Neo4j.rb Documentation

Release 10.0.0

Chris Grigg, Brian Underwood

Jul 07, 2020

Contents

Contents:

CHAPTER 1

Introduction

ActiveGraph is an ActiveRecord-inspired OGM (Object Graph Mapping, like ORM) for Ruby supporting Neo4j 3.4+.

1.1 Terminology

1.1.1 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)

1.1.2 ActiveGraph

ActiveGraph consists of the *activegraph* gem and suitable driver at level 1.7.x. There are currently 2 such available drivers gems: *neo4j-ruby-driver* and *neo4j-java-driver* (jruby only). Both drivers implement exactly the same api and can be swapped seamlessly on jruby.

activegraph Provides `Node` and `Relationship` modules for object modeling. Introduces *Model* and *Association* concepts (see below).

drivers Provide low-level connectivity, transactions, and response object wrapping with api and functionality consistent with the official neo4j drivers. Please see https://github.com/neo4jrb/neo4j-ruby-driver/tree/1.7

Model A Ruby class including either the `ActiveGraph::Node` module (for modeling nodes) or the `ActiveGraph::Relationship` 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 `Node` model. Defines either a `has_one` or `has_many` relationship to a model. A higher level abstraction of a **Relationship**

1.2 Code Examples

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

1.2.1 Node

Node 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
```

1.3 Setup

See the next section for instructions on *Setup*

CHAPTER 2

Setup

The `activegraph` gem supports both Ruby and JRuby and can be used with many different frameworks and services.

Below are some instructions on how to get started:

2.1 Ruby on Rails

The following contains instructions on how to setup ActiveGraph 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 ActiveGraph as the default model mapper or you can *add it manually*.

2.1.1 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 -O -m https://raw.githubusercontent.com/neo4jrb/activegraph/master/
↪docs/activegraph.rb
```

An example series of setup commands:

```
rails new myapp -O -m https://raw.githubusercontent.com/neo4jrb/activegraph/master/
↪docs/activegraph.rb
cd myapp
rake neo4j:install[community-4.0.6]
db/neo4j/development/bin/neo4j-admin set-initial-password password
rake neo4j:start
rails generate scaffold User name:string email:string
rake neo4j:migrate
```

(continues on next page)

(continued from previous page)

```
rails s
open http://localhost:3000/users
```

See also:

2.1.2 Adding the gem to an existing project

Include in your `Gemfile`:

```
# for rubygems
gem 'activegraph', '>= 10.0.0' # For example, see https://rubygems.org/gems/
↪activegraph/versions for the latest versions
gem 'neo4j-ruby-driver'
```

In `application.rb`:

```
require 'active_graph/railtie'
```

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

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

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

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

2.1.3 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=bolt://localhost:7687
```

For the `config/neo4j.yml` file:

```
development:
  url: neo4j://localhost:7687

test:
  url: neo4j://localhost:7688

production:
  url:
    - neo4j://core1:7687
    - neo4j://core2:7687
    - neo4j://core3:7687
```

(continues on next page)

(continued from previous page)

```
  username: neo4j
  password: password
```

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.driver.url = 'bolt://localhost:7687'
```

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.driver.url = 'neo4j://localhost:7687'
config.neo4j.driver.username = 'neo4j'
config.neo4j.driver.username = 'password'
```

2.2 Any Ruby Project

Include `activegrah` and either `neo4j-ruby-driver` or `neo4j-java-driver` in your `Gemfile`:

```
gem 'activegraph', '>= 10.0.0' # For example, see https://rubygems.org/gems/
↪activegraph/versions for the latest versions
gem 'neo4j-ruby-driver' # For example, see https://rubygems.org/gems/neo4j-ruby-
↪driver/versions for the latest versions
```

```
# Both are optional

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

2.2.1 Driver Instance

To start interacting with neo4j a driver instance is required:

In Ruby

When the railtie is included, this happens automatically.

Using the `acivegraph` gem (`Node` and `Relationship`) without Rails

To define your own driver for the `activegraph` gem you create a driver object and establish it as the default driver with the `Base` module (this is done automatically in Rails):

```
ActiveGraph::Base.driver = Neo4j::Driver::GraphDatabase.driver('neo4j://localhost:7687
↪', Neo4j::Driver.AuthTokens.basic('user','pass'), encryption: false)
```

Driver instances are thread-safe. Session and transactions can be created explicitly to guarantee reading your own writes and atomic operations with the following methods:

```
ActiveGraph::Base.session
ActiveGraph::Base.write_transaction
ActiveGraph::Base.read_transaction
```

In the absense of those method calls `activegraph` automatically creates a session and write transaction and associates them with the thread.

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

When trying to get the `activegraph` 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 `Node` 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 `ActiveGraph::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*.

2.4 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

This guide outlines changes from the last `neo4j`` gem version 9.x to `activegraph` version 10.x.

`activegraph` is an extensive refactoring of the `neo4j` gem. The major changes comprise of:

- full bolt support
- full causal cluster support
- removal of http support
- removal of embedded support (neo4j embedded is still supported via bolt)
- support for a neo4j ruby driver with an api of the official drivers
- discontinuation of the `neo4j-core` gem. Its functionality is replaced partially by `neo4j-ruby-driver` and partially by `activegraph`
- higher naming consistency with `activerecord` and the official `neo4j-java-driver`
- configuration more consistent with `activerecord`
- changed transaction API
- support for sessions with bookmarks and read and write transaction

3.1 How to upgrade to `activegraph`?

Your *neo4j* application is unlikely to work with `activegraph` out of the box. The good news is that the changes required are rather straightforward. To start follow the Setup guide. Once configured there few class name changes:

- Neo4j::ActiveNode became ActiveGraph::Node
- Neo4j::AciveRel became ActiveGrah::Relationship
- Neo4j::ActiveBase became ActiveGrapph::Base
- all other classes changed their namespace from Neo4j to ActiveGraph

If you use explicit cypher with `{parameter}` syntax you will need to change it to `$parameter` if using neo4j 4

3.1.1 Transaction API

The previous transaction api has been modified to support causal cluster and be a bit more intutive to `activerecord` users. The following methods provide that api:

- ActiveGraph::Base.session - corresponds to driver's session, if called multiple times from the same thread will use the same instance
- ActiveGraph::Base.transaction - corresponds to driver's begin_transaction, the most basic way of creating transactions
- ActiveGraph::Base.read_transaction - corresponds to a driver read_transaction, with retry logic, routed to a follower or read replica
- ActiveGraph::Base.write_transaction - corresponds to a driver witer_transaction, with retry logic, routed to the leader

All the above methods can be called on concrete model classes as well.

3.1.2 Exceptions

Several Exception types which previously were defined in the ``neo4j`` gem have been replaced with neo4j driver exceptions.

CHAPTER 4

Rake Tasks

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

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:shell **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

Node

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

To use Node, include ActiveGraph::Node in a class.

```
class Post
  include ActiveGraph::Node
end
```

5.1 Properties

Properties for ActiveGraph::Node objects must be declared by default. Properties are declared using the property method which is the same as attribute from the active_attr gem.

Example:

```
class Post
  include ActiveGraph::Node
  property :title
  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
```

See the Properties section for additional information.

See also:

5.1.1 Labels

By default `Node` 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 `Node` by using `mapped_label_name` like so:

```ruby
class Post
  include ActiveGraph::Node

  self.mapped_label_name = 'BlogPost'
end
```

5.1.2 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 `Node` 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.

5.1.3 Labels

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

```ruby
class Post
  include ActiveGraph::Node
end
```

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

```ruby
class Post
  include ActiveGraph::Node

  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.

```ruby
class Post
  include ActiveGraph::Node
end

class Article < Post
end
```

5.1.4 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 ActiveGraph::Node

  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.

5.1.5 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 ActiveGraph::Node

  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 require you to add an associated index to improve query performance. If you want to disable this, simply pass `_index: false` to `enum`:

```
class Media
  include ActiveGraph::Node

  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 pass the `_default` option when defining the enum:

```
class Media
  include ActiveGraph::Node

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

By default, enum setters are *case insensitive* (in the example below, `Media.create(type: 'VIDEO').type == :video`). If you wish to disable this for a specific enum, pass the `_case_sensitive: true` option. if you wish to change the global default for `_case_sensitive` to `true`, use Neo4jrb's `enums_case_sensitive` config option (detailed in the *Variables* section).

```
class Media
  include ActiveGraph::Node

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

5.2 Scopes

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

```ruby
class Person
  include ActiveGraph::Node

  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 ActiveGraph::Node

  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 ActiveGraph::Node

  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 ActiveGraph::Node

  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:

```
class Person
  include ActiveGraph::Node

  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.

5.3 Wrapping

When loading a node from the database there is a process to determine which `Node` 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.

5.4 Callbacks

Implements like Active Records the following callback hooks:

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

5.5 created_at, updated_at

```
class Blog
  include ActiveGraph::Node

  include ActiveGraph::Timestamps # will give model created_at and updated_at␣
↪timestamps
```

(continues on next page)

(continued from previous page)

```
  include ActiveGraph::Timestamps::Created # will give model created_at timestamp
  include ActiveGraph::Timestamps::Updated # will give model updated_at timestamp
end
```

5.6 Validation

Support the Active Model validation, such as:

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

5.7 id property (primary key)

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

5.8 Associations

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

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

class Comment
  include ActiveGraph::Node
  has_one :out, :post, type: :post
  has_one :out, :author, type: :author, model_class: :Person
end

class Person
  include ActiveGraph::Node
  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
```

When querying `has_one` associations, by default `.first` will be called on the result. This makes the result non-chainable if the result is `nil`. If you want to ensure a chainable result, you can call `has_one` with a `chainable: true` argument.

```
comment.post                    # Post object
comment.post(chainable: true)   # Association proxy object wrapping post
```

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
```

5.8.1 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)
```

5.8.2 Polymorphic Associations

`has_one` or `has_many` associations which target multiple `model_class` are called polymorphic associations. This is done by setting `model_class: false` or `model_class: [:ModelOne, :ModelTwo, :Etc]`. In our example, the `Person` class has a polymorphic association `written_things`

```
class Person
  include ActiveGraph::Node

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

You can't perform standard association chains on a polymorphic association. For example, while you *can* call `post.comments.author.written_things`, you *cannot* call `post.comments.author.written_things.post.comments` (an exception will be raised). In this example, the return of `.written_things` can be either a `Post` object or a `Comment` object, any method you called on an association made up of them both could have a different meaning for the `Post` object vs the `Comment` object. So how can you execute `post.comments.author.written_things.post.comments`? This is where `.query_as` and `.proxy_as` come to the rescue! While

`Node` doesn't know how to handle the `.post` call on `.written_things`, you *know* that the path from the return of `.written_things` to `Post` nodes is `(written_thing)-[:post]->(post:Post)`. To help `Node` out, convert the *AssociationProxy*` object returned by `post.comments.author.written_things` into a `Query` object with `.query_as()`, then manually specify the path of `.post`. Like so:

```
post.comments.author.written_things.query_as(:written_thing).match("(written_thing)-
↪[:post]->(post:Post)")
```

It's worth noting that the object returned by this chain is now a `Query` object, meaning that if you wish to get the result (`(post:Post)`), you'll need to `.pluck(:post)` it. However, we don't want to get the result yet. Instead, we wish to perform further queries. Because the end of the chain is now a `Query`, we could continue to manually describe the path to the nodes we want using the `Query` API of `.match`, `.where`, `.return`, etc. For example, to get `post.comments.author.written_things.post.comments` we could

```
post.comments.author.written_things.query_as(:written_thing).match("(written_thing)-
↪[:post]->(post:Post)").match("(post)<-[:post]-(comment:Comment)").pluck(:comment)
```

But this isn't ideal. It would be nice to make use of `Node`'s association chains to complete our query. We *know* that the return of `post.comments.author.written_things.query_as(:written_thing).match("(written_thing)-[:post]->(post:Post)")` is a `Post` object, after all. To allow for association chains in this circumstance, `.proxy_as()` comes to the rescue! If we *know* that a `Query` will return a specific model class, `proxy_as` allows us to tell Neo4jrb this, and begin association chaining from that point. For example

```
post.comments.author.written_things.query_as(:written_thing).match("(written_thing)-
↪[:post]->(post:Post)").proxy_as(Post, :post).comments.author
```

See also:

#query_as http://www.rubydoc.info/gems/activegraph/ActiveGraph/Node/Query/QueryProxy#query_as-instance_method and #proxy_as http://www.rubydoc.info/gems/activegraph/ActiveGraph/Core/Query#proxy_as-instance_method

5.8.3 Dependent Associations

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

```
class Route
  include ActiveGraph::Node
  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 http://www.rubydoc.info/gems/activegraph/ActiveGraph/Node/HasN/ClassMethods#has_many-instance_method and #has_one http://www.rubydoc.info/gems/activegraph/ActiveGraph/Node/HasN/ClassMethods#has_one-instance_method

5.8.4 Association Options

By default, when you call an association `Node` will add the `model_class` labels to the query (as a filter). For example:

```
person.friends
# =>
# MATCH (person125)
# WHERE (ID(person125) = $ID_person125)
# MATCH (person125)-[rel1:`FRIEND`]->(node3:`Person`)
```

The exception to this is if `model_class: false`, in which case `MATCH (person125)-[rel1:`FRIEND`]->(node3)`. More advanced Neo4j users may prefer to skip adding labels to the target node, even if `model_class != false`. This can be accomplished on a case-by-case basis by calling the association with a *labels: false*‘ options argument. For example: `person.friends(labels: false)`.

You can also make `labels: false` the default settings by creating the association with a `labels: false` option. For example:

```
class Person
  has_many :out, :friends, type: :FRIEND, model_class: self, labels: false
end
```

5.8.5 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.

5.8.6 Eager Loading

Node 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 *Node* 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. *Node* then automatically structures them into a nested set of *Node* objects for you.

You can also use `with_associations` with multiple levels like:

```
person.blog_posts.with_associations(:tags, comments: :hashtags)
```

You can use `*` to eager load relationships with variable length like:

```
person.blog_posts.with_associations('comments.owner.friends*')
```

To get fixed length relationships you can use `*<length>` like:

```
person.blog_posts.with_associations('comments.owner.friends*2')
```

This will eager load `friends` relationship till 2 levels deep.

CHAPTER 6

Relationship

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

6.1 When to Use?

It is not always necessary to use Relationship 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 `Relationship` doesn't implement a `uuid` property like `Node`.

6.2 Setup

Relationship model definitions have three requirements:

- `include ActiveGraph::Relationship`
- Call `from_class` with a symbol/string referring to an `Node` model or :any
- Call `to_class` with a symbol/string referring to an `Node` model or :any

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

```ruby
# app/models/enrolled_in.rb
class EnrolledIn
  include ActiveGraph::Relationship
  before_save :do_this

  from_class :Student
  to_class   :Lesson
  # `type` can be specified, but it is assumed from the model name
```

(continues on next page)

(continued from previous page)

```
  # 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 `Relationship` model in `Node` models:
# app/models/student.rb
class Student
  include ActiveGraph::Node

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

# app/models/lesson.rb
class Lesson
  include ActiveGraph::Node

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

See also:

6.3 Relationship Creation

6.3.1 From an Relationship Model

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

Relationship 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)
```

6.3.2 From a *has_many* or *has_one* association

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

```
class Student
  include ActiveGraph::Node
  has_many :out, :lessons, rel_class: :EnrolledIn
end
```

6.3.3 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.

6.4 Query and Loading existing relationships

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

6.4.1 :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.

6.5 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
=> #<ActiveGraph::Relationship::RelatedNode:0x00000104589d78 @node=#<Student␣
↪property: 'value'>>
rel.to_node
=> #<ActiveGraph::Relationship::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.

6.6 Advanced Usage

6.6.1 Separation of Relationship Logic

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

```
class User
  include ActiveGraph::Node
  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 ActiveGraph::Relationship
  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
```

(continues on next page)

(continued from previous page)

```
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
```

6.7 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

6.8 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 ActiveGraph::Core API while offering what may be more natural options for Rails users.

CHAPTER 7

Properties

In classes that mixin the `ActiveGraph::Node` or `ActiveGraph::Relationship` 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 ActiveGraph::Node

  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`

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 ActiveGraph::Node

  property :title, type: String, default: 'This ia new post'
  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.

7.1 Validations

The `Node` and `Relationship` modules in the `neo4j` gem are based off of `ActiveModel`. Because of this you can use any validations defined by `ActiveModel` as well as create your own in the same style. For the best documentation on validations, see the Active Record Validations page. The `neo4j` gem isn't based off of `ActiveRecord` aside from being inspired by it, but they both use `ActiveModel` under the covers.

One validation to note in particular is `validates_uniqueness_of`. Whereas most validations work only on the model in memory, this validation requires connecting to the database. The `neo4j` gem implements it's own version of `validates_uniqueness_of` for Neo4j.

7.2 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 `ActiveGraph::UndeclaredProperties` no exceptions will be thrown if unknown attributes are passed to selected methods.

```
class Post
  include ActiveGraph::Node
  include ActiveGraph::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.

7.2.1 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.

```ruby
class Post
  include ActiveGraph::Node

  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.

7.2.2 Custom Converters

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

```ruby
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 ActiveGraph::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 `ActiveGraph::Node` classes instead of using the internal Neo4j ids. By default, Node 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 Relationship 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 Node class!

8.1 Defining your own ID

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

```ruby
class Person
  include ActiveGraph::Node
  id_property :personal_id, on: :phone_and_name

  property :name
  property :phone

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

8.2 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 ActiveGraph::Node
  id_property :neo_id
end
```

8.3 A note regarding constraints

A constraint is required for the `id_property` of an `Node` 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.

8.4 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 < ActiveGraph::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).

8.5 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 *Node*. 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
```

```ruby
class AddConstraintToTag < ActiveGraph::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 Node

```ruby
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

9.1 Introduction

Using the `activegraph` gem provides the entry point is `ActiveGraph::Base`. So you could make a simple query with:

```
ActiveGraph::Base.query('MATCH (n) RETURN n LIMIT $limit', limit: 10)
```

Most of the time, though, using the `activegraph` gem involves using the `Node` and `Relationship` APIs as described below.

9.2 Node

9.2.1 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')
```

9.2.2 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 `Node` 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)
```

(continues on next page)

(continued from previous page)

```
# 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!

9.2.3 The Query API

The `activegraph` 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 `ActiveGraph::Base` or by building a `Query` off of a scope such as above.

```
ActiveGraph::Base.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:

9.2.4 #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.

9.2.5 `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.

9.2.6 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)

9.2.7 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

9.2.8 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

Query Examples

In the rest of the documentation for this site we try to lay out all of the pieces of the Neo4j.rb gems to explain them one at a time. Sometimes, though, it can be instructive to see examples. The following are examples of code where somebody had a question and the resulting code after fixes / refactoring. This section will expand over time as new examples are found.

10.1 Example 1: Find all contacts for a user two hops away, but don't include contacts which are only one hop away

```
user.contacts(:contact, :knows, rel_length: 2).where_not(
  uuid: user.contacts.pluck(:uuid)
)
```

This works, though it makes two queries. The first to get the `uuid` s for the `where_not` and the second for the full query. For the first query, `user.contacts.pluck(:id)` could be also used instead, though associations already have a pre-defined method to get IDs, so this could instead be `user.contact_ids`.

This doesn't take care of the problem of having two queries, though. If we keep the `rel_length: 2`, however, we won't be able to reference the nodes which are one hop away in order. This seems like it would be a straightforward solution:

```
user.contacts(:contact1).contacts(:contact2).where_not('contact1 = contact2')
```

And it is straightforward, but it won't work. Because Cypher matches one subgraph at a time (in this case roughly `(:User)--(contact1:User)--(contact2:User)`), `contact` one is always just going to be the node which is in between the user in question and `contact2`. It doesn't represent "all users which are one step away". So if we want to do this as one query, we do need to first get all of the first-level nodes together so that we can then check if the second level nodes are in that list. This can be done as:

```
user.as(:user).contacts
  .query_as(:contact).with(:user, first_level_ids: 'collect(ID(contact))')
```

(continues on next page)

(continued from previous page)

```
  .proxy_as(User, :user)
  .contacts(:other_contact, nil, rel_length: 2)
  .where_not('ID(other_contact) IN first_level_ids')
```

And there we have a query which is much more verbose than the original code, but accomplishes the goal in a single query. Having two queries isn't neccessarily bad, so the code's complexity should be weighed against how both versions perform on real datasets.

10.2 Example 2: Simple Recommendation Engine

If you are interested in more complex collaborative filter methods check out this article.

Let's assume you have the following schema:

```
(:User)-[:FOLLOW|:SKIP]->(:Page)
```

We want to recommend pages for a user to follow based on their current followed pages.

Constraints:

- We want to include the source of the recommendation. i.e (we recommend you follow X because you follow Y).

  Note : To do this part, we are going to use an APOC function `apoc.coll.sortMaps`.

- We want to exclude pages the user has skipped or already follows.
- The recommended pages must have a name field.

Given our schema, we could write the following Cypher to accomplish this:

```
MATCH (user:User { id: "1" })
MATCH (user)-[:FOLLOW]->(followed_page:Page)<-[:FOLLOW]-(co_user:User)
MATCH (co_user)-[:FOLLOW]->(rec_page:Page)
WHERE exists(rec_page.name)
AND NOT (user)-[:FOLLOW|:SKIP]->(rec_page)
WITH rec_page, count(rec_page) AS score, collect(followed_page.name) AS source_names
ORDER BY score DESC LIMIT {limit}
UNWIND source_names AS source_name
WITH rec_page, score, source_name, count(source_name) AS contrib
WITH rec_page, score, apoc.coll.sortMaps(collect({name:source_name, contrib:contrib*-
↪1}), 'contrib') AS sources
RETURN rec_page.name AS name, score, extract(source IN sources[0..3] | source.name)␣
↪AS top_sources,
  size(sources) AS sources_count
ORDER BY score DESC
```

Now let's see how we could write this using Node syntax in a `User` Ruby class.

```
class User
  include ActiveGraph::Node

  property :id, type: Integer

  has_many :out, :followed_pages, type: :FOLLOW, model_class: :Page
  has_many :out, :skipped_pages, type: :SKIP, model_class: :Page

```

(continues on next page)

(continued from previous page)

```
  def recommended_pages
    as(:user)
      .followed_pages(:followed_page)
        .where("exists(followed_page.name)")
      .followers(:co_user)
      .followed_pages
      .query_as(:rec_page) # Transition into Core Query
        .where("exists(rec_page.name)")
        .where_not("(user)-[:FOLLOW|:SKIP]->(rec_page)")
      .with("rec_page, count(rec_page) AS score, collect(followed_page.name) AS␣
↪source_names")
        .order_by('score DESC').limit(25)
      .unwind(source_name: :source_names) # This generates "UNWIND source_names AS␣
↪source_name"
      .with("rec_page, score, source_name, count(source_name) AS contrib")
      .with("rec_page, score, apoc.coll.sortMaps(collect({name:source_name,
↪contrib:contrib*-1}), 'contrib') AS sources")
      .with("rec_page.name AS name, score, extract(source in sources[0..3] | source.
↪name) AS top_sources, size(sources) AS sources_count")
        .order_by('score DESC')
      .pluck(:name, :score, :top_sources, :sources_count)
  end
end
```

Note : The *contrib*-1* value is a way of getting the desired order out of the *sortMaps* APOC function without needing to reverse the resulting list.

This assumes we have a `Page` class like the following:

```
class Page
  include ActiveGraph::Node

  property name, type: String

  has_many :in, :followers, type: :FOLLOW, model_class: :User
  has_many :in, :skippers, type: :SKIP, model_class: :User
end
```

CHAPTER 11

QueryClauseMethods

The `ActiveGraph::Core::Query` class 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 `Node` / `Relationship` in the `neo4j` gem can be retrieved␣
↪with `ActiveGraph::Base.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.

11.1 ActiveGraph::Core::Query

11.1.1 #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)
```

11.1.2 #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)
```

11.1.3 #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
```

11.1.4 #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}`

11.1.5 #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)
```

11.1.6 #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}`

11.1.7 #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
```

11.1.8 #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
```

11.1.9 #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
```

11.1.10 #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

```

```

11.1.11 #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}`

11.1.12 #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
```

11.2 #with_distinct

Ruby

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

Cypher

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

Ruby

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

Cypher

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

Ruby

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

Cypher

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

Ruby

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

Cypher

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

11.2.1 #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}`

11.2.2 #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}`

11.2.3 #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}`

11.2.4 #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
```

11.2.5 #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}}`

11.2.6 #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

```

```

11.2.7 #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}`

11.2.8 #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}`

11.2.9 #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`
```

11.2.10 #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")
```

11.2.11 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 12

Configuration

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

12.1 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.

12.2 Other Ruby apps

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

12.3 Variables

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'`.

class_name_property Default: `:_classname`

Which property should be used to determine the `Node` 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 `Node` class

See also:

Wrapping

enums_case_sensitive **Default:** `false`

Determins whether enums property setters should be case sensitive or not.

See also:

Enums

include_root_in_json **Default:** `true`

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

See also:

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

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 `Node` and `Relationship` modules).

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 `Node` model name to the node label or an `Relationship` 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.

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 Node and Relationship models will include the Timestamps module and have `:created_at` and `:updated_at` properties.

skip_migration_check **Default:** `false`

Prevents the `neo4j` gem from raising `ActiveGraph::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

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`).

transform_rel_type **Default:** `:upcase`

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

Determines how relationship types for `Relationship` models are transformed when stored in the database. By default this is upper-case to match with Neo4j convention so if you specify an `Relationship` model of `HasPost` 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

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

verbose_query_logs Default: `false`

Specifies that queries outputted to the log also get a source file / line outputted to aid debugging.

12.4 Instrumented events

The `activegraph` 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 gem:

```
ActiveGraph::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:

ActiveGraph::Base.subscribe_to_query neo4j.core.cypher_query

ActiveGraph::Base.subscribe_to_request neo4j.core.http.request

CHAPTER 13

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 `Node` or `Relationship` 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.

13.1 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 < ActiveGraph::Migrations::Base
  def up
    rename_property :User, :name, :first_name
  end

  def down
    rename_property :User, :first_name, :name
```

(continues on next page)

(continued from previous page)

```
  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.

13.2 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 < ActiveGraph::Migrations::Base
  disable_transactions!

  ...
end
```

13.3 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.

13.4 Tasks

Neo4j.rb implements a clone of the `ActiveRecord` migration tasks API to migrate.

13.4.1 neo4j:migrate:all

Runs any pending migration.

```
rake neo4j:migrate:all
```

13.4.2 neo4j:migrate

An alias for `rake neo4j:migrate:all`.

```
rake neo4j:migrate:all
```

13.4.3 neo4j:migrate:up

Executes a migration given it's version id.

```
rake neo4j:migrate:up VERSION=some_version
```

13.4.4 neo4j:migrate:down

Reverts a migration given it's version id.

```
rake neo4j:migrate:down VERSION=some_version
```

13.4.5 neo4j:migrate:status

Prints a detailed migration state report, showing up and down migrations together with their own version id.

```
rake neo4j:migrate:status
```

13.4.6 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
```

13.4.7 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
```

13.4.8 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
```

13.5 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`

13.6 Migration Helpers

13.6.1 #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')
```

13.6.2 #query

An alias for `ActiveGraph::Session.query`. You can use it as root for the query builder:

```
query.match(:n).where(name: 'John').delete(:n).exec
```

13.6.3 #remove_property

Removes a property given a label.

```
remove_property(:User, :money)
```

13.6.4 #rename_property

Renames a property given a label.

```
rename_property(:User, :name, :first_name)
```

13.6.5 #drop_nodes

Removes all nodes with a certain label

```
drop_nodes(:User)
```

13.6.6 #add_label

Adds a label to nodes, given their current label

```
add_label(:User, :Person)
```

13.6.7 #add_labels

Adds labels to nodes, given their current label

```
add_label(:User, [:Person, :Boy])
```

13.6.8 #remove_label

Removes a label from nodes, given a label

```
remove_label(:User, :Person)
```

13.6.9 #remove_labels

Removes labels from nodes, given a label

```
remove_label(:User, [:Person, :Boy])
```

13.6.10 #rename_label

Renames a label

```
rename_label(:User, :Person)
```

13.6.11 #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.

13.6.12 #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.

13.6.13 #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.

13.6.14 #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.

13.6.15 #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
```

13.6.16 #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
```

13.6.17 #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.

13.6.18 #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
```

13.6.19 #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 14

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*)

14.1 How to clear the database

14.1.1 Cypher DELETE

This is the most reliable way to clear your database in Neo4j

```
// For version of Neo4j after 2.3.0
// DETACH DELETE takes care of removing relationships for you
MATCH (n) DETACH DELETE n
```

In Ruby:

```
ActiveGraph::Base.query('MATCH (n) DETACH DELETE n')
```

If you are using `Node` and/or `Relationship` from the `activegraph` 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:`ActiveGraph::Migrations::SchemaMigration`
DETACH DELETE n
```

14.1.2 The `database_cleaner` gem

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. (see below)

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.

14.1.3 Delete data files

Completely delete the database files (slower, by removeds schema). If you installed Neo4j via the `neo4j-rake_tasks` gem, 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.

14.1.4 RSpec Transaction Rollback

If you are using RSpec you can perform tests in a transaction as you would using ActiveRecord. Just add the following to your rspec configuration in `spec/rails_helper.rb` or `spec/spec_helper.rb`

```
# For the `neo4j` gem
config.around do |example|
  ActiveGraph::Base.transaction do |tx|
    example.run
    tx.failure
  end
end
```

There is one big disadvantage to this approach though: In Neo4j, nested transactions still act as one big transaction. If the code you are testing has a transaction which, for example, gets marked as failed, then the transaction around the RSpec example will be marked as failed.

14.1.5 Using Rack::Test

If you're using the *Rack::Test <https://github.com/rack-test/rack-test>* gem to test your Neo4j-enabled web application from the outside, be aware that the *Rack::Test::Methods* mixin won't work with this driver. Instead, use the *Rack::Test::Session* approach as described in the *Sinatra documentation <http://sinatrarb.com/testing.html>*.

CHAPTER 15

Contributing

We very much welcome contributions! Before contributing there are a few things that you should know about the neo4j.rb projects:

15.1 The Neo4j.rb Project

We have two main gems: activegraph, neo4j-ruby-driver.

We try to follow semantic versioning based on *semver.org <http://semver.org/>*

15.2 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!

15.3 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 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.

15.4 Running Specs

For running the specs, see our spec/README.md

15.5 Before you submit your pull request

15.5.1 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.

15.5.2 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

CHAPTER 17

Helper Gems

17.1 devise-activegraph

devise-activegraph is an adaptor gem for using the devise authentication library with ActiveGraph.

17.2 cancancan-activegraph

The cancancan-neo4j gem is the neo4j adapter for the CanCanCan authorisation library. This gem will help you seamlessly integrate cancan gem to your Ruby/Rails app wich has Neo4j as database.

17.3 neo4j-paperclip

Currently not compatible with `activegraph` The neo4jrb-paperclip gem allows easy use of the `paperclip` gem in `Node` and `Relationship` models.

17.4 neo4jrb_spatial

Obsolete due to native neo4j data types. The neo4jrb_spatial gem add the ability to work with the Neo4j Spatial server plugin via the `neo4j` and `neo4j-core` gems

17.5 neo4j-rspec

Currently not compatible with `activegraph` The neo4j-rspec gem adds RSpec matchers for easier testing of `Node` and `Relationship` models.

ActiveGraph (the activegraph gem) 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
- Associations

Because relationships are first-class citizens in Neo4j, models can be created for both nodes and relationships.

CHAPTER 18

Additional features include

- A chainable arel-inspired query builder
- Transactions
- Migration framework

CHAPTER 19

Requirements

- Ruby 2.5 + (tested in MRI and JRuby)
- Neo4j 3.4 +

CHAPTER 20

Indices and tables

- genindex
- modindex
- search

Index