
Neo4j.rb Documentation

Release 7.0.1

Chris Grigg, Brian Underwood

March 21, 2016

1	Introduction	3
1.1	Terminology	3
1.1.1	Neo4j	3
1.1.2	Neo4j.rb	3
1.2	Code Examples	4
1.2.1	ActiveNode	4
1.3	Setup	4
2	Setup	5
2.1	Ruby on Rails	5
2.1.1	Generating a new app	5
2.1.2	Adding the gem to an existing project	6
2.1.3	Rails configuration	6
2.1.4	Configuring Faraday	7
2.2	Any Ruby Project	7
2.2.1	Connection	7
2.3	Heroku	7
2.3.1	Rails configuration	8
3	Rake Tasks	9
4	ActiveNode	11
4.1	Properties	11
4.1.1	Labels	12
4.1.2	Indexes	12
4.1.3	Constraints	12
4.1.4	Labels	12
4.1.5	Serialization	13
4.1.6	Enums	13
4.2	Wrapping	14
4.3	Callbacks	15
4.4	created_at, updated_at	15
4.5	Validation	15
4.6	id property (primary key)	15
4.7	Associations	15
4.7.1	Creating Unique Relationships	16
4.7.2	Eager Loading	17

5	ActiveRel	19
5.1	When to Use?	19
5.2	Setup	19
5.3	Relationship Creation	20
5.3.1	From an ActiveRel Model	20
5.3.2	From a <i>has_many</i> or <i>has_one</i> association	20
5.3.3	Creating Unique Relationships	20
5.4	Query and Loading existing relationships	21
5.4.1	:each_rel, :each_with_rel, or :pluck methods	21
5.5	Accessing related nodes	21
5.6	Advanced Usage	21
5.6.1	Separation of Relationship Logic	21
5.7	Additional methods	22
5.8	Regarding: from and to	22
6	Properties	23
6.1	Undeclared Properties	24
6.1.1	Types and Conversion	24
6.1.2	Custom Converters	25
7	Querying	27
7.1	Simple Query Methods	27
7.1.1	.find	27
7.1.2	.find_by	27
7.2	Scope Method Chaining	27
7.2.1	Querying the scope	28
7.2.2	Chaining associations	28
7.2.3	Associations and Unpersisted Nodes	28
7.2.4	Parameters	29
7.2.5	Variable-length relationships	29
7.3	The Query API	30
7.4	#proxy_as	31
7.5	match_to and first_rel_to	31
7.6	Finding in Batches	31
7.7	Orm_Adapter	31
7.8	Find or Create By...	31
8	QueryClauseMethods	33
8.1	Neo4j::Core::Query	33
8.1.1	#match	33
8.2	#optional_match	35
8.3	#using	36
8.4	#where	36
8.5	#where_not	40
8.6	#match_nodes	41
8.6.1	one node object	41
8.7	integer	42
8.8	two node objects	42
8.9	node object and integer	42
8.10	#unwind	42
8.11	#return	43
8.12	#order	44
8.13	#limit	45
8.14	#skip	46

8.15	#with	47
8.16	#create	48
8.17	#create_unique	49
8.18	#merge	50
8.19	#delete	51
8.20	#set_props	51
8.21	#set	52
8.22	#on_create_set	53
8.23	#on_match_set	54
8.24	#remove	55
8.25	#start	56
8.26	clause combinations	57
9	Configuration	61
9.1	In Rails	61
9.2	Other Ruby apps	61
9.2.1	Variables	61
10	Contributing	63
10.1	The Neo4j.rb Project	63
10.2	Low Hanging Fruit	63
10.3	Communicating With the Neo4j.rb Team	63
10.4	Running Specs	63
10.5	Before you submit your pull request	64
10.5.1	Automated Tools	64
10.5.2	Documentation	64
11	API	65
11.1	Neo4j	65
11.1.1	Error	65
11.1.2	RecordNotFound	65
11.1.3	InvalidPropertyOptionsError	66
11.1.4	InvalidParameterError	66
11.1.5	UnknownTypeConverterError	66
11.1.6	DangerousAttributeError	67
11.1.7	UnknownAttributeError	67
11.1.8	Config	67
11.1.9	Shared	69
11.1.10	Railtie	116
11.1.11	ClassWrapper	118
11.1.12	Paginated	118
11.1.13	Migration	119
11.1.14	ActiveRel	121
11.1.15	Core	150
11.1.16	Timestamps	151
11.1.17	ActiveNode	152
11.1.18	TypeConverters	236
11.1.19	ClassArguments	236
11.1.20	Schema	237
11.1.21	Relationship	242
11.1.22	Node	243
11.1.23	Generators	244
11.1.24	Constants	246
11.1.25	Files	246

11.1.26	Methods	248
11.2	Rails	248
11.2.1	Generators	248
11.2.2	Constants	248
11.2.3	Files	248
11.2.4	Methods	249
12	Additional features include	251
13	Requirements	253
14	Indices and tables	255

Contents:

Introduction

- *Terminology*
 - *Neo4j*
 - *Neo4j.rb*
- *Code Examples*
 - *ActiveNode*
- *Setup*

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

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 Neo4j.rb

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

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

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

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

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

1.2 Code Examples

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

1.2.1 ActiveNode

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

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

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

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

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

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

1.3 Setup

See the next section for instructions on [Setup](#)

Setup

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

Below are some instructions on how to get started:

2.1 Ruby on Rails

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

There are two ways to add `neo4j` to your Rails project. You can [LINK](#) generate a new project [LINK](#) with Neo4j as the default model mapper or you can [LINK](#) add it manually [LINK](#).

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

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

An example series of setup commands:

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

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

See also:

2.1.2 Adding the gem to an existing project

Include in your Gemfile:

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

In application.rb:

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

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

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

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

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

2.1.3 Rails configuration

For both new apps and existing apps the following configuration applies:

An example config/neo4j.yml file:

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

test:
  type: server_db
  url: http://localhost:7575

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

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

```
config.neo4j.session_type = :server_db
config.neo4j.session_path = 'http://localhost:7474'
```

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

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

Of course it's often the case that you don't want to expose your username / password / URL in your repository. In these cases you can use the NEO4J_TYPE (either server_db or embedded_db) and NEO4J_URL/NEO4J_PATH environment variables.

2.1.4 Configuring Faraday

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

```
config.neo4j.session_options = {initialize: { ssl: { verify: true }}}}
```

2.2 Any Ruby Project

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

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

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

```
# Both are optional

# This provides tasks to install/start/stop/configure Neo4j
load 'neo4j/tasks/neo4j_server.rake'
# This provides tasks to have migrations
load 'neo4j/tasks/migration.rake'
```

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 Connection

To open a session to the neo4j server database:

In Ruby

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

Embedded mode in JRuby

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

```
session = Neo4j::Session.open(:embedded_db, '/folder/db')
session.start
```

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

2.3 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>

2.3.1 Rails configuration

config/application.rb

```
config.neo4j.session_type = :server_db
# GrapheneDB
config.neo4j.session_path = ENV["GRAPHENEDB_URL"] || 'http://localhost:7474'
# Graph Story
config.neo4j.session_path = ENV["GRAPHSTORY_URL"] || 'http://localhost:7474'
```

Rake Tasks

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

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

neo4j: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`

neo4j:config Arguments: environment and port

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

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

neo4j:start Arguments: environment

Example: `rake neo4j:start[development]`

Start the Neo4j server

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

neo4j:start Arguments: environment

Example: `rake neo4j:shell[development]`

Open a Neo4j shell console (REPL shell).

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

neo4j:start_no_wait Arguments: environment

Example: `rake neo4j:start_no_wait[development]`

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

neo4j:stop Arguments: environment

Example: rake neo4j:stop[development]

Stop the Neo4j server

neo4j:restart Arguments: environment

Example: rake neo4j:restart[development]

Restart the Neo4j server

ActiveNode

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

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

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

4.1 Properties

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

Example:

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

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

  before_save do
    self.score = score * 100
  end

  has_n :friends
end
```

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

See the Properties section for additional information.

See also:

4.1.1 Labels

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

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

```
class Post
  include Neo4j::ActiveNode

  self.mapped_label_name = 'BlogPost'
end
```

4.1.2 Indexes

To declare a index on a property

```
class Person
  include Neo4j::ActiveNode
  property :name, index: :exact
end
```

Only exact index is currently possible.

Indexes can also be declared like this:

```
class Person
  include Neo4j::ActiveNode
  property :name
  index :name
end
```

4.1.3 Constraints

You can declare that a property should have a unique value.

```
class Person
  property :id_number, constraint: :unique # will raise an exception if id_number is not unique
end
```

Notice an unique validation is not enough to be 100% sure that a property is unique (because of concurrency issues, just like `ActiveRecord`). Constraints can also be declared just like indexes separately, see above.

4.1.4 Labels

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

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

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

```
class Post
  include Neo4j::ActiveNode

  self.mapped_label_name = 'BlogPost'
end
```

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

```
class Post
  include Neo4j::ActiveNode
end

class Article < Post
end
```

4.1.5 Serialization

Pass a property name as a symbol to the serialize method if you want to save a hash or an array with mixed object types* to the database.

```
class Student
  include Neo4j::ActiveNode

  property :links

  serialize :links
end

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

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

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

4.1.6 Enums

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

```
class Media
  include Neo4j::ActiveNode

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

media = Media.create(type: :video)
media.type
```

```
# => :video
media.image!
media.image?
# => true
```

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

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

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

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

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

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

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

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

Or by using where:

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

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

```
class Media
  include Neo4j::ActiveNode

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

4.2 Wrapping

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

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

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

4.3 Callbacks

Implements like Active Records the following callback hooks:

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

4.4 created_at, updated_at

```
class Blog
  include Neo4j::ActiveNode

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

4.5 Validation

Support the Active Model validation, such as:

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

4.6 id property (primary key)

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

4.7 Associations

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

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

class Comment
  include Neo4j::ActiveNode
  has_one :out, :post, type: :post
end
```

```
has_one :out, :author, type: :author, model_class: :Person
end

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

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

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

  # or if you want 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.authors.posts # All posts of people who have commented on the post. Still makes just one query
```

You can create associations

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

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

See also:

See also:

#has_many and *#has_one*

4.7.1 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 or 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.

4.7.2 Eager Loading

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

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

Only three Cypher queries will be made:

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

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

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

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

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

ActiveRel

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

5.1 When to Use?

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

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

... Documentation notes

Separation of relationship logic instead of shoehorning it into Node models

Validations, callbacks, custom methods, etc.

Centralize relationship type, no longer need to use `:type` or `:origin` options in models

5.2 Setup

ActiveRel model definitions have four requirements:

- include `Neo4j::ActiveRel`
- call `from_class` with a valid model constant or `:any`
- call `to_class` with a valid model constant or `:any`
- call `type` with a Symbol or String to define the Neo4j relationship type

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

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

  from_class :Student
  to_class   :Lesson
  type 'enrolled_in'
```

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

validates_presence_of :since

def do_this
  #a callback
end
end
```

See also:

5.3 Relationship Creation

5.3.1 From an ActiveRel Model

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

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

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

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

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

5.3.2 From a *has_many* or *has_one* association

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

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

5.3.3 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 or 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.

5.4 Query and Loading existing relationships

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

5.4.1 :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.

5.5 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
=> #<Neo4j::ActiveRel::RelatedNode:0x00000104589d78 @node=#<Student property: 'value'>>
rel.to_node
=> #<Neo4j::ActiveRel::RelatedNode:0x00000104589d50 @node=#<Lesson property: 'value'>>
```

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

5.6 Advanced Usage

5.6.1 Separation of Relationship Logic

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

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

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

  def update_stats
    managed_stats += 1
    save
  end
end

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

  def update_user_stats
    from_node.update_stats
  end

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

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

5.7 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

5.8 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 Neo4j::Core API while offering what may be more natural options for Rails users.

Properties

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

```
class Post
  include Neo4j::ActiveNode

  property :title
end
```

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

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

Node models have two more options:

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

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

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

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

```
class Post
  include Neo4j::ActiveNode

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

  serialize :links
end
```

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

6.1 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. For instance:

```
Neo4j::Node.create({ property: 'MyProp', secret_val: 123 }, :Post)
post = Post.first
post.secret_val
=> NoMethodError: undefined method `secret_val`...
```

In this case, simply adding the `secret_val` property to your model will make it available through the `secret_val` method. Alternatively, you can also access the properties of the “unwrapped node” through `post._persisted_obj.props`. See the `Neo4j::Core` API for more details about working with `CypherNode` objects.

6.1.1 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
- Fixnum
- BigDecimal
- Date
- Time
- DateTime
- Boolean (`TrueClass` or `FalseClass`)

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

```
class Post
  include Neo4j::ActiveNode

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

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

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

6.1.2 Custom Converters

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

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

    def convert_type
      Range
    end

    def to_db(value)
      value.to_s
    end

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

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

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

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

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

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

`to_ruby` provides logic to transform the DB-provided value back into the class expected by code using the property. It should 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.

Querying

7.1 Simple Query Methods

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

7.1.1 `.find`

Find an object by `id_property` (TODO: LINK TO `id_property` documentation)

7.1.2 `.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')
```

7.2 Scope 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

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

7.2.1 Querying the scope

Similar to ActiveRecord you can perform various operations on a scope 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.

7.2.2 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 return all of the teachers for all of the lessons which the student 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 association 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)`

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:

7.2.3 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.

7.2.4 Parameters

If you need to use a string in where, you should set the parameter manually.

```
Student.all.where("s.age < {age} AND s.name = {name} AND s.home_town = {home_town}")
  .params(age: params[:age], name: params[:name], home_town: params[:home_town])
  .pluck(:s)
```

7.2.5 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: 'male', date: Date.new(2010, 1, 1))
```

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

# As a Range:
```

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

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

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

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

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

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

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

7.3 The Query API

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

```
Neo4j::Session.current.query # Get a new Query object

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

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.

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:

7.4 #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.

7.5 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 student.
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.

7.6 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)

7.7 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

7.8 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.

QueryClauseMethods

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

```
# The current session can be retrieved with `Neo4j::Session.current`
a_session.query
```

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.

8.1 Neo4j::Core::Query

8.1.1 #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').match('o--p')
```

Cypher

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

8.2 #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
```

8.3 #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
```

8.4 #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.name =~ ?', '.*test.*')
```

Cypher

```
WHERE (q.name =~ {question_mark_param1})
```

Parameters: {:question_mark_param1=>".*test.*"}

Ruby

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

Cypher

```
WHERE (q.age IN {question_mark_param1})
```

Parameters: {:question_mark_param1=>[30, 32, 34]}

Ruby

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

Cypher

```
WHERE (q.age IN {question_mark_param1}) AND (q.age != {question_mark_param2})
```

Parameters: {:question_mark_param1=>[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(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 }

8.5 #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_param1})
```

Parameters: { :question_mark_param1=>[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)
```

8.6 #match_nodes

8.6.1 one node object

Ruby

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

Cypher

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

Parameters: { :ID_var=>246 }

8.7 integer

Ruby

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

Cypher

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

Parameters: { :ID_var=>924 }

8.8 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 }

8.9 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 }

8.10 #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
```

8.11 #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
```

8.12 #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: [:age, {name: :desc}])
```

Cypher

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

Ruby

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

Cypher

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

Ruby

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

Cypher

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

Ruby

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

Cypher

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

8.13 #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_5=>5}

Ruby

```
.limit(nil)
```

Cypher

8.14 #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_10=>10 }

Ruby

```
.offset(6)
```

Cypher

```
SKIP {skip_6}
```

Parameters: { :skip_6=>6 }

8.15 #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
```

8.16 #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 }

8.17 #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 }

8.18 #merge

Ruby

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

Cypher

```
MERGE (:Person)
```

Ruby

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

Cypher

```
MERGE (:Person)
```

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 }

8.19 #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
```

8.20 #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} }

8.21 #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

8.22 #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 }

8.23 #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 }

8.24 #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`
```

8.25 #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")
```

8.26 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

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

Cypher

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

Parameters: { :age=>15 }

Configuration

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

9.1 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.

9.2 Other Ruby apps

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

9.2.1 Variables

class_name_property Default: `:_classname`

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

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

See also:

Wrapping

include_root_in_json Default: `true`

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

See also:

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

transform_rel_type Default: `:upcase`

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

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

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

:none Uses the type as specified

module_handling **Default:** :none

Available values: :demodulize, :none, proc

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

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

association_model_namespace **Default:** nil

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

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

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

A Ruby Logger object which is used to log Cypher queries (*info* level is used)

pretty_logged_cypher_queries **Default:** nil

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

record_timestamps **Default:** false

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

timestamp_type **Default:** DateTime

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

wait_for_connection **Default:** false

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

Contributing

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

10.1 The Neo4j.rb Project

We have three main gems: `neo4j`, `neo4j-core`, `neo4j-rake_tasks`.

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

10.2 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!

10.3 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 like tracking our GitHub issues with waffle.io (`neo4j`, `neo4j-core`, `neo4j-rake_tasks`) but just through GitHub also works.

We hang out mostly in our [Gitter.im](https://gitter.im) chat room and are happy to talk or answer questions. We also are often around on the [Neo4j-Users Slack group](#).

10.4 Running Specs

For running the specs, see our [spec/README.md](#)

10.5 Before you submit your pull request

10.5.1 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.

10.5.2 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/`.

11.1 Neo4j

11.1.1 Error

Neo4j.rb Errors Generic Neo4j.rb exception class.

Constants

Files

- lib/neo4j/errors.rb:4

Methods

11.1.2 RecordNotFound

Raised when Neo4j.rb cannot find record by given id.

Constants

Files

- lib/neo4j/errors.rb:8

Methods

#id Returns the value of attribute id

```
def id
  @id
end
```

#initialize

```
def initialize(message = nil, model = nil, primary_key = nil, id = nil)
  @primary_key = primary_key
  @model = model
  @id = id

  super(message)
end
```

#model Returns the value of attribute model

```
def model
  @model
end
```

#primary_key Returns the value of attribute primary_key

```
def primary_key
  @primary_key
end
```

11.1.3 InvalidPropertyOptionsError

Constants

Files

- lib/neo4j/errors.rb:20

Methods

11.1.4 InvalidParameterError

Constants

Files

- lib/neo4j/errors.rb:22

Methods

11.1.5 UnknownTypeConverterError

Constants

Files

- lib/neo4j/errors.rb:24

Methods

11.1.6 DangerousAttributeError

Constants

Files

- `lib/neo4j/errors.rb:26`

Methods

11.1.7 UnknownAttributeError

Constants

Files

- `lib/neo4j/errors.rb:27`

Methods

11.1.8 Config

== Keeps configuration for neo4j

== Configurations keys

Constants

- `DEFAULT_FILE`
- `CLASS_NAME_PROPERTY_KEY`

Files

- `lib/neo4j/config.rb:5`

Methods

`.[]`

```
def [] (key)
  configuration[key.to_s]
end
```

`.[]=` Sets the value of a config entry.

```
def []=(key, val)
  configuration[key.to_s] = val
end
```

.association_model_namespace

```
def association_model_namespace
  Neo4j::Config[:association_model_namespace] || nil
end
```

.association_model_namespace_string

```
def association_model_namespace_string
  namespace = Neo4j::Config[:association_model_namespace]
  return nil if namespace.nil?
  "::#{namespace}"
end
```

.configuration Reads from the default_file if configuration is not set already

```
def configuration
  return @configuration if @configuration

  @configuration = ActiveSupport::HashWithIndifferentAccess.new
  @configuration.merge!(defaults)
  @configuration
end
```

.default_file

```
def default_file
  @default_file ||= DEFAULT_FILE
end
```

.default_file= Sets the location of the configuration YAML file and old deletes configurations.

```
def default_file=(file_path)
  delete_all
  @defaults = nil
  @default_file = File.expand_path(file_path)
end
```

.defaults

```
def defaults
  require 'yaml'
  @defaults ||= ActiveSupport::HashWithIndifferentAccess.new(YAML.load_file(default_file))
end
```

.delete Remove the value of a config entry.

```
def delete(key)
  configuration.delete(key)
end
```

.delete_all Remove all configuration. This can be useful for testing purpose.

```
def delete_all
  @configuration = nil
end
```

.include_root_in_json

```
def include_root_in_json
  # we use ternary because a simple || will always evaluate true
  Neo4j::Config[:include_root_in_json].nil? ? true : Neo4j::Config[:include_root_in_json]
end
```

.module_handling

```
def module_handling
  Neo4j::Config[:module_handling] || :none
end
```

.timestamp_type

```
def timestamp_type
  Neo4j::Config[:timestamp_type] || DateTime
end
```

.to_hash

```
def to_hash
  configuration.to_hash
end
```

.to_yaml

```
def to_yaml
  configuration.to_yaml
end
```

.use Yields the configuration

```
def use
  @configuration ||= ActiveSupport::HashWithIndifferentAccess.new
  yield @configuration
  nil
end
```

11.1.9 Shared

ClassMethods**Constants****Files**

- lib/neo4j/shared.rb:10

Methods**#neo4j_session**

```
def neo4j_session
  if @neo4j_session_name
    Neo4j::Session.named(@neo4j_session_name) ||
      fail("#{self.name} is configured to use a neo4j session named #{@neo4j_session_name}, but")
  else
    Neo4j::Session.current!
  end
end
```

#neo4j_session_name

```
def neo4j_session_name(name)
  ActiveSupport::Deprecation.warn 'neo4j_session_name is deprecated and may be removed from future versions'

  @neo4j_session_name = name
end
```

#neo4j_session_name= Sets the attribute neo4j_session_name

```
def neo4j_session_name=(value)
  @neo4j_session_name = value
end
```

Enum

ConflictingEnumMethodError

Constants

Files

- lib/neo4j/shared/enum.rb:5

Methods

ClassMethods

Constants

- VALID_OPTIONS_FOR_ENUMS
- DEFAULT_OPTIONS_FOR_ENUMS

Files

- lib/neo4j/shared/enum.rb:7

Methods

#enum Similar to ActiveRecord enum, maps an integer value on the database to a set of enum keys.

```
def enum(parameters = {})
  options, parameters = *split_options_and_parameters(parameters)
  parameters.each do |property_name, enum_keys|
    enum_keys = normalize_key_list(enum_keys)
    @neo4j_enum_data ||= {}
    @neo4j_enum_data[property_name] = enum_keys
    define_property(property_name, enum_keys, options)
    define_enum_methods(property_name, enum_keys, options)
  end
end
```

#neo4j_enum_data Returns the value of attribute neo4j_enum_data

```
def neo4j_enum_data
  @neo4j_enum_data
end
```

Constants

Files

- lib/neo4j/shared/enum.rb:2

Methods

Cypher

CreateMethod

Constants

Files

- lib/neo4j/shared/cypher.rb:3

Methods #create_method

```
def create_method
  creates_unique? ? :create_unique : :create
end
```

#creates_unique

```
def creates_unique(option = :none)
  option = :none if option == true
  @creates_unique = option
end
```

#creates_unique?

```
def creates_unique?
  !!@creates_unique
end
```

#creates_unique_option

```
def creates_unique_option
  @creates_unique || :none
end
```

#unique?

```
def creates_unique?
  !!@creates_unique
end
```

RelIdentifiers

Constants

Files

- `lib/neo4j/shared/cypher.rb:23`

Methods

Constants

Files

- `lib/neo4j/shared/cypher.rb:2`

Methods

Marshal

Constants

Files

- `lib/neo4j/shared/marshal.rb:3`

Methods

#marshal_dump

```
def marshal_dump
  marshal_instance_variables.map(&method(:instance_variable_get))
end
```

#marshal_load

```
def marshal_load(array)
  marshal_instance_variables.zip(array).each do |var, value|
    instance_variable_set(var, value)
  end
end
```

Property

UndefinedPropertyError

Constants

Files

- `lib/neo4j/shared/property.rb:9`

Methods**MultiparameterAssignmentError****Constants****Files**

- `lib/neo4j/shared/property.rb:10`

Methods**ClassMethods****Constants****Files**

- `lib/neo4j/shared/property.rb:111`

Methods

#attributes_nil_hash an extra call to a slow dependency method.

```
def attributes_nil_hash
  declared_properties.attributes_nil_hash
end
```

#build_property

```
def build_property(name, options)
  DeclaredProperty.new(name, options).tap do |prop|
    prop.register
    declared_properties.register(prop)
    yield name
    constraint_or_index(name, options)
  end
end
```

#declared_properties

```
def declared_properties
  @_declared_properties ||= DeclaredProperties.new(self)
end
```

#extract_association_attributes!

```
def extract_association_attributes!(props)
  props
end
```

#inherit_property

```
def inherit_property(name, attr_def, options = {})
  build_property(name, options) do |prop_name|
    attributes[prop_name] = attr_def
  end
end
```

#property Defines a property on the class

See active_attr gem for allowed options, e.g which type Notice, in Neo4j you don't have to declare properties before using them, see the neo4j-core api.

```
def property(name, options = {})
  build_property(name, options) do |prop|
    attribute(prop)
  end
end
```

#undef_property

```
def undef_property(name)
  undef_constraint_or_index(name)
  declared_properties.unregister(name)
  attribute_methods(name).each { |method| undef_method(method) }
end
```

Constants

- DATE_KEY_REGEX
- DEPRECATED_OBJECT_METHODS

Files

- lib/neo4j/shared/property.rb:2

Methods

#== Performs equality checking on the result of attributes and its type.

```
def ==(other)
  return false unless other.instance_of? self.class
  attributes == other.attributes
end
```

#[]

```
def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end
```

#[]= Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  end
end
```



```

    else
      fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
    end
  end
end

```

#_persisted_obj Returns the value of attribute `_persisted_obj`

```

def _persisted_obj
  @_persisted_obj
end

```

#assign_attributes Mass update a model's attributes

```

def assign_attributes(new_attributes = nil)
  return unless new_attributes.present?
  new_attributes.each do |name, value|
    writer = :"#{name}="
    send(writer, value) if respond_to?(writer)
  end
end

```

#attribute_before_type_cast Read the raw attribute value

```

def attribute_before_type_cast(name)
  @attributes ||= {}
  @attributes[name.to_s]
end

```

#attributes Returns a Hash of all attributes

```

def attributes
  attributes_map { |name| send name }
end

```

#attributes= Mass update a model's attributes

```

def attributes=(new_attributes)
  assign_attributes(new_attributes)
end

```

#initialize

```

def initialize(attributes = nil)
  attributes = process_attributes(attributes)
  modded_attributes = inject_defaults!(attributes)
  validate_attributes!(modded_attributes)
  writer_method_props = extract_writer_methods!(modded_attributes)
  send_props(writer_method_props)
  @_persisted_obj = nil
end

```

#inject_defaults!

```

def inject_defaults!(starting_props)
  return starting_props if self.class.declared_properties.declared_property_defaults.empty?
  self.class.declared_properties.inject_defaults!(self, starting_props || {})
end

```

#inspect

```
def inspect
  attribute_descriptions = inspect_attributes.map do |key, value|
    "#{Neo4j::ANSI::CYAN}#{key}: #{Neo4j::ANSI::CLEAR}#{value.inspect}"
  end.join(', ')

  separator = ' ' unless attribute_descriptions.empty?
  "#<#{Neo4j::ANSI::YELLOW}#{self.class.name}#{Neo4j::ANSI::CLEAR}#{separator}#{attribute_descriptions}"
end
```

#read_attribute

```
def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end
```

#reload_properties!

```
def reload_properties!(properties)
  @attributes = nil
  convert_and_assign_attributes(properties)
end
```

#send_props

```
def send_props(hash)
  return hash if hash.blank?
  hash.each { |key, value| send("#{key}=", value) }
end
```

#write_attribute Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

Identity

Constants

Files

- lib/neo4j/shared/identity.rb:2

Methods

#==

```
def ==(other)
  other.class == self.class && other.id == id
end
```

#eq!?

```
def ==(other)
  other.class == self.class && other.id == id
end
```

#hash

```
def hash
  id.hash
end
```

#id

```
def id
  id = neo_id
  id.is_a?(Integer) ? id : nil
end
```

#neo_id

```
def neo_id
  _persisted_obj ? _persisted_obj.neo_id : nil
end
```

#to_key Returns an Enumerable of all (primary) key attributes or nil if model.persisted? is false

```
def to_key
  _persisted_obj ? [id] : nil
end
```

Callbacks

nodoc

ClassMethods

Constants

Files

- lib/neo4j/shared/callbacks.rb:6

Methods

Constants

Files

- lib/neo4j/shared/callbacks.rb:3

Methods

#conditional_callback Allows you to perform a callback if a condition is not satisfied.

```
def conditional_callback(kind, guard)
  return yield if guard
  run_callbacks(kind) { yield }
end
```

#destroy

nodoc

```
def destroy #:nodoc:
  tx = Neo4j::Transaction.new
  run_callbacks(:destroy) { super }
rescue
  @_deleted = false
  @attributes = @attributes.dup
  tx.mark_failed
  raise
ensure
  tx.close if tx
end
```

#initialize

```
def initialize(args = nil)
  run_callbacks(:initialize) { super }
end
```

#touch

nodoc

```
def touch #:nodoc:
  run_callbacks(:touch) { super }
end
```

Initialize

Constants

Files

- lib/neo4j/shared/initialize.rb:2

Methods

#wrapper Implements the Neo4j::Node#wrapper and Neo4j::Relationship#wrapper method so that we don't have to care if the node is wrapped or not.

```
def wrapper
  self
end
```

Attributes

Attributes provides a set of class methods for defining an attributes schema and instance methods for reading and writing attributes.

Originally part of ActiveSupport, https://github.com/cgriego/active_attr

ClassMethods

Constants

Files

- `lib/neo4j/shared/attributes.rb:98`

Methods

#attribute Defines an attribute

For each attribute that is defined, a getter and setter will be added as an instance method to the model. An {AttributeDefinition} instance will be added to result of the attributes class method.

```
def attribute(name)
  if dangerous_attribute?(name)
    fail Neo4j::DangerousAttributeError, %(an attribute method named "#{name}" would conflict with
  else
    attribute!(name)
  end
end
```

#attribute_names Returns an Array of attribute names as Strings

```
def attribute_names
  attributes.keys
end
```

#attributes Returns a Hash of AttributeDefinition instances

```
def attributes
  @attributes ||= ActiveSupport::HashWithIndifferentAccess.new
end
```

#dangerous_attribute? Determine if a given attribute name is dangerous

Some attribute names can cause conflicts with existing methods on an object. For example, an attribute named “timeout” would conflict with the timeout method that Ruby’s Timeout library mixes into Object.

```
def dangerous_attribute?(name)
  attribute_methods(name).detect do |method_name|
    !DEPRECATED_OBJECT_METHODS.include?(method_name.to_s) && allocate.respond_to?(method_name, true)
  end unless attribute_names.include? name.to_s
end
```

#inspect Returns the class name plus its attribute names

```
def inspect
  inspected_attributes = attribute_names.sort
  attributes_list = "(#{inspected_attributes.join(', ')}" unless inspected_attributes.empty?
```

```
    "#{name}#{attributes_list}"  
  end
```

Constants

- DEPRECATED_OBJECT_METHODS

Files

- lib/neo4j/shared/attributes.rb:15

Methods

#== Performs equality checking on the result of attributes and its type.

```
def ==(other)  
  return false unless other.instance_of? self.class  
  attributes == other.attributes  
end
```

#[]= Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)  
  if respond_to? "#{name}="   
    send "#{name}=", value  
  else  
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"  
  end  
end
```

#attributes Returns a Hash of all attributes

```
def attributes  
  attributes_map { |name| send name }  
end
```

#write_attribute Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)  
  if respond_to? "#{name}="   
    send "#{name}=", value  
  else  
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"  
  end  
end
```

Typecaster

This module provides a convenient way of registering a custom Typecasting class. Custom Typecasters all follow a simple pattern.

EXAMPLE:

```

class RangeConverter
  class << self
    def primitive_type
      String
    end

    def convert_type
      Range
    end

    def to_db(value)
      value.to_s
    end

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

  include Neo4j::Shared::Typecaster
end

```

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

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

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

to_db provides logic required to transform your value into the class defined by *primitive_type*

to_ruby provides logic to transform the DB-provided value back into the class expected by code using the property. In other words, it should match the *convert_type*.

Note that *alias_method* is used to make *to_ruby* respond to *call*. This is to provide compatibility with ActiveAttr.

Constants

Files

- lib/neo4j/shared/typecaster.rb:47

Methods

.included

```

def self.included(other)
  Neo4j::Shared::TypeConverters.register_converter(other)
end

```

Persistence

Constants

Files

- lib/neo4j/shared/persistence.rb:2

Methods

#apply_default_values

```
def apply_default_values
  return if self.class.declared_property_defaults.empty?
  self.class.declared_property_defaults.each_pair do |key, value|
    self.send("#{key}=", value) if self.send(key).nil?
  end
end
```

#cache_key

```
def cache_key
  if self.new_record?
    "#{model_cache_key}/new"
  elsif self.respond_to?(:updated_at) && !self.updated_at.blank?
    "#{model_cache_key}/#{neo_id}-#{self.updated_at.utc.to_s(:number)}"
  else
    "#{model_cache_key}/#{neo_id}"
  end
end
```

#concurrent_increment! Increments concurrently a numeric attribute by a certain amount

```
def concurrent_increment!(_attribute, _by = 1)
  fail 'not_implemented'
end
```

#create_or_update

```
def create_or_update
  # since the same model can be created or updated twice from a relationship we have to have this
  @_create_or_updating = true
  apply_default_values
  result = _persisted_obj ? update_model : create_model
  if result == false
    Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  else
    true
  end
rescue => e
  Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  raise e
ensure
  @_create_or_updating = nil
end
```

#destroy


```
def destroy
  freeze
  _persisted_obj && _persisted_obj.del
  @_deleted = true
end
```

#destroyed? Returns +true+ if the object was destroyed.

```
def destroyed?
  @_deleted
end
```

#exist?

```
def exist?
  _persisted_obj && _persisted_obj.exist?
end
```

#freeze

```
def freeze
  @attributes.freeze
  self
end
```

#frozen?

```
def frozen?
  @attributes.frozen?
end
```

#increment Increments a numeric attribute by a certain amount

```
def increment(attribute, by = 1)
  self[attribute] ||= 0
  self[attribute] += by
  self
end
```

#increment! Convenience method to increment numeric attribute and #save at the same time

```
def increment!(attribute, by = 1)
  increment(attribute, by).update_attribute(attribute, self[attribute])
end
```

#new? Returns +true+ if the record hasn't been saved to Neo4j yet.

```
def new_record?
  !_persisted_obj
end
```

#new_record? Returns +true+ if the record hasn't been saved to Neo4j yet.

```
def new_record?
  !_persisted_obj
end
```

#persisted? Returns +true+ if the record is persisted, i.e. it's not a new record and it was not destroyed

```
def persisted?
  !new_record? && !destroyed?
end
```

#props

```
def props
  attributes.reject { |_, v| v.nil? }.symbolize_keys
end
```

#props_for_create Returns a hash containing: * All properties and values for insertion in the database * A *uuid* (or equivalent) key and value * Timestamps, if the class is set to include them. Note that the UUID is added to the hash but is not set on the node. The timestamps, by comparison, are set on the node prior to addition in this hash.

```
def props_for_create
  inject_timestamps!
  props_with_defaults = inject_defaults!(props)
  converted_props = props_for_db(props_with_defaults)
  return converted_props unless self.class.respond_to?(:default_property_values)
  inject_primary_key!(converted_props)
end
```

#props_for_persistence

```
def props_for_persistence
  _persisted_obj ? props_for_update : props_for_create
end
```

#props_for_update

```
def props_for_update
  update_magic_properties
  changed_props = attributes.select { |k, _| changed_attributes.include?(k) }
  changed_props.symbolize_keys!
  inject_defaults!(changed_props)
  props_for_db(changed_props)
end
```

#reload

```
def reload
  return self if new_record?
  association_proxy_cache.clear if respond_to?(:association_proxy_cache)
  changed_attributes && changed_attributes.clear
  unless reload_from_database
    @_deleted = true
    freeze
  end
  self
end
```

#reload_from_database

```
def reload_from_database
  reloaded = self.class.load_entity(neo_id)
  reloaded ? init_on_reload(reloaded._persisted_obj) : nil
end
```

#touch

```
def touch
  fail 'Cannot touch on a new record object' unless persisted?
  update_attribute!(:updated_at, Time.now) if respond_to?(:updated_at=)
end
```

#update Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end
```

#update! Same as {#update_attributes}, but raises an exception if saving fails.

```
def update!(attributes)
  self.attributes = process_attributes(attributes)
  save!
end
```

#update_attribute Convenience method to set attribute and #save at the same time

```
def update_attribute(attribute, value)
  send("#{attribute}=", value)
  self.save
end
```

#update_attribute! Convenience method to set attribute and #save! at the same time

```
def update_attribute!(attribute, value)
  send("#{attribute}=", value)
  self.save!
end
```

#update_attributes Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end
```

#update_attributes! Same as {#update_attributes}, but raises an exception if saving fails.

```
def update!(attributes)
  self.attributes = process_attributes(attributes)
  save!
end
```

#update_model

```
def update_model
  return if !changed_attributes || changed_attributes.empty?
  _persisted_obj.update_props(props_for_update)
  changed_attributes.clear
end
```

Validations

Constants

Files

- `lib/neo4j/shared/validations.rb:3`

Methods

#read_attribute_for_validation Implements the ActiveRecord::Validation hook method.

```
def read_attribute_for_validation(key)
  respond_to?(key) ? send(key) : self[key]
end
```

#save The validation process on save can be skipped by passing false. The regular Model#save method is replaced with this when the validations module is mixed in, which it is by default.

```
def save(options = {})
  result = perform_validations(options) ? super : false
  if !result
    Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  end
  result
end
```

#valid?

```
def valid?(context = nil)
  context ||= (new_record? ? :create : :update)
  super(context)
  errors.empty?
end
```

FilteredHash

InvalidHashFilterType

Constants

Files

- lib/neo4j/shared/filtered_hash.rb:3

Methods

Constants

- VALID_SYMBOL_INSTRUCTIONS
- VALID_HASH_INSTRUCTIONS
- VALID_INSTRUCTIONS_TYPES

Files

- lib/neo4j/shared/filtered_hash.rb:2

Methods

#base Returns the value of attribute base

```
def base
  @base
end
```

#filtered_base

```
def filtered_base
  case instructions
  when Symbol
    filtered_base_by_symbol
  when Hash
    filtered_base_by_hash
  end
end
```

#initialize

```
def initialize(base, instructions)
  @base = base
  @instructions = instructions
  @instructions_type = instructions.class
  validate_instructions!(instructions)
end
```

#instructions Returns the value of attribute instructions

```
def instructions
  @instructions
end
```

#instructions_type Returns the value of attribute instructions_type

```
def instructions_type
  @instructions_type
end
```

QueryFactory

Acts as a bridge between the node and rel models and Neo4j::Core::Query. If the object is persisted, it returns a query matching; otherwise, it returns a query creating it. This class does not execute queries, so it keeps no record of what identifiers have been set or what has happened in previous factories.

Constants

Files

- lib/neo4j/shared/query_factory.rb:5

Methods

#base_query

```
def base_query
  @base_query ||= Neo4j::Session.current.query
end
```

#base_query=

```
def base_query=(query)
  return if query.blank?
  @base_query = query
end
```

.create

```
def self.create(graph_object, identifier)
  factory_for(graph_object).new(graph_object, identifier)
end
```

.factory_for

```
def self.factory_for(graph_obj)
  case
  when graph_obj.respond_to?(:labels_for_create)
    NodeQueryFactory
  when graph_obj.respond_to?(:rel_type)
    RelQueryFactory
  else
    fail "Unable to find factory for #{graph_obj}"
  end
end
```

#graph_object Returns the value of attribute graph_object

```
def graph_object
  @graph_object
end
```

#identifier Returns the value of attribute identifier

```
def identifier
  @identifier
end
```

#initialize

```
def initialize(graph_object, identifier)
  @graph_object = graph_object
  @identifier = identifier.to_sym
end
```

#query

```
def query
  graph_object.persisted? ? match_query : create_query
end
```

NodeQueryFactory

Constants

Files

- lib/neo4j/shared/query_factory.rb:63

Methods

#base_query

```
def base_query
  @base_query || Neo4j::Session.current.query
end
```

#base_query=

```
def base_query=(query)
  return if query.blank?
  @base_query = query
end
```

.create

```
def self.create(graph_object, identifier)
  factory_for(graph_object).new(graph_object, identifier)
end
```

.factory_for

```
def self.factory_for(graph_obj)
  case
  when graph_obj.respond_to?(:labels_for_create)
    NodeQueryFactory
  when graph_obj.respond_to?(:rel_type)
    RelQueryFactory
  else
    fail "Unable to find factory for #{graph_obj}"
  end
end
```

#graph_object Returns the value of attribute graph_object

```
def graph_object
  @graph_object
end
```

#identifier Returns the value of attribute identifier

```
def identifier
  @identifier
end
```

#initialize

```
def initialize(graph_object, identifier)
  @graph_object = graph_object
end
```

```
@identifier = identifier.to_sym
end
```

#query

```
def query
  graph_object.persisted? ? match_query : create_query
end
```

RelQueryFactory

Constants

Files

- lib/neo4j/shared/query_factory.rb:76

Methods

#base_query

```
def base_query
  @base_query || Neo4j::Session.current.query
end
```

#base_query=

```
def base_query=(query)
  return if query.blank?
  @base_query = query
end
```

.create

```
def self.create(graph_object, identifier)
  factory_for(graph_object).new(graph_object, identifier)
end
```

.factory_for

```
def self.factory_for(graph_obj)
  case
  when graph_obj.respond_to?(:labels_for_create)
    NodeQueryFactory
  when graph_obj.respond_to?(:rel_type)
    RelQueryFactory
  else
    fail "Unable to find factory for #{graph_obj}"
  end
end
```

#graph_object Returns the value of attribute graph_object

```
def graph_object
  @graph_object
end
```


#identifier Returns the value of attribute identifier

```
def identifier
  @identifier
end
```

#initialize

```
def initialize(graph_object, identifier)
  @graph_object = graph_object
  @identifier = identifier.to_sym
end
```

#query

```
def query
  graph_object.persisted? ? match_query : create_query
end
```

MassAssignment

MassAssignment allows you to bulk set and update attributes

Including MassAssignment into your model gives it a set of mass assignment methods, similar to those found in ActiveRecord.

Originally part of ActiveAttr, https://github.com/cgriego/active_attr

Constants

Files

- lib/neo4j/shared/mass_assignment.rb:13

Methods

#assign_attributes Mass update a model's attributes

```
def assign_attributes(new_attributes = nil)
  return unless new_attributes.present?
  new_attributes.each do |name, value|
    writer = :"#{name}="
    send(writer, value) if respond_to?(writer)
  end
end
```

#attributes= Mass update a model's attributes

```
def attributes=(new_attributes)
  assign_attributes(new_attributes)
end
```

#initialize Initialize a model with a set of attributes

```
def initialize(attributes = nil)
  assign_attributes(attributes)
  super()
end
```

Boolean

Constants

Files

- lib/neo4j/shared/type_converters.rb:7

Methods

TypeConverters

Boolean

Constants

Files

- lib/neo4j/shared/type_converters.rb:12

Methods

BaseConverter

Constants

Files

- lib/neo4j/shared/type_converters.rb:14

Methods .converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

IntegerConverter

Constants

Files

- lib/neo4j/shared/type_converters.rb:22

Methods .convert_type

```
def convert_type
  Integer
end
```

.converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

.db_type

```
def db_type
  Integer
end
```

.to_db

```
def to_db(value)
  value.to_i
end
```

.to_ruby

```
def to_db(value)
  value.to_i
end
```

FloatConverter**Constants****Files**

- lib/neo4j/shared/type_converters.rb:40

Methods .convert_type

```
def convert_type
  Float
end
```

.converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

.db_type

```
def db_type
  Float
end
```

.to_db

```
def to_db(value)
  value.to_f
end
```

.to_ruby

```
def to_db(value)
  value.to_f
end
```

BigDecimalConverter

Constants

Files

- lib/neo4j/shared/type_converters.rb:57

Methods .convert_type

```
def convert_type
  BigDecimal
end
```

.converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

.db_type

```
def db_type
  BigDecimal
end
```

.to_db

```
def to_db(value)
  case value
  when Rational
    value.to_f.to_d
  when respond_to?(:to_d)
    value.to_d
  else
    BigDecimal.new(value.to_s)
  end
end
```

.to_ruby

```
def to_db(value)
  case value
  when Rational
    value.to_f.to_d
  when respond_to?(:to_d)
    value.to_d
```

```
    else
      BigDecimal.new(value.to_s)
    end
  end
end
```

StringConverter

Constants

Files

- lib/neo4j/shared/type_converters.rb:81

Methods .convert_type

```
def convert_type
  String
end
```

.converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

.db_type

```
def db_type
  String
end
```

.to_db

```
def to_db(value)
  value.to_s
end
```

.to_ruby

```
def to_db(value)
  value.to_s
end
```

BooleanConverter

Constants

- FALSE_VALUES

Files

- lib/neo4j/shared/type_converters.rb:98

Methods .convert_type

```
def db_type
  Neo4j::Shared::Boolean
end
```

.converted?

```
def converted?(value)
  converted_values.include?(value)
end
```

.converted_values

```
def converted_values
  [true, false]
end
```

.db_type

```
def db_type
  Neo4j::Shared::Boolean
end
```

.to_db

```
def to_db(value)
  return false if FALSE_VALUES.include?(value)
  case value
  when TrueClass, FalseClass
    value
  when Numeric, /^\-?[0-9]/
    !value.to_f.zero?
  else
    value.present?
  end
end
```

.to_ruby

```
def to_db(value)
  return false if FALSE_VALUES.include?(value)
  case value
  when TrueClass, FalseClass
    value
  when Numeric, /^\-?[0-9]/
    !value.to_f.zero?
  else
    value.present?
  end
end
```

DateConverter

Converts Date objects to Java long types. Must be timezone UTC.

Constants

Files

- lib/neo4j/shared/type_converters.rb:133

Methods `.convert_type`

```
def convert_type
  Date
end
```

`.converted?`

```
def converted?(value)
  value.is_a?(db_type)
end
```

`.db_type`

```
def db_type
  Integer
end
```

`.to_db`

```
def to_db(value)
  Time.utc(value.year, value.month, value.day).to_i
end
```

`.to_ruby`

```
def to_ruby(value)
  value.respond_to?(:to_date) ? value.to_date : Time.at(value).utc.to_date
end
```

DateTimeConverter

Converts DateTime objects to and from Java long types. Must be timezone UTC.

Constants

- DATETIME_FORMAT

Files

- lib/neo4j/shared/type_converters.rb:154

Methods `.convert_type`

```
def convert_type
  DateTime
end
```

`.converted?`

```
def converted?(value)
  value.is_a?(db_type)
end
```

.db_type

```
def db_type
  Integer
end
```

.to_db Converts the given DateTime (UTC) value to an Integer. DateTime values are automatically converted to UTC.

```
def to_db(value)
  value = value.new_offset(0) if value.respond_to?(:new_offset)

  args = [value.year, value.month, value.day]
  args += (value.class == Date ? [0, 0, 0] : [value.hour, value.min, value.sec])

  Time.utc(*args).to_i
end
```

.to_ruby

```
def to_ruby(value)
  return value if value.is_a?(DateTime)
  t = case value
      when Time
        return value.to_datetime.utc
      when Integer
        Time.at(value).utc
      when String
        DateTime.strptime(value, DATETIME_FORMAT)
      else
        fail ArgumentError, "Invalid value type for DateType property: #{value.inspect}"
      end

  DateTime.civil(t.year, t.month, t.day, t.hour, t.min, t.sec)
end
```

TimeConverter**Constants****Files**

- lib/neo4j/shared/type_converters.rb:194

Methods .convert_type

```
def convert_type
  Time
end
```

.converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

.db_type


```
def db_type
  Integer
end
```

.primitive_type ActiveAttr, which assists with property management, does not recognize Time as a valid type. We tell it to interpret it as Integer, as it will be when saved to the database.

```
def primitive_type
  Integer
end
```

.to_db Converts the given DateTime (UTC) value to an Integer. Only utc times are supported !

```
def to_db(value)
  if value.class == Date
    Time.utc(value.year, value.month, value.day, 0, 0, 0).to_i
  else
    value.utc.to_i
  end
end
```

.to_ruby

```
def to_ruby(value)
  Time.at(value).utc
end
```

YAMLConverter

Converts hash to/from YAML

Constants

Files

- lib/neo4j/shared/type_converters.rb:227

Methods .convert_type

```
def convert_type
  Hash
end
```

.converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

.db_type

```
def db_type
  String
end
```

.to_db

```
def to_db(value)
  Psych.dump(value)
end
```

.to_ruby

```
def to_ruby(value)
  Psych.load(value)
end
```

JSONConverter

Converts hash to/from JSON

Constants

Files

- `lib/neo4j/shared/type_converters.rb:248`

Methods .convert_type

```
def convert_type
  JSON
end
```

.converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

.db_type

```
def db_type
  String
end
```

.to_db

```
def to_db(value)
  value.to_json
end
```

.to_ruby

```
def to_ruby(value)
  JSON.parse(value, quirks_mode: true)
end
```

EnumConverter

Constants

Files

- lib/neo4j/shared/type_converters.rb:268

Methods #call

```
def to_ruby(value)
  @enum_keys.key(value) unless value.nil?
end
```

#convert_type

```
def convert_type
  Symbol
end
```

#converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

#db_type

```
def db_type
  Integer
end
```

#initialize

```
def initialize(enum_keys)
  @enum_keys = enum_keys
end
```

#to_db

```
def to_db(value)
  @enum_keys[value.to_s.to_sym] || 0
end
```

#to_ruby

```
def to_ruby(value)
  @enum_keys.key(value) unless value.nil?
end
```

ObjectConverter

Constants

Files

- lib/neo4j/shared/type_converters.rb:296

Methods .convert_type

```
def convert_type
  Object
end
```

.converted?

```
def converted?(value)
  value.is_a?(db_type)
end
```

.to_ruby

```
def to_ruby(value)
  value
end
```

Constants

- CONVERTERS

Files

- lib/neo4j/shared/type_converters.rb:9

Methods

#convert_properties_to Modifies a hash's values to be of types acceptable to Neo4j or matching what the user defined using *type* in property definitions.

```
def convert_properties_to(obj, medium, properties)
  direction = medium == :ruby ? :to_ruby : :to_db
  properties.each_pair do |key, value|
    next if skip_conversion?(obj, key, value)
    properties[key] = convert_property(key, value, direction)
  end
end
```

#convert_property Converts a single property from its current format to its db- or Ruby-expected output type.

```
def convert_property(key, value, direction)
  converted_property(primitive_type(key.to_sym), value, direction)
end
```

.converter_for

```
def converter_for(type)
  type.respond_to?(:db_type) ? type : CONVERTERS[type]
end
```

.formatted_for_db? Attempts to determine whether conversion should be skipped because the object is already of the anticipated output type.

```
def formatted_for_db?(found_converter, value)
  return false unless found_converter.respond_to?(:db_type)
  found_converter.respond_to?(:converted) ? found_converter.converted?(value) : value.is_a?(found_converter.db_type)
end
```

.included

```
def included(_)
  Neo4j::Shared::TypeConverters.constants.each do |constant_name|
    constant = Neo4j::Shared::TypeConverters.const_get(constant_name)
    register_converter(constant) if constant.respond_to?(:convert_type)
  end
end
```

.register_converter

```
def register_converter(converter)
  CONVERTERS[converter.convert_type] = converter
end
```

.to_other

```
def to_other(direction, value, type)
  fail "Unknown direction given: #{direction}" unless direction == :to_ruby || direction == :to_db
  found_converter = converter_for(type)
  return value unless found_converter
  return value if direction == :to_db && formatted_for_db?(found_converter, value)
  found_converter.send(direction, value)
end
```

.typecast_attribute

```
def typecast_attribute(typecaster, value)
  fail ArgumentError, "A typecaster must be given, #{typecaster} is invalid" unless typecaster.respond_to?(:to_ruby)
  return value if value.nil?
  typecaster.to_ruby(value)
end
```

#typecast_attribute

```
def typecast_attribute(typecaster, value)
  Neo4j::Shared::TypeConverters.typecast_attribute(typecaster, value)
end
```

#typecaster_for

```
def typecaster_for(value)
  Neo4j::Shared::TypeConverters.typecaster_for(value)
end
```

.typecaster_for

```
def typecaster_for(primitive_type)
  return nil if primitive_type.nil?
  CONVERTERS[primitive_type]
end
```

DeclaredProperty

Contains methods related to the management

IllegalPropertyError**Constants**

Files

- `lib/neo4j/shared/declared_property.rb:6`

Methods

Index

None of these methods interact with the database. They only keep track of property settings in models. It could (should?) handle the actual indexing/constraining, but that's TBD.

Constants

Files

- `lib/neo4j/shared/declared_property/index.rb:5`

Methods `#constraint!`

```
def constraint!(type = :unique)
  fail Neo4j::InvalidPropertyOptionsError, "Unable to set constraint on indexed property #{name}"
  options[:constraint] = type
end
```

`#constraint?`

```
def constraint?(type = :unique)
  options.key?(:constraint) && options[:constraint] == type
end
```

`#index!`

```
def index!(type = :exact)
  fail Neo4j::InvalidPropertyOptionsError, "Unable to set index on constrained property #{name}"
  options[:index] = type
end
```

`#index?`

```
def index?(type = :exact)
  options.key?(:index) && options[:index] == type
end
```

`#index_or_constraint?`

```
def index_or_constraint?
  index?(:exact) || constraint?(:unique)
end
```

`#unconstraint!`

```
def unconstraint!(type = :unique)
  options.delete(:constraint) if constraint?(type)
end
```

`#unindex!`

```
def unindex!(type = :exact)
  options.delete(:index) if index?(type)
end
```

Constants

- `ILLEGAL_PROPS`

Files

- `lib/neo4j/shared/declared_property.rb:3`
- `lib/neo4j/shared/declared_property/index.rb:2`

Methods

`#<=>` Compare attribute definitions

```
def <=>(other)
  return nil unless other.instance_of? self.class
  return nil if name == other.name && options != other.options
  self.to_s <=> other.to_s
end
```

`#[]`

```
def [](key)
  respond_to?(key) ? public_send(key) : nil
end
```

`#constraint!`

```
def constraint!(type = :unique)
  fail Neo4j::InvalidPropertyOptionsError, "Unable to set constraint on indexed property #{name}"
  options[:constraint] = type
end
```

`#constraint?`

```
def constraint?(type = :unique)
  options.key?(:constraint) && options[:constraint] == type
end
```

`#default`

```
def default_value
  options[:default]
end
```

`#default_value`

```
def default_value
  options[:default]
end
```

`#fail_invalid_options!`

```
def fail_invalid_options!  
  case  
  when index?(:exact) && constraint?(:unique)  
    fail Neo4j::InvalidPropertyOptionsError,  
      "#Uniqueness constraints also provide exact indexes, cannot set both options on property"  
  end  
end
```

#index!

```
def index!(type = :exact)  
  fail Neo4j::InvalidPropertyOptionsError, "Unable to set index on constrained property #{name}"  
  options[:index] = type  
end
```

#index?

```
def index?(type = :exact)  
  options.key?(:index) && options[:index] == type  
end
```

#index_or_constraint?

```
def index_or_constraint?  
  index?(:exact) || constraint?(:unique)  
end
```

#initialize

```
def initialize(name, options = {})  
  fail IllegalPropertyError, "#{name} is an illegal property" if ILLEGAL_PROPS.include?(name.to_s)  
  fail TypeError, "can't convert #{name.class} into Symbol" unless name.respond_to?(:to_sym)  
  @name = @name_sym = name.to_sym  
  @name_string = name.to_s  
  @options = options  
  fail_invalid_options!  
end
```

#inspect

```
def inspect  
  options_description = options.map { |key, value| "#{key.inspect} => #{value.inspect}" }.sort.  
  inspected_options = ", #{options_description}" unless options_description.empty?  
  "attribute :#{name}#{inspected_options}"  
end
```

#magic_typecaster Returns the value of attribute magic_typecaster

```
def magic_typecaster  
  @magic_typecaster  
end
```

#name Returns the value of attribute name

```
def name  
  @name  
end
```

#name_string Returns the value of attribute name_string


```
def name_string
  @name_string
end
```

#name_sym Returns the value of attribute name_sym

```
def name_sym
  @name_sym
end
```

#options Returns the value of attribute options

```
def options
  @options
end
```

#register

```
def register
  register_magic_properties
end
```

#to_s

```
def to_s
  name.to_s
end
```

#to_sym

```
def to_sym
  name
end
```

#type

```
def type
  options[:type]
end
```

#typecaster

```
def typecaster
  options[:typecaster]
end
```

#unconstraint!

```
def unconstraint!(type = :unique)
  options.delete(:constraint) if constraint?(type)
end
```

#unindex!

```
def unindex!(type = :exact)
  options.delete(:index) if index?(type)
end
```

RelTypeConverters

This module controls changes to relationship type based on `Neo4j::Config.transform_rel_type`. It's used whenever a rel type is automatically determined based on ActiveRecord model name or association type.

Constants

Files

- `lib/neo4j/shared/rel_type_converters.rb:5`

Methods

`#decorated_rel_type`

```
def decorated_rel_type(type)
  @decorated_rel_type ||= Neo4j::Shared::RelTypeConverters.decorated_rel_type(type)
end
```

`.decorated_rel_type`

```
def decorated_rel_type(type)
  type = type.to_s
  decorated_type = case rel_transformer
                   when :upcase
                     type.underscore.upcase
                   when :downcase
                     type.underscore.downcase
                   when :legacy
                     "#{type.underscore.downcase}"
                   when :none
                     type
                   else
                     type.underscore.upcase
                   end
  decorated_type.tap { |s| s.gsub!('/', '::') if type.include?('::') }
end
```

.rel_transformer Determines how relationship types should look when inferred based on association or ActiveRecord model name. With the exception of `:none`, all options will call `underscore`, so `ThisClass` becomes `this_class`, with capitalization determined by the specific option passed. Valid options: `* :upcase` - `this_class`, `ThisClass`, `this_cLaSs` (if you don't like yourself) becomes `THIS_CLASS` `* :downcase` - same as above, only... downcased. `* :legacy` - downcases and prepends #, so `ThisClass` becomes `#this_class` `* :none` - uses the string version of whatever is passed with no modifications

```
def rel_transformer
  @rel_transformer ||= Neo4j::Config[:transform_rel_type].nil? ? :upcase : Neo4j::Config[:transform_rel_type]
end
```

DeclaredProperties

The `DeclaredPropertyManager` holds details about objects created as a result of calling the `#property` class method on a class that includes `Neo4j::ActiveNode` or `Neo4j::ActiveRel`. There are many options that are referenced frequently,

particularly during load and save, so this provides easy access and a way of separating behavior from the general Active{obj} modules.

See Neo4j::Shared::DeclaredProperty for definitions of the property objects themselves.

Constants

- EXCLUDED_TYPES
- CONVERTERS

Files

- lib/neo4j/shared/declared_properties.rb:8

Methods

#[]

```
def [] (key)
  registered_properties[key.to_sym]
end
```

#attributes_nil_hash During object wrap, a hash is needed that contains each declared property with a nil value. The active_attr dependency is capable of providing this but it is expensive and calculated on the fly each time it is called. Rather than rely on that, we build this progressively as properties are registered. When the node or rel is loaded, this is used as a template.

```
def attributes_nil_hash
  @attributes_nil_hash ||= {}.tap do |attr_hash|
    registered_properties.each_pair do |k, prop_obj|
      val = prop_obj.default_value
      attr_hash[k.to_s] = val
    end
  end.freeze
end
```

#attributes_string_map During object wrapping, a props hash is built with string keys but Neo4j-core provides symbols. Rather than a *to_s* or *symbolize_keys* during every load, we build a map of symbol-to-string to speed up the process. This increases memory used by the gem but reduces object allocation and GC, so it is faster in practice.

```
def attributes_string_map
  @attributes_string_map ||= {}.tap do |attr_hash|
    attributes_nil_hash.each_key { |k| attr_hash[k.to_sym] = k }
  end.freeze
end
```

#constraint_or_fail!

```
def constraint_or_fail!(key, id_property_name, type = :unique)
  return if key == id_property_name
  fail "Cannot constraint undeclared property #{property}" unless property?(key)
  registered_properties[key].constraint!(type)
end
```

#convert_properties_to Modifies a hash's values to be of types acceptable to Neo4j or matching what the user defined using *type* in property definitions.

```
def convert_properties_to(obj, medium, properties)
  direction = medium == :ruby ? :to_ruby : :to_db
  properties.each_pair do |key, value|
    next if skip_conversion?(obj, key, value)
    properties[key] = convert_property(key, value, direction)
  end
end
```

#convert_property Converts a single property from its current format to its db- or Ruby-expected output type.

```
def convert_property(key, value, direction)
  converted_property(primitive_type(key.to_sym), value, direction)
end
```

#declared_property_defaults The `:default` option in `Neo4j::ActiveNode#property` class method allows for setting a default value instead of `nil` on declared properties. This holds those values.

```
def declared_property_defaults
  @_default_property_values ||= {}
end
```

#index_or_fail!

```
def index_or_fail!(key, id_property_name, type = :exact)
  return if key == id_property_name
  fail "Cannot index undeclared property #{key}" unless property?(key)
  registered_properties[key].index!(type)
end
```

#indexed_properties

```
def indexed_properties
  registered_properties.select { |_, p| p.index_or_constraint? }
end
```

#initialize Each class that includes `Neo4j::ActiveNode` or `Neo4j::ActiveRel` gets one instance of this class.

```
def initialize(klass)
  @klass = klass
end
```

#inject_defaults!

```
def inject_defaults!(object, props)
  declared_property_defaults.each_pair do |k, v|
    props[k.to_sym] = v if object.send(k).nil? && props[k.to_sym].nil?
  end
  props
end
```

#klass Returns the value of attribute `klass`

```
def klass
  @klass
end
```

#magic_typecast_properties

```
def magic_typecast_properties
  @magic_typecast_properties ||= {}
end
```

#magic_typecast_properties_keys

```
def magic_typecast_properties_keys
  @magic_typecast_properties_keys ||= magic_typecast_properties.keys
end
```

#property?

```
def property?(key)
  registered_properties.key?(key.to_sym)
end
```

#register #property on an ActiveNode or ActiveRel class. The DeclaredProperty has specifics about the property, but registration makes the management object aware of it. This is necessary for type conversion, defaults, and inclusion in the nil and string hashes.

```
def register(property)
  @attributes_nil_hash = nil
  @attributes_string_map = nil
  registered_properties[property.name] = property
  register_magic_typecaster(property) if property.magic_typecaster
  declared_property_defaults[property.name] = property.default_value if !property.default_value.
end
```

#registered_properties

```
def registered_properties
  @registered_properties ||= {}
end
```

#serialize

```
def serialize(name, coder = JSON)
  @serialize ||= {}
  @serialize[name] = coder
end
```

#serialized_properties

```
def serialized_properties
  @serialize ||= {}
end
```

#serialized_properties=

```
def serialized_properties=(serialize_hash)
  @serialized_property_keys = nil
  @serialize = serialize_hash.clone
end
```

#serialized_properties_keys

```
def serialized_properties_keys
  @serialized_property_keys ||= serialized_properties.keys
end
```

#string_key but when this happens many times while loading many objects, it results in a surprisingly significant slowdown. The branching logic handles what happens if a property can't be found. The first option attempts to

find it in the existing hash. The second option checks whether the key is the class's id property and, if it is, the string hash is rebuilt with it to prevent future lookups. The third calls *to_s*. This would happen if undeclared properties are found on the object. We could add them to the string map but that would result in unchecked, un-GCed memory consumption. In the event that someone is adding properties dynamically, maybe through user input, this would be bad.

```
def string_key(k)
  attributes_string_map[k] || string_map_id_property(k) || k.to_s
end
```

#typecast_attribute

```
def typecast_attribute(typecaster, value)
  Neo4j::Shared::TypeConverters.typecast_attribute(typecaster, value)
end
```

#typecaster_for

```
def typecaster_for(value)
  Neo4j::Shared::TypeConverters.typecaster_for(value)
end
```

#unregister

```
def unregister(name)
  # might need to be include?(name.to_s)
  fail ArgumentError, "Argument `#{name}` not an attribute" if not registered_properties[name]
  registered_properties.delete(name)
  unregister_magic_typecaster(name)
  unregister_property_default(name)
end
```

#value_for_db

```
def value_for_db(key, value)
  return value unless registered_properties[key]
  convert_property(key, value, :to_db)
end
```

#value_for_ruby

```
def value_for_ruby(key, value)
  return unless registered_properties[key]
  convert_property(key, value, :to_ruby)
end
```

#value_for_where

```
def value_for_where(key, value)
  return value unless prop = registered_properties[key]
  return value_for_db(key, value) if prop.typecaster && prop.typecaster.convert_type == value.class
  EXCLUDED_TYPES.include?(value.class) ? value : value_for_db(key, value)
end
```

SerializedProperties

This module adds the *serialize* class method. It lets you store hashes and arrays in Neo4j properties. Be aware that you won't be able to search within serialized properties and stuff use indexes. If you do a regex search for portion of a string property, the search happens in Cypher and you may take a performance hit.

See `type_converters.rb` for the serialization process.

ClassMethods

Constants

Files

- `lib/neo4j/shared/serialized_properties.rb:19`

Methods `#inherit_serialized_properties`

```
def inherit_serialized_properties(other)
  other.serialized_properties = self.serialized_properties
end
```

`#inherited`

```
def inherited(other)
  inherit_serialized_properties(other) if self.respond_to?(:serialized_properties)
  super
end
```

Constants

Files

- `lib/neo4j/shared/serialized_properties.rb:7`

Methods

`#serializable_hash`

```
def serializable_hash(*args)
  super.merge(id: id)
end
```

`#serialized_properties`

```
def serialized_properties
  self.class.serialized_properties
end
```

TypecastedAttributes

TypecastedAttributes allows types to be declared for your attributes

Types are declared by passing the `:type` option to the attribute class method. After a type is declared, attribute readers will convert any assigned attribute value to the declared type. If the assigned value cannot be cast, `nil` will be returned instead. You can access the original assigned value using the `before_type_cast` methods.

See {Typecasting} for the currently supported types.

Originally part of ActiveAttr, https://github.com/cgriego/active_attr

ClassMethods

Constants

Files

- `lib/neo4j/shared/typecasted_attributes.rb:73`

Methods

#_attribute_type Calculates an attribute type

```
def _attribute_type(attribute_name)
  attributes[attribute_name].type || Object
end
```

#inspect Returns the class name plus its attribute names and types

```
def inspect
  inspected_attributes = attribute_names.sort.map { |name| "#{name}: #{_attribute_type(name)}" }
  attributes_list = "(#{inspected_attributes.join(', ')})" unless inspected_attributes.empty?
  "#{name}#{attributes_list}"
end
```

#typecast_attribute

```
def typecast_attribute(typecaster, value)
  Neo4j::Shared::TypeConverters.typecast_attribute(typecaster, value)
end
```

Constants

- `DEPRECATED_OBJECT_METHODS`

Files

- `lib/neo4j/shared/typecasted_attributes.rb:24`

Methods

#== Performs equality checking on the result of attributes and its type.

```
def ==(other)
  return false unless other.instance_of? self.class
  attributes == other.attributes
end
```

#[]= Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```



```
end
end
```

#attribute_before_type_cast Read the raw attribute value

```
def attribute_before_type_cast(name)
  @attributes ||= {}
  @attributes[name.to_s]
end
```

#attributes Returns a Hash of all attributes

```
def attributes
  attributes_map { |name| send name }
end
```

#write_attribute Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

Constants

Files

- lib/neo4j/shared.rb:2
- lib/neo4j/shared/enum.rb:1
- lib/neo4j/shared/cypher.rb:1
- lib/neo4j/shared/marshal.rb:2
- lib/neo4j/shared/property.rb:1
- lib/neo4j/shared/identity.rb:1
- lib/neo4j/shared/callbacks.rb:2
- lib/neo4j/shared/initialize.rb:1
- lib/neo4j/shared/attributes.rb:1
- lib/neo4j/shared/typecaster.rb:2
- lib/neo4j/shared/persistence.rb:1
- lib/neo4j/shared/validations.rb:2
- lib/neo4j/shared/filtered_hash.rb:1
- lib/neo4j/shared/query_factory.rb:1
- lib/neo4j/shared/mass_assignment.rb:1
- lib/neo4j/shared/type_converters.rb:6
- lib/neo4j/shared/declared_property.rb:1

- lib/neo4j/shared/rel_type_converters.rb:1
- lib/neo4j/shared/declared_properties.rb:1
- lib/neo4j/shared/serialized_properties.rb:1
- lib/neo4j/shared/typecasted_attributes.rb:1
- lib/neo4j/shared/declared_property/index.rb:1

Methods

#declared_properties

```
def declared_properties
  self.class.declared_properties
end
```

11.1.10 Raitie

Constants

Files

- lib/neo4j/raitie.rb:7

Methods

.config_data

```
def config_data
  @config_data ||= if yaml_path
    HashWithIndifferentAccess.new(YAML.load(ERB.new(yaml_path.read).result) [Rai
  else
    {}
  end
end
```

.default_session_path

```
def default_session_path
  ENV['NEO4J_URL'] || ENV['NEO4J_PATH'] ||
  config_data[:url] || config_data[:path] ||
  'http://localhost:7474'
end
```

.default_session_type

```
def default_session_type
  if ENV['NEO4J_TYPE']
    :embedded_db
  else
    config_data[:type] || :server_db
  end.to_sym
end
```

.java_platform?

```
def java_platform?
  RUBY_PLATFORM =~ /java/
end
```

.open_neo4j_session

```
def open_neo4j_session(options, wait_for_connection = false)
  type, name, default, path = options.values_at(:type, :name, :default, :path)

  if !java_platform? && type == :embedded_db
    fail "Tried to start embedded Neo4j db without using JRuby (got #{RUBY_PLATFORM}), please ru
  end

  session = wait_for_value(wait_for_connection) do
    if options.key?(:name)
      Neo4j::Session.open_named(type, name, default, path)
    else
      Neo4j::Session.open(type, path, options[:options])
    end
  end

  start_embedded_session(session) if type == :embedded_db
end
```

#register_neo4j_cypher_logging

```
def register_neo4j_cypher_logging
  return if @neo4j_cypher_logging_registered

  Neo4j::Core::Query.pretty_cypher = Neo4j::Config[:pretty_logged_cypher_queries]

  Neo4j::Server::CypherSession.log_with do |message|
    (Neo4j::Config[:logger] || Rails.logger).debug message
  end

  @neo4j_cypher_logging_registered = true
end
```

.setup_config_defaults!

```
def setup_config_defaults!(cfg)
  cfg.session_type ||= default_session_type
  cfg.session_path ||= default_session_path
  cfg.session_options ||= {}
  cfg.sessions ||= []
end
```

.setup_default_session

```
def setup_default_session(cfg)
  setup_config_defaults!(cfg)

  return if !cfg.sessions.empty?

  cfg.sessions << {type: cfg.session_type, path: cfg.session_path, options: cfg.session_options.
end
```

.start_embedded_session

```
def start_embedded_session(session)
  # See https://github.com/jruby/jruby/wiki/UnlimitedStrengthCrypto
  security_class = java.lang.Class.forName('javax.crypto.JceSecurity')
  restricted_field = security_class.get_declared_field('isRestricted')
  restricted_field.accessible = true
  restricted_field.set nil, false
  session.start
end
```

#wait_for_value

```
def wait_for_value(wait)
  session = nil
  Timeout.timeout(60) do
    until session
      begin
        if session = yield
          puts
          return session
        end
      rescue Faraday::ConnectionFailed => e
        raise e if !wait

        puts '.'
        sleep(1)
      end
    end
  end
end
```

.yaml_path

```
def yaml_path
  @yaml_path ||= %w(config/neo4j.yml config/neo4j.yaml).map do |path|
    Rails.root.join(path)
  end.detect(&:exist?)
end
```

11.1.11 ClassWrapper

Constants

Files

- lib/neo4j/wrapper.rb:2

Methods

11.1.12 Paginated

Constants

Files

- lib/neo4j/paginated.rb:2

Methods

.create_from

```
def self.create_from(source, page, per_page, order = nil)
  target = source.node_var || source.identity
  partial = source.skip((page - 1) * per_page).limit(per_page)
  ordered_partial, ordered_source = if order
    [partial.order_by(order), source.query.with("#{target} as
    else
    [partial, source.count]
  end
  Paginated.new(ordered_partial, ordered_source, page)
end
```

#current_page Returns the value of attribute current_page

```
def current_page
  @current_page
end
```

#initialize

```
def initialize(items, total, current_page)
  @items = items
  @total = total
  @current_page = current_page
end
```

#items Returns the value of attribute items

```
def items
  @items
end
```

#total Returns the value of attribute total

```
def total
  @total
end
```

11.1.13 Migration

AddIdProperty

Constants

Files

- lib/neo4j/migration.rb:25

Methods

#default_path

```
def default_path
  Rails.root if defined? Rails
end
```

#initialize

```
def initialize(path = default_path)
  @models_filename = File.join(joined_path(path), 'add_id_property.yml')
end
```

#joined_path

```
def joined_path(path)
  File.join(path.to_s, 'db', 'neo4j-migrate')
end
```

#migrate

```
def migrate
  models = ActiveSupport::HashWithIndifferentAccess.new(YAML.load_file(models_filename)[:models])
  output 'This task will add an ID Property every node in the given file.'
  output 'It may take a significant amount of time, please be patient.'
  models.each do |model|
    output
    output
    output "Adding IDs to #{model}"
    add_ids_to model.constantize
  end
end
```

#models_filename Returns the value of attribute models_filename

```
def models_filename
  @models_filename
end
```

#output

```
def output(string = '')
  puts string unless !!ENV['silenced']
end
```

#print_output

```
def print_output(string)
  print string unless !!ENV['silenced']
end
```

#setup

```
def setup
  FileUtils.mkdir_p('db/neo4j-migrate')

  return if File.file?(models_filename)

  File.open(models_filename, 'w') do |file|
    message = <<MESSAGE
    # Provide models to which IDs should be added.
    # # It will only modify nodes that do not have IDs. There is no danger of overwriting data.
    # # models: [Student,Lesson,Teacher,Exam]\nmodels: []
    MESSAGE
  end
end
```

```

      file.write(message)
    end
  end
end

```

Constants

Files

- lib/neo4j/migration.rb:4

Methods

#default_path

```

def default_path
  Rails.root if defined? Rails
end

```

#joined_path

```

def joined_path(path)
  File.join(path.to_s, 'db', 'neo4j-migrate')
end

```

#migrate

```

def migrate
  fail 'not implemented'
end

```

#output

```

def output(string = '')
  puts string unless !!ENV['silenced']
end

```

#print_output

```

def print_output(string)
  print string unless !!ENV['silenced']
end

```

11.1.14 ActiveRel

Makes Neo4j Relationships more or less act like ActiveRecord objects. See documentation at <https://github.com/neo4jrb/neo4j/wiki/Neo4j%3A%3AActiveRel>

FrozenRelError

Constants

Files

- lib/neo4j/active_rel.rb:22

Methods

ClassMethods

Constants

Files

- `lib/neo4j/active_rel.rb:66`

Methods

Types

provides mapping of type to model name

ClassMethods

Constants

Files

- `lib/neo4j/active_rel/types.rb:24`

Methods

#_type When called without arguments, it will return the current setting or supply a default. When called with arguments, it will change the current setting. should be deprecated

```
def type(given_type = nil, auto = false)
  case
  when !given_type && rel_type?
    @rel_type
  when given_type
    assign_type!(given_type, auto)
  else
    assign_type!(namespaced_model_name, true)
  end
end
```

#_wrapped_classes

```
def _wrapped_classes
  WRAPPED_CLASSES
end
```

#add_wrapped_class

```
def add_wrapped_class(type)
  # WRAPPED_CLASSES[type.to_sym.downcase] = self.name
  _wrapped_classes[type.to_sym] = self.name
end
```

#decorated_rel_type


```
def decorated_rel_type(type)
  @decorated_rel_type ||= Neo4j::Shared::RelTypeConverters.decorated_rel_type(type)
end
```

#inherited

```
def inherited(subclass)
  subclass.type subclass.namespaced_model_name, true
end
```

#namespaced_model_name

```
def namespaced_model_name
  case Neo4j::Config[:module_handling]
  when :demodulize
    self.name.demodulize
  when Proc
    Neo4j::Config[:module_handling].call(self.name)
  else
    self.name
  end
end
```

#rel_type When called without arguments, it will return the current setting or supply a default. When called with arguments, it will change the current setting.

```
def type(given_type = nil, auto = false)
  case
  when !given_type && rel_type?
    @rel_type
  when given_type
    assign_type!(given_type, auto)
  else
    assign_type!(namespaced_model_name, true)
  end
end
```

#rel_type?

```
def rel_type?
  !!@rel_type
end
```

#type When called without arguments, it will return the current setting or supply a default. When called with arguments, it will change the current setting.

```
def type(given_type = nil, auto = false)
  case
  when !given_type && rel_type?
    @rel_type
  when given_type
    assign_type!(given_type, auto)
  else
    assign_type!(namespaced_model_name, true)
  end
end
```

Constants

- WRAPPED_CLASSES

Files

- lib/neo4j/active_rel/types.rb:4

Methods

Query

ClassMethods

Constants

Files

- lib/neo4j/active_rel/query.rb:5

Methods

#all Performs a basic match on the relationship, returning all results. This is not executed lazily, it will immediately return matching objects.

```
def all
  all_query.pluck(:r1)
end
```

#find Returns the object with the specified neo4j id.

```
def find(id, session = self.neo4j_session)
  fail "Unknown argument #{id.class} in find method (expected String or Integer)" if !(id.is_a?(String || Integer))
  find_by_id(id, session)
end
```

#find_by_id Loads the relationship using its neo_id.

```
def find_by_id(key, session = Neo4j::Session.current!)
  session.query.match('()-[r]-()').where('ID(r)' => key.to_i).limit(1).return(:r).first.r
end
```

#first

```
def first
  all_query.limit(1).order('ID(r1)').pluck(:r1).first
end
```

#last

```
def last
  all_query.limit(1).order('ID(r1) DESC').pluck(:r1).first
end
```

#where Performs a very basic match on the relationship. This is not executed lazily, it will immediately return matching objects. To use a string, prefix the property with “r1”

```
def where(args = {})
  where_query.where(where_string(args)).pluck(:r1)
end
```

Constants

Files

- lib/neo4j/active_rel/query.rb:2

Methods

Property

ClassMethods

Constants

Files

- lib/neo4j/active_rel/property.rb:39

Methods #create_method

```
def create_method
  creates_unique? ? :create_unique : :create
end
```

#creates_unique

```
def creates_unique(option = :none)
  option = :none if option == true
  @creates_unique = option
end
```

#creates_unique?

```
def creates_unique?
  !!@creates_unique
end
```

#creates_unique_option

```
def creates_unique_option
  @creates_unique || :none
end
```

#end_class

```
alias_method :end_class, :to_class
```

#extract_association_attributes! Extracts keys from attributes hash which are relationships of the model TODO: Validate separately that relationships are getting the right values? Perhaps also store the values and persist relationships on save?

```
def extract_association_attributes!(attributes)
  return if attributes.blank?
  {}.tap do |relationship_props|
    attributes.each_key do |key|
      relationship_props[key] = attributes.delete(key) if [:from_node, :to_node].include?(key)
    end
  end
end
```

#id_property_name

```
def id_property_name
  false
end
```

#load_entity

```
def load_entity(id)
  Neo4j::Node.load(id)
end
```

#start_class

```
alias_method :start_class, :from_class
```

#unique?

```
def creates_unique?
  !!@creates_unique
end
```

#valid_class_argument?

```
def valid_class_argument?(class_argument)
  [String, Symbol, FalseClass].include?(class_argument.class) ||
    (class_argument.is_a?(Array) && class_argument.all? { |c| [String, Symbol].include?(c.class) })
end
```

Constants

- DATE_KEY_REGEX
- DEPRECATED_OBJECT_METHODS

Files

- lib/neo4j/active_rel/property.rb:4

Methods

#== Performs equality checking on the result of attributes and its type.

```
def ==(other)
  return false unless other.instance_of? self.class
  attributes == other.attributes
end
```

#[]

```
def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end
```

#[]= Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

#_persisted_obj Returns the value of attribute _persisted_obj

```
def _persisted_obj
  @_persisted_obj
end
```

#assign_attributes Mass update a model's attributes

```
def assign_attributes(new_attributes = nil)
  return unless new_attributes.present?
  new_attributes.each do |name, value|
    writer = "#{name}="
    send(writer, value) if respond_to?(writer)
  end
end
```

#attribute_before_type_cast Read the raw attribute value

```
def attribute_before_type_cast(name)
  @attributes ||= {}
  @attributes[name.to_s]
end
```

#attributes Returns a Hash of all attributes

```
def attributes
  attributes_map { |name| send name }
end
```

#attributes= Mass update a model's attributes

```
def attributes=(new_attributes)
  assign_attributes(new_attributes)
end
```

#creates_unique_option

```
def creates_unique_option
  self.class.create_unique_option
end
```

#end_node

```
alias_method :end_node, :to_node
```

#from_node_neo_id

```
alias_method :from_node_neo_id, :start_node_neo_id
```

#initialize

```
def initialize(attributes = nil)
  super(attributes)
end
```

#inject_defaults!

```
def inject_defaults!(starting_props)
  return starting_props if self.class.declared_properties.declared_property_defaults.empty?
  self.class.declared_properties.inject_defaults!(self, starting_props || {})
end
```

#inspect

```
def inspect
  attribute_descriptions = inspect_attributes.map do |key, value|
    "#{Neo4j::ANSI::CYAN}#{key}: #{Neo4j::ANSI::CLEAR}#{value.inspect}"
  end.join(', ')

  separator = ' ' unless attribute_descriptions.empty?
  "#<#{Neo4j::ANSI::YELLOW}#{self.class.name}#{Neo4j::ANSI::CLEAR}#{separator}#{attribute_descriptions}"
end
```

#read_attribute

```
def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end
```

#rel_type

```
def type
  self.class.type
end
```

#reload_properties!

```
def reload_properties!(properties)
  @attributes = nil
  convert_and_assign_attributes(properties)
end
```

#send_props

```
def send_props(hash)
  return hash if hash.blank?
  hash.each { |key, value| send("#{key}=", value) }
end
```

#start_node

```
alias_method :start_node, :from_node
```

#to_node_neo_id

```
alias_method :to_node_neo_id, :end_node_neo_id
```

#type

```
def type
  self.class.type
end
```

#write_attribute Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

Callbacks**nodoc****Constants****Files**

- lib/neo4j/active_rel/callbacks.rb:3

Methods**#conditional_callback** Allows you to perform a callback if a condition is not satisfied.

```
def conditional_callback(kind, guard)
  return yield if guard
  run_callbacks(kind) { yield }
end
```

#destroy**nodoc**

```
def destroy #:nodoc:
  tx = Neo4j::Transaction.new
  run_callbacks(:destroy) { super }
rescue
  @_deleted = false
  @attributes = @attributes.dup
  tx.mark_failed
  raise
ensure
  tx.close if tx
end
```

#initialize

```
def initialize(args = nil)
  run_callbacks(:initialize) { super }
end
```

#save

```
def save(*args)
  unless _persisted_obj || (from_node.respond_to?(:neo_id) && to_node.respond_to?(:neo_id))
    fail Neo4j::ActiveRel::Persistence::RelInvalidError, 'from_node and to_node must be node obj'
  end
  super(*args)
end
```

#touch

nodoc

```
def touch #:nodoc:
  run_callbacks(:touch) { super }
end
```

Initialize

Constants

Files

- lib/neo4j/active_rel/initialize.rb:2

Methods

#init_on_load called when loading the rel from the database

```
def init_on_load(persisted_rel, from_node_id, to_node_id, type)
  @rel_type = type
  @_persisted_obj = persisted_rel
  changed_attributes && changed_attributes.clear
  @attributes = convert_and_assign_attributes(persisted_rel.props)
  load_nodes(from_node_id, to_node_id)
end
```

#init_on_reload

```
def init_on_reload(unwrapped_reloaded)
  @attributes = nil
  init_on_load(unwrapped_reloaded,
               unwrapped_reloaded._start_node_id,
               unwrapped_reloaded._end_node_id,
               unwrapped_reloaded.rel_type)

  self
end
```

#wrapper Implements the Neo4j::Node#wrapper and Neo4j::Relationship#wrapper method so that we don't have to care if the node is wrapped or not.


```
def wrapper
  self
end
```

Persistence

RelInvalidError

Constants

Files

- lib/neo4j/active_rel/persistence.rb:7

Methods

ModelClassInvalidError

Constants

Files

- lib/neo4j/active_rel/persistence.rb:8

Methods

RelCreateFailedError

Constants

Files

- lib/neo4j/active_rel/persistence.rb:9

Methods

ClassMethods

Constants

Files

- lib/neo4j/active_rel/persistence.rb:55

Methods

#create Creates a new relationship between objects

```
def create(props = {})
  relationship_props = extract_association_attributes!(props) || {}
  new(props).tap do |obj|
    relationship_props.each do |prop, value|
      obj.send("#{prop}=", value)
    end
    obj.save
  end
end
```

#create! Same as #create, but raises an error if there is a problem during save.

```
def create!(*args)
  props = args[0] || {}
  relationship_props = extract_association_attributes!(props) || {}
  new(props).tap do |obj|
    relationship_props.each do |prop, value|
      obj.send("#{prop}=", value)
    end
    obj.save!
  end
end
```

#create_method

```
def create_method
  creates_unique? ? :create_unique : :create
end
```

#load_entity

```
def load_entity(id)
  Neo4j::Relationship.load(id)
end
```

QueryFactory

This class builds and executes a Cypher query, using information from the graph objects to determine whether they need to be created simultaneously. It keeps the rel instance from being responsible for inspecting the nodes or talking with Shared::QueryFactory.

Constants

- NODE_SYMBOLS

Files

- lib/neo4j/active_rel/persistence/query_factory.rb:5

Methods

#build! TODO: This feels like it should also wrap the rel, but that is handled in Neo4j::ActiveRel::Persistence at the moment. Builds and executes the query using the objects giving during init. It holds the process:

- Execute node callbacks if needed
- Create and execute the query
- Mix the query response into the unpersisted objects given during init

```
def build!
  node_before_callbacks! do
    res = query_factory(rel, rel_id, iterative_query).query.unwrapped.return(*unpersisted_return)
    node_symbols.each { |n| wrap!(send(n), res, n) }
    @unwrapped_rel = res.send(rel_id)
  end
end
```

#from_node Returns the value of attribute from_node

```
def from_node
  @from_node
end
```

#initialize

```
def initialize(from_node, to_node, rel)
  @from_node = from_node
  @to_node = to_node
  @rel = rel
end
```

#rel Returns the value of attribute rel

```
def rel
  @rel
end
```

#to_node Returns the value of attribute to_node

```
def to_node
  @to_node
end
```

#unwrapped_rel Returns the value of attribute unwrapped_rel

```
def unwrapped_rel
  @unwrapped_rel
end
```

Constants

Files

- lib/neo4j/active_rel/persistence.rb:2
- lib/neo4j/active_rel/persistence/query_factory.rb:1

Methods

#apply_default_values

```
def apply_default_values
  return if self.class.declared_property_defaults.empty?
  self.class.declared_property_defaults.each_pair do |key, value|
    self.send("#{key}=", value) if self.send(key).nil?
  end
end
```

#cache_key

```
def cache_key
  if self.new_record?
    "#{model_cache_key}/new"
  elsif self.respond_to?(:updated_at) && !self.updated_at.blank?
    "#{model_cache_key}/#{neo_id}-#{self.updated_at.utc.to_s(:number)}"
  else
    "#{model_cache_key}/#{neo_id}"
  end
end
```

#concurrent_increment! Increments concurrently a numeric attribute by a certain amount

```
def concurrent_increment!(attribute, by = 1)
  query_rel = Neo4j::Session.query.match('()-[n]-()').where(n: {neo_id: neo_id})
  increment_by_query! query_rel, attribute, by
end
```

#create_method

```
def create_method
  self.class.create_method
end
```

#create_model

```
def create_model
  validate_node_classes!
  rel = _create_rel
  return self unless rel.respond_to?(:props)
  init_on_load(rel, from_node, to_node, @rel_type)
  true
end
```

#create_or_update

```
def create_or_update
  # since the same model can be created or updated twice from a relationship we have to have thi
  @create_or_updating = true
  apply_default_values
  result = _persisted_obj ? update_model : create_model
  if result == false
    Neo4j::Transaction.current.failure if Neo4j::Transaction.current
    false
  else
    true
  end
rescue => e
  Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  raise e
ensure
```

```

    @_create_or_updating = nil
  end

```

#cypher_identifier

```

def cypher_identifier
  @cypher_identifier || :rel
end

```

#destroy

```

def destroy
  freeze
  _persisted_obj && _persisted_obj.del
  @_deleted = true
end

```

#destroyed? Returns +true+ if the object was destroyed.

```

def destroyed?
  @_deleted
end

```

#exist?

```

def exist?
  _persisted_obj && _persisted_obj.exist?
end

```

#freeze

```

def freeze
  @attributes.freeze
  self
end

```

#from_node_identifier

```

def from_node_identifier
  @from_node_identifier || :from_node
end

```

#from_node_identifier=

```

def from_node_identifier=(id)
  @from_node_identifier = id.to_sym
end

```

#frozen?

```

def frozen?
  @attributes.frozen?
end

```

#increment Increments a numeric attribute by a certain amount

```

def increment(attribute, by = 1)
  self[attribute] ||= 0
  self[attribute] += by
  self
end

```

#increment! Convenience method to increment numeric attribute and #save at the same time

```
def increment!(attribute, by = 1)
  increment(attribute, by).update_attribute(attribute, self[attribute])
end
```

#new? Returns +true+ if the record hasn't been saved to Neo4j yet.

```
def new_record?
  !_persisted_obj
end
```

#new_record? Returns +true+ if the record hasn't been saved to Neo4j yet.

```
def new_record?
  !_persisted_obj
end
```

#persisted? Returns +true+ if the record is persisted, i.e. it's not a new record and it was not destroyed

```
def persisted?
  !new_record? && !destroyed?
end
```

#props

```
def props
  attributes.reject { |_, v| v.nil? }.symbolize_keys
end
```

#props_for_create Returns a hash containing: * All properties and values for insertion in the database * A *uuid* (or equivalent) key and value * Timestamps, if the class is set to include them. Note that the UUID is added to the hash but is not set on the node. The timestamps, by comparison, are set on the node prior to addition in this hash.

```
def props_for_create
  inject_timestamps!
  props_with_defaults = inject_defaults!(props)
  converted_props = props_for_db(props_with_defaults)
  return converted_props unless self.class.respond_to?(:default_property_values)
  inject_primary_key!(converted_props)
end
```

#props_for_persistence

```
def props_for_persistence
  _persisted_obj ? props_for_update : props_for_create
end
```

#props_for_update

```
def props_for_update
  update_magic_properties
  changed_props = attributes.select { |k, _| changed_attributes.include?(k) }
  changed_props.symbolize_keys!
  inject_defaults!(changed_props)
  props_for_db(changed_props)
end
```

#reload

```
def reload
  return self if new_record?
  association_proxy_cache.clear if respond_to?(:association_proxy_cache)
  changed_attributes && changed_attributes.clear
  unless reload_from_database
    @_deleted = true
    freeze
  end
  self
end
```

#reload_from_database

```
def reload_from_database
  reloaded = self.class.load_entity(neo_id)
  reloaded ? init_on_reload(reloaded._persisted_obj) : nil
end
```

#save

```
def save(*)
  create_or_update
end
```

#save!

```
def save!(*args)
  save(*args) or fail(RelInvalidError, inspect) # rubocop:disable Style/AndOr
end
```

#to_node_identifier

```
def to_node_identifier
  @_to_node_identifier || :to_node
end
```

#to_node_identifier=

```
def to_node_identifier=(id)
  @_to_node_identifier = id.to_sym
end
```

#touch

```
def touch
  fail 'Cannot touch on a new record object' unless persisted?
  update_attribute!(:updated_at, Time.now) if respond_to?(:updated_at=)
end
```

#update Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end
```

#update! Same as {#update_attributes}, but raises an exception if saving fails.

```
def update!(attributes)
  self.attributes = process_attributes(attributes)
```

```
    save!  
  end
```

#update_attribute Convenience method to set attribute and #save at the same time

```
def update_attribute(attribute, value)  
  send("#{attribute}=", value)  
  self.save  
end
```

#update_attribute! Convenience method to set attribute and #save! at the same time

```
def update_attribute!(attribute, value)  
  send("#{attribute}=", value)  
  self.save!  
end
```

#update_attributes Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)  
  self.attributes = process_attributes(attributes)  
  save  
end
```

#update_attributes! Same as {#update_attributes}, but raises an exception if saving fails.

```
def update!(attributes)  
  self.attributes = process_attributes(attributes)  
  save!  
end
```

#update_model

```
def update_model  
  return if !changed_attributes || changed_attributes.empty?  
  _persisted_obj.update_props(props_for_update)  
  changed_attributes.clear  
end
```

Validations

Constants

Files

- `lib/neo4j/active_rel/validations.rb:3`

Methods

#read_attribute_for_validation Implements the ActiveRecord::Validation hook method.

```
def read_attribute_for_validation(key)  
  respond_to?(key) ? send(key) : self[key]  
end
```


#save The validation process on save can be skipped by passing false. The regular Model#save method is replaced with this when the validations module is mixed in, which it is by default.

```
def save(options = {})
  result = perform_validations(options) ? super : false
  if !result
    Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  end
  result
end
```

#valid?

```
def valid?(context = nil)
  context ||= (new_record? ? :create : :update)
  super(context)
  errors.empty?
end
```

RelatedNode

A container for ActiveRecord's :inbound and :outbound methods. It provides lazy loading of nodes. It's important (or maybe not really IMPORTANT, but at least worth mentioning) that calling method_missing will result in a query to load the node if the node is not already loaded.

UnsetRelatedNodeError

Constants

Files

- lib/neo4j/active_rel/related_node.rb:6

Methods

Constants

Files

- lib/neo4j/active_rel/related_node.rb:5

Methods

#== Loads the node if needed, then conducts comparison.

```
def ==(other)
  loaded if @node.is_a?(Integer)
  @node == other
end
```

#class

```
def class
  loaded.send(:class)
end
```

#cypher_representation

```
def cypher_representation(clazz)
  case
  when !set?
    "(#{formatted_label_list(clazz)})"
  when set? && !loaded?
    "(Node with neo_id #{@node})"
  else
    node_class = self.class
    id_name = node_class.id_property_name
    labels = ':' + node_class.mapped_label_names.join(':')

    "(#{labels} {#{id_name}: #{@node.id.inspect}})"
  end
end
```

#initialize ActiveRel's related nodes can be initialized with nothing, an integer, or a fully wrapped node.

Initialization with nothing happens when a new, non-persisted ActiveRel object is first initialized.

Initialization with an integer happens when a relationship is loaded from the database. It loads using the ID because that is provided by the Cypher response and does not require an extra query.

```
def initialize(node = nil)
  @node = valid_node_param?(node) ? node : (fail Neo4j::InvalidParameterError, 'RelatedNode must be a Node or Integer')
end
```

#loaded Loads a node from the database or returns the node if already loaded

```
def loaded
  fail UnsetRelatedNodeError, 'Node not set, cannot load' if @node.nil?
  @node = @node.respond_to?(:neo_id) ? @node : Neo4j::Node.load(@node)
end
```

#loaded?

```
def loaded?
  @node.respond_to?(:neo_id)
end
```

#method_missing

```
def method_missing(*args, &block)
  loaded.send(*args, &block)
end
```

#neo_id Returns the neo_id of a given node without loading.

```
def neo_id
  loaded? ? @node.neo_id : @node
end
```

#respond_to_missing?

```
def respond_to_missing?(method_name, include_private = false)
  loaded if @node.is_a?(Numeric)
end
```

```
@node.respond_to?(method_name) ? true : super
end
```

#set?

```
def set?
  !@node.nil?
end
```

Constants

- MARSHAL_INSTANCE_VARIABLES
- WRAPPED_CLASSES
- DATE_KEY_REGEX
- DEPRECATED_OBJECT_METHODS

Files

- lib/neo4j/active_rel.rb:4
- lib/neo4j/active_rel/types.rb:2
- lib/neo4j/active_rel/query.rb:1
- lib/neo4j/active_rel/property.rb:3
- lib/neo4j/active_rel/callbacks.rb:2
- lib/neo4j/active_rel/initialize.rb:1
- lib/neo4j/active_rel/persistence.rb:1
- lib/neo4j/active_rel/validations.rb:2
- lib/neo4j/active_rel/related_node.rb:1

Methods

#== Performs equality checking on the result of attributes and its type.

```
def ==(other)
  return false unless other.instance_of? self.class
  attributes == other.attributes
end
```

#[]

```
def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end
```

#[]= Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
```

```
      fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
    end
  end
```

#_persisted_obj Returns the value of attribute `_persisted_obj`

```
def _persisted_obj
  @_persisted_obj
end
```

#apply_default_values

```
def apply_default_values
  return if self.class.declared_property_defaults.empty?
  self.class.declared_property_defaults.each_pair do |key, value|
    self.send("#{key}=", value) if self.send(key).nil?
  end
end
```

#assign_attributes Mass update a model's attributes

```
def assign_attributes(new_attributes = nil)
  return unless new_attributes.present?
  new_attributes.each do |name, value|
    writer = "#{name}="
    send(writer, value) if respond_to?(writer)
  end
end
```

#attribute_before_type_cast Read the raw attribute value

```
def attribute_before_type_cast(name)
  @attributes ||= {}
  @attributes[name.to_s]
end
```

#attributes Returns a Hash of all attributes

```
def attributes
  attributes_map { |name| send name }
end
```

#attributes= Mass update a model's attributes

```
def attributes=(new_attributes)
  assign_attributes(new_attributes)
end
```

#cache_key

```
def cache_key
  if self.new_record?
    "#{model_cache_key}/new"
  elsif self.respond_to?(:updated_at) && !self.updated_at.blank?
    "#{model_cache_key}/#{neo_id}-#{self.updated_at.utc.to_s(:number)}"
  else
    "#{model_cache_key}/#{neo_id}"
  end
end
```

#concurrent_increment! Increments concurrently a numeric attribute by a certain amount

```
def concurrent_increment!(attribute, by = 1)
  query_rel = Neo4j::Session.query.match('()-[n]-()').where(n: {neo_id: neo_id})
  increment_by_query! query_rel, attribute, by
end
```

#conditional_callback Allows you to perform a callback if a condition is not satisfied.

```
def conditional_callback(kind, guard)
  return yield if guard
  run_callbacks(kind) { yield }
end
```

#create_method

```
def create_method
  self.class.create_method
end
```

#creates_unique_option

```
def creates_unique_option
  self.class.create_unique_option
end
```

#cypher_identifier

```
def cypher_identifier
  @cypher_identifier || :rel
end
```

#declared_properties

```
def declared_properties
  self.class.declared_properties
end
```

#destroy

nodoc

```
def destroy #:nodoc:
  tx = Neo4j::Transaction.new
  run_callbacks(:destroy) { super }
rescue
  @_deleted = false
  @attributes = @attributes.dup
  tx.mark_failed
  raise
ensure
  tx.close if tx
end
```

#destroyed? Returns +true+ if the object was destroyed.

```
def destroyed?
  @_deleted
end
```

#end_node

```
alias_method :end_node, :to_node
```

#eq?

```
def ==(other)
  other.class == self.class && other.id == id
end
```

#exist?

```
def exist?
  _persisted_obj && _persisted_obj.exist?
end
```

#freeze

```
def freeze
  @attributes.freeze
  self
end
```

#from_node_identifier

```
def from_node_identifier
  @from_node_identifier || :from_node
end
```

#from_node_identifier=

```
def from_node_identifier=(id)
  @from_node_identifier = id.to_sym
end
```

#from_node_neo_id

```
alias_method :from_node_neo_id, :start_node_neo_id
```

#frozen?

```
def frozen?
  @attributes.frozen?
end
```

#hash

```
def hash
  id.hash
end
```

#id

```
def id
  id = neo_id
  id.is_a?(Integer) ? id : nil
end
```

#increment

 Increments a numeric attribute by a certain amount

```
def increment(attribute, by = 1)
  self[attribute] ||= 0
  self[attribute] += by
  self
end
```

#increment!

 Convenience method to increment numeric attribute and #save at the same time

```
def increment!(attribute, by = 1)
  increment(attribute, by).update_attribute(attribute, self[attribute])
end
```

#init_on_load called when loading the rel from the database

```
def init_on_load(persisted_rel, from_node_id, to_node_id, type)
  @rel_type = type
  @persisted_obj = persisted_rel
  changed_attributes && changed_attributes.clear
  @attributes = convert_and_assign_attributes(persisted_rel.props)
  load_nodes(from_node_id, to_node_id)
end
```

#init_on_reload

```
def init_on_reload(unwrapped_reloaded)
  @attributes = nil
  init_on_load(unwrapped_reloaded,
               unwrapped_reloaded._start_node_id,
               unwrapped_reloaded._end_node_id,
               unwrapped_reloaded.rel_type)

  self
end
```

#initialize

```
def initialize(from_node = nil, to_node = nil, args = nil)
  load_nodes(node_or_nil(from_node), node_or_nil(to_node))
  resolved_args = hash_or_nil(from_node, args)
  symbol_args = resolved_args.is_a?(Hash) ? resolved_args.symbolize_keys : resolved_args
  super(symbol_args)
end
```

#inject_defaults!

```
def inject_defaults!(starting_props)
  return starting_props if self.class.declared_properties.declared_property_defaults.empty?
  self.class.declared_properties.inject_defaults!(self, starting_props || {})
end
```

#inspect

```
def inspect
  attribute_descriptions = inspect_attributes.map do |key, value|
    "#{Neo4j::ANSI::CYAN}#{key}: #{Neo4j::ANSI::CLEAR}#{value.inspect}"
  end.join(', ')

  separator = ' ' unless attribute_descriptions.empty?
  "<#{Neo4j::ANSI::YELLOW}#{self.class.name}#{Neo4j::ANSI::CLEAR}#{separator}#{attribute_descriptions}"
end
```

#marshal_dump

```
def marshal_dump
  marshal_instance_variables.map(&method(:instance_variable_get))
end
```

#marshal_load

```
def marshal_load(array)
  marshal_instance_variables.zip(array).each do |var, value|
    instance_variable_set(var, value)
  end
end
```

#neo4j_obj

```
def neo4j_obj
  _persisted_obj || fail('Tried to access native neo4j object on a non persisted object')
end
```

#neo_id

```
def neo_id
  _persisted_obj ? _persisted_obj.neo_id : nil
end
```

#new? Returns +true+ if the record hasn't been saved to Neo4j yet.

```
def new_record?
  !_persisted_obj
end
```

#new_record? Returns +true+ if the record hasn't been saved to Neo4j yet.

```
def new_record?
  !_persisted_obj
end
```

#node_cypher_representation

```
def node_cypher_representation(node)
  node_class = node.class
  id_name = node_class.id_property_name
  labels = ':' + node_class.mapped_label_names.join(':')

  "(#{labels} {#{id_name}: #{node.id.inspect}})"
end
```

#persisted? Returns +true+ if the record is persisted, i.e. it's not a new record and it was not destroyed

```
def persisted?
  !new_record? && !destroyed?
end
```

#props

```
def props
  attributes.reject { |_, v| v.nil? }.symbolize_keys
end
```

#props_for_create Returns a hash containing: * All properties and values for insertion in the database * A *uuid* (or equivalent) key and value * Timestamps, if the class is set to include them. Note that the UUID is added to the hash but is not set on the node. The timestamps, by comparison, are set on the node prior to addition in this hash.

```
def props_for_create
  inject_timestamps!
  props_with_defaults = inject_defaults!(props)
  converted_props = props_for_db(props_with_defaults)
```



```

    return converted_props unless self.class.respond_to?(:default_property_values)
    inject_primary_key!(converted_props)
  end

```

#props_for_persistence

```

def props_for_persistence
  _persisted_obj ? props_for_update : props_for_create
end

```

#props_for_update

```

def props_for_update
  update_magic_properties
  changed_props = attributes.select { |k, _| changed_attributes.include?(k) }
  changed_props.symbolize_keys!
  inject_defaults!(changed_props)
  props_for_db(changed_props)
end

```

#read_attribute

```

def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end

```

#read_attribute_for_validation Implements the ActiveRecord::Validation hook method.

```

def read_attribute_for_validation(key)
  respond_to?(key) ? send(key) : self[key]
end

```

#rel_type

```

def type
  self.class.type
end

```

#reload

```

def reload
  return self if new_record?
  association_proxy_cache.clear if respond_to?(:association_proxy_cache)
  changed_attributes && changed_attributes.clear
  unless reload_from_database
    @_deleted = true
    freeze
  end
  self
end

```

#reload_from_database

```

def reload_from_database
  reloaded = self.class.load_entity(neo_id)
  reloaded ? init_on_reload(reloaded._persisted_obj) : nil
end

```

#reload_properties!

```
def reload_properties!(properties)
  @attributes = nil
  convert_and_assign_attributes(properties)
end
```

#save

```
def save(*args)
  unless _persisted_obj || (from_node.respond_to?(:neo_id) && to_node.respond_to?(:neo_id))
    fail Neo4j::ActiveRel::Persistence::RelInvalidError, 'from_node and to_node must be node obj'
  end
  super(*args)
end
```

#save!

```
def save!(*args)
  save(*args) or fail(RelInvalidError, inspect) # rubocop:disable Style/AndOr
end
```

#send_props

```
def send_props(hash)
  return hash if hash.blank?
  hash.each { |key, value| send("#{key}=", value) }
end
```

#serializable_hash

```
def serializable_hash(*args)
  super.merge(id: id)
end
```

#serialized_properties

```
def serialized_properties
  self.class.serialized_properties
end
```

#start_node

```
alias_method :start_node, :from_node
```

#to_key Returns an Enumerable of all (primary) key attributes or nil if model.persisted? is false

```
def to_key
  _persisted_obj ? [id] : nil
end
```

#to_node_identifier

```
def to_node_identifier
  @to_node_identifier || :to_node
end
```

#to_node_identifier=

```
def to_node_identifier=(id)
  @to_node_identifier = id.to_sym
end
```

#to_node_neo_id

```
alias_method :to_node_neo_id, :end_node_neo_id
```

#touch**nodoc**

```
def touch #:nodoc:
  run_callbacks(:touch) { super }
end
```

#type

```
def type
  self.class.type
end
```

#update Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end
```

#update! Same as {#update_attributes}, but raises an exception if saving fails.

```
def update!(attributes)
  self.attributes = process_attributes(attributes)
  save!
end
```

#update_attribute Convenience method to set attribute and #save at the same time

```
def update_attribute(attribute, value)
  send("#{attribute}=", value)
  self.save
end
```

#update_attribute! Convenience method to set attribute and #save! at the same time

```
def update_attribute!(attribute, value)
  send("#{attribute}=", value)
  self.save!
end
```

#update_attributes Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end
```

#update_attributes! Same as {#update_attributes}, but raises an exception if saving fails.

```
def update!(attributes)
  self.attributes = process_attributes(attributes)
  save!
end
```

#valid?

```
def valid?(context = nil)
  context ||= (new_record? ? :create : :update)
  super(context)
  errors.empty?
end
```

#wrapper Implements the Neo4j::Node#wrapper and Neo4j::Relationship#wrapper method so that we don't have to care if the node is wrapped or not.

```
def wrapper
  self
end
```

#write_attribute Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

11.1.15 Core

Query

Constants

Files

- lib/neo4j/core/query.rb:2

Methods

#proxy_as Creates a Neo4j::ActiveNode::Query::QueryProxy object that builds off of a Core::Query object.

```
def proxy_as(model, var, optional = false)
  # TODO: Discuss whether it's necessary to call `break` on the query or if this should be left
  Neo4j::ActiveNode::Query::QueryProxy.new(model, nil, node: var, optional: optional, starting_q
end
```

#proxy_as_optional Calls proxy_as with *optional* set true. This doesn't offer anything different from calling *proxy_as* directly but it may be more readable.

```
def proxy_as_optional(model, var)
  proxy_as(model, var, true)
end
```

#proxy_chain_level For instances where you turn a QueryProxy into a Query and then back to a QueryProxy with *#proxy_as*

```
def proxy_chain_level
  @proxy_chain_level
end
```

#proxy_chain_level= For instances where you turn a QueryProxy into a Query and then back to a QueryProxy with *#proxy_as*

```
def proxy_chain_level=(value)
  @proxy_chain_level = value
end
```

Constants

Files

- `lib/neo4j/core/query.rb:1`

Methods

11.1.16 Timestamps

This mixin includes timestamps in the included class

Updated

This mixin includes a `updated_at` timestamp property

Constants

Files

- `lib/neo4j/timestamps/updated.rb:4`

Methods

Created

This mixin includes a `created_at` timestamp property

Constants

Files

- `lib/neo4j/timestamps/created.rb:4`

Methods

Constants

Files

- `lib/neo4j/timestamps.rb:6`

- lib/neo4j/timestamps/updated.rb:2
- lib/neo4j/timestamps/created.rb:2

Methods

11.1.17 ActiveNode

Makes Neo4j nodes and relationships behave like ActiveRecord objects. By including this module in your class it will create a mapping for the node to your ruby class by using a Neo4j Label with the same name as the class. When the node is loaded from the database it will check if there is a ruby class for the labels it has. If there Ruby class with the same name as the label then the Neo4j node will be wrapped in a new object of that class.

= ClassMethods * {Neo4j::ActiveNode::Labels::ClassMethods} defines methods like: `index` and `find` * {Neo4j::ActiveNode::Persistence::ClassMethods} defines methods like: `create` and `create!` * {Neo4j::ActiveNode::Property::ClassMethods} defines methods like: `property`.

ClassMethods

Constants

Files

- lib/neo4j/active_node.rb:57
- lib/neo4j/active_node/orm_adapter.rb:5

Methods

#nodeify

```
def nodeify(object)
  if object.is_a?(::Neo4j::ActiveNode) || object.nil?
    object
  else
    self.find(object)
  end
end
```

Enum

ClassMethods

Constants

Files

- lib/neo4j/active_node/enum.rb:6

Methods

Constants

Files

- `lib/neo4j/active_node/enum.rb:2`

Methods

Rels

Constants

Files

- `lib/neo4j/active_node/rels.rb:2`

Methods

#_rels_delegator

```
def _rels_delegator
  fail "Can't access relationship on a non persisted node" unless _persisted_obj
  _persisted_obj
end
```

Query

Helper methods to return Neo4j::Core::Query objects. A query object can be used to successively build a cypher query

```
person.query_as(:n).match('n-[:friend]-o').return(o: :name) # Return the names of all the person's friends
```

ClassMethods

Constants

Files

- `lib/neo4j/active_node/query.rb:35`

Methods

#as Start a new QueryProxy with the starting identifier set to the given argument. This method does not exist within QueryProxy, it can only be called at the class level to create a new QP object. To set an identifier within a QueryProxy chain, give it as the first argument to a chained association.

```
def as(node_var)
  query_proxy(node: node_var, context: self.name)
end
```

#query_as Returns a Query object with all nodes for the model matched as the specified variable name
an early Cypher match has already filtered results) where including labels will degrade performance.

```
def query_as(var, with_labels = true)
  query_proxy.query_as(var, with_labels)
end
```

#query_proxy

```
def query_proxy(options = {})
  Neo4j::ActiveNode::Query::QueryProxy.new(self, nil, options)
end
```

QueryProxy

Link

Constants

Files

- lib/neo4j/active_node/query/query_proxy_link.rb:5

Methods #args

```
def args(var, rel_var)
  @arg.respond_to?(:call) ? @arg.call(var, rel_var) : [@arg, @args].flatten
end
```

#clause Returns the value of attribute clause

```
def clause
  @clause
end
```

.converted_value

```
def converted_value(model, key, value)
  model.declared_properties.value_for_where(key, value)
end
```

.for_arg

```
def for_arg(model, clause, arg, *args)
  default = [Link.new(clause, arg, *args)]

  Link.for_clause(clause, arg, model, *args) || default
rescue NoMethodError
  default
end
```

.for_args

```
def for_args(model, clause, args, association = nil)
  if [:where, :where_not].include?(clause) && args[0].is_a?(String) # Better way?
    [for_arg(model, clause, args[0], *args[1..-1])]
  elsif clause == :rel_where
```



```

      args.map { |arg| for_arg(model, clause, arg, association) }
    else
      args.map { |arg| for_arg(model, clause, arg) }
    end
  end
end

```

.for_association

```

def for_association(name, value, n_string, model)
  neo_id = value.try(:neo_id) || value
  fail ArgumentError, "Invalid value for '#{name}' condition" if not neo_id.is_a?(Integer)

  [
    new(:match, ->(v, _) { "#{v}#{model.associations[name].arrow_cypher}({#{n_string}})" }),
    new(:where, ->(_, _) { {"ID(#{n_string})" => neo_id.to_i} })
  ]
end

```

.for_clause

```

def for_clause(clause, arg, model, *args)
  method_to_call = "for_#{clause}_clause"

  send(method_to_call, arg, model, *args)
end

```

.for_node_where_clause

```

def for_where_clause(arg, model, *args)
  node_num = 1
  result = []
  if arg.is_a?(Hash)
    arg.each do |key, value|
      if model && model.association?(key)
        result += for_association(key, value, "n#{node_num}", model)
        node_num += 1
      else
        result << new_for_key_and_value(model, key, value)
      end
    end
  elsif arg.is_a?(String)
    result << new(:where, arg, args)
  end
  result
end

```

.for_order_clause

```

def for_order_clause(arg, _)
  [new(:order, ->(v, _) { arg.is_a?(String) ? arg : {v => arg} })]
end

```

.for_rel_order_clause

```

def for_rel_order_clause(arg, _)
  [new(:order, ->(_, v) { arg.is_a?(String) ? arg : {v => arg} })]
end

```

.for_rel_where_clause We don't accept strings here. If you want to use a string, just use where.

```
def for_rel_where_clause(arg, _, association)
  arg.each_with_object([]) do |(key, value), result|
    rel_class = association.relationship_class if association.relationship_class
    val = rel_class ? converted_value(rel_class, key, value) : value
    result << new(:where, ->(_, rel_var) { {rel_var => {key => val}} })
  end
end
```

.for_where_clause

```
def for_where_clause(arg, model, *args)
  node_num = 1
  result = []
  if arg.is_a?(Hash)
    arg.each do |key, value|
      if model && model.association?(key)
        result += for_association(key, value, "n#{node_num}", model)
        node_num += 1
      else
        result << new_for_key_and_value(model, key, value)
      end
    end
  elsif arg.is_a?(String)
    result << new(:where, arg, args)
  end
  result
end
```

.for_where_not_clause

```
def for_where_not_clause(*args)
  for_where_clause(*args).each do |link|
    link.instance_variable_set('@clause', :where_not)
  end
end
```

#initialize

```
def initialize(clause, arg, args = [])
  @clause = clause
  @arg = arg
  @args = args
end
```

.new_for_key_and_value

```
def new_for_key_and_value(model, key, value)
  key = (key.to_sym == :id ? model.id_property_name : key)

  val = if !model
    value
  elsif key == model.id_property_name && value.is_a?(Neo4j::ActiveNode)
    value.id
  else
    converted_value(model, key, value)
  end

  new(:where, ->(v, _) { {v => {key => val}} })
end
```

Constants

- METHODS
- FIRST
- LAST

Files

- lib/neo4j/active_node/query/query_proxy.rb:4
- lib/neo4j/active_node/query/query_proxy_link.rb:4

Methods

#<< To add a relationship for the node for the association on this QueryProxy

```
def <<(other_node)
  if @start_object._persisted_obj
    create(other_node, {})
  elsif @association
    @start_object.defer_create(@association.name, other_node)
  else
    fail 'Another crazy error!'
  end
  self
end
```

#== Does exactly what you would hope. Without it, comparing *bobby.lessons* == *sandy.lessons* would evaluate to false because it would be comparing the QueryProxy objects, not the lessons themselves.

```
def ==(other)
  self.to_a == other
end
```

#[]

```
def [](index)
  # TODO: Maybe for this and other methods, use array if already loaded, otherwise
  # use OFFSET and LIMIT 1?
  self.to_a[index]
end
```

#_create_relationship

```
def _create_relationship(other_node_or_nodes, properties)
  association._create_relationship(@start_object, other_node_or_nodes, properties)
end
```

#_model_label_string param [TrueClass, FalseClass] with_labels This param is used by certain QueryProxy methods that already have the neo_id and therefore do not need labels. The @association_labels instance var is set during init and used during association chaining to keep labels out of Cypher queries.

```
def _model_label_string(with_labels = true)
  return if !@model || (!with_labels || @association_labels == false)
  @model.mapped_label_names.map { |label_name| ":{label_name}" }.join
end
```

#_nodeify!

```

def _nodeify!(*args)
  other_nodes = [args].flatten!.map! do |arg|
    (arg.is_a?(Integer) || arg.is_a?(String)) ? @model.find_by(id: arg) : arg
  end.compact

  if @model && other_nodes.any? { |other_node| !other_node.class.mapped_label_names.include?(@model.mapped_label_name) }
    fail ArgumentError, "Node must be of the association's class when model is specified"
  end

  other_nodes
end

```

#all_rels_to Returns all relationships across a QueryProxy chain between a given node or array of nodes and the preceeding link.

```

def rels_to(node)
  self.match_to(node).pluck(rel_var)
end

```

#as_models Takes an Array of ActiveRecord models and applies the appropriate WHERE clause So for a *Teacher* model inheriting from a *Person* model and an *Article* model if you called `.as_models([Teacher, Article])` The where clause would look something like:

```
WHERE (node_var:Teacher:Person OR node_var:Article)
```

```

def as_models(models)
  where_clause = models.map do |model|
    "`#{identity}`:" + model.mapped_label_names.map do |mapped_label_name|
      "`#{mapped_label_name}`"
    end.join(':')
  end.join(' OR ')

  where(" (#{where_clause}) ")
end

```

#association The most recent node to start a QueryProxy chain. Will be nil when using QueryProxy chains on class methods.

```

def association
  @association
end

```

#base_query

```

def base_query(var, with_labels = true)
  if @association
    chain_var = _association_chain_var
    (_association_query_start(chain_var) & _query).break.send(@match_type,
                                                              "#{chain_var}#{@association_arrow}")
  else
    starting_query ? starting_query : _query_model_as(var, with_labels)
  end
end

```

#blank?

```

def empty?(target = nil)
  query_with_target(target) { |var| !self.exists?(nil, var) }
end

```

#branch Executes the relation chain specified in the block, while keeping the current scope

```
def branch(&block)
  if block
    instance_eval(&block).query.proxy_as(self.model, identity)
  else
    fail LocalJumpError, 'no block given'
  end
end
```

#context Returns the value of attribute context

```
def context
  @context
end
```

#count

```
def count(distinct = nil, target = nil)
  fail(Neo4j::InvalidParameterError, ':count accepts `distinct` or nil as a parameter') unless distinct || target
  query_with_target(target) do |var|
    q = distinct.nil? ? var : "DISTINCT #{var}"
    limited_query = self.query.clause?(:limit) ? self.query.break.with(var) : self.query.reorder
    limited_query.pluck("count(#{q}) AS #{var}").first
  end
end
```

#create

```
def create(other_nodes, properties)
  fail 'Can only create relationships on associations' if !@association
  other_nodes = _nodeify!(*other_nodes)

  Neo4j::Transaction.run do
    other_nodes.each do |other_node|
      other_node.save unless other_node.neo_id

      return false if @association.perform_callback(@start_object, other_node, :before) == false

      @start_object.association_proxy_cache.clear

      _create_relationship(other_node, properties)

      @association.perform_callback(@start_object, other_node, :after)
    end
  end
end
```

#delete Deletes the relationship between a node and its last link in the QueryProxy chain. Executed in the database, callbacks will not run.

```
def delete(node)
  self.match_to(node).query.delete(rel_var).exec
  clear_source_object_cache
end
```

#delete_all Deletes a group of nodes and relationships within a QP chain. When identifier is omitted, it will remove the last link in the chain. The optional argument must be a node identifier. A relationship identifier will result in a Cypher Error

```
def delete_all(identifier = nil)
  query_with_target(identifier) do |target|
    begin
      self.query.with(target).optional_match("#{target}-[#{target}_rel]-()").delete("#{target}")
    rescue Neo4j::Session::CypherError
      self.query.delete(target).exec
    end
    clear_source_object_cache
  end
end
```

#delete_all_rels Deletes the relationships between all nodes for the last step in the QueryProxy chain. Executed in the database, callbacks will not be run.

```
def delete_all_rels
  return unless start_object && start_object._persisted_obj
  self.query.delete(rel_var).exec
end
```

#destroy Returns all relationships between a node and its last link in the QueryProxy chain, destroys them in Ruby. Callbacks will be run.

```
def destroy(node)
  self.rels_to(node).map!(&:destroy)
  clear_source_object_cache
end
```

#each

```
def each(node = true, rel = nil, &block)
  return super if with_associations_spec.size.zero?

  query_from_association_spec.pluck(identity, with_associations_return_clause).map do |record, eager_data|
    eager_data.each_with_index do |eager_records, index|
      record.association_proxy(with_associations_spec[index]).cache_result(eager_records)
    end

    block.call(record)
  end
end
```

#each_for_destruction Used as part of *dependent::destroy* and may not have any utility otherwise. It keeps track of the node responsible for a cascading *destroy* process. but this is not always available, so we require it explicitly.

```
def each_for_destruction(owning_node)
  target = owning_node.called_by || owning_node
  objects = pluck(identity).compact.reject do |obj|
    target.dependent_children.include?(obj)
  end

  objects.each do |obj|
    obj.called_by = target
    target.dependent_children << obj
    yield obj
  end
end
```

#each_rel When called at the end of a QueryProxy chain, it will return the resultant relationship objects instead of nodes. For example, to return the relationship between a given student and their lessons:

```
student.lessons.each_rel do |rel|
```

```
def each_rel(&block)
  block_given? ? each(false, true, &block) : to_enum(:each, false, true)
end
```

#each_with_rel When called at the end of a QueryProxy chain, it will return the nodes and relationships of the last link. For example, to return a lesson and each relationship to a given student:

```
student.lessons.each_with_rel do |lesson, rel|
```

```
def each_with_rel(&block)
  block_given? ? each(true, true, &block) : to_enum(:each, true, true)
end
```

#empty?

```
def empty?(target = nil)
  query_with_target(target) { |var| !self.exists?(nil, var) }
end
```

#exists?

```
def exists?(node_condition = nil, target = nil)
  unless node_condition.is_a?(Integer) || node_condition.is_a?(Hash) || node_condition.nil?
    fail(Neo4j::InvalidParameterError, ':exists? only accepts neo_ids')
  end
  query_with_target(target) do |var|
    start_q = exists_query_start(node_condition, var)
    start_q.query.reorder.return("COUNT(#{var}) AS count").first.count > 0
  end
end
```

#fetch_result_cache

```
def fetch_result_cache
  @result_cache ||= yield
end
```

#find Give ability to call *#find* on associations to get a scoped find Doesn't pass through via *method_missing* because Enumerable has a *#find* method

```
def find(*args)
  scoping { @model.find(*args) }
end
```

#find_each

```
def find_each(options = {})
  query.return(identity).find_each(identity, @model.primary_key, options) do |result|
    yield result.send(identity)
  end
end
```

#find_in_batches

```
def find_in_batches(options = {})
  query.return(identity).find_in_batches(identity, @model.primary_key, options) do |batch|
    yield batch.map(&:identity)
  end
end
```

#find_or_create_by When called, this method returns a single node that satisfies the match specified in the params hash. If no existing node is found to satisfy the match, one is created or associated as expected.

```
def find_or_create_by(params)
  fail 'Method invalid when called on Class objects' unless source_object
  result = self.where(params).first
  return result unless result.nil?
  Neo4j::Transaction.run do
    node = model.find_or_create_by(params)
    self << node
    return node
  end
end
```

#first

```
def first(target = nil)
  first_and_last(FIRST, target)
end
```

#first_rel_to Gives you the first relationship between the last link of a QueryProxy chain and a given node Shorthand for *MATCH (start)-[r]-(other_node) WHERE ID(other_node) = #{other_node.neo_id} RETURN r*

```
def first_rel_to(node)
  self.match_to(node).limit(1).pluck(rel_var).first
end
```

#identity

```
def identity
  @node_var || _result_string
end
```

#include?

```
def include?(other, target = nil)
  query_with_target(target) do |var|
    where_filter = if other.respond_to?(:neo_id)
      "ID(#{var}) = {other_node_id}"
    else
      " #{var}.#{association_id_key} = {other_node_id}"
    end
    node_id = other.respond_to?(:neo_id) ? other.neo_id : other
    self.where(where_filter).params(other_node_id: node_id).query.reorder.return("count(#{var})")
  end
end
```

#initialize QueryProxy is ActiveNode's Cypher DSL. While the name might imply that it creates queries in a general sense, it is actually referring to `<tt>Neo4j::Core::Query</tt>`, which is a pure Ruby Cypher DSL provided by the `<tt>neo4j-core</tt>` gem. QueryProxy provides ActiveRecord-like methods for common patterns. When it's not handling CRUD for relationships and queries, it provides ActiveNode's association chaining (*student.lessons.teachers.where(age: 30).hobbies*) and enjoys long walks on the beach.

It should not ever be necessary to instantiate a new QueryProxy object directly, it always happens as a result of calling a method that makes use of it.

originated. `<tt>has_many</tt>` that created this object. QueryProxy objects are evaluated lazily.

```
def initialize(model, association = nil, options = {})
  @model = model
  @association = association
end
```



```

    @context = options.delete(:context)
    @options = options
    @associations_spec = []

    instance_vars_from_options!(options)

    @match_type = @optional ? :optional_match : :match

    @rel_var = options[:rel] || _rel_chain_var

    @chain = []
    @params = @query_proxy ? @query_proxy.instance_variable_get('@params') : {}
  end

```

#inspect

```

def inspect
  "#<QueryProxy #{@context} CYPHER: #{self.to_cypher.inspect}>"
end

```

#last

```

def last(target = nil)
  first_and_last(LAST, target)
end

```

#limit_value TODO: update this with public API methods if/when they are exposed

```

def limit_value
  return unless self.query.clause?(:limit)
  limit_clause = self.query.send(:clauses).find { |clause| clause.is_a?(Neo4j::Core::QueryClause) }
  limit_clause.instance_variable_get(:@arg)
end

```

#match_to Shorthand for *MATCH (start)-[r]-(other_node) WHERE ID(other_node) = #{other_node.neo_id}* The *node* param can be a persisted *ActiveNode* instance, any string or integer, or nil. When it's a node, it'll use the object's *neo_id*, which is fastest. When not nil, it'll figure out the primary key of that model. When nil, it uses *1 = 2* to prevent matching all records, which is the default behavior when nil is passed to *where* in *QueryProxy*.

```

def match_to(node)
  first_node = node.is_a?(Array) ? node.first : node
  where_arg = if first_node.respond_to?(:neo_id)
    {neo_id: node.is_a?(Array) ? node.map(&:neo_id) : node}
  elsif !node.nil?
    {association_id_key => node.is_a?(Array) ? ids_array(node) : node}
  else
    # support for null object pattern
    '1 = 2'
  end

  self.where(where_arg)
end

```

#method_missing *QueryProxy* objects act as a representation of a model at the class level so we pass through calls. This allows us to define class functions for reusable query chaining or for end-of-query aggregation/summarizing.

```

def method_missing(method_name, *args, &block)
  if @model && @model.respond_to?(method_name)
    scoping { @model.public_send(method_name, *args, &block) }
  else

```

```
      super
    end
  end
end
```

#model The most recent node to start a QueryProxy chain. Will be nil when using QueryProxy chains on class methods.

```
def model
  @model
end
```

#new_link

```
def new_link(node_var = nil)
  self.clone.tap do |new_query_proxy|
    new_query_proxy.instance_variable_set('@result_cache', nil)
    new_query_proxy.instance_variable_set('@node_var', node_var) if node_var
  end
end
```

#node_identity

```
def identity
  @node_var || _result_string
end
```

#node_order

```
alias_method :node_order, :order
```

#node_var The current node identifier on deck, so to speak. It is the object that will be returned by calling *each* and the last node link in the QueryProxy chain.

```
def node_var
  @node_var
end
```

#node_where Since there are *rel_where* and *rel_order* methods, it seems only natural for there to be *node_where* and *node_order*

```
alias_method :node_where, :where
```

#offset

```
alias_method :offset, :skip
```

#optional A shortcut for attaching a new, optional match to the end of a QueryProxy chain.

```
def optional(association, node_var = nil, rel_var = nil)
  self.send(association, node_var, rel_var, optional: true)
end
```

#optional?

```
def optional?
  @optional == true
end
```

#order_by

```
alias_method :order_by, :order
```

#order_property

```
def order_property
  # This should maybe be based on a setting in the association
  # rather than a hardcoded `nil`
  model ? model.id_property_name : nil
end
```

#params

```
def params(params)
  new_link.tap { |new_query| new_query._add_params(params) }
end
```

#pluck For getting variables which have been defined as part of the association chain

```
def pluck(*args)
  transformable_attributes = (model ? model.attribute_names : []) + %w(uuid neo_id)
  arg_list = args.map do |arg|
    if transformable_attributes.include?(arg.to_s)
      {identity => arg}
    else
      arg
    end
  end

  self.query.pluck(*arg_list)
end
```

#query Like calling #query_as, but for when you don't care about the variable name

```
def query
  query_as(identity)
end
```

#query_as Build a Neo4j::Core::Query object for the QueryProxy. This is necessary when you want to take an existing QueryProxy chain and work with it from the more powerful (but less friendly) Neo4j::Core::Query. ..

```
code-block:: ruby
  student.lessons.query_as(:l).with('your cypher here...')

def query_as(var, with_labels = true)
  result_query = @chain.inject(base_query(var, with_labels).params(@params)) do |query, link|
    args = link.args(var, rel_var)

    args.is_a?(Array) ? query.send(link.clause, *args) : query.send(link.clause, args)
  end

  result_query.tap { |query| query.proxy_chain_level = _chain_level }
end
```

#query_proxy Returns the value of attribute query_proxy

```
def query_proxy
  @query_proxy
end
```

#read_attribute_for_serialization

```
def read_attribute_for_serialization(*args)
  to_a.map { |o| o.read_attribute_for_serialization(*args) }
end
```

#rel

```
def rel
  rels.first
end
```

#rel_identity

```
def rel_identity
  ActiveSupport::Deprecation.warn 'rel_identity is deprecated and may be removed from future rel

  @rel_var
end
```

#rel_var The relationship identifier most recently used by the QueryProxy chain.

```
def rel_var
  @rel_var
end
```

#rels

```
def rels
  fail 'Cannot get rels without a relationship variable.' if !@rel_var

  pluck(@rel_var)
end
```

#rels_to Returns all relationships across a QueryProxy chain between a given node or array of nodes and the preceding link.

```
def rels_to(node)
  self.match_to(node).pluck(rel_var)
end
```

#replace_with Deletes the relationships between all nodes for the last step in the QueryProxy chain and replaces them with relationships to the given nodes. Executed in the database, callbacks will not be run.

```
def replace_with(node_or_nodes)
  nodes = Array(node_or_nodes)

  self.delete_all_rels
  nodes.each { |node| self << node }
end
```

#respond_to_missing?

```
def respond_to_missing?(method_name, include_all = false)
  (@model && @model.respond_to?(method_name, include_all)) || super
end
```

#result

```
def result(node = true, rel = nil)
  @result_cache ||= {}
  return result_cache_for(node, rel) if result_cache?(node, rel)

  pluck_vars = []
  pluck_vars << identity if node
  pluck_vars << @rel_var if rel

  result = pluck(*pluck_vars)
```

```

    result.each do |object|
      object.instance_variable_set('@source_query_proxy', self)
      object.instance_variable_set('@source_proxy_result_cache', result)
    end

    @result_cache[[node, rel]] ||= result
  end

```

#result_cache?

```

def result_cache?(node = true, rel = nil)
  !!result_cache_for(node, rel)
end

```

#result_cache_for

```

def result_cache_for(node = true, rel = nil)
  (@result_cache || {})[[node, rel]]
end

```

#scoping Scope all queries to the current scope.

```

Comment.where(post_id: 1).scoping do
  Comment.first
end

```

TODO: unscoped Please check unscoped if you want to remove all previous scopes (including the default_scope) during the execution of a block.

```

def scoping
  previous = @model.current_scope
  @model.current_scope = self
  yield
ensure
  @model.current_scope = previous
end

```

#size

```

def size
  result_cache? ? result_cache_for.length : count
end

```

#source_object The most recent node to start a QueryProxy chain. Will be nil when using QueryProxy chains on class methods.

```

def source_object
  @source_object
end

```

#start_object Returns the value of attribute start_object

```

def start_object
  @start_object
end

```

#starting_query The most recent node to start a QueryProxy chain. Will be nil when using QueryProxy chains on class methods.

```
def starting_query
  @starting_query
end
```

#to_cypher_with_params Returns a string of the cypher query with return objects and params

```
def to_cypher_with_params(columns = [self.identity])
  final_query = query.return_query(columns)
  "#{final_query.to_cypher} | params: #{final_query.send(:merge_params)}"
end
```

#unique_nodes This will match nodes who only have a single relationship of a given type. It's used by *dependent: :delete_orphans* and *dependent: :destroy_orphans* and may not have much utility otherwise.

```
def unique_nodes(association, self_idenfier, other_node, other_rel)
  fail 'Only supported by in QueryProxy chains started by an instance' unless source_object
  return false if send(association.name).empty?
  unique_nodes_query(association, self_idenfier, other_node, other_rel)
    .proxy_as(association.target_class, other_node)
end
```

#update_all Updates some attributes of a group of nodes within a QP chain. The optional argument makes sense only if *updates* is a string.

```
def update_all(updates, params = {})
  # Move this to ActiveNode module?
  update_all_with_query(identity, updates, params)
end
```

#update_all_rels Updates some attributes of a group of relationships within a QP chain. The optional argument makes sense only if *updates* is a string.

```
def update_all_rels(updates, params = {})
  fail 'Cannot update rels without a relationship variable.' unless @rel_var
  update_all_with_query(@rel_var, updates, params)
end
```

#with_associations

```
def with_associations(*spec)
  invalid_association_names = spec.reject { |association_name|
    model.associations[association_name]
  }
  end

  if invalid_association_names.size > 0
    fail "Invalid associations: #{invalid_association_names.join(', ')}"
  end

  new_link.tap { |new_query_proxy|
    new_spec = new_query_proxy.with_associations_spec + spec
    new_query_proxy.with_associations_spec.replace(new_spec)
  }
  end
end
```

#with_associations_return_clause

```
def with_associations_return_clause
  '[' + with_associations_spec.map { |n| "collect(#{n})" }.join(',') + ']'
end
```

#with_associations_spec

```
def with_associations_spec
  @with_associations_spec ||= []
end
```

QueryProxyMethods

Constants

- FIRST
- LAST

Files

- lib/neo4j/active_node/query/query_proxy_methods.rb:4

Methods

#all_rels_to Returns all relationships across a QueryProxy chain between a given node or array of nodes and the preceeding link.

```
def rels_to(node)
  self.match_to(node).pluck(rel_var)
end
```

#as_models Takes an Array of ActiveNode models and applies the appropriate WHERE clause So for a *Teacher* model inheriting from a *Person* model and an *Article* model if you called .as_models([Teacher, Article]) The where clause would look something like:

```
WHERE (node_var:Teacher:Person OR node_var:Article)
```

```
def as_models(models)
  where_clause = models.map do |model|
    "`#{identity}`:" + model.mapped_label_names.map do |mapped_label_name|
      "`#{mapped_label_name}`"
    end.join(':')
  end.join(' OR ')

  where("(#{where_clause})")
end
```

#blank?

```
def empty?(target = nil)
  query_with_target(target) { |var| !self.exists?(nil, var) }
end
```

#count

```
def count(distinct = nil, target = nil)
  fail(Neo4j::InvalidParameterError, ':count accepts `distinct` or nil as a parameter') unless distinct || target
  query_with_target(target) do |var|
    q = distinct.nil? ? var : "DISTINCT #{var}"
    limited_query = self.query.clause?(:limit) ? self.query.break.with(var) : self.query.reorder
    limited_query.pluck("count(#{q}) AS #{var}").first
  end
end
```

#empty?

```
def empty?(target = nil)
  query_with_target(target) { |var| !self.exists?(nil, var) }
end
```

#exists?

```
def exists?(node_condition = nil, target = nil)
  unless node_condition.is_a?(Integer) || node_condition.is_a?(Hash) || node_condition.nil?
    fail(Neo4j::InvalidParameterError, ':exists? only accepts neo_ids')
  end
  query_with_target(target) do |var|
    start_q = exists_query_start(node_condition, var)
    start_q.query.reorder.return("COUNT(#{var}) AS count").first.count > 0
  end
end
```

#find Give ability to call *#find* on associations to get a scoped find Doesn't pass through via *method_missing* because Enumerable has a *#find* method

```
def find(*args)
  scoping { @model.find(*args) }
end
```

#find_or_create_by When called, this method returns a single node that satisfies the match specified in the params hash. If no existing node is found to satisfy the match, one is created or associated as expected.

```
def find_or_create_by(params)
  fail 'Method invalid when called on Class objects' unless source_object
  result = self.where(params).first
  return result unless result.nil?
  Neo4j::Transaction.run do
    node = model.find_or_create_by(params)
    self << node
    return node
  end
end
```

#first

```
def first(target = nil)
  first_and_last(FIRST, target)
end
```

#first_rel_to Gives you the first relationship between the last link of a QueryProxy chain and a given node Shorthand for *MATCH (start)-[r]-(other_node) WHERE ID(other_node) = #{other_node.neo_id} RETURN r*

```
def first_rel_to(node)
  self.match_to(node).limit(1).pluck(rel_var).first
end
```

#include?

```
def include?(other, target = nil)
  query_with_target(target) do |var|
    where_filter = if other.respond_to?(:neo_id)
      "ID(#{var}) = {other_node_id}"
    else
      "#{var}.#{association_id_key} = {other_node_id}"
    end
  end
end
```



```

      node_id = other.respond_to?(:neo_id) ? other.neo_id : other
      self.where(where_filter).params(other_node_id: node_id).query.reorder.return("count(#{var})")
    end
  end
end

```

#last

```

def last(target = nil)
  first_and_last(LAST, target)
end

```

#limit_value TODO: update this with public API methods if/when they are exposed

```

def limit_value
  return unless self.query.clause?(:limit)
  limit_clause = self.query.send(:clauses).find { |clause| clause.is_a?(Neo4j::Core::QueryClause) }
  limit_clause.instance_variable_get(:@arg)
end

```

#match_to Shorthand for *MATCH (start)-[r]-(other_node) WHERE ID(other_node) = #{other_node.neo_id}* The *node* param can be a persisted *ActiveNode* instance, any string or integer, or nil. When it's a node, it'll use the object's *neo_id*, which is fastest. When not nil, it'll figure out the primary key of that model. When nil, it uses *l = 2* to prevent matching all records, which is the default behavior when nil is passed to *where* in *QueryProxy*.

```

def match_to(node)
  first_node = node.is_a?(Array) ? node.first : node
  where_arg = if first_node.respond_to?(:neo_id)
    {neo_id: node.is_a?(Array) ? node.map(&:neo_id) : node}
  elsif !node.nil?
    {association_id_key => node.is_a?(Array) ? ids_array(node) : node}
  else
    # support for null object pattern
    '1 = 2'
  end

  self.where(where_arg)
end

```

#optional A shortcut for attaching a new, optional match to the end of a *QueryProxy* chain.

```

def optional(association, node_var = nil, rel_var = nil)
  self.send(association, node_var, rel_var, optional: true)
end

```

#order_property

```

def order_property
  # This should maybe be based on a setting in the association
  # rather than a hardcoded `nil`
  model ? model.id_property_name : nil
end

```

#rel

```

def rel
  rels.first
end

```

#rels

```
def rels
  fail 'Cannot get rels without a relationship variable.' if !@rel_var

  pluck(@rel_var)
end
```

#rels_to Returns all relationships across a QueryProxy chain between a given node or array of nodes and the preceding link.

```
def rels_to(node)
  self.match_to(node).pluck(rel_var)
end
```

#size

```
def size
  result_cache? ? result_cache_for.length : count
end
```

QueryProxyEnumerable

Methods related to returning nodes and rels from QueryProxy

Constants

Files

- `lib/neo4j/active_node/query/query_proxy_enumerable.rb:5`

Methods

== Does exactly what you would hope. Without it, comparing *bobby.lessons* == *sandy.lessons* would evaluate to false because it would be comparing the QueryProxy objects, not the lessons themselves.

```
def ==(other)
  self.to_a == other
end
```

#each Just like every other `each` but it allows for optional params to support the versions that also return relationships. The `node` and `rel` params are typically used by those other methods but there's nothing stopping you from using *your_node.each(true, true)* instead of *your_node.each_with_rel*.

```
def each(node = true, rel = nil, &block)
  result(node, rel).each(&block)
end
```

#each_rel When called at the end of a QueryProxy chain, it will return the resultant relationship objects instead of nodes. For example, to return the relationship between a given student and their lessons:

```
student.lessons.each_rel do |rel|
```

```
def each_rel(&block)
  block_given? ? each(false, true, &block) : to_enum(:each, false, true)
end
```

#each_with_rel When called at the end of a QueryProxy chain, it will return the nodes and relationships of the last link. For example, to return a lesson and each relationship to a given student:

```
student.lessons.each_with_rel do |lesson, rel|
```

```
def each_with_rel(&block)
  block_given? ? each(true, true, &block) : to_enum(:each, true, true)
end
```

#fetch_result_cache

```
def fetch_result_cache
  @result_cache ||= yield
end
```

#pluck For getting variables which have been defined as part of the association chain

```
def pluck(*args)
  transformable_attributes = (model ? model.attribute_names : []) + %w(uuid neo_id)
  arg_list = args.map do |arg|
    if transformable_attributes.include?(arg.to_s)
      {identity => arg}
    else
      arg
    end
  end

  self.query.pluck(*arg_list)
end
```

#result

```
def result(node = true, rel = nil)
  @result_cache ||= {}
  return result_cache_for(node, rel) if result_cache?(node, rel)

  pluck_vars = []
  pluck_vars << identity if node
  pluck_vars << @rel_var if rel

  result = pluck(*pluck_vars)

  result.each do |object|
    object.instance_variable_set('@source_query_proxy', self)
    object.instance_variable_set('@source_proxy_result_cache', result)
  end

  @result_cache[[node, rel]] ||= result
end
```

#result_cache?

```
def result_cache?(node = true, rel = nil)
  !!result_cache_for(node, rel)
end
```

#result_cache_for

```
def result_cache_for(node = true, rel = nil)
  (@result_cache || {})[[node, rel]]
end
```

QueryProxyEagerLoading

Constants

Files

- lib/neo4j/active_node/query/query_proxy_eager_loading.rb:4

Methods #each

```
def each(node = true, rel = nil, &block)
  return super if with_associations_spec.size.zero?

  query_from_association_spec.pluck(identity, with_associations_return_clause).map do |record, eager_data|
    eager_data.each_with_index do |eager_records, index|
      record.association_proxy(with_associations_spec[index]).cache_result(eager_records)
    end

    block.call(record)
  end
end
```

#with_associations

```
def with_associations(*spec)
  invalid_association_names = spec.reject do |association_name|
    model.associations[association_name]
  end

  if invalid_association_names.size > 0
    fail "Invalid associations: #{invalid_association_names.join(', ')}"
  end

  new_link.tap do |new_query_proxy|
    new_spec = new_query_proxy.with_associations_spec + spec
    new_query_proxy.with_associations_spec.replace(new_spec)
  end
end
```

#with_associations_return_clause

```
def with_associations_return_clause
  '[' + with_associations_spec.map { |n| "collect(#{n})" }.join(',') + ']'
end
```

#with_associations_spec

```
def with_associations_spec
  @with_associations_spec ||= []
end
```

QueryProxyFindInBatches

Constants

Files

- lib/neo4j/active_node/query/query_proxy_find_in_batches.rb:4

Methods #find_each

```
def find_each(options = {})
  query.return(identity).find_each(identity, @model.primary_key, options) do |result|
    yield result.send(identity)
  end
end
```

#find_in_batches

```
def find_in_batches(options = {})
  query.return(identity).find_in_batches(identity, @model.primary_key, options) do |batch|
    yield batch.map(&:identity)
  end
end
```

QueryProxyMethodsOfMassUpdating

Constants

Files

- lib/neo4j/active_node/query/query_proxy_methods_of_mass_updating.rb:4

Methods

#delete Deletes the relationship between a node and its last link in the QueryProxy chain. Executed in the database, callbacks will not run.

```
def delete(node)
  self.match_to(node).query.delete(rel_var).exec
  clear_source_object_cache
end
```

#delete_all Deletes a group of nodes and relationships within a QP chain. When identifier is omitted, it will remove the last link in the chain. The optional argument must be a node identifier. A relationship identifier will result in a Cypher Error

```
def delete_all(identifier = nil)
  query_with_target(identifier) do |target|
    begin
      self.query.with(target).optional_match("#{target}-[#{target}_rel]-()").delete("#{target}")
    rescue Neo4j::Session::CypherError
      self.query.delete(target).exec
    end
    clear_source_object_cache
  end
end
```

#delete_all_rels Deletes the relationships between all nodes for the last step in the QueryProxy chain. Executed in the database, callbacks will not be run.

```
def delete_all_rels
  return unless start_object && start_object._persisted_obj
  self.query.delete(rel_var).exec
end
```

#destroy Returns all relationships between a node and its last link in the QueryProxy chain, destroys them in Ruby. Callbacks will be run.

```
def destroy(node)
  self.rels_to(node).map!(&:destroy)
  clear_source_object_cache
end
```

#replace_with Deletes the relationships between all nodes for the last step in the QueryProxy chain and replaces them with relationships to the given nodes. Executed in the database, callbacks will not be run.

```
def replace_with(node_or_nodes)
  nodes = Array(node_or_nodes)

  self.delete_all_rels
  nodes.each { |node| self << node }
end
```

#update_all Updates some attributes of a group of nodes within a QP chain. The optional argument makes sense only if *updates* is a string.

```
def update_all(updates, params = {})
  # Move this to ActiveNode module?
  update_all_with_query(identity, updates, params)
end
```

#update_all_rels Updates some attributes of a group of relationships within a QP chain. The optional argument makes sense only if *updates* is a string.

```
def update_all_rels(updates, params = {})
  fail 'Cannot update rels without a relationship variable.' unless @rel_var
  update_all_with_query(@rel_var, updates, params)
end
```

Constants

Files

- lib/neo4j/active_node/query.rb:7
- lib/neo4j/active_node/query/query_proxy.rb:3
- lib/neo4j/active_node/query/query_proxy_link.rb:3
- lib/neo4j/active_node/query/query_proxy_methods.rb:3
- lib/neo4j/active_node/query/query_proxy_enumerable.rb:3
- lib/neo4j/active_node/query/query_proxy_eager_loading.rb:3
- lib/neo4j/active_node/query/query_proxy_find_in_batches.rb:3
- lib/neo4j/active_node/query/query_proxy_methods_of_mass_updating.rb:3

Methods

#as Starts a new QueryProxy with the starting identifier set to the given argument and QueryProxy source_object set to the node instance. This method does not exist within QueryProxy and can only be used to start a new chain.

```
def as(node_var)
  self.class.query_proxy(node: node_var, source_object: self).match_to(self)
end
```

#query_as Returns a Query object with the current node matched the specified variable name

```
def query_as(node_var)
  self.class.query_as(node_var, false).where("ID(#{node_var})" => self.neo_id)
end
```

HasN

NonPersistedNodeError

Constants

Files

- lib/neo4j/active_node/has_n.rb:5

Methods

AssociationProxy

Return this object from associations It uses a QueryProxy to get results But also caches results and can have results cached on it

Constants

- QUERY_PROXY_METHODS
- CACHED_RESULT_METHODS

Files

- lib/neo4j/active_node/has_n.rb:10

Methods #+

```
def +(other)
  self.to_a + other
end
```

#==

```
def ==(other)
  self.to_a == other.to_a
end
```

#add_to_cache

```
def add_to_cache(object)
  @cached_result ||= []
  @cached_result << object
end
```

#cache_query_proxy_result

```
def cache_query_proxy_result
  @query_proxy.to_a.tap { |result| cache_result(result) }
end
```

#cache_result

```
def cache_result(result)
  @cached_result = result
  @enumerable = (@cached_result || @query_proxy)
end
```

#cached?

```
def cached?
  !!@cached_result
end
```

#clear_cache_result

```
def clear_cache_result
  cache_result(nil)
end
```

#each

```
def each(&block)
  result_nodes.each(&block)
end
```

#initialize

```
def initialize(query_proxy, deferred_objects = [], cached_result = nil)
  @query_proxy = query_proxy
  @deferred_objects = deferred_objects

  cache_result(cached_result)

  # Represents the thing which can be enumerated
  # default to @query_proxy, but will be set to
  # @cached_result if that is set
  @enumerable = @query_proxy
end
```

#inspect States: Default

```
def inspect
  if @cached_result
    result_nodes.inspect
  else
    "<AssociationProxy @query_proxy=#{@query_proxy.inspect}>"
  end
end
```


#method_missing

```
def method_missing(method_name, *args, &block)
  target = target_for_missing_method(method_name)
  super if target.nil?

  cache_query_proxy_result if !cached? && !target.is_a?(Neo4j::ActiveNode::Query::QueryProxy)
  clear_cache_result if !QUERY_PROXY_METHODS.include?(method_name) && target.is_a?(Neo4j::ActiveNode::Query::QueryProxy)

  target.public_send(method_name, *args, &block)
end
```

#replace_with

```
def replace_with(*args)
  @cached_result = nil

  @query_proxy.public_send(:replace_with, *args)
end
```

#result

```
def result
  (@deferred_objects || []) + result_without_deferred
end
```

#result_ids

```
def result_ids
  result.map do |object|
    object.is_a?(Neo4j::ActiveNode) ? object.id : object
  end
end
```

#result_nodes

```
def result_nodes
  return result_objects if !@query_proxy.model

  result_objects.map do |object|
    object.is_a?(Neo4j::ActiveNode) ? object : @query_proxy.model.find(object)
  end
end
```

#result_objects

```
def result_objects
  @deferred_objects + result_without_deferred
end
```

#result_without_deferred

```
def result_without_deferred
  cache_query_proxy_result if !@cached_result

  @cached_result
end
```

#serializable_hash

```
def serializable_hash(options = {})
  to_a.map { |record| record.serializable_hash(options) }
end
```

ClassMethods

Constants

Files

- lib/neo4j/active_node/has_n.rb:206

Methods

#association? rubocop:disable Style/PredicateName

```
def association?(name)
  !!associations[name.to_sym]
end
```

#associations

```
def associations
  @associations ||= {}
end
```

#associations_keys

```
def associations_keys
  @associations_keys ||= associations.keys
end
```

#has_association?

nocov

```
def has_association?(name)
  ActiveSupport::Deprecation.warn 'has_association? is deprecated and may be removed from future
  versions'

  association?(name)
end
```

#has_many For defining a “has many” association on a model. This defines a set of methods on your model instances. For instance, if you define the association on a Person model:

```
has_many :out, :vehicles, type: :has_vehicle
```

This would define the following methods:

#vehicles Returns a QueryProxy object. This is an Enumerable object and thus can be iterated over. It also has the ability to accept class-level methods from the Vehicle model (including calls to association methods)

#vehicles= Takes an array of Vehicle objects and replaces all current :HAS_VEHICLE relationships with new relationships referring to the specified objects

.vehicles Returns a QueryProxy object. This would represent all Vehicle objects associated with either all Person nodes (if Person.vehicles is called), or all Vehicle objects associated with the Person nodes thus far represented in the QueryProxy chain. For example:

```
company.people.where(age: 40).vehicles
```

Arguments:

direction: Available values: `:in`, `:out`, or `:both`.

Refers to the relative to the model on which the association is being defined.

Example:

```
Person.has_many :out, :posts, type: :wrote
```

means that a *WROTE* relationship goes from a *Person* node to a *Post* node

name: The name of the association. The affects the methods which are created (see above). The name is also used to form default assumptions about the model which is being referred to

Example:

```
Person.has_many :out, :posts, type: :wrote
```

will assume a *model_class* option of `'Post'` unless otherwise specified

options: A Hash of options. Allowed keys are:

type: The Neo4j relationship type. This option is required unless either the *origin* or *rel_class* options are specified

origin: The name of the association from another model which the *type* and *model_class* can be gathered.

Example:

```
# `model_class` of `Post` is assumed here
Person.has_many :out, :posts, origin: :author

Post.has_one :in, :author, type: :has_author, model_class: :Person
```

model_class: The model class to which the association is referring. Can be a Symbol/String (or an Array of same) with the name of the *ActiveNode* class, *false* to specify any model, or *nil* to specify that it should be guessed.

rel_class: The *ActiveRel* class to use for this association. Can be either a model object include ing *ActiveRel* or a Symbol/String (or an Array of same). A Symbol or String is recommended to avoid load-time issues

dependent: Enables deletion cascading. Available values: `:delete`, `:delete_orphans`, `:destroy`, `:destroy_orphans` (note that the `:destroy_orphans` option is known to be “very metal”. Caution advised)

```
def has_many(direction, name, options = {}) # rubocop:disable Style/PredicateName
  name = name.to_sym
  build_association(:has_many, direction, name, options)

  define_has_many_methods(name)
end
```

#has_one For defining an “has one” association on a model. This defines a set of methods on your model instances. For instance, if you define the association on a *Person* model:

```
has_one :out, :vehicle, type: :has_vehicle
```

This would define the methods: `#vehicle`, `#vehicle=`, and `.vehicle`.

See `#has_many` for anything not specified here

```
def has_one(direction, name, options = {}) # rubocop:disable Style/PredicateName
  name = name.to_sym
  build_association(:has_one, direction, name, options)

  define_has_one_methods(name)
end
```

#inherited make sure the inherited classes inherit the `<tt>_decl_rels</tt>` hash

```
def inherited(klass)
  klass.instance_variable_set(:@associations, associations.clone)
  @associations_keys = klass.associations_keys.clone
  super
end
```

Association

RelWrapper Provides the interface needed to interact with the ActiveRecord query factory.

Constants

Files

- `lib/neo4j/active_node/has_n/association/rel_wrapper.rb:3`

Methods `#create_method`

```
def create_method
  creates_unique? ? :create_unique : :create
end
```

`#creates_unique`

```
def creates_unique(option = :none)
  option = :none if option == true
  @creates_unique = option
end
```

`#creates_unique?`

```
def creates_unique?
  !!@creates_unique
end
```

`#creates_unique_option`

```
def creates_unique_option
  @creates_unique || :none
end
```

`#initialize`

```
def initialize(association, properties = {})
  @association = association
  @properties = properties
  @type = association.relationship_type(true)
  creates_unique(association.creates_unique_option) if association.unique?
end
```

#persisted?

```
def persisted?
  false
end
```

#properties Returns the value of attribute properties

```
def properties
  @properties
end
```

#properties= Sets the attribute properties

```
def properties=(value)
  @properties = value
end
```

#props_for_create Returns the value of attribute properties

```
def properties
  @properties
end
```

#type Returns the value of attribute type

```
def type
  @type
end
```

#unique?

```
def creates_unique?
  !!@creates_unique
end
```

RelFactory**Constants****Files**

- `lib/neo4j/active_node/has_n/association/rel_factory.rb:3`

Methods #_create_relationship

```
def _create_relationship
  creator = association.relationship_class ? :rel_class : :factory
  send(:"_create_relationship_with_#{creator}")
end
```

.create

```
def self.create(start_object, other_node_or_nodes, properties, association)
  factory = new(start_object, other_node_or_nodes, properties, association)
  factory._create_relationship
end
```

Constants

- VALID_ASSOCIATION_OPTION_KEYS
- VALID_REL_LENGTH_SYMBOLS

Files

- lib/neo4j/active_node/has_n/association.rb:7
- lib/neo4j/active_node/has_n/association/rel_wrapper.rb:1
- lib/neo4j/active_node/has_n/association/rel_factory.rb:2

Methods #_create_relationship

```
def _create_relationship(start_object, node_or_nodes, properties)
  RelFactory.create(start_object, node_or_nodes, properties, self)
end
```

#add_destroy_callbacks

```
def add_destroy_callbacks(model)
  return if dependent.nil?

  model.before_destroy(&method("dependent_#{dependent}_callback"))
rescue NameError
  raise "Unknown dependent option #{dependent}"
end
```

#arrow_cypher Return cypher partial query string for the relationship part of a MATCH (arrow / relationship definition)

```
def arrow_cypher(var = nil, properties = {}, create = false, reverse = false, length = nil)
  validate_origin!

  if create && length.present?
    fail(ArgumentError, 'rel_length option cannot be specified when creating a relationship')
  end

  direction_cypher(get_relationship_cypher(var, properties, create, length), create, reverse)
end
```

#callback

```
def callback(type)
  @callbacks[type]
end
```

#create_method

```
def create_method
  unique? ? :create_unique : :create
end
```

#creates_unique_option

```
def creates_unique_option
  @unique || :none
end
```

#decorated_rel_type

```
def decorated_rel_type(type)
  @decorated_rel_type ||= Neo4j::Shared::RelTypeConverters.decorated_rel_type(type)
end
```

#dependent Returns the value of attribute dependent

```
def dependent
  @dependent
end
```

#derive_model_class

```
def derive_model_class
  refresh_model_class! if pending_model_refresh?
  return @model_class unless @model_class.nil?
  return nil if relationship_class.nil?
  dir_class = direction == :in ? :from_class : :to_class
  return false if relationship_class.send(dir_class).to_s.to_sym == :any
  relationship_class.send(dir_class)
end
```

#direction Returns the value of attribute direction

```
def direction
  @direction
end
```

#discovered_model

```
def discovered_model
  target_classes.select do |constant|
    constant.ancestors.include?(:Neo4j::ActiveNode)
  end
end
```

#initialize

```
def initialize(type, direction, name, options = {type: nil})
  validate_init_arguments(type, direction, name, options)
  @type = type.to_sym
  @name = name
  @direction = direction.to_sym
  @target_class_name_from_name = name.to_s.classify
  apply_vars_from_options(options)
end
```

#model_class Returns the value of attribute model_class

```
def model_class
  @model_class
end
```

#name Returns the value of attribute name

```
def name
  @name
end
```

#pending_model_refresh?

```
def pending_model_refresh?
  !!@pending_model_refresh
end
```

#perform_callback

```
def perform_callback(caller, other_node, type)
  return if callback(type).nil?
  caller.send(callback(type), other_node)
end
```

#queue_model_refresh!

```
def queue_model_refresh!
  @pending_model_refresh = true
end
```

#refresh_model_class!

```
def refresh_model_class!
  @pending_model_refresh = @target_classes_or_nil = nil

  # Using #to_s on purpose here to take care of classes/strings/symbols
  @model_class = ClassArguments.constantize_argument(@model_class.to_s) if @model_class
end
```

#rel_class?

```
def relationship_class?
  !!relationship_class
end
```

#relationship Returns the value of attribute relationship

```
def relationship
  @relationship
end
```

#relationship_class

```
def relationship_class
  @relationship_class ||= @relationship_class_name && @relationship_class_name.constantize
end
```

#relationship_class?

```
def relationship_class?
  !!relationship_class
end
```


#relationship_class_name Returns the value of attribute relationship_class_name

```
def relationship_class_name
  @relationship_class_name
end
```

#relationship_class_type

```
def relationship_class_type
  relationship_class._type.to_sym
end
```

#relationship_type

```
def relationship_type(create = false)
  case
  when relationship_class
    relationship_class_type
  when !@relationship_type.nil?
    @relationship_type
  when @origin
    origin_type
  else
    (create || exceptional_target_class?) && decorated_rel_type(@name)
  end
end
```

#target_class

```
def target_class
  return @target_class if @target_class

  return if !(target_class_names && target_class_names.size == 1)

  class_const = ClassArguments.constantize_argument(target_class_names[0])

  @target_class = class_const
end
```

#target_class_names

```
def target_class_names
  option = target_class_option(derive_model_class)

  @target_class_names ||= if option.is_a?(Array)
    option.map(&:to_s)
  elsif option
    [option.to_s]
  end
end
```

#target_class_option

```
def target_class_option(model_class)
  case model_class
  when nil
    @target_class_name_from_name ? "#{association_model_namespace}::#{@target_class_name_from_name}"
  when Array
    model_class.map { |sub_model_class| target_class_option(sub_model_class) }
  when false
    false
  end
end
```

```
    else
      model_class.to_s[0, 2] == '::' ? model_class.to_s : "::#{model_class}"
    end
  end
end
```

#target_classes

```
def target_classes
  ClassArguments.constantize_argument(target_class_names)
end
```

#target_classes_or_nil

```
def target_classes_or_nil
  @target_classes_or_nil ||= discovered_model if target_class_names
end
```

#target_where_clause

```
def target_where_clause
  return if model_class == false

  Array.new(target_classes).map do |target_class|
    "#{name}:#{target_class.mapped_label_name}"
  end.join(' OR ')
end
```

#type Returns the value of attribute type

```
def type
  @type
end
```

#unique?

```
def unique?
  return relationship_class.unique? if rel_class?
  @origin ? origin_association.unique? : !!@unique
end
```

#validate_dependent

```
def validate_dependent(value)
  fail ArgumentError, "Invalid dependent value: #{value.inspect}" if not valid_dependent_value?(value)
end
```

AssociationCypherMethods

Constants

- VALID_REL_LENGTH_SYMBOLS

Files

- lib/neo4j/active_node/has_n/association_cypher_methods.rb:4

Methods

#arrow_cypher Return cypher partial query string for the relationship part of a MATCH (arrow / relationship definition)

```
def arrow_cypher(var = nil, properties = {}, create = false, reverse = false, length = nil)
  validate_origin!

  if create && length.present?
    fail(ArgumentError, 'rel_length option cannot be specified when creating a relationship')
  end

  direction_cypher(get_relationship_cypher(var, properties, create, length), create, reverse)
end
```

Constants

Files

- lib/neo4j/active_node/has_n.rb:2
- lib/neo4j/active_node/has_n/association.rb:6
- lib/neo4j/active_node/has_n/association/rel_factory.rb:1
- lib/neo4j/active_node/has_n/association_cypher_methods.rb:3

Methods

#association_proxy

```
def association_proxy(name, options = {})
  name = name.to_sym
  hash = association_proxy_hash(name, options)
  association_proxy_cache_fetch(hash) do
    if result_cache = self.instance_variable_get('@source_proxy_result_cache')
      result_by_previous_id = previous_proxy_results_by_previous_id(result_cache, name)

      result_cache.inject(nil) do |proxy_to_return, object|
        proxy = fresh_association_proxy(name, options.merge(start_object: object), result_by_pre

        object.association_proxy_cache[hash] = proxy

        (self == object ? proxy : proxy_to_return)
      end
    else
      fresh_association_proxy(name, options)
    end
  end
end
```

#association_proxy_cache Returns the current AssociationProxy cache for the association cache. It is in the format { :association_name => AssociationProxy } This is so that we * don't need to re-build the QueryProxy objects * also because the QueryProxy object caches it's results * so we don't need to query again * so that we can cache results from association calls or eager loading

```
def association_proxy_cache
  @association_proxy_cache ||= {}
end
```

#association_proxy_cache_fetch

```
def association_proxy_cache_fetch(key)
  association_proxy_cache.fetch(key) do
    value = yield
    association_proxy_cache[key] = value
  end
end
```

#association_proxy_hash

```
def association_proxy_hash(name, options = {})
  [name.to_sym, options.values_at(:node, :rel, :labels, :rel_length)].hash
end
```

#association_query_proxy

```
def association_query_proxy(name, options = {})
  self.class.send(:association_query_proxy, name, {start_object: self}.merge!(options))
end
```

Scope

ClassMethods

Constants

Files

- lib/neo4j/active_node/scope.rb:7

Methods #_call_scope_context

```
def _call_scope_context(eval_context, query_params, proc)
  if proc.arity == 1
    eval_context.instance_exec(query_params, &proc)
  else
    eval_context.instance_exec(&proc)
  end
end
```

#_scope

```
def _scope
  @_scope ||= {}
end
```

#all

```
def all(new_var = nil)
  var = new_var || (current_scope ? current_scope.node_identity : :n)
  if current_scope
    current_scope.new_link(var)
  end
end
```

```

    else
      self.as(var)
    end
  end
end

```

#current_scope**nodoc**

```

def current_scope # :nodoc:
  ScopeRegistry.value_for(:current_scope, base_class.to_s)
end

```

#current_scope=**nodoc**

```

def current_scope=(scope) # :nodoc:
  ScopeRegistry.set_value_for(:current_scope, base_class.to_s, scope)
end

```

#has_scope? rubocop:disable Style/PredicateName

```

def has_scope?(name)
  ActiveSupport::Deprecation.warn 'has_scope? is deprecated and may be removed from future releases'

  scope?(name)
end

```

#scope Similar to ActiveRecord scope

```

def scope(name, proc)
  _scope[name.to_sym] = proc

  klass = class << self; self; end
  klass.instance_eval do
    define_method(name) do |query_params = nil, _ = nil|
      eval_context = ScopeEvalContext.new(self, current_scope || self.query_proxy)
      proc = _scope[name.to_sym]
      _call_scope_context(eval_context, query_params, proc)
    end
  end
end

```

#scope? rubocop:enable Style/PredicateName

```

def scope?(name)
  _scope.key?(name.to_sym)
end

```

ScopeEvalContext**Constants****Files**

- lib/neo4j/active_node/scope.rb:93

Methods #initialize

```
def initialize(target, query_proxy)
  @query_proxy = query_proxy
  @target = target
end
```

ScopeRegistry

Stolen from ActiveRecord https://github.com/rails/rails/blob/08754f12e65a9ec79633a605e986d0f1ffa4b251/activerecord/lib/active_record

Constants

- VALID_SCOPE_TYPES

Files

- lib/neo4j/active_node/scope.rb:112

Methods #initialize

```
def initialize
  @registry = Hash.new { |hash, key| hash[key] = {} }
end
```

#set_value_for Sets the +value+ for a given +scope_type+ and +variable_name+.

```
def set_value_for(scope_type, variable_name, value)
  raise_invalid_scope_type!(scope_type)
  @registry[scope_type][variable_name] = value
end
```

#value_for Obtains the value for a given +scope_name+ and +variable_name+.

```
def value_for(scope_type, variable_name)
  raise_invalid_scope_type!(scope_type)
  @registry[scope_type][variable_name]
end
```

Constants

Files

- lib/neo4j/active_node/scope.rb:4

Methods

Labels

Provides a mapping between neo4j labels and Ruby classes

RecordNotFound

Constants

Files

- lib/neo4j/active_node/labels.rb:23

Methods

#id Returns the value of attribute id

```
def id
  @id
end
```

#initialize

```
def initialize(message = nil, model = nil, primary_key = nil, id = nil)
  @primary_key = primary_key
  @model = model
  @id = id

  super(message)
end
```

#model Returns the value of attribute model

```
def model
  @model
end
```

#primary_key Returns the value of attribute primary_key

```
def primary_key
  @primary_key
end
```

ClassMethods

Constants

Files

- lib/neo4j/active_node/labels.rb:77

Methods **#base_class**

```
def base_class
  unless self < Neo4j::ActiveNode
    fail "#{name} doesn't belong in a hierarchy descending from ActiveNode"
  end

  if superclass == Object
    self
  else

```

```
    superclass.base_class
  end
end
```

#blank?

```
def empty?
  !self.all.exists?
end
```

#count

```
def count(distinct = nil)
  fail(Neo4j::InvalidParameterError, ':count accepts `distinct` or nil as a parameter') unless distinct.nil?
  q = distinct.nil? ? 'n' : 'DISTINCT n'
  self.query_as(:n).return("count(#{q}) AS count").first.count
end
```

#delete_all Deletes all nodes and connected relationships from Cypher.

```
def delete_all
  self.neo4j_session._query("MATCH (n:`#{mapped_label_name}`) OPTIONAL MATCH n-[r]-() DELETE n,r")
  self.neo4j_session._query("MATCH (n:`#{mapped_label_name}`) DELETE n")
end
```

#destroy_all Returns each node to Ruby and calls *destroy*. Be careful, as this can be a very slow operation if you have many nodes. It will generate at least one database query per node in the database, more if callbacks require them.

```
def destroy_all
  all.each(&:destroy)
end
```

#empty?

```
def empty?
  !self.all.exists?
end
```

#exists?

```
def exists?(node_condition = nil)
  unless node_condition.is_a?(Integer) || node_condition.is_a?(Hash) || node_condition.nil?
    fail(Neo4j::InvalidParameterError, ':exists? only accepts ids or conditions')
  end
  query_start = exists_query_start(node_condition)
  start_q = query_start.respond_to?(:query_as) ? query_start.query_as(:n) : query_start
  start_q.return('COUNT(n) AS count').first.count > 0
end
```

#find Returns the object with the specified neo4j id.

```
def find(id)
  map_id = proc { |object| object.respond_to?(:id) ? object.send(:id) : object }

  result = find_by_id_or_ids(map_id, id)

  fail RecordNotFound.new(
    "Couldn't find #{name} with '#{id_property_name}'=#{id}",
    name, id_property_name, id) if result.blank?
end
```



```

    result.tap { |r| find_callbacks!(r) }
  end

```

#find_by Finds the first record matching the specified conditions. There is no implied ordering so if order matters, you should specify it yourself.

```

def find_by(values)
  all.where(values).limit(1).query_as(:n).pluck(:n).first
end

```

#find_by! Like find_by, except that if no record is found, raises a RecordNotFound error.

```

def find_by!(values)
  find_by(values) || fail(RecordNotFound, "#{self.query_as(:n).where(n: values).limit(1).to_cyph
end

```

#find_each

```

def find_each(options = {})
  self.query_as(:n).return(:n).find_each(:n, primary_key, options) do |batch|
    yield batch.n
  end
end

```

#find_in_batches

```

def find_in_batches(options = {})
  self.query_as(:n).return(:n).find_in_batches(:n, primary_key, options) do |batch|
    yield batch.map(&:n)
  end
end

```

#first Returns the first node of this class, sorted by ID. Note that this may not be the first node created since Neo4j recycles IDs.

```

def first
  self.query_as(:n).limit(1).order(n: primary_key).pluck(:n).first
end

```

#last Returns the last node of this class, sorted by ID. Note that this may not be the first node created since Neo4j recycles IDs.

```

def last
  self.query_as(:n).limit(1).order(n: {primary_key => :desc}).pluck(:n).first
end

```

#length

```

def count(distinct = nil)
  fail(Neo4j::InvalidParameterError, ':count accepts `distinct` or nil as a parameter') unless distinct
  q = distinct.nil? ? 'n' : 'DISTINCT n'
  self.query_as(:n).return("count(#{q}) AS count").first.count
end

```

#mapped_label

```

def mapped_label
  Neo4j::Label.create(mapped_label_name)
end

```

#mapped_label_name

```
def mapped_label_name
  @mapped_label_name || label_for_model
end
```

#mapped_label_names

```
def mapped_label_names
  self.ancestors.find_all { |a| a.respond_to?(:mapped_label_name) }.map { |a| a.mapped_label_name }
end
```

#size

```
def count(distinct = nil)
  fail(Neo4j::InvalidParameterError, ':count accepts `distinct` or nil as a parameter') unless distinct
  q = distinct.nil? ? 'n' : 'DISTINCT n'
  self.query_as(:n).return("count(#{q}) AS count").first.count
end
```

Index

ClassMethods

Constants

Files

- `lib/neo4j/active_node/labels/index.rb:5`

Methods

#constraint Creates a neo4j constraint on this class for given property

```
def constraint(property, constraints = {type: :unique})
  Neo4j::Session.on_next_session_available do
    declared_properties.constraint_or_fail!(property, id_property_name)
    schema_create_operation(:constraint, property, constraints)
  end
end
```

#constraint?

```
def constraint?(property)
  mapped_label.unique_constraints[:property_keys].include?([property])
end
```

#drop_constraint

```
def drop_constraint(property, constraint = {type: :unique})
  Neo4j::Session.on_next_session_available do
    declared_properties[property].unconstraint! if declared_properties[property]
    schema_drop_operation(:constraint, property, constraint)
  end
end
```

#drop_index

```
def drop_index(property, options = {})
  Neo4j::Session.on_next_session_available do
    declared_properties[property].unindex! if declared_properties[property]
    schema_drop_operation(:index, property, options)
  end
end
```

#index Creates a Neo4j index on given property

This can also be done on the property directly, see `Neo4j::ActiveNode::Property::ClassMethods#property`.

```
def index(property)
  Neo4j::Session.on_next_session_available do |_|
    declared_properties.index_or_fail!(property, id_property_name)
    schema_create_operation(:index, property)
  end
end
```

#index?

```
def index?(property)
  mapped_label.indexes[:property_keys].include?([property])
end
```

Constants

Files

- `lib/neo4j/active_node/labels/index.rb:2`

Methods

Reloading

ClassMethods

Constants

Files

- `lib/neo4j/active_node/labels/reloading.rb:12`

Methods **#before_remove_const**

```
def before_remove_const
  associations.each_value(&:queue_model_refresh!)
  MODELS_FOR_LABELS_CACHE.clear
  WRAPPED_CLASSES.each { |c| MODELS_TO_RELOAD << c.name }
  WRAPPED_CLASSES.clear
end
```

Constants

- `MODELS_TO_RELOAD`

Files

- `lib/neo4j/active_node/labels/reloading.rb:2`

Methods `.reload_models!`

```
def self.reload_models!  
  MODELS_TO_RELOAD.each(&:constantize)  
  MODELS_TO_RELOAD.clear  
end
```

Constants

- `WRAPPED_CLASSES`
- `MODELS_FOR_LABELS_CACHE`
- `MODELS_TO_RELOAD`

Files

- `lib/neo4j/active_node/labels.rb:4`
- `lib/neo4j/active_node/labels/index.rb:1`
- `lib/neo4j/active_node/labels/reloading.rb:1`

Methods

`._wrapped_classes`

```
def self._wrapped_classes  
  WRAPPED_CLASSES  
end
```

#add_label adds one or more labels

```
def add_label(*label)  
  @_persisted_obj.add_label(*label)  
end
```

`.add_wrapped_class`

```
def self.add_wrapped_class(model)  
  _wrapped_classes << model  
end
```

`.clear_wrapped_models`

```
def self.clear_wrapped_models
  WRAPPED_CLASSES.clear
  MODELS_FOR_LABELS_CACHE.clear
  Neo4j::Node::Wrapper::CONSTANTS_FOR_LABELS_CACHE.clear
end
```

#labels

```
def labels
  @_persisted_obj.labels
end
```

.model_for_labels Finds an appropriate matching model given a set of labels which are assigned to a node

```
def self.model_for_labels(labels)
  return MODELS_FOR_LABELS_CACHE[labels] if MODELS_FOR_LABELS_CACHE[labels]

  models = WRAPPED_CLASSES.select do |model|
    (model.mapped_label_names - labels).size == 0
  end

  MODELS_FOR_LABELS_CACHE[labels] = models.max do |model|
    (model.mapped_label_names & labels).size
  end
end
```

#remove_label Removes one or more labels Be careful, don't remove the label representing the Ruby class.

```
def remove_label(*label)
  @_persisted_obj.remove_label(*label)
end
```

Property**ClassMethods****Constants****Files**

- lib/neo4j/active_node/property.rb:11

Methods #association_key?

```
def association_key?(key)
  association_method_keys.include?(key.to_sym)
end
```

#extract_association_attributes! Extracts keys from attributes hash which are associations of the model TODO: Validate separately that relationships are getting the right values? Perhaps also store the values and persist relationships on save?

```
def extract_association_attributes!(attributes)
  return unless contains_association?(attributes)
  attributes.each_with_object({}) do |(key, _), result|
    result[key] = attributes.delete(key) if self.association_key?(key)
  end
end
```

```
end
end
```

Constants

- DATE_KEY_REGEX
- DEPRECATED_OBJECT_METHODS

Files

- lib/neo4j/active_node/property.rb:2

Methods

#== Performs equality checking on the result of attributes and its type.

```
def ==(other)
  return false unless other.instance_of? self.class
  attributes == other.attributes
end
```

#[]

```
def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end
```

#[]= Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

#_persisted_obj Returns the value of attribute `_persisted_obj`

```
def _persisted_obj
  @_persisted_obj
end
```

#assign_attributes Mass update a model's attributes

```
def assign_attributes(new_attributes = nil)
  return unless new_attributes.present?
  new_attributes.each do |name, value|
    writer = "#{name}="
    send(writer, value) if respond_to?(writer)
  end
end
```

#attribute_before_type_cast Read the raw attribute value

```
def attribute_before_type_cast(name)
  @attributes ||= {}
  @attributes[name.to_s]
end
```

#attributes Returns a Hash of all attributes

```
def attributes
  attributes_map { |name| send name }
end
```

#attributes= Mass update a model's attributes

```
def attributes=(new_attributes)
  assign_attributes(new_attributes)
end
```

#initialize

```
def initialize(attributes = nil)
  super(attributes)
  @attributes ||= Hash[self.class.attributes_nil_hash]
end
```

#inject_defaults!

```
def inject_defaults!(starting_props)
  return starting_props if self.class.declared_properties.declared_property_defaults.empty?
  self.class.declared_properties.inject_defaults!(self, starting_props || {})
end
```

#inspect

```
def inspect
  attribute_descriptions = inspect_attributes.map do |key, value|
    "#{Neo4j::ANSI::CYAN}#{key}: #{Neo4j::ANSI::CLEAR}#{value.inspect}"
  end.join(', ')

  separator = ' ' unless attribute_descriptions.empty?
  "<#{Neo4j::ANSI::YELLOW}#{self.class.name}#{Neo4j::ANSI::CLEAR}#{separator}#{attribute_descriptions}"
end
```

#read_attribute

```
def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end
```

#reload_properties!

```
def reload_properties!(properties)
  @attributes = nil
  convert_and_assign_attributes(properties)
end
```

#send_props

```
def send_props(hash)
  return hash if hash.blank?
  hash.each { |key, value| send("#{key}=", value) }
end
```

#write_attribute Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

Dependent

AssociationMethods

Constants

Files

- lib/neo4j/active_node/dependent/association_methods.rb:4

Methods #add_destroy_callbacks

```
def add_destroy_callbacks(model)
  return if dependent.nil?

  model.before_destroy(&method("dependent_#{dependent}_callback"))
rescue NameError
  raise "Unknown dependent option #{dependent}"
end
```

#validate_dependent

```
def validate_dependent(value)
  fail ArgumentError, "Invalid dependent value: #{value.inspect}" if not valid_dependent_value?(value)
end
```

QueryProxyMethods

methods used to resolve association dependencies

Constants

Files

- lib/neo4j/active_node/dependent/query_proxy_methods.rb:5

Methods

#each_for_destruction Used as part of *dependent: :destroy* and may not have any utility otherwise. It keeps track of the node responsible for a cascading *destroy* process. but this is not always available, so we require it explicitly.


```

def each_for_destruction(owning_node)
  target = owning_node.called_by || owning_node
  objects = pluck(identity).compact.reject do |obj|
    target.dependent_children.include?(obj)
  end

  objects.each do |obj|
    obj.called_by = target
    target.dependent_children << obj
    yield obj
  end
end

```

#unique_nodes This will match nodes who only have a single relationship of a given type. It's used by *dependent::delete_orphans* and *dependent::destroy_orphans* and may not have much utility otherwise.

```

def unique_nodes(association, self_idenfier, other_node, other_rel)
  fail 'Only supported by in QueryProxy chains started by an instance' unless source_object
  return false if send(association.name).empty?
  unique_nodes_query(association, self_idenfier, other_node, other_rel)
    .proxy_as(association.target_class, other_node)
end

```

Constants

Files

- lib/neo4j/active_node/dependent.rb:3
- lib/neo4j/active_node/dependent/association_methods.rb:3
- lib/neo4j/active_node/dependent/query_proxy_methods.rb:3

Methods

#called_by= Sets the attribute called_by

```

def called_by=(value)
  @called_by = value
end

```

#dependent_children

```

def dependent_children
  @dependent_children ||= []
end

```

Callbacks

nodoc

Constants

Files

- lib/neo4j/active_node/callbacks.rb:3

Methods

#conditional_callback Allows you to perform a callback if a condition is not satisfied.

```
def conditional_callback(kind, guard)
  return yield if guard
  run_callbacks(kind) { yield }
end
```

#destroy

nodoc

```
def destroy #:nodoc:
  tx = Neo4j::Transaction.new
  run_callbacks(:destroy) { super }
rescue
  @_deleted = false
  @attributes = @attributes.dup
  tx.mark_failed
  raise
ensure
  tx.close if tx
end
```

#initialize

```
def initialize(args = nil)
  run_callbacks(:initialize) { super }
end
```

#touch

nodoc

```
def touch #:nodoc:
  run_callbacks(:touch) { super }
end
```

Initialize

Constants

Files

- lib/neo4j/active_node/initialize.rb:1

Methods

#called_by Returns the value of attribute called_by

```
def called_by
  @called_by
end
```

#init_on_load called when loading the node from the database

```
def init_on_load(persisted_node, properties)
  self.class.extract_association_attributes!(properties)
  @_persisted_obj = persisted_node
  changed_attributes && changed_attributes.clear
  @attributes = convert_and_assign_attributes(properties)
end
```

#init_on_reload

```
def init_on_reload(reloaded)
  @attributes = nil
  init_on_load(reloaded, reloaded.props)
end
```

#wrapper Implements the Neo4j::Node#wrapper and Neo4j::Relationship#wrapper method so that we don't have to care if the node is wrapped or not.

```
def wrapper
  self
end
```

Reflection

A reflection contains information about an association. They are often used in connection with form builders to determine associated classes. This module contains methods related to the creation and retrieval of reflections.

ClassMethods

Adds methods to the class related to creating and retrieving reflections.

Constants

Files

- `lib/neo4j/active_node/reflection.rb:14`

Methods

#reflect_on_all_associations Returns an array containing one reflection for each association declared in the model.

```
def reflect_on_all_associations(macro = nil)
  association_reflections = reflections.values
  macro ? association_reflections.select { |reflection| reflection.macro == macro } : association_reflections
end
```

#reflect_on_association

```
def reflect_on_association(association)
  reflections[association.to_sym]
end
```

AssociationReflection

The actual reflection object that contains information about the given association. These should never need to be created manually, they will always be created by declaring a `:has_many` or `:has_one` association on a model.

Constants**Files**

- `lib/neo4j/active_node/reflection.rb:39`

Methods

#association The association object referenced by this reflection

```
def association
  @association
end
```

#class_name Returns the name of the target model

```
def class_name
  @class_name ||= association.target_class.name
end
```

#collection?

```
def collection?
  macro == :has_many
end
```

#initialize

```
def initialize(macro, name, association)
  @macro      = macro
  @name       = name
  @association = association
end
```

#klass Returns the target model

```
def klass
  @klass ||= class_name.constantize
end
```

#macro The type of association

```
def macro
  @macro
end
```

#name The name of the association

```
def name
  @name
end
```

#rel_class_name

```
def rel_class_name
  @rel_class_name ||= association.relationship_class.name.to_s
end
```

#rel_class

```
def rel_class
  @rel_class ||= rel_class_name.constantize
end
```

#type

```
def type
  @type ||= association.relationship_type
end
```

#validate?

```
def validate?
  true
end
```

Constants**Files**

- lib/neo4j/active_node/reflection.rb:5

Methods**Validations**

This mixin replace the original save method and performs validation before the save.

ClassMethods**Constants****Files**

- lib/neo4j/active_node/validations.rb:16

Methods #validates_uniqueness_of

```
def validates_uniqueness_of(*attr_names)
  validates_with UniquenessValidator, _merge_attributes(attr_names)
end
```

UniquenessValidator

Constants

Files

- `lib/neo4j/active_node/validations.rb:23`

Methods `#found`

```
def found(record, attribute, value)
  conditions = scope_conditions(record)

  # TODO: Added as find(:name => nil) throws error
  value = '' if value.nil?

  conditions[attribute] = options[:case_sensitive] ? value : /#{Regexp.escape(value.to_s)}/i

  found = record.class.as(:result).where(conditions)
  found = found.where_not(neo_id: record.neo_id) if record._persisted_obj
  found
end
```

`#initialize`

```
def initialize(options)
  super(options.reverse_merge(case_sensitive: true))
end
```

`#message`

```
def message(instance)
  super || 'has already been taken'
end
```

`#scope_conditions`

```
def scope_conditions(instance)
  Array(options[:scope] || []).inject({}) do |conditions, key|
    conditions.merge(key => instance[key])
  end
end
```

`#validate_each`

```
def validate_each(record, attribute, value)
  return unless found(record, attribute, value).exists?

  record.errors.add(attribute, :taken, options.except(:case_sensitive, :scope).merge(value: value))
end
```

Constants

Files

- `lib/neo4j/active_node/validations.rb:4`

Methods

#read_attribute_for_validation Implements the ActiveRecord::Validation hook method.

```
def read_attribute_for_validation(key)
  respond_to?(key) ? send(key) : self[key]
end
```

#save The validation process on save can be skipped by passing false. The regular Model#save method is replaced with this when the validations module is mixed in, which it is by default.

```
def save(options = {})
  result = perform_validations(options) ? super : false
  if !result
    Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  end
  result
end
```

#valid?

```
def valid?(context = nil)
  context ||= (new_record? ? :create : :update)
  super(context)
  errors.empty?
end
```

OrmAdapter

ClassMethods

Constants

Files

- lib/neo4j/active_node/orm_adapter.rb:10

Methods

Constants

Files

- lib/neo4j/active_node/orm_adapter.rb:9

Methods

#column_names

```
def column_names
  klass._decl_props.keys
end
```

#create! Create a model using attributes

```
def create!(attributes = {})  
  klass.create!(attributes)  
end
```

#destroy

```
def destroy(object)  
  object.destroy && true if valid_object?(object)  
end
```

#find_all Find all models matching conditions

```
def find_all(options = {})  
  conditions, order, limit, offset = extract_conditions!(options)  
  extract_id!(conditions)  
  order = hasherize_order(order)  
  
  result = klass.where(conditions)  
  result = result.order(order) unless order.empty?  
  result = result.skip(offset) if offset  
  result = result.limit(limit) if limit  
  result.to_a  
end
```

#find_first Find the first instance matching conditions

```
def find_first(options = {})  
  conditions, order = extract_conditions!(options)  
  extract_id!(conditions)  
  order = hasherize_order(order)  
  
  result = klass.where(conditions)  
  result = result.order(order) unless order.empty?  
  result.first  
end
```

#get Get an instance by id of the model

```
def get(id)  
  klass.find_by(klass.id_property_name => wrap_key(id))  
end
```

#get! Get an instance by id of the model

```
def get!(id)  
  klass.find(wrap_key(id)).tap do |node|  
    fail 'No record found' if node.nil?  
  end  
end
```

#i18n_scope

```
def i18n_scope  
  :neo4j  
end
```


Persistence

RecordInvalidError

Constants

Files

- lib/neo4j/active_node/persistence.rb:3

Methods #initialize

```
def initialize(record)
  @record = record
  super(@record.errors.full_messages.join(', '))
end
```

#record Returns the value of attribute record

```
def record
  @record
end
```

ClassMethods

Constants

Files

- lib/neo4j/active_node/persistence.rb:102

Methods

#create Creates and saves a new node

```
def create(props = {})
  association_props = extract_association_attributes!(props) || {}
  new(props).tap do |obj|
    yield obj if block_given?
    obj.save
    association_props.each do |prop, value|
      obj.send("#{prop}=", value)
    end
  end
end
```

#create! Same as #create, but raises an error if there is a problem during save.

```
def create!(*args)
  props = args[0] || {}
  association_props = extract_association_attributes!(props) || {}

  new(*args).tap do |o|
    yield o if block_given?
    o.save!
  end
end
```

```
      association_props.each do |prop, value|
        o.send("#{prop}=", value)
      end
    end
  end
end
```

#find_or_create

```
def find_or_create(find_attributes, set_attributes = {})
  on_create_attributes = set_attributes.reverse_merge(on_create_props(find_attributes))

  neo4j_session.query.merge(n: {self.mapped_label_names => find_attributes})
    .on_create_set(n: on_create_attributes)
    .pluck(:n).first
end
```

#find_or_create_by Finds the first node with the given attributes, or calls create if none found

```
def find_or_create_by(attributes, &block)
  find_by(attributes) || create(attributes, &block)
end
```

#find_or_create_by! Same as #find_or_create_by, but calls #create! so it raises an error if there is a problem during save.

```
def find_or_create_by!(attributes, &block)
  find_by(attributes) || create!(attributes, &block)
end
```

#load_entity

```
def load_entity(id)
  Neo4j::Node.load(id)
end
```

#merge

```
def merge(match_attributes, optional_attrs = {})
  options = [:on_create, :on_match, :set]
  optional_attrs.assert_valid_keys(*options)

  optional_attrs.default = {}
  on_create_attrs, on_match_attrs, set_attrs = optional_attrs.values_at(*options)

  neo4j_session.query.merge(n: {self.mapped_label_names => match_attributes})
    .on_create_set(n: on_create_props(on_create_attrs))
    .on_match_set(n: on_match_props(on_match_attrs))
    .break.set(n: set_attrs)
    .pluck(:n).first
end
```

Constants

Files

- [lib/neo4j/active_node/persistence.rb:2](#)

Methods

#_create_node TODO: This does not seem like it should be the responsibility of the node. Creates an unwrapped node in the database.

```
def _create_node(node_props, labels = labels_for_create)
  self.class.neo4j_session.create_node(node_props, labels)
end
```

#apply_default_values

```
def apply_default_values
  return if self.class.declared_property_defaults.empty?
  self.class.declared_property_defaults.each_pair do |key, value|
    self.send("#{key}=", value) if self.send(key).nil?
  end
end
```

#cache_key

```
def cache_key
  if self.new_record?
    "#{model_cache_key}/new"
  elsif self.respond_to?(:updated_at) && !self.updated_at.blank?
    "#{model_cache_key}/#{neo_id}-#{self.updated_at.utc.to_s(:number)}"
  else
    "#{model_cache_key}/#{neo_id}"
  end
end
```

#concurrent_increment! Increments concurrently a numeric attribute by a certain amount

```
def concurrent_increment!(attribute, by = 1)
  query_node = Neo4j::Session.query.match_nodes(n: neo_id)
  increment_by_query! query_node, attribute, by
end
```

#create_model Creates a model with values matching those of the instance attributes and returns its id.

```
def create_model
  node = _create_node(props_for_create)
  init_on_load(node, node.props)
  @deferred_nodes = nil
  true
end
```

#create_or_update

```
def create_or_update
  # since the same model can be created or updated twice from a relationship we have to have thi
  @_create_or_updating = true
  apply_default_values
  result = _persisted_obj ? update_model : create_model
  if result == false
    Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  else
    true
  end
rescue => e
```

```
Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  raise e
ensure
  @_create_or_updating = nil
end
```

#destroy

```
def destroy
  freeze
  _persisted_obj && _persisted_obj.del
  @_deleted = true
end
```

#destroyed? Returns +true+ if the object was destroyed.

```
def destroyed?
  @_deleted
end
```

#exist?

```
def exist?
  _persisted_obj && _persisted_obj.exist?
end
```

#freeze

```
def freeze
  @attributes.freeze
  self
end
```

#frozen?

```
def frozen?
  @attributes.frozen?
end
```

#increment Increments a numeric attribute by a certain amount

```
def increment(attribute, by = 1)
  self[attribute] ||= 0
  self[attribute] += by
  self
end
```

#increment! Convenience method to increment numeric attribute and #save at the same time

```
def increment!(attribute, by = 1)
  increment(attribute, by).update_attribute(attribute, self[attribute])
end
```

#inject_primary_key! As the name suggests, this inserts the primary key (id property) into the properties hash. The method called here, *default_property_values*, is a holdover from an earlier version of the gem. It does NOT contain the default values of properties, it contains the Default Property, which we now refer to as the ID Property. It will be deprecated and renamed in a coming refactor.

```
def inject_primary_key!(converted_props)
  self.class.default_property_values(self).tap do |destination_props|
    destination_props.merge!(converted_props) if converted_props.is_a?(Hash)
  end
end
```

```

    end
  end

```

#labels_for_create

```

def labels_for_create
  self.class.mapped_label_names
end

```

#new? Returns +true+ if the record hasn't been saved to Neo4j yet.

```

def new_record?
  !_persisted_obj
end

```

#new_record? Returns +true+ if the record hasn't been saved to Neo4j yet.

```

def new_record?
  !_persisted_obj
end

```

#persisted? Returns +true+ if the record is persisted, i.e. it's not a new record and it was not destroyed

```

def persisted?
  !new_record? && !destroyed?
end

```

#props

```

def props
  attributes.reject { |_, v| v.nil? }.symbolize_keys
end

```

#props_for_create Returns a hash containing: * All properties and values for insertion in the database * A *uuid* (or equivalent) key and value * Timestamps, if the class is set to include them. Note that the UUID is added to the hash but is not set on the node. The timestamps, by comparison, are set on the node prior to addition in this hash.

```

def props_for_create
  inject_timestamps!
  props_with_defaults = inject_defaults!(props)
  converted_props = props_for_db(props_with_defaults)
  return converted_props unless self.class.respond_to?(:default_property_values)
  inject_primary_key!(converted_props)
end

```

#props_for_persistence

```

def props_for_persistence
  _persisted_obj ? props_for_update : props_for_create
end

```

#props_for_update

```

def props_for_update
  update_magic_properties
  changed_props = attributes.select { |k, _| changed_attributes.include?(k) }
  changed_props.symbolize_keys!
  inject_defaults!(changed_props)
  props_for_db(changed_props)
end

```

#reload

```
def reload
  return self if new_record?
  association_proxy_cache.clear if respond_to?(:association_proxy_cache)
  changed_attributes && changed_attributes.clear
  unless reload_from_database
    @_deleted = true
    freeze
  end
  self
end
```

#reload_from_database

```
def reload_from_database
  reloaded = self.class.load_entity(neo_id)
  reloaded ? init_on_reload(reloaded._persisted_obj) : nil
end
```

#save Saves the model.

If the model is new a record gets created in the database, otherwise the existing record gets updated. If `perform_validation` is true validations run. If any of them fail the action is cancelled and save returns false. If the flag is false validations are bypassed altogether. See `ActiveRecord::Validations` for more information. There's a series of callbacks associated with save. If any of the `before_*` callbacks return false the action is cancelled and save returns false.

```
def save(*)
  cascade_save do
    association_proxy_cache.clear
    create_or_update
  end
end
```

#save! Persist the object to the database. Validations and Callbacks are included by default but validation can be disabled by passing `:validate => false` to `#save!`. Creates a new transaction.

```
def save!(*args)
  save(*args) or fail(RecordInvalidError, self) # rubocop:disable Style/AndOr
end
```

#touch

```
def touch
  fail 'Cannot touch on a new record object' unless persisted?
  update_attribute!(:updated_at, Time.now) if respond_to?(:updated_at=)
end
```

#update Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end
```

#update! Same as `{#update_attributes}`, but raises an exception if saving fails.

```
def update!(attributes)
  self.attributes = process_attributes(attributes)
```

```

    save!
  end

```

#update_attribute Convenience method to set attribute and #save at the same time

```

def update_attribute(attribute, value)
  send("#{attribute}=", value)
  self.save
end

```

#update_attribute! Convenience method to set attribute and #save! at the same time

```

def update_attribute!(attribute, value)
  send("#{attribute}=", value)
  self.save!
end

```

#update_attributes Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```

def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end

```

#update_attributes! Same as {#update_attributes}, but raises an exception if saving fails.

```

def update!(attributes)
  self.attributes = process_attributes(attributes)
  save!
end

```

#update_model

```

def update_model
  return if !changed_attributes || changed_attributes.empty?
  _persisted_obj.update_props(props_for_update)
  changed_attributes.clear
end

```

IdProperty

This module makes it possible to use other IDs than the build it neo4j id (neo_id)

TypeMethods

Constants

Files

- `lib/neo4j/active_node/id_property.rb:39`

Methods #define_id_methods

```
def define_id_methods(clazz, name, conf)
  validate_conf!(conf)

  if conf[:on]
    define_custom_method(clazz, name, conf[:on])
  elsif conf[:auto]
    define_uuid_method(clazz, name)
  elsif conf.empty?
    define_property_method(clazz, name)
  end
end
```

.define_id_methods

```
def define_id_methods(clazz, name, conf)
  validate_conf!(conf)

  if conf[:on]
    define_custom_method(clazz, name, conf[:on])
  elsif conf[:auto]
    define_uuid_method(clazz, name)
  elsif conf.empty?
    define_property_method(clazz, name)
  end
end
```

ClassMethods

Constants

Files

- `lib/neo4j/active_node/id_property.rb:124`

Methods `#find_by_id`

```
def find_by_id(id)
  all.where(id_property_name => id).first
end
```

#find_by_ids

```
def find_by_ids(ids)
  all.where(id_property_name => ids).to_a
end
```

#find_by_neo_id

```
def find_by_neo_id(id)
  Neo4j::Node.load(id)
end
```

#has_id_property? `rubocop:disable Style/PredicateName`

```
def has_id_property?
  ActiveSupport::Deprecation.warn 'has_id_property? is deprecated and may be removed from future
```



```

    id_property?
  end

```

#id_property

```

def id_property(name, conf = {})
  self.manual_id_property = true
  Neo4j::Session.on_next_session_available do |_|
    @id_property_info = {name: name, type: conf}
    TypeMethods.define_id_methods(self, name, conf)
    constraint(name, type: :unique) unless conf[:constraint] == false
  end
end

```

#id_property? rubocop:disable Style/PredicateName

```

def id_property?
  id_property_info && !id_property_info.empty?
end

```

#id_property_info

```

def id_property_info
  @id_property_info ||= {}
end

```

#id_property_name

```

def id_property_name
  id_property_info[:name]
end

```

#manual_id_property Returns the value of attribute manual_id_property

```

def manual_id_property
  @manual_id_property
end

```

#manual_id_property= Sets the attribute manual_id_property

```

def manual_id_property=(value)
  @manual_id_property = value
end

```

#manual_id_property?

```

def manual_id_property?
  !!manual_id_property
end

```

#primary_key

```

def id_property_name
  id_property_info[:name]
end

```

Accessor

Provides get/set of the Id Property values. Some methods

ClassMethods

Constants

Files

- `lib/neo4j/active_node/id_property/accessor.rb:26`

Methods #default_properties

```
def default_properties
  @default_property ||= {}
end
```

#default_properties_keys

```
def default_properties_keys
  @default_properties_keys ||= default_properties.keys
end
```

#default_property TODO: Move this to the DeclaredProperties

```
def default_property(name, &block)
  reset_default_properties(name) if default_properties.respond_to?(:size)
  default_properties[name] = block
end
```

#default_property_key

```
def default_property_key
  @default_property_key ||= default_properties_keys.first
end
```

#default_property_values

```
def default_property_values(instance)
  default_properties.each_with_object({}) do |(key, block), result|
    result[key] = block.call(instance)
  end
end
```

#reset_default_properties

```
def reset_default_properties(name_to_keep)
  default_properties.each_key do |property|
    @default_properties_keys = nil
    undef_method(property) unless property == name_to_keep
  end
  @default_properties_keys = nil
  @default_property = {}
end
```

Constants

Files

- `lib/neo4j/active_node/id_property/accessor.rb:4`

Methods #default_properties

```
def default_properties
  @default_properties ||= Hash.new(nil)
end
```

#default_properties=

```
def default_properties=(properties)
  @default_property_value = properties[default_property_key]
end
```

#default_property

```
def default_property(key)
  return nil unless key == default_property_key
  default_property_value
end
```

#default_property_key

```
def default_property_key
  self.class.default_property_key
end
```

#default_property_value Returns the value of attribute default_property_value

```
def default_property_value
  @default_property_value
end
```

Constants**Files**

- lib/neo4j/active_node/id_property.rb:35
- lib/neo4j/active_node/id_property/accessor.rb:1

Methods**#default_properties**

```
def default_properties
  @default_properties ||= Hash.new(nil)
end
```

#default_properties=

```
def default_properties=(properties)
  @default_property_value = properties[default_property_key]
end
```

#default_property

```
def default_property(key)
  return nil unless key == default_property_key
  default_property_value
end
```

#default_property_key

```
def default_property_key
  self.class.default_property_key
end
```

#default_property_value Returns the value of attribute default_property_value

```
def default_property_value
  @default_property_value
end
```

Unpersisted**Constants****Files**

- lib/neo4j/active_node/unpersisted.rb:3

Methods**#clear_deferred_nodes_for_association**

```
def clear_deferred_nodes_for_association(association_name)
  deferred_nodes_for_association(association_name.to_sym).clear
end
```

#defer_create

```
def defer_create(association_name, object, options = {})
  clear_deferred_nodes_for_association(association_name) if options[:clear]

  deferred_nodes_for_association(association_name) << object
end
```

#deferred_create_cache The values in this Hash are returned and used outside by reference so any modifications to the Array should be in-place

```
def deferred_create_cache
  @deferred_create_cache ||= {}
end
```

#deferred_nodes_for_association

```
def deferred_nodes_for_association(association_name)
  deferred_create_cache[association_name.to_sym] ||= []
end
```

#pending_deferred_creations?

```
def pending_deferred_creations?
  !deferred_create_cache.values.all?(&:empty?)
end
```

QueryMethods

Constants

Files

- lib/neo4j/active_node/query_methods.rb:3

Methods

#blank?

```
def empty?
  !self.all.exists?
end
```

#count

```
def count(distinct = nil)
  fail(Neo4j::InvalidParameterError, ':count accepts `distinct` or nil as a parameter') unless distinct.nil?
  q = distinct.nil? ? 'n' : 'DISTINCT n'
  self.query_as(:n).return("count(#{q}) AS count").first.count
end
```

#empty?

```
def empty?
  !self.all.exists?
end
```

#exists?

```
def exists?(node_condition = nil)
  unless node_condition.is_a?(Integer) || node_condition.is_a?(Hash) || node_condition.nil?
    fail(Neo4j::InvalidParameterError, ':exists? only accepts ids or conditions')
  end
  query_start = exists_query_start(node_condition)
  start_q = query_start.respond_to?(:query_as) ? query_start.query_as(:n) : query_start
  start_q.return('COUNT(n) AS count').first.count > 0
end
```

#find_each

```
def find_each(options = {})
  self.query_as(:n).return(:n).find_each(:n, primary_key, options) do |batch|
    yield batch.n
  end
end
```

#find_in_batches

```
def find_in_batches(options = {})
  self.query_as(:n).return(:n).find_in_batches(:n, primary_key, options) do |batch|
    yield batch.map(&:n)
  end
end
```

#first Returns the first node of this class, sorted by ID. Note that this may not be the first node created since Neo4j recycles IDs.

```
def first
  self.query_as(:n).limit(1).order(n: primary_key).pluck(:n).first
end
```

#last Returns the last node of this class, sorted by ID. Note that this may not be the first node created since Neo4j recycles IDs.

```
def last
  self.query_as(:n).limit(1).order(n: {primary_key => :desc}).pluck(:n).first
end
```

#length

```
def count(distinct = nil)
  fail(Neo4j::InvalidParameterError, ':count accepts `distinct` or nil as a parameter') unless distinct
  q = distinct.nil? ? 'n' : 'DISTINCT n'
  self.query_as(:n).return("count(#{q}) AS count").first.count
end
```

#size

```
def count(distinct = nil)
  fail(Neo4j::InvalidParameterError, ':count accepts `distinct` or nil as a parameter') unless distinct
  q = distinct.nil? ? 'n' : 'DISTINCT n'
  self.query_as(:n).return("count(#{q}) AS count").first.count
end
```

Constants

- MARSHAL_INSTANCE_VARIABLES
- WRAPPED_CLASSES
- MODELS_FOR_LABELS_CACHE
- MODELS_TO_RELOAD
- DATE_KEY_REGEX
- DEPRECATED_OBJECT_METHODS

Files

- lib/neo4j/active_node.rb:23
- lib/neo4j/active_node/enum.rb:1
- lib/neo4j/active_node/rels.rb:1
- lib/neo4j/active_node/query.rb:2
- lib/neo4j/active_node/has_n.rb:1
- lib/neo4j/active_node/scope.rb:3
- lib/neo4j/active_node/labels.rb:2
- lib/neo4j/active_node/property.rb:1
- lib/neo4j/active_node/dependent.rb:2
- lib/neo4j/active_node/callbacks.rb:2

- lib/neo4j/active_node/reflection.rb:1
- lib/neo4j/active_node/validations.rb:2
- lib/neo4j/active_node/orm_adapter.rb:4
- lib/neo4j/active_node/persistence.rb:1
- lib/neo4j/active_node/id_property.rb:1
- lib/neo4j/active_node/unpersisted.rb:2
- lib/neo4j/active_node/query_methods.rb:2
- lib/neo4j/active_node/has_n/association.rb:5
- lib/neo4j/active_node/query/query_proxy.rb:2
- lib/neo4j/active_node/query/query_proxy_link.rb:2
- lib/neo4j/active_node/query/query_proxy_methods.rb:2
- lib/neo4j/active_node/query/query_proxy_enumerable.rb:2
- lib/neo4j/active_node/dependent/association_methods.rb:2
- lib/neo4j/active_node/dependent/query_proxy_methods.rb:2
- lib/neo4j/active_node/query/query_proxy_eager_loading.rb:2
- lib/neo4j/active_node/has_n/association_cypher_methods.rb:2
- lib/neo4j/active_node/query/query_proxy_find_in_batches.rb:2
- lib/neo4j/active_node/query/query_proxy_methods_of_mass Updating.rb:2

Methods

#== Performs equality checking on the result of attributes and its type.

```
def ==(other)
  return false unless other.instance_of? self.class
  attributes == other.attributes
end
```

#[]

```
def read_attribute(name)
  respond_to?(name) ? send(name) : nil
end
```

#[]= Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

#_create_node TODO: This does not seem like it should be the responsibility of the node. Creates an unwrapped node in the database.

```
def _create_node(node_props, labels = labels_for_create)
  self.class.neo4j_session.create_node(node_props, labels)
end
```

#_persisted_obj Returns the value of attribute `_persisted_obj`

```
def _persisted_obj
  @_persisted_obj
end
```

#_rels_delegator

```
def _rels_delegator
  fail "Can't access relationship on a non persisted node" unless _persisted_obj
  _persisted_obj
end
```

#add_label adds one or more labels

```
def add_label(*label)
  @_persisted_obj.add_label(*label)
end
```

#apply_default_values

```
def apply_default_values
  return if self.class.declared_property_defaults.empty?
  self.class.declared_property_defaults.each_pair do |key, value|
    self.send("#{key}=", value) if self.send(key).nil?
  end
end
```

#as Starts a new QueryProxy with the starting identifier set to the given argument and QueryProxy `source_object` set to the node instance. This method does not exist within QueryProxy and can only be used to start a new chain.

```
def as(node_var)
  self.class.query_proxy(node: node_var, source_object: self).match_to(self)
end
```

#assign_attributes Mass update a model's attributes

```
def assign_attributes(new_attributes = nil)
  return unless new_attributes.present?
  new_attributes.each do |name, value|
    writer = "#{name}="
    send(writer, value) if respond_to?(writer)
  end
end
```

#association_proxy

```
def association_proxy(name, options = {})
  name = name.to_sym
  hash = association_proxy_hash(name, options)
  association_proxy_cache_fetch(hash) do
    if result_cache = self.instance_variable_get('@source_proxy_result_cache')
      result_by_previous_id = previous_proxy_results_by_previous_id(result_cache, name)

      result_cache.inject(nil) do |proxy_to_return, object|
        proxy = fresh_association_proxy(name, options.merge(start_object: object), result_by_pre
```



```

        object.association_proxy_cache[hash] = proxy

        (self == object ? proxy : proxy_to_return)
      end
    else
      fresh_association_proxy(name, options)
    end
  end
end
end

```

#association_proxy_cache Returns the current AssociationProxy cache for the association cache. It is in the format { :association_name => AssociationProxy } This is so that we * don't need to re-build the QueryProxy objects * also because the QueryProxy object caches it's results * so we don't need to query again * so that we can cache results from association calls or eager loading

```

def association_proxy_cache
  @association_proxy_cache ||= {}
end

```

#association_proxy_cache_fetch

```

def association_proxy_cache_fetch(key)
  association_proxy_cache.fetch(key) do
    value = yield
    association_proxy_cache[key] = value
  end
end

```

#association_proxy_hash

```

def association_proxy_hash(name, options = {})
  [name.to_sym, options.values_at(:node, :rel, :labels, :rel_length)].hash
end

```

#association_query_proxy

```

def association_query_proxy(name, options = {})
  self.class.send(:association_query_proxy, name, {start_object: self}.merge!(options))
end

```

#attribute_before_type_cast Read the raw attribute value

```

def attribute_before_type_cast(name)
  @attributes ||= {}
  @attributes[name.to_s]
end

```

#attributes Returns a Hash of all attributes

```

def attributes
  attributes_map { |name| send name }
end

```

#attributes= Mass update a model's attributes

```

def attributes=(new_attributes)
  assign_attributes(new_attributes)
end

```

#cache_key

```
def cache_key
  if self.new_record?
    "#{model_cache_key}/new"
  elsif self.respond_to?(:updated_at) && !self.updated_at.blank?
    "#{model_cache_key}/#{neo_id}-#{self.updated_at.utc.to_s(:number)}"
  else
    "#{model_cache_key}/#{neo_id}"
  end
end
```

#called_by Returns the value of attribute called_by

```
def called_by
  @called_by
end
```

#called_by= Sets the attribute called_by

```
def called_by=(value)
  @called_by = value
end
```

#clear_deferred_nodes_for_association

```
def clear_deferred_nodes_for_association(association_name)
  deferred_nodes_for_association(association_name.to_sym).clear
end
```

#concurrent_increment! Increments concurrently a numeric attribute by a certain amount

```
def concurrent_increment!(attribute, by = 1)
  query_node = Neo4j::Session.query.match_nodes(n: neo_id)
  increment_by_query! query_node, attribute, by
end
```

#conditional_callback Allows you to perform a callback if a condition is not satisfied.

```
def conditional_callback(kind, guard)
  return yield if guard
  run_callbacks(kind) { yield }
end
```

#declared_properties

```
def declared_properties
  self.class.declared_properties
end
```

#default_properties

```
def default_properties
  @default_properties ||= Hash.new(nil)
end
```

#default_properties=

```
def default_properties=(properties)
  @default_property_value = properties[default_property_key]
end
```

#default_property

```
def default_property(key)
  return nil unless key == default_property_key
  default_property_value
end
```

#default_property_key

```
def default_property_key
  self.class.default_property_key
end
```

#default_property_value Returns the value of attribute default_property_value

```
def default_property_value
  @default_property_value
end
```

#defer_create

```
def defer_create(association_name, object, options = {})
  clear_deferred_nodes_for_association(association_name) if options[:clear]

  deferred_nodes_for_association(association_name) << object
end
```

#deferred_create_cache The values in this Hash are returned and used outside by reference so any modifications to the Array should be in-place

```
def deferred_create_cache
  @deferred_create_cache ||= {}
end
```

#deferred_nodes_for_association

```
def deferred_nodes_for_association(association_name)
  deferred_create_cache[association_name.to_sym] ||= []
end
```

#dependent_children

```
def dependent_children
  @dependent_children ||= []
end
```

#destroy

nodoc

```
def destroy #:nodoc:
  tx = Neo4j::Transaction.new
  run_callbacks(:destroy) { super }
rescue
  @_deleted = false
  @attributes = @attributes.dup
  tx.mark_failed
  raise
ensure
  tx.close if tx
end
```

#destroyed? Returns +true+ if the object was destroyed.

```
def destroyed?  
  @_deleted  
end
```

#eq?

```
def ==(other)  
  other.class == self.class && other.id == id  
end
```

#exist?

```
def exist?  
  _persisted_obj && _persisted_obj.exist?  
end
```

#freeze

```
def freeze  
  @attributes.freeze  
  self  
end
```

#frozen?

```
def frozen?  
  @attributes.frozen?  
end
```

#hash

```
def hash  
  id.hash  
end
```

#id

```
def id  
  id = neo_id  
  id.is_a?(Integer) ? id : nil  
end
```

#increment Increments a numeric attribute by a certain amount

```
def increment(attribute, by = 1)  
  self[attribute] ||= 0  
  self[attribute] += by  
  self  
end
```

#increment! Convenience method to increment numeric attribute and #save at the same time

```
def increment!(attribute, by = 1)  
  increment(attribute, by).update_attribute(attribute, self[attribute])  
end
```

#init_on_load called when loading the node from the database

```
def init_on_load(persisted_node, properties)  
  self.class.extract_association_attributes!(properties)  
  @_persisted_obj = persisted_node  
  changed_attributes && changed_attributes.clear
```

```
@attributes = convert_and_assign_attributes(properties)
end
```

#init_on_reload

```
def init_on_reload(reloaded)
  @attributes = nil
  init_on_load(reloaded, reloaded.props)
end
```

#initialize

```
def initialize(args = nil)
  symbol_args = args.is_a?(Hash) ? args.symbolize_keys : args
  super(symbol_args)
end
```

#inject_defaults!

```
def inject_defaults!(starting_props)
  return starting_props if self.class.declared_properties.declared_property_defaults.empty?
  self.class.declared_properties.inject_defaults!(self, starting_props || {})
end
```

#inject_primary_key! As the name suggests, this inserts the primary key (id property) into the properties hash. The method called here, *default_property_values*, is a holdover from an earlier version of the gem. It does NOT contain the default values of properties, it contains the Default Property, which we now refer to as the ID Property. It will be deprecated and renamed in a coming refactor.

```
def inject_primary_key!(converted_props)
  self.class.default_property_values(self).tap do |destination_props|
    destination_props.merge!(converted_props) if converted_props.is_a?(Hash)
  end
end
```

#inspect

```
def inspect
  attribute_descriptions = inspect_attributes.map do |key, value|
    "#{Neo4j::ANSI::CYAN}#{key}: #{Neo4j::ANSI::CLEAR}#{value.inspect}"
  end.join(', ')

  separator = ' ' unless attribute_descriptions.empty?
  "<#{Neo4j::ANSI::YELLOW}#{self.class.name}#{Neo4j::ANSI::CLEAR}#{separator}#{attribute_descriptions}"
end
```

#labels

```
def labels
  @_persisted_obj.labels
end
```

#labels_for_create

```
def labels_for_create
  self.class.mapped_label_names
end
```

#marshal_dump

```
def marshal_dump
  marshal_instance_variables.map(&method(:instance_variable_get))
end
```

#marshal_load

```
def marshal_load(array)
  marshal_instance_variables.zip(array).each do |var, value|
    instance_variable_set(var, value)
  end
end
```

#neo4j_obj

```
def neo4j_obj
  _persisted_obj || fail('Tried to access native neo4j object on a non persisted object')
end
```

#neo_id

```
def neo_id
  _persisted_obj ? _persisted_obj.neo_id : nil
end
```

#new? Returns +true+ if the record hasn't been saved to Neo4j yet.

```
def new_record?
  !_persisted_obj
end
```

#new_record? Returns +true+ if the record hasn't been saved to Neo4j yet.

```
def new_record?
  !_persisted_obj
end
```

#pending_deferred_creations?

```
def pending_deferred_creations?
  !deferred_create_cache.values.all?(&:empty?)
end
```

#persisted? Returns +true+ if the record is persisted, i.e. it's not a new record and it was not destroyed

```
def persisted?
  !new_record? && !destroyed?
end
```

#props

```
def props
  attributes.reject { |_, v| v.nil? }.symbolize_keys
end
```

#props_for_create Returns a hash containing: * All properties and values for insertion in the database * A *uuid* (or equivalent) key and value * Timestamps, if the class is set to include them. Note that the UUID is added to the hash but is not set on the node. The timestamps, by comparison, are set on the node prior to addition in this hash.

```
def props_for_create
  inject_timestamps!
  props_with_defaults = inject_defaults!(props)
end
```

```

    converted_props = props_for_db(props_with_defaults)
    return converted_props unless self.class.respond_to?(:default_property_values)
    inject_primary_key!(converted_props)
  end

```

#props_for_persistence

```

  def props_for_persistence
    _persisted_obj ? props_for_update : props_for_create
  end

```

#props_for_update

```

  def props_for_update
    update_magic_properties
    changed_props = attributes.select { |k, _| changed_attributes.include?(k) }
    changed_props.symbolize_keys!
    inject_defaults!(changed_props)
    props_for_db(changed_props)
  end

```

#query_as Returns a Query object with the current node matched the specified variable name

```

  def query_as(node_var)
    self.class.query_as(node_var, false).where("ID(#{node_var})" => self.neo_id)
  end

```

#read_attribute

```

  def read_attribute(name)
    respond_to?(name) ? send(name) : nil
  end

```

#read_attribute_for_validation Implements the ActiveRecord::Validation hook method.

```

  def read_attribute_for_validation(key)
    respond_to?(key) ? send(key) : self[key]
  end

```

#reload

```

  def reload
    return self if new_record?
    association_proxy_cache.clear if respond_to?(:association_proxy_cache)
    changed_attributes && changed_attributes.clear
    unless reload_from_database
      @_deleted = true
      freeze
    end
    self
  end

```

#reload_from_database

```

  def reload_from_database
    reloaded = self.class.load_entity(neo_id)
    reloaded ? init_on_reload(reloaded._persisted_obj) : nil
  end

```

#reload_properties!

```
def reload_properties!(properties)
  @attributes = nil
  convert_and_assign_attributes(properties)
end
```

#remove_label Removes one or more labels Be careful, don't remove the label representing the Ruby class.

```
def remove_label(*label)
  @_persisted_obj.remove_label(*label)
end
```

#save The validation process on save can be skipped by passing false. The regular Model#save method is replaced with this when the validations module is mixed in, which it is by default.

```
def save(options = {})
  result = perform_validations(options) ? super : false
  if !result
    Neo4j::Transaction.current.failure if Neo4j::Transaction.current
  end
  result
end
```

#save! Persist the object to the database. Validations and Callbacks are included by default but validation can be disabled by passing :validate => false to #save! Creates a new transaction.

```
def save!(*args)
  save(*args) or fail(RecordInvalidError, self) # rubocop:disable Style/AndOr
end
```

#send_props

```
def send_props(hash)
  return hash if hash.blank?
  hash.each { |key, value| send("#(key)=", value) }
end
```

#serializable_hash

```
def serializable_hash(*args)
  super.merge(id: id)
end
```

#serialized_properties

```
def serialized_properties
  self.class.serialized_properties
end
```

#to_key Returns an Enumerable of all (primary) key attributes or nil if model.persisted? is false

```
def to_key
  _persisted_obj ? [id] : nil
end
```

#touch

nodoc

```
def touch #:nodoc:
  run_callbacks(:touch) { super }
end
```


#update Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end
```

#update! Same as {#update_attributes}, but raises an exception if saving fails.

```
def update!(attributes)
  self.attributes = process_attributes(attributes)
  save!
end
```

#update_attribute Convenience method to set attribute and #save at the same time

```
def update_attribute(attribute, value)
  send("#{attribute}=", value)
  self.save
end
```

#update_attribute! Convenience method to set attribute and #save! at the same time

```
def update_attribute!(attribute, value)
  send("#{attribute}=", value)
  self.save!
end
```

#update_attributes Updates this resource with all the attributes from the passed-in Hash and requests that the record be saved. If saving fails because the resource is invalid then false will be returned.

```
def update(attributes)
  self.attributes = process_attributes(attributes)
  save
end
```

#update_attributes! Same as {#update_attributes}, but raises an exception if saving fails.

```
def update!(attributes)
  self.attributes = process_attributes(attributes)
  save!
end
```

#valid?

```
def valid?(context = nil)
  context ||= (new_record? ? :create : :update)
  super(context)
  errors.empty?
end
```

#wrapper Implements the Neo4j::Node#wrapper and Neo4j::Relationship#wrapper method so that we don't have to care if the node is wrapped or not.

```
def wrapper
  self
end
```

#write_attribute Write a single attribute to the model's attribute hash.

```
def write_attribute(name, value)
  if respond_to? "#{name}="
    send "#{name}=", value
  else
    fail Neo4j::UnknownAttributeError, "unknown attribute: #{name}"
  end
end
```

11.1.18 TypeConverters

Constants

- CONVERTERS

Files

- lib/neo4j/type_converters.rb:2

Methods

#convert_properties_to Modifies a hash's values to be of types acceptable to Neo4j or matching what the user defined using *type* in property definitions.

```
def convert_properties_to(obj, medium, properties)
  direction = medium == :ruby ? :to_ruby : :to_db
  properties.each_pair do |key, value|
    next if skip_conversion?(obj, key, value)
    properties[key] = convert_property(key, value, direction)
  end
end
```

#convert_property Converts a single property from its current format to its db- or Ruby-expected output type.

```
def convert_property(key, value, direction)
  converted_property(primitive_type(key.to_sym), value, direction)
end
```

#typecast_attribute

```
def typecast_attribute(typecaster, value)
  Neo4j::Shared::TypeConverters.typecast_attribute(typecaster, value)
end
```

#typecaster_for

```
def typecaster_for(value)
  Neo4j::Shared::TypeConverters.typecaster_for(value)
end
```

11.1.19 ClassArguments

Constants

- INVALID_CLASS_ARGUMENT_ERROR

Files

- lib/neo4j/class_arguments.rb:2

Methods

.active_node_model?

```
def active_node_model?(class_constant)
  class_constant.included_modules.include?(Neo4j::ActiveNode)
end
```

.constantize_argument

```
def constantize_argument(class_argument)
  case class_argument
  when 'any', :any, false, nil
    nil
  when Array
    class_argument.map(&method(:constantize_argument))
  else
    class_argument.to_s.constantize.tap do |class_constant|
      if !active_node_model?(class_constant)
        fail ArgumentError, "#{class_constant} is not an ActiveNode model"
      end
    end
  end
end
rescue NameError
  raise ArgumentError, "Could not find class: #{class_argument}"
end
```

.valid_argument?

```
def valid_argument?(class_argument)
  [NilClass, String, Symbol, FalseClass].include?(class_argument.class) ||
  (class_argument.is_a?(Array) && class_argument.all? { |c| [Symbol, String].include?(c.class) })
end
```

.validate_argument!

```
def validate_argument!(class_argument, context)
  return if valid_argument?(class_argument)

  fail ArgumentError, "#{context} #{INVALID_CLASS_ARGUMENT_ERROR} (was #{class_argument.inspect})"
end
```

11.1.20 Schema

Operation

Constants

Files

- lib/neo4j/schema/operation.rb:3

Methods

#create!

```
def create!
  drop_incompatible!
  return if exist?
  label_object.send(:"create_#{type}", property, options)
end
```

#default_options

```
def default_options
  {}
end
```

#drop!

```
def drop!
  label_object.send(:"drop_#{type}", property, options)
end
```

#drop_incompatible!

```
def drop_incompatible!
  incompatible_operation_classes.each do |clazz|
    operation = clazz.new(label_name, property)
    operation.drop! if operation.exist?
  end
end
```

#exist?

```
def exist?
  fail 'Abstract class, not implemented'
end
```

#incompatible_operation_classes

```
def incompatible_operation_classes
  self.class.incompatible_operation_classes
end
```

.incompatible_operation_classes

```
def self.incompatible_operation_classes
  []
end
```

#initialize

```
def initialize(label_name, property, options = default_options)
  @label_name = label_name.to_sym
  @property = property.to_sym
  @options = options
end
```

#label_name Returns the value of attribute label_name

```
def label_name
  @label_name
end
```

#label_object

```
def label_object
  @label_object ||= Neo4j::Label.create(label_name)
end
```

#options Returns the value of attribute options

```
def options
  @options
end
```

#property Returns the value of attribute property

```
def property
  @property
end
```

#type

```
def type
  fail 'Abstract class, not implemented'
end
```

ExactIndexOperation**Constants****Files**

- lib/neo4j/schema/operation.rb:54

Methods**#create!**

```
def create!
  drop_incompatible!
  return if exist?
  label_object.send(:"create_#{type}", property, options)
end
```

#default_options

```
def default_options
  {}
end
```

#drop!

```
def drop!
  label_object.send(:"drop_#{type}", property, options)
end
```

#drop_incompatible!

```
def drop_incompatible!  
  incompatible_operation_classes.each do |clazz|  
    operation = clazz.new(label_name, property)  
    operation.drop! if operation.exist?  
  end  
end
```

#exist?

```
def exist?  
  label_object.indexes[:property_keys].include?([property])  
end
```

#incompatible_operation_classes

```
def incompatible_operation_classes  
  self.class.incompatible_operation_classes  
end
```

.incompatible_operation_classes

```
def self.incompatible_operation_classes  
  [UniqueConstraintOperation]  
end
```

#initialize

```
def initialize(label_name, property, options = default_options)  
  @label_name = label_name.to_sym  
  @property = property.to_sym  
  @options = options  
end
```

#label_name Returns the value of attribute label_name

```
def label_name  
  @label_name  
end
```

#label_object

```
def label_object  
  @label_object ||= Neo4j::Label.create(label_name)  
end
```

#options Returns the value of attribute options

```
def options  
  @options  
end
```

#property Returns the value of attribute property

```
def property  
  @property  
end
```

#type

```
def type  
  'index'  
end
```

UniqueConstraintOperation

Constants

Files

- lib/neo4j/schema/operation.rb:68

Methods

#create!

```
def create!
  return if exist?
  super
end
```

#default_options

```
def default_options
  {type: :unique}
end
```

#drop!

```
def drop!
  label_object.send(:"drop_#{type}", property, options)
end
```

#drop_incompatible!

```
def drop_incompatible!
  incompatible_operation_classes.each do |clazz|
    operation = clazz.new(label_name, property)
    operation.drop! if operation.exist?
  end
end
```

#exist?

```
def exist?
  Neo4j::Label.constraint?(label_name, property)
end
```

#incompatible_operation_classes

```
def incompatible_operation_classes
  self.class.incompatible_operation_classes
end
```

.incompatible_operation_classes

```
def self.incompatible_operation_classes
  [ExactIndexOperation]
end
```

#initialize

```
def initialize(label_name, property, options = default_options)
  @label_name = label_name.to_sym
  @property = property.to_sym
  @options = options
end
```

#label_name Returns the value of attribute label_name

```
def label_name
  @label_name
end
```

#label_object

```
def label_object
  @label_object ||= Neo4j::Label.create(label_name)
end
```

#options Returns the value of attribute options

```
def options
  @options
end
```

#property Returns the value of attribute property

```
def property
  @property
end
```

#type

```
def type
  'constraint'
end
```

Constants

Files

- lib/neo4j/schema/operation.rb:2

Methods

11.1.21 Relationship

Wrapper

Constants

Files

- lib/neo4j/active_rel/rel_wrapper.rb:2

Methods

#wrapper

```
def wrapper
  props.symbolize_keys!
  begin
    most_concrete_class = class_from_type
    wrapped_rel = most_concrete_class.constantize.new
  rescue NameError
    return self
  end

  wrapped_rel.init_on_load(self, self._start_node_id, self._end_node_id, self.rel_type)
  wrapped_rel
end
```

Constants

Files

- lib/neo4j/active_rel/rel_wrapper.rb:1

Methods

11.1.22 Node

Wrapper

The wrapping process is what transforms a raw CypherNode or EmbeddedNode from Neo4j::Core into a healthy ActiveNode (or ActiveRel) object.

Constants

- CONSTANTS_FOR_LABELS_CACHE

Files

- lib/neo4j/active_node/node_wrapper.rb:5

Methods

#class_to_wrap

```
def class_to_wrap
  load_classes_from_labels
  Neo4j::ActiveNode::Labels.model_for_labels(labels).tap do |model_class|
    Neo4j::Node::Wrapper.populate_constants_for_labels_cache(model_class, labels)
  end
end
```

#wrapper this is a plugin in the neo4j-core so that the Ruby wrapper will be wrapped around the Neo4j::Node objects

```
def wrapper
  found_class = class_to_wrap
  return self if not found_class

  found_class.new.tap do |wrapped_node|
    wrapped_node.init_on_load(self, self.props)
  end
end
```

Constants

Files

- lib/neo4j/active_node/node_wrapper.rb:3

Methods

11.1.23 Generators

nodoc

Base

nodoc

Constants

Files

- lib/rails/generators/neo4j_generator.rb:10

Methods

.source_root

```
def self.source_root
  @_neo4j_source_root ||= File.expand_path(File.join(File.dirname(__FILE__),
                                                         'neo4j', generator_name, 'templates'))
end
```

ActiveModel

nodoc

Constants

Files

- lib/rails/generators/neo4j_generator.rb:17

Methods

.all

```
def self.all(klass)
  "#{klass}.all"
end
```

.build

```
def self.build(klass, params = nil)
  if params
    "#{klass}.new({params}) "
  else
    "#{klass}.new"
  end
end
```

#destroy

```
def destroy
  "#{name}.destroy"
end
```

#errors

```
def errors
  "#{name}.errors"
end
```

.find

```
def self.find(klass, params = nil)
  "#{klass}.find({params}) "
end
```

#save

```
def save
  "#{name}.save"
end
```

#update_attributes

```
def update_attributes(params = nil)
  "#{name}.update_attributes({params}) "
end
```

ModelGenerator

nodoc

Constants

Files

- lib/rails/generators/neo4j/model/model_generator.rb:3

Methods

#create_model_file

```
def create_model_file
  template 'model.erb', File.join('app/models', class_path, "#{singular_name}.rb")
end
```

.source_root

```
def self.source_root
  @_neo4j_source_root ||= File.expand_path(File.join(File.dirname(__FILE__),
                                                         'neo4j', generator_name, 'templates'))
end
```

Constants

Files

- lib/rails/generators/neo4j_generator.rb:6

Methods

11.1.24 Constants

- VERSION

11.1.25 Files

- lib/neo4j/errors.rb:1
- lib/neo4j/config.rb:1
- lib/neo4j/shared.rb:1
- lib/neo4j/railtie.rb:6
- lib/neo4j/version.rb:1
- lib/neo4j/wrapper.rb:1
- lib/neo4j/paginated.rb:1
- lib/neo4j/migration.rb:3
- lib/neo4j/active_rel.rb:1
- lib/neo4j/timestamps.rb:4
- lib/neo4j/active_node.rb:1

- lib/neo4j/shared/marshal.rb:1
- lib/neo4j/type_converters.rb:1
- lib/neo4j/class_arguments.rb:1
- lib/neo4j/active_rel/types.rb:1
- lib/neo4j/shared/callbacks.rb:1
- lib/neo4j/schema/operation.rb:1
- lib/neo4j/active_node/query.rb:1
- lib/neo4j/shared/typecaster.rb:1
- lib/neo4j/timestamps/updated.rb:1
- lib/neo4j/timestamps/created.rb:1
- lib/neo4j/active_node/labels.rb:1
- lib/neo4j/shared/validations.rb:1
- lib/neo4j/active_rel/callbacks.rb:1
- lib/neo4j/active_node/dependent.rb:1
- lib/neo4j/active_node/callbacks.rb:1
- lib/neo4j/active_rel/validations.rb:1
- lib/neo4j/active_node/validations.rb:1
- lib/neo4j/active_node/orm_adapter.rb:3
- lib/neo4j/active_node/unpersisted.rb:1
- lib/neo4j/active_node/query_methods.rb:1
- lib/rails/generators/neo4j_generator.rb:5
- lib/neo4j/active_node/has_n/association.rb:4
- lib/neo4j/active_node/query/query_proxy.rb:1
- lib/neo4j/active_node/query/query_proxy_link.rb:1
- lib/neo4j/active_node/query/query_proxy_methods.rb:1
- lib/neo4j/active_node/query/query_proxy_enumerable.rb:1
- lib/neo4j/active_node/dependent/association_methods.rb:1
- lib/neo4j/active_node/dependent/query_proxy_methods.rb:1
- lib/neo4j/active_node/query/query_proxy_eager_loading.rb:1
- lib/neo4j/active_node/has_n/association_cypher_methods.rb:1
- lib/neo4j/active_node/query/query_proxy_find_in_batches.rb:1
- lib/neo4j/active_node/query/query_proxy_methods_of_mass_updating.rb:1

11.1.26 Methods

11.2 Rails

11.2.1 Generators

GeneratedAttribute

nodoc

Constants

Files

- lib/rails/generators/neo4j_generator.rb:53

Methods

#type_class

```
def type_class
  case type.to_s.downcase
  when 'any' then 'any'
  when 'datetime' then 'DateTime'
  when 'date' then 'Date'
  when 'integer', 'number', 'fixnum' then 'Integer'
  when 'float' then 'Float'
  else
    'String'
  end
end
```

Constants

Files

- lib/rails/generators/neo4j_generator.rb:52

Methods

11.2.2 Constants

11.2.3 Files

- lib/rails/generators/neo4j_generator.rb:51

11.2.4 Methods

Neo4j.rb (the `neo4j` and `neo4j-core` gems) is a [Ruby](#) Object-Graph-Mapper (OGM) for the [Neo4j](#) graph database. It tries to follow API conventions established by [ActiveRecord](#) and familiar to most Ruby developers but with a Neo4j flavor.

Ruby (software) A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

Graph Database (computer science) A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way.

Neo4j (databases) The world's leading graph database

If you're already familiar with ActiveRecord, DataMapper, or Mongoid, you'll find the Object Model features you've come to expect from an O*M:

- Properties
- Indexes / Constraints
- Callbacks
- Validation
- Associations

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

Additional features include

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

Requirements

- Ruby 1.9.3+ (tested in MRI and JRuby)
- Neo4j 2.1.0 + (version 4.0+ of the gem is required to use neo4j 2.2+)

Indices and tables

- `genindex`
- `modindex`
- `search`

A

association_model_namespace, [62](#)

C

class_name_property, [61](#)

I

include_root_in_json, [61](#)

L

logger, [62](#)

M

module_handling, [62](#)

N

neo4j:config, [9](#)

neo4j:install, [9](#)

neo4j:restart, [10](#)

neo4j:start, [9](#)

neo4j:start_no_wait, [9](#)

neo4j:stop, [10](#)

P

pretty_logged_cypher_queries, [62](#)

R

record_timestamps, [62](#)

T

timestamp_type, [62](#)

transform_rel_type, [61](#)

W

wait_for_connection, [62](#)