when stored in the database. By default this is upper-case to match with Neo4j convention so if you specify an association of `has_many :in, :posts, type: :has_post` then the relationship type in the database will be `HAS_POST`

`:legacy` Causes the type to be downcased and preceded by a *#*

`:none` Uses the type as specified

module_handling **Default:** `:none`

Available values: `:demodulize`, `:none`, `proc`

Determines what, if anything, should be done to module names when a model's class is set. By default, there is a direct mapping of an `ActiveNode` model name to the node label or an `ActiveRel` model to the relationship type, so *MyModule::MyClass* results in a label with the same name.

The *:demodulize* option uses ActiveSupport's method of the same name to strip off modules. If you use a *proc*, it will the class name as an argument and you should return a string that modifies it as you see fit.

association_model_namespace **Default:** `nil`

Associations defined in node models will try to match association names to classes. For example, *has_many :out, :student* will look for a *Student* class. To avoid having to use *model_class: 'MyModule::Student'*, this config option lets you specify the module that should be used globally for class name discovery.

Of course, even with this option set, you can always override it by calling *model_class: 'ClassName'*.

logger **Default:** `nil` (or `Rails.logger` in Rails)

A Ruby `Logger` object which is used to log Cypher queries (*info* level is used). This is only for the `neo4j` gem (that is, for models created with the `ActiveNode` and `ActiveRel` modules).

pretty_logged_cypher_queries **Default:** `nil`

If true, format outputted queries with newlines and colors to be more easily readable by humans

record_timestamps **Default:** `false`

A Rails-inspired configuration to manage inclusion of the Timestamps module. If set to true, all ActiveNode and ActiveRel models will include the Timestamps module and have `:created_at` and `:updated_at` properties.

timestamp_type **Default:** `DateTime`

This method returns the specified default type for the `:created_at` and `:updated_at` timestamps. You can also specify another type (e.g. `Integer`).

wait_for_connection **Default:** `false`

This allows you to tell the gem to wait for up to 60 seconds for Neo4j to be available. This is useful in environments such as Docker Compose. This is currently only for Rails

Instrumented events

The `neo4j-core` gem instruments a handful of events so that users can subscribe to them to do logging, metrics, or anything else that they need. For example, to create a block which is called any time a query is made via the `neo4j-core` gem:

```
Neo4j::Core::CypherSession::Adaptors::Base.subscribe_to_query do |message|
  puts message
end
```

The argument to the block (`message` in this case) will be an ANSI formatted string which can be outputted or stored. If you want to access this event at a lower level, `subscribe_to_query` is actually tied to the `neo4j.core.cypher_query` event to which you could subscribe to like:

```
ActiveSupport::Notifications.subscribe('neo4j.core.cypher_query') do |name, start,␣
↪finish, id, payload|
  puts payload[:query].to_cypher
  # or
  payload[:query].print_cypher

  puts "Query took: #{(finish - start)} seconds"
end
```

All methods and their corresponding events:

Neo4j::Core::CypherSession::Adaptors::Base.subscribe_to_query **neo4j.core.cypher_query**

Neo4j::Core::CypherSession::Adaptors::HTTP.subscribe_to_request **neo4j.core.http.request**

Neo4j::Core::CypherSession::Adaptors::Bolt.subscribe_to_request **neo4j.core.bolt.request**

Neo4j::Core::CypherSession::Adaptors::Embedded.subscribe_to_transaction **neo4j.core.embedded.transaction**

CHAPTER 12

Migrations

Neo4j does not have a set schema like relational databases, but sometimes changes to the schema and the data are required. To help with this, Neo4j.rb provides an `ActiveRecord`-like migration framework and a set of helper methods to manipulate both database schema and data. Just like `ActiveRecord`, a record of which transactions have been run will be stored in the database so that a migration is automatically only run once per environment.

Note: If you are new to Neo4j, note that properties on nodes and relationships are not defined ahead of time. Properties can be added and removed on the fly, and so adding a `property` to your `ActiveNode` or `ActiveRel` model is sufficient to start storing data. No migration is needed to add properties, but if you remove a property from your model you may want a migration to cleanup the data (by using the `remove_property`, for example).

Note: The migration functionality described on this page was introduced in version 8.0 of the `neo4j` gem.

Generators

Migrations can be created by using the built-in Rails generator:

```
rails generate neo4j:migration RenameUserNameToFirstName
```

This will generate a new file located in `db/neo4j/migrate/xxxxxxxxxx_rename_user_name_to_first_name.rb`

```
class RenameUserNameToFirstName < Neo4j::Migrations::Base
  def up
    rename_property :User, :name, :first_name
  end

  def down
    rename_property :User, :first_name, :name
```

```
  end
end
```

In the same way as `ActiveRecord` does, you should fill up the `up` and `down` methods to define the migration and (eventually) the rollback steps.

Transactions

Every migrations runs inside a transaction by default. So, if some statement fails inside a migration fails, the database rollbacks to the previous state.

However this behaviour is not always good. For instance, neo4j doesn't allow schema and data changes in the same transaction.

To disable this, you can use the `disable_transactions!` helper in your migration definition:

```
class SomeMigration < Neo4j::Migrations::Base
  disable_transactions!

  ...
end
```

The schema file

When generating an empty database for your app you could run all of your migrations, but this strategy gets slower over time and can even cause issues if your older migrations become incompatible with your newer code. For this reason, whenever you run migrations a `db/neo4j/schema.yml` file is created which keeps track of constraints, indexes (which aren't automatically created by constraints), and which migrations have been run. This schema file can then be loaded with the `neo4j:schema:load` rake task to quickly and safely setup a blank database for testing or for a new environment. While the `neo4j:migrate` rake task automatically creates the `schema.yml` file, if you ever need to generate it yourself you can use the `neo4j:schema:dump` rake task.

It is suggested that you check in the `db/neo4j/schema.yml` to your repository whenever you have new migrations.

Tasks

Neo4j.rb implements a clone of the `ActiveRecord` migration tasks API to migrate.

neo4j:migrate:all

Runs any pending migration.

```
rake neo4j:migrate:all
```

neo4j:migrate

An alias for `rake neo4j:migrate:all`.

```
rake neo4j:migrate:all
```

neo4j:migrate:up

Executes a migration given it's version id.

```
rake neo4j:migrate:up VERSION=some_version
```

neo4j:migrate:down

Reverts a migration given it's version id.

```
rake neo4j:migrate:down VERSION=some_version
```

neo4j:migrate:status

Prints a detailed migration state report, showing up and down migrations together with their own version id.

```
rake neo4j:migrate:status
```

neo4j:rollback

Reverts the last up migration. You can additionally pass a `STEPS` parameter, specifying how many migration you want to revert.

```
rake neo4j:rollback
```

neo4j:schema:dump

Reads the current database and generates a `db/neo4j/schema.yml` file to track constraints, indexes, and migrations which have been run (runs automatically after the `neo4j:migrate` task)

```
rake neo4j:schema:dump
```

neo4j:schema:load

Reads the `db/neo4j/schema.yml` file and loads the constraints, indexes, and migration nodes into the database. The default behavior is to only add, but an argument can be passed in to tell the task to remove any indexes / constraints that were found in the database which were not in the `schema.yml` file.

```
rake neo4j:schema:load
rake neo4j:schema:load[true] # Remove any constraints or indexes which aren't in the␣
↪``schema.yml`` file
```

Integrate Neo4j.rb with ActiveRecord migrations

You can setup Neo4j migration tasks to run together with standard ActiveRecord ones. Simply create a new rake task in `lib/tasks/neo4j_migrations.rake`:

```
Rake::Task['db:migrate'].enhance ['neo4j:migrate']
```

This will run the `neo4j:migrate` every time you run a `rake db:migrate`

Migration Helpers

#execute

Executes a pure neo4j cypher query, interpolating parameters.

```
execute('MATCH (n) WHERE n.name = {node_name} RETURN n', node_name: 'John')
```

```
execute('MATCH (n)-[r:`friend`]->() WHERE n.age = 7 DELETE r')
```

#query

An alias for `Neo4j::Session.query`. You can use it as root for the query builder:

```
query.match(:n).where(name: 'John').delete(:n).exec
```

#remove_property

Removes a property given a label.

```
remove_property(:User, :money)
```

#rename_property

Renames a property given a label.

```
rename_property(:User, :name, :first_name)
```

#drop_nodes

Removes all nodes with a certain label

```
drop_nodes(:User)
```

#add_label

Adds a label to nodes, given their current label

```
add_label(:User, :Person)
```

#add_labels

Adds labels to nodes, given their current label

```
add_label(:User, [:Person, :Boy])
```

#remove_label

Removes a label from nodes, given a label

```
remove_label(:User, :Person)
```

#remove_labels

Removes labels from nodes, given a label

```
remove_label(:User, [:Person, :Boy])
```

#rename_label

Renames a label

```
rename_label(:User, :Person)
```

#add_constraint

Adds a new unique constraint on a given label attribute.

Warning it would fail if you make data changes in the same migration. To fix, define `disable_transactions!` in your migration file.

```
add_constraint(:User, :name)
```

Use *force: true* as an option in the third argument to ignore errors about an already existing constraint.

#drop_constraint

Drops an unique constraint on a given label attribute.

Warning it would fail if you make data changes in the same migration. To fix, define `disable_transactions!` in your migration file.

```
drop_constraint(:User, :name)
```

Use *force: true* as an option in the third argument to ignore errors about the constraint being missing.

#add_index

Adds a new exact index on a given label attribute.

Warning it would fail if you make data changes in the same migration. To fix, define `disable_transactions!` in your migration file.

```
add_index(:User, :name)
```

Use *force: true* as an option in the third argument to ignore errors about an already existing index.

#drop_index

Drops an exact index on a given label attribute.

Warning it would fail if you make data changes in the same migration. To fix, define `disable_transactions!` in your migration file.

```
drop_index(:User, :name)
```

Use *force: true* as an option in the third argument to ignore errors about the index being missing.

#say

Writes some text while running the migration.

Ruby

```
say 'Hello'
```

Output

```
-- Hello
```

When passing `true` as second parameter, it writes it more indented.

Ruby

```
say 'Hello', true
```

Output

```
-> Hello
```

#say_with_time

Wraps a set of statements inside a block, printing the given and the execution time. When an `Integer` is returned, it assumes it's the number of affected rows.

Ruby

```
say_with_time 'Trims all names' do
  query.match(n: :User).set('n.name = TRIM(n.name)').pluck('count(*)').
↪first
end
```

Output

```
-- Trims all names.
   -> 0.3451s
   -> 2233 rows
```

#populate_id_property

Populates the `uuid` property (or any `id_property` you defined) of nodes given their model name.

```
populate_id_property :User
```

Check *Adding IDs to Existing Data* for more usage details.

#relabel_relation

Relabels a relationship, keeping intact any relationship attribute.

```
relabel_relation :old_label, :new_label
```

Additionally you can specify the starting and the destination node, using `:from` and `:to`.

You can specify also the `:direction` (one if `:in`, `:out` or `:both`).

Example:

```
relabel_relation :friends, :FRIENDS, from: :Animal, to: :Person, direction: :both
```

#change_relations_style

Relabels relationship nodes from one format to another.

Usage:

```
change_relations_style list_of_labels, old_style, new_style
```

For example, if you created a relationship `#foo` in 3.x, and you want to convert it to the 4.x+ `foo` syntax, you could run this.

```
change_relations_style [:all, :your, :labels, :here], :lower_hash, :lower
```

Allowed styles are:

- `:lower`: lowercase string, like `my_relation`
- `:upper`: uppercase string, like `MY_RELATION`
- `:lower_hash`: Lowercase string starting with hash, like `#my_relation`

CHAPTER 13

Testing

To run your tests, you must have a Neo4j server running (ideally a different server than the development database on a different port). One quick way to get a test database up and running is to use the built in rake task:

```
rake neo4j:install[community-latest,test]
# or a specific version
rake neo4j:install[community-3.1.0,test]
```

You can configure it to respond on a different port like so:

```
rake neo4j:config[test,7475]
```

If you are using Rails, you can edit the test configuration `config/environments/test.rb` or the `config/neo4j.yml` file (see *Setup*)

How to clear the database

Cypher

Faster, but does not does not remove the database schema (indexes and constraints):

```
Neo4j::ActiveBase.current_session.query('MATCH (n) DETACH DELETE n')

# For Neo4j < 2.3
Neo4j::ActiveBase.current_session.query('MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n,
↪r')
```

Delete data files

Completely delete the database files (slower, by removeds schema). If you installed Neo4j via the rake tasks, you can run:

```
rake neo4j:reset_yes_i_am_sure[test]
```

If you are using embedded Neo4j, stop embedded db, delete the db path, start embedded db.

RSpec Transaction Rollback

If you are using RSpec you can perform tests in a transaction as you would using active record. Just add the following to your rspec configuration in `spec/rails_helper.rb` or `spec/spec_helper.rb`

```
config.around do |example|
  Neo4j::Transaction.run do |tx|
    example.run
      tx.mark_failed
    end
  end
```

CHAPTER 14

Miscellany

Mass / Batch Importing

Importing many nodes or relationships at once is a common use case. Often the naive approach can be slow because each query is done over a separate HTTP request. There are a number of ways to improve this:

- The neo4j-core gem (starting with version 7.0) supports batch execution of queries by calling the *queries* method an a *CypherSession* (There is not yet a means of doing this in *ActiveNode* and *ActiveRecord* in the *neo4j* gem)
- Since even batched queries require sending a large payload of queries, you might consider making a single Cypher query with an array parameter which can be turned into a series of rows with the UNWIND clause which can then be used to execute a CREATE clause to make one creation per row from the *UNWIND*
- The neo4apis gem offers a way to create a DSL for defining and loading data and will batch creations for you (see the neo4apis-github and neo4apis-twitter gems for examples of implementing a *neo4apis* DSL)

Outside of Ruby, there are also standard ways of importing large sets of data:

- The LOAD CSV clause allows you to take a CSV in any format and create your own custom Cypher logic to import the data
- The Neo4j import tool requires a specific CSV format for nodes and relationships, but it can be extremely fast. (Note that the import tool can only be used to create a new database, not to add to an existing one)

Cleaning Your Database for Testing

Often when writing tests for Neo4j it is desirable to start with a fresh database for each test. In general this can be as easy as writing a Cypher query which runs before each test:

```
// For version of Neo4j before 2.3.0
// Nodes cannot be deleted without first deleting their relationships
MATCH (n)
OPTIONAL MATCH (n)-[r]-()
DELETE n,r
```

```
// For version of Neo4j after 2.3.0
// DETACH DELETE takes care of removing relationships for you
MATCH (n) DETACH DELETE n
```

In Ruby:

```
# Just using the `neo4j-core` gem:
neo4j_session.query('MATCH (n) DETACH DELETE n')

# When using the `neo4j` gem:
Neo4j::ActiveBase.current_session.query('MATCH (n) DETACH DELETE n')
```

If you are using `ActiveNode` and/or `ActiveRel` from the `neo4j` gem you will no doubt have `SchemaMigration` nodes in the database. If you delete these nodes the gem will complain that your migrations haven't been run. To get around this you could modify the query to exclude those nodes:

```
MATCH (n) WHERE NOT n:`Neo4j::Migrations::SchemaMigration`
DETACH DELETE n
```

Separately, the `database_cleaner` gem is a popular and useful tool for abstracting away the cleaning of databases in tests. There is support for Neo4j in the `database_cleaner` gem, but there are a couple of problems with it:

- Neo4j does not currently support truncation (wiping of the entire database designed to be faster than a `DELETE`)
- Neo4j supports transactions, but nested transactions do not work the same as in relational databases. A failure in a nested transaction will cause the entire set of outer transactions to be rolled back. Therefore running tests inside of a transaction and rolling back a nested transaction for each test isn't viable.

Because of this, all strategies in the `database_cleaner` gem amount to it's "Deletion" strategy. Therefore, while you are welcome to use the `database_cleaner` gem, is is generally simpler to execute one of the above Cypher queries.

CHAPTER 15

Contributing

We very much welcome contributions! Before contributing there are a few things that you should know about the neo4j.rb projects:

The Neo4j.rb Project

We have three main gems: neo4j, neo4j-core, neo4j-rake_tasks.

We try to follow semantic versioning based on *semver.org <http://semver.org/>*

Low Hanging Fruit

Just reporting issues is helpful, but if you want to help with some code we label our GitHub issues with `low-hanging-fruit` to make it easy for somebody to start helping out:

https://github.com/neo4jrb/neo4j/labels/low-hanging-fruit

https://github.com/neo4jrb/neo4j-core/labels/low-hanging-fruit

https://github.com/neo4jrb/neo4j-rake_tasks/labels/low-hanging-fruit

Help or discussion on other issues is welcome, just let us know!

Communicating With the Neo4j.rb Team

GitHub issues are a great way to submit new bugs / ideas. Of course pull requests are welcome (though please check with us first if it's going to be a large change). We like tracking our GitHub issues with waffle.io (neo4j, neo4j-core, neo4j-rake_tasks) but just through GitHub also works.

We hang out mostly in our Gitter.im chat room and are happy to talk or answer questions. We also are often around on the Neo4j-Users Slack group.

Running Specs

For running the specs, see our spec/README.md

Before you submit your pull request

Automated Tools

We use:

- RSpec
- Rubocop
- Coveralls

Please try to check at least the RSpec tests and Rubocop before making your pull request. `Guardfile` and `.overcommit.yml` files are available if you would like to use `guard` (for RSpec and rubocop) and/or overcommit.

We also use Travis CI to make sure all of these pass for each pull request. Travis runs the specs across multiple versions of Ruby and multiple Neo4j databases, so be aware of that for potential build failures.

Documentation

To aid our users, we try to keep a complete `CHANGELOG.md` file. We use keepachangelog.com as a guide. We appreciate a line in the `CHANGELOG.md` as part of any changes.

We also use Sphinx / reStructuredText for our documentation which is published on readthedocs.org. We also appreciate your help in documenting any user-facing changes.

Notes about our documentation setup:

- YARD documentation in code is also parsed and placed into the Sphinx site so that is also welcome. Note that reStructuredText inside of your YARD docs will render more appropriately.
- You can use `rake docs` to build the documentation locally and `rake docs:open` to open it in your web browser.
- Please make sure that you run `rake docs` before committing any documentation changes and checkin all changes to `docs/`.

CHAPTER 16

Additional Resources

The following is a list of resources where you can learn more about using Neo4j with Ruby.

- Neo4j.rb Screencast Series
- How NEO4J Saved my Relationship by Coraline Ada Ehmke
- Why You Should Use Neo4j in Your Next Ruby App
- Query or QueryProxy?
- Getting Started with Neo4j and Ruby
- Example Sinatra applications
  - Using the neo4j gem
  - Using only the neo4j-core gem

Neo4j.rb (the neo4j and neo4j-core gems) is a Ruby Object-Graph-Mapper (OGM) for the Neo4j graph database. It tries to follow API conventions established by ActiveRecord and familiar to most Ruby developers but with a Neo4j flavor.

Ruby (software) A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

Graph Database (computer science) A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way.

Neo4j (databases) The world's leading graph database

If you're already familiar with ActiveRecord, DataMapper, or Mongoid, you'll find the Object Model features you've come to expect from an O*M:

- Properties
- Indexes / Constraints
- Callbacks
- Validation

- Assocations

Because relationships are first-class citizens in Neo4j, models can be created for both nodes and relationships.

CHAPTER 17

Additional features include

- A chainable arel-inspired query builder
- Transactions
- Migration framework

CHAPTER 18

Requirements

- Ruby 1.9.3+ (tested in MRI and JRuby)
- Neo4j 2.1.0 + (version 4.0+ of the gem is required to use neo4j 2.2+)

CHAPTER 19

Indices and tables

- genindex
- modindex
- search

Index