Neo4j.rb

Graph Database

The Natural Way to Persist Data?

Andreas Kollegge

Andreas Ronge
The problem with SQL: not designed for

- Accelerating growth of data
- Huge clustered environments
- Complex and evolving data models

Choose the right tools - Not Only SQL
The CAP Theorem

Can't achieve all three, pick two

- **Consistency** – all readers will see the same write
- **Availability** – tolerant of node failures
- **Partition tolerant** – if lost interconnect between nodes

Many NOSQL databases choose

- Sacrifice consistency over availability
- Eventual consistency instead of ACID (but not Neo4j)
NOSQL data models

Size

Key-value stores

Bigtable clones

Document databases

Graph databases

Complexity
How can I talk to Neo4j?

- Embedded with JRuby
  - Ruby Gems: neo4j.rb, neo4jr-simple
  - Ruby Gems: pacer
    - friends.out_e(:friend).in_v(:type => 'person').except(friends).except(person).most_frequent(0...10)
- Neo4j Server - HTTP/REST
  - Ruby Gems: neography
What is a Graph Database?
Graph DB vs. SQL

Node.new :name => 'andreas'

name: andreas

r = Relationship.new(:friends, a, b)
r[:since] = 2002

Node.new :name => 'peter'

People

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>andreas</td>
</tr>
<tr>
<td>2</td>
<td>peter</td>
</tr>
</tbody>
</table>

Friends

<table>
<thead>
<tr>
<th>id</th>
<th>p1_id</th>
<th>p2_id</th>
<th>since</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>1</td>
<td>2</td>
<td>2002</td>
</tr>
</tbody>
</table>

friend since: 2002
Graph DB vs SQL

```java
node.outgoing(:friends)
```

```sql
SELECT p2.name FROM People p1, People p2
JOIN Friends f ON p2.id = f.p2_id OR p2.id = f.p1_id
WHERE p1.name = "peter" AND p1.id != p2.id
```
Benefit 1
Domain Modeling

[Diagram of relationships between user, comment, entry, blog, and authorship]
Benefit 2
No O/R mismatch
Benefit 3
Semi-structured information
thomas = Node.new :age => 29
morpheus = Node.new :rank => 'Captain', :occupation => 'Total badass'
trinity = Node.new :name => 'Trinity'
cypher = Node.new :last_name => 'Cypher'
smith = Node.new :language => 'C++', :name => 'Agent Smith', :version => '1.0b'
architect = Node.new :name => 'The Architect'

Relationship.new(:root, Neo4j.ref_node, thomas)
Relationship.new(:knows, thomas, trinity)[:age] = '3 days'
Relationship.new(:knows, thomas, morpheus)
Relationship.new(:knows, morpheus, trinity)[:age] = '12 years'
Relationship.new(:knows, morpheus, cypher)[:disclojure] = 'public'
r = Relationship.new(:knows, cypher, smith)
r[:age] = '6 month'
r[:disclojure] = 'secret'
Relationship.new(:coded_by, smith, architect)
Benefit 4
No Schema
Benefit 5
Deep traversals

Does Thomas Andersson know someone [who knows]* called Agent Smith?

thomas.outgoing(:knows).depth(:all).find{|node| node[:name] == 'Agent Smith'}
What does Neo4j.rb provide?

The Embedded Java Neo4j

- Nodes, Properties, Relationship, Traversals
- ACID Transactions
- Lucene Integration
- Graph Algorithms
- High Availability Clustering

Neo4j.rb

- Object Oriented Mapping
- “Drop in” replacement for Rails Active Model
- Improved/extended API (lucene, rules, migrations,...)
Neo4j.rb Architecture

Active Model Compliant API
Neo4j::Rails::Model
Neo4j::Rails::Relationship

Mapping Layer to Ruby Classes
Neo4j::NodeMixin
Neo4j::RelationshipMixin

Mapping to Java API
Neo4j::Node
Neo4j::Relationship
Embedded

Easier to install, deploy & test
Is running in same thread as your application
No network connection to DB needed
No Database Tier
Embedded DB = Direct Access to Data

With Neo4j::NodeMixin

```ruby
Neo4j::Transaction.run do
  Person.new(:name => 'foo')
end
```

With Neo4j::Rails::Model

```ruby
person = Person.new(:name => 'foo')
person.save # callbacks/validation
```
Transactions

ACID

- atomicity, consistency, isolation, durability
- only write locks, no read locks

```
Neo4j::Transaction.run do
  # do stuff
end
```
Object Oriented Mapping

A Neo4j Node

```
_class: Person
name: andreas
```

Ruby Class

```
class Person
  include NodeMixin
  property :name
end

node = Person.new
node.name = 'andreas'
```
How do I find things?

1. Start from Reference Node
2. Graph as an Index
3. Use Lucene
Find Thomas Andersson

Neo4j.ref_node.outgoing(:root).first
Use the Graph as an Index

```java
Neo4j.ref_node.outgoing('A').each {...}
```
Lucene

Full-featured text search engine

Features

- Phrase queries, wildcard queries, proximity queries, range queries and more
- Ranked searching
- Sorting
- Date-range
- Sorting by any field
class Person
  include NodeMixin
  index :name
end

Transaction.run do
  Person.new :name => 'andreas'
end

Person.find('name: andreas')
NodeMixin

Lucene Integration
Accessors for properties
Accessors for relationships
Migrations
Works with inheritance
Ruby Class Mapping: Relationships

class Person
  include NodeMixin
  has_n :friends
end

andreas = Person.new(:name => 'andreas')
peter = Person.new(:name => 'peter')
david = Person.new(:name => 'david')

andreas.friends << peter << david

andreas.friends.each { |n| puts n[:name]}

andreas = Node.new :name => 'andreas'
peter = Node.new :name => 'peter'
david = Node.new :name => 'david'
andreas.outgoing(:friend) << peter << david
Incoming Relationship

class Actor
  include Neo4j::NodeMixin
  has_n(acted_in).to(Movie)
end

keanu = Actor.new :name=>'keanu'

matrix = Movie.new :name => 'matrix'

keanu.acted_in << matrix
matrix.actors << keanu

Same relationship different direction

Same
Ruby on Rails/Active Record
“drop in” replacement
class User < ActiveRecord::Base
  attr_accessor :password
  attr_accessible :name, :email, :password, :password_confirmation

  after_save :encrypt_password

  email_regex = /\A[\w+\.-]+@[a-z\d\-]+\.[a-z]+\z/i

  validates :name,  :presence => true,
             :length   => { :maximum => 50 }
  validates :email, :presence => true,
              :format    => { :with => email_regex }
  validates :password, :presence => true,
               :confirmation => true,
               :length      => { :within => 6..40 }

  has_one :profile

private

  def encrypt_password
    self.salt = make_salt if new_record?
    self.encrypted_password = encrypt(password)
  end
class User < Neo4j::Model
  attr_accessor :password
  attr_accessible :name, :email, :password, :password_confirmation
  after_save :encrypt_password

  email_regex = /\A[\w+\-]+@[a-z\d\-]+\.[a-z]+\z/i

  validates :name, :presence => true,
            :length => { :maximum => 50 }
  validates :email, :presence => true,
              :format => { :with => email_regex }
  validates :password, :presence => true,
               :confirmation => true,
               :length => { :within => 6..40 }

  property :name
  property :email
  property :salt
  property :encrypted_password
  index :email

  has_one :profile

  private

  def encrypt_password
    self.salt = make_salt if new_record?
    self.encrypted_password = encrypt(password)
  end
Active Record like API
Examples:

Create Relationship
actor = Actor.new
matrix = actor.acted_in.build(:title => 'matrix')
actor.save

Find Relationships
rel = actor.acted_in.find(matrix) # ret Neo4j::Rails::Relationship
rel[:role] = 'trinity'
actor.save

Delete Relationship
actor.acted_in.delete(matrix)
actor.acted_in.destroy_all

Updated relationships in nested forms
using accepts_nested_attributes_for :acted_in
actor.update_attributes(:acted_in_attributes => {...})
class Role < Neo4j::Rails::Relationship
  property :role_name
  index :role_name
end

class Actor < Neo4j::Rails::Model
  has_n(:acted_in).relationship(Role)
end

actor = Actor.new

# create a node but return the relationship -
#   use the "_rels" accessor
role = actor.acted_in.rels.build(:title => 'matrix')
role.role_name = 'trinity'
role.save!

role = Role.find_by_role_name('trinity')
role.start_node #=> actor
role.end_node #=> the created matrix node
A Common Problem

I have a

- System already in production
- Huge database

I need to

- Change the structure of the database

Solution:

- Migrations
Migrations: Direct

```ruby
Neo4j.migration 1, "My First Migration" do
  up do
    Neo4j::Transaction.run { DO_STUFF }
  end
  down do
    Neo4j::Transaction.run { DO_STUFF }
  end
end
```
Actor.migration 1, :split_name do
  up do
    self[:given_name] = self[:name].split[0]
    self[:surname] = self[:name].split[1]
    self[:name] = nil
  end

  down do
    self[:name] = "#{self[:given_name]} #{self[:surname]}"
    self[:surename] = nil
    self[:given_name] = nil
  end
end
Migrations is **NOT** needed when developing unlike Active Record migrations
Inheritance

class Vehicle
  include Neo4j::NodeMixin
  property :name, :year
  index :name, :year
end

class Car < Vehicle
end

Neo4j::Transaction.run do
  Car.new :name => 'volvo', :year => 2000
end

Car.find(:name => 'volvo', :year => 1999..2001).first
Vehicle.find(:name => 'volvo', :year => 1999..2001).first
Recommendation Engine

Adam

Likes

1
2
3
4

Likes

Bertil

Likes

Caesar

5

Recommend This
Example, Recommendation

def composers_for(person)
    [*person.outgoing(:likes)]
end

def recommend(person)
    # which composers does this person like?
    my_composers = composers_for(person)

    # find all other people liking those composers
    other_people = person.outgoing(:likes).incoming(:likes).depth(2).filter{|f| f.depth == 2}

    # for each of those people, sort by the number of matching composers
    # so that the most relevant recommendations are printed first
    sorted = other_people.sort_by{|p| (composers_for(p) & my_composers).size}.reverse
    sorted.each do |other_person|
        # then print out those composers that he don't have
        puts "Recommendation from #{other_person[:name]}
        (composers_for(other_person) - my_composers).each do |s|
            puts " composer #{s[:name]}"
        end
    end
end
Aggregation/Rules

• How to make a flat structure into a graph?
  - Use the Graph DB as an index

```ruby
Person.old
```

```ruby
a.outgoing(:friends).find_all{|f| f.age > 20}
```
Included Graph Algorithms

Shortest paths, Simple paths, Graph measures ...

```
Neo4j::GraphAlgo.all_simple_paths(from(node1).outgoing(:knows).to(node2).depth(5)
```
High Availability

Online Backup - hot spare
Read-slave replication
Write master election
Neo4j – An Object DB?

Neo4j has very fast traversals
  - Avoids loading properties

No need to declare two way relationships
  - A relationship has a start and end node

Does have two ways of finding objects
  - Traversals
  - Lucene

Optimized for Graph Algorithms
Conclusions: Benefits

Express your domain as a Graph

- Domain Modeling
- No O/R mismatch
- Efficient storage of Semi Structured Information
- Schema Less

Express Queries as Traversals

- Fast deep traversal instead of slow SQL queries that span many table joins
When **NOT** use Graph DB

Don't have a graph related problem?

Not too much changing requirements?

Easy to organized data into:
- Tables, Documents or Key-Value models?

Few & well defined relationships in the domain?

Don't have SQL queries that span many table joins?

Many YES => *maybe* Graph DB **not** a good choice
When should I use a Graph DB?

- Need to solve a graph related problem?
- Recommendations, Shortest path, Social Networks
- Have a complex and evolving data model?
- Few mandatory and many optional attributes?
- Big part of domain is expressed as relationships?
- Have SQL queries that span many table joins?

Many YES => maybe a Graph DB is a good choice
Neo4j Spatial.rb

A wrapper around Java Neo4j Spatial using Neo4j.rb

git clone git@github.com:craigtaverner/neo4j-spatial.rb.git
cd neo4j-spatial.rb/examples
jruby osm_import.rb map2.osm
jruby osm_layer.rb map2.osm highway highway-residential waterway
natural natural-water
  jruby osm_layer.rb -l
jruby export_layer.rb highway-residential
jruby export_layer.rb -F shp highway-residential natural
Neo4j Spatial Queries

AbstractSearchIntersection
SearchAll
SearchClosest
SearchContain
SearchCover
SearchCoveredBy
SearchCross
SearchDisjoint
SearchEmpty
SearchEqual
SearchInRelation
SearchIntersect
SearchIntersectWindow
SearchInvalid
SearchOverlap
SearchPointsWithinOrthodromicDistance
SearchTouch
SearchWithin
SearchWithinDistance
Spatial Graph
GeoServer

GeoServer is an open source software server written in Java that allows users to share and edit geospatial data. Designed for interoperability, it publishes data from any major spatial data source using open standards.

Neo4j Spatial includes built-in support for a GeoTools data store.
GraphDatabaseService database = new EmbeddedGraphDatabase(storeDir);
try {
    SpatialDatabaseService s = new SpatialDatabaseService(database);
    Layer layer = s.getLayer("layer_roads");
    SpatialIndexReader index = layer.getIndex();

    Search q = new SearchIntersectWindow(new Envelope(xmin, xmax, ymin, ymax));
    index.executeSearch(searchQuery);
    List<SpatialDatabaseRecord> results = q.getResults();
} finally {
    database.shutdown();
}