Neo4j.rb Documentation

Release 4.1

Chris Grigg, Brian Underwood

March 29, 2015

Contents

Contents:

CHAPTER 1

Basic Setup

CHAPTER 2

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
```

2.1 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.

2.1.1 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
```

2.1.2 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.

2.1.3 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.

2.2 Callbacks

Implements like Active Records the following callback hooks:

- initialize

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

2.3 created_at, updated_at

See http://neo4j.rubyforge.org/classes/Neo4j/Rails/Timestamps.html

```
class Blog
  include Neo4j::ActiveNode
  property :updated_at  # will automatically be set when model changes
end
```

2.4 Validation

Support the Active Model validation, such as:

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

2.5 id property (primary key)

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

2.6 Associations

What follows is an overview of adding associations to models. For more detailed information, see Declared Relationships.

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

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

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

class Person
```

```
  include Neo4j::ActiveNode
  has_many :in, :posts, origin: :author
  has_many :in, :comments, origin: :author
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
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
```

CHAPTER 3

ActiveRel

Note: See https://github.com/neo4jrb/neo4j/wiki/Neo4j.rb-v4-Introduction if you are using the master branch from this repo. It contains information about changes to the API.

ActiveRel is Neo4j.rb 3.0's the relationship wrapper. ActiveRel objects share most of their behavior with ActiveNode objects. It is purely optional and offers advanced functionality for complex relationships.

3.1 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.

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

3.2 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 string to define the relationship type Name the file as you would any other model. 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
```

3.3 Relationship Creation

From an ActiveRel Model

Once setup, ActiveRel models follow the same rules as ActiveNode in regard to properties. Declare them to create setter/getter methods, set them to 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)
```

3.4 From a *has_many* or *has_one* association

Pass the :rel_type option in a declared association with the constant of an ActiveRel model. When that relationship is created, it will add a hidden _classname property with that model's name. The association will use the type declared in the ActiveRel model and it will raise an error if it is included in more than one place.

To take advantage of callbacks and validations, declare your relationship using your ActiveRel model as described above.

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

3.5 Query and Loading existing relationships

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

3.5.1 :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
```

```
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 perform a Cypher query that matches all students with relationships of type enrolled_in to Lesson.first, then it would call select on that.

3.5.2 The :where method

Because you cannot search for a relationship the way you search for a node, ActiveRel's where method searches for the relationship relative to the labels found in the from_class and to_class models. Therefore:

```
EnrolledIn.where(since: 2002)
# "MATCH (node1:`Student`)-[rel1:`enrolled_in`]->(node2:`Lesson`) WHERE rel1.since = 2002 RETURN rel
```

If your from_class is :any, the same query looks like this:

```
"MATCH (node1)-[rel1:`enrolled_in`]->(node2:`Lesson`) WHERE rel1.since = 2002 RETURN rel1"
```

And if to_class is also :any, you end up with:

```
"MATCH (node1)-[rel1:`enrolled_in`]->(node2) WHERE rel1.since = 2002 RETURN rel1"
```

As a result, this combined with the inability to index relationship properties can result in extremely inefficient queries.

3.6 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 fo
```

3.7 Advanced Usage

3.7.1 Separation of Relationship Logic

ActiveRel really shines when you have multiple associations that share a relationship type. You can use a rel 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
```

3.8 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

3.9 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.

Neo4j.rb (the neo4j and neo4j-core gems) is a Ruby Object-Graph-Mapper (OGM) for the Neo4j graph database. It tries to follow API conventions established by ActiveRecord and familiar to most Ruby developers but with a Neo4j flavor.

Ruby (software) A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

Graph Database (computer science) A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way.

Neo4j (databases) The world's leading graph database

If you're already familiar with ActiveRecord, DataMapper, or Mongoid, you'll find the Object Model features you've come to expect from an O*M:

- Properties
- Indexes / Constraints
- Callbacks
- Validation
- Assocations

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

Additional features include

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

CHAPTER 5

Requirements

- Ruby 1.9.3+ (tested in MRI and JRuby)
- Neo4j 2.1.0 + (version 4.0+ of the gem is required to use neo4j 2.2+)

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*