Django and Neo4j
Domain modeling that kicks ass!

Tobias Ivarsson
Hacker @ Neo Technology

twitter: @thobe / #neo4j
email: tobias@neotechnology.com
web: http://www.neo4j.org/
web: http://www.thobe.org/
It all started with this guy. Emil Eifrem, CEO of Neo Technology. We picked him apart, and inside his brain we found the base for a database that models the connections in between entities.
It all started with this guy. Emil Eifrem, CEO of Neo Technology. We picked him apart, and inside his brain we found the base for a database that models the connections in between entities.
It all started with this guy. Emil Eifrem, CEO of Neo Technology. We picked him apart, and inside his brain we found the base for a database that models the connections in between entities.
It all started with this guy, Emil Eifrem, CEO of Neo Technology. We picked him apart, and inside his brain we found the base for a database that models the connections in between entities.
NOSQL is a wide area
The problems NOSQL focuses on

Focus area of many NOSQL Databases

- Huge amounts of data
- (mostly) Disjoint data
- Heavy load
- Many concurrent writers

All NOSQL databases focus on solving problems where RDBMSes fail.

While this handles the load, it lacks in "social"
The evolution of data

... but it turns out that data evolves to become MORE interconnected (as well as greater sizes)
Neo4j is a Graph Database

Graph databases FOCUS on the interconnection between entities.
Graph databases FOCUS on the interconnection between entities.
Scaling to size vs. Scaling to complexity

- Key/Value stores
- Bigtable clones
- Document databases
- Graph databases
Scaling to size vs. Scaling to complexity

- Key/Value stores
- Bigtable clones
- Document databases
- Graph databases

> 90% of use cases

Billions of nodes and relationships
What is Neo4j?

- Neo4j is a Graph Database
  - Non-relational ("#nosql"), transactional (ACID), embedded
  - Data is stored as a Graph / Network
    - Nodes and relationships with properties
    - "Property Graph" or "edge-labeled multidigraph"
  - Schema free, bottom-up data model design
- Neo4j is Open Source / Free (as in speech) Software
  - AGPLv3
  - Commercial ("dual license") license available
    - First server is free (as in beer), next is inexpensive

Contact us if you have questions and/or special license needs (e.g. if you want an evaluation license)
More about Neo4j

- Neo4j is stable
  - In 24/7 operation since 2003
- Neo4j is in active development
  - Neo Technology received VC funding October 2009
- Neo4j delivers high performance graph operations
  - traverses 1’000’000+ relationships / second on commodity hardware (1000~2500 traversals/ms)
Building business applications with Neo4j

- Try it out! It’s all open source!
  - Build a prototype, find out your needs and how Neo4j matches
  - AGPL says all your users should have access to your code
    - at this stage this is your employees / co-workers
Building business applications with Neo4j

- Try it out! It’s all open source!
  - Build a prototype, find out your needs and how Neo4j matches
  - AGPL says all your users should have access to your code
    - at this stage this is your employees / co-workers
- Put it in front of users! The license is free for the first server!
  - Contact Neo Technology sales to get a free single server license
  - You’ll (probably) not have massive load the first days
Building business applications with Neo4j

- Try it out! It’s all open source!
  - Build a prototype, find out your needs and how Neo4j matches
  - AGPL says all your users should have access to your code
    - at this stage this is your employees / co-workers
- Put it in front of users! The license is free for the first server!
  - Contact Neo Technology sales to get a free single server license
  - You’ll (probably) not have massive load the first days
- As you grow, Neo4j grows with you!
  - As your needs and revenue increase you can buy an advanced license (prices are reasonable)
Graphs are all around us

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17</td>
<td>3.14</td>
<td>3</td>
<td>17.7933333333333333</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>10.11</td>
<td>14</td>
<td>30.33</td>
</tr>
<tr>
<td>3</td>
<td>316</td>
<td>6.66</td>
<td>1</td>
<td>2104.56</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>9.11</td>
<td>592</td>
<td>0.492432432432</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>2153.175765766</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Even if this spreadsheet looks like it could be a fit for a RDBMS it isn’t:
- RDBMSes have problems with extending indefinitely on both rows and columns
- Formulas and data dependencies would quickly lead to heavy join operations
Graphs are all around us

|     | A  | B    | C  | D                      |   ...
|-----|----|------|----|------------------------|-------
| 1   | 17 | 3.14 | 3  | = A1 * B1 / C1         |       |
| 2   | 42 | 10.11| 14 | = A2 * B2 / C2         |       |
| 3   | 316| 6.66 | 1  | = A3 * B3 / C3         |       |
| 4   | 32 | 9.11 | 592| = A4 * B4 / C4         |       |
| 5   |    |      |    | = SUM(D2:D5)           |       |
| ... |    |      |    |                         |       |

With data dependencies, the spreadsheet turns out to be a graph.
Graphs are all around us

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17</td>
<td>3.14</td>
<td>3</td>
<td>= A1 * B1 / C1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>10.11</td>
<td>14</td>
<td>= A2 * B2 / C2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>316</td>
<td>6.66</td>
<td>1</td>
<td>= A3 * B3 / C3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>9.11</td>
<td>592</td>
<td>= A4 * B4 / C4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>= SUM(D2:D5)</td>
<td></td>
</tr>
</tbody>
</table>

With data dependencies the spreadsheet turns out to be a graph.
Graphs are all around us

If we add external data sources the problem becomes even more interesting...

17 3.14 3 = A1 * B1 / C1

42 10.1 14 = A2 * B2 / C2

316 6.66 1 = A3 * B3 / C3

32 9.1 592 = A4 * B4 / C4

= SUM(D2:D5)
Graphs are all around us

If we add external data sources the problem becomes even more interesting...

- \[ A1 \times B1 / C1 \]
- \[ A2 \times B2 / C2 \]
- \[ A3 \times B3 / C3 \]
- \[ A4 \times B4 / C4 \]
- \[ \text{SUM(D2:D5)} \]
The Neo4j Graph data model

- Nodes
- Relationships between Nodes
- Relationships have Labels
- Relationships are directed, but traversed at equal speed in both directions
- The semantics of the direction is up to the application (LIVES WITH is reflexive, LOVES is not)
- Nodes have key-value properties
- Relationships have key-value properties
The Neo4j Graph data model

- Nodes
- Relationships between Nodes
- Relationships have Labels
- Relationships are directed, but traversed at equal speed in both directions
- The semantics of the direction is up to the application (LIVES WITH is reflexive, LOVES is not)
- Nodes have key-value properties
- Relationships have key-value properties
The Neo4j Graph data model

- Nodes
- Relationships between Nodes
- Relationships have Labels
- Relationships are directed, but traversed at equal speed in both directions
- The semantics of the direction is up to the application (LIVES WITH is reflexive, LOVES is not)
- Nodes have key-value properties
- Relationships have key-value properties
The Neo4j Graph data model

- Nodes
- Relationships between Nodes
- Relationships have Labels
- Relationships are directed, but traversed at equal speed in both directions
- The semantics of the direction is up to the application (LIVES WITH is reflexive, LOVES is not)
- Nodes have key-value properties
- Relationships have key-value properties
The Neo4j Graph data model

- **Nodes**
- **Relationships between Nodes**
- **Relationships have Labels**
- **Relationships are directed, but traversed at equal speed in both directions**
- The semantics of the direction is up to the application (LIVES WITH is reflexive, LOVES is not)
- **Nodes have key-value properties**
- **Relationships have key-value properties**

- **Name:** James
  - Age: 32
  - Twitter: @spam
- **Name:** Mary
  - Age: 35

- **Brand:** Volvo
  - Model: V70
The Neo4j Graph data model

- Nodes
- Relationships between Nodes
- Relationships have Labels
- Relationships are directed, but traversed at equal speed in both directions
- The semantics of the direction is up to the application (LIVES WITH is reflexive, LOVES is not)
- Nodes have key-value properties
- Relationships have key-value properties

Example:

- Name: James
  - Age: 32
  - Twitter: @spam
- Name: Mary
  - Age: 35

Relationships:
- Lives With: James to Mary
- Loves: James to Mary
- Owns: James to Car
- Drives: James to Volvo V70
- Loves: Mary to Volvo V70
Graphs are Whiteboard Friendly

The domain I specify is the domain I implement. No mismatch, no ER-modeling.
Graphs are Whiteboard Friendly

The domain I specify is the domain I implement. No mismatch, no ER-modeling.
Graphs are Whiteboard Friendly

username: “thobe”
name: “Tobias Ivarsson”
twitter: “@thobe”
password: “**********”
Graphs are Whiteboard Friendly

address: “http://journal.thobe.org”

title: “Wardrobe Strength”

tagline: “Good enough thoughts”
import neo4j

graphDb = neo4j.GraphDatabase( PATH_TO_YOUR_NEO4J_DATASTORE )

with graphDb.transaction: # All writes require transactions

    # Create Thomas 'Neo' Anderson
    mrAnderson = graphDb.node(name="Thomas Anderson", age=29)

    # Create Morpheus
    morpheus = graphDb.node(name="Morpheus", rank="Captain", occupation="Total bad ass")

    # Create relationship representing they know each other
    mrAnderson.KNOWS( morpheus )

    # ... similarly for Trinity, Cypher, Agent Smith, Architect
Graph traversals

name: “Thomas Anderson”
age: 29

name: “Morpheus”
rank: “Captain”
occupation: “Total badass”

name: “Trinity”
since: “meeting the oracle”

name: “Cypher”
last name: “Reagan”
disclosure: “secret”

name: “Agent Smith”
version: “1.0b”
language: “C++”

name: “The Architect”

KNOWS

LOVES

since: “a year before the movie”
cooperates on: “The Nebuchadnezzar”

disclosure: “public”
Graph traversals

import neo4j

class Friends(neo4j.Traversal):
    # Traversals ≈ queries in Neo4j
    types = [neo4j.Outgoing.KNOWS]
    order = neo4j.BREADTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

name: “Thomas Anderson”
age: 29

name: “Morpheus”
rank: “Captain”
occupation: “Total badass”
since: “meeting the oracle”

name: “Trinity”

name: “Cypher”
last name: “Reagan”
disclosure: “secret”
since: “a year before the movie”
cooperates on: “The Nebuchadnezzar”

name: “Agent Smith”
version: “1.0b”
language: “C++”

name: “The Architect”

CODED BY

LOVES

KNOWS

KNOWS

KNOWS
Graph traversals

import neo4j
class Friends(neo4j.Traversal):  # Traversals ≈ queries in Neo4j
types = [neo4j.Outgoing.KNOWS]
order = neo4j.BREADTH_FIRST
stop = neo4j.STOP_AT_END_OF_GRAPH
returnable = neo4j.RETURN_ALL_BUT_START_NODE

for friend_node in Friends(mr_anderson):
    print "%s (@ depth=%s)" % (friend_node['name'],
                                friend_node.node.depth)
import neo4j
class Friends(neo4j.Traversal):  # Traversals == queries in Neo4j
    types = [neo4j.Outgoing.KNOWS]
    order = neo4j.BREADTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

for friend_node in Friends(mr_anderson):
    print "%s (@ depth=%s)" % (friend_node["name"], friend_node.depth)
Graph traversals

import neo4j

class Friends(neo4j.Traversal):
    # Traversals ≈ queries in Neo4j
    types = [neo4j.Outgoing.KNOWS]
    order = neo4j.BREADTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

    for friend_node in Friends(mr_anderson):
        print "%s (@ depth=%s) % (friend_node["name"], friend_node_node.depth )

name: “Thomas Anderson”
age: 29

name: “Morpheus”
rank: “Captain”
ockupation: “Total badass”
since: “a year before the movie”
cooperates on: “The Nebuchadnezzar”
disclosure: “secret”
since: “meeting the oracle”

name: “Trinity”
disclosure: “public”

name: “Cypher”
last name: “Reagan”

name: “Agent Smith”
version: “1.0b”
language: “C++”

name: “The Architect”

Morpheus (@ depth=1)
import neo4j

class Friends(neo4j.Traversal):
    # Traversals ≈ queries in Neo4j
    types = [neo4j.Outgoing.KNOWS]
    order = neo4j.BREADTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

    for friend_node in Friends(mr_anderson):
        print "%s (@ depth=%s)" % (friend_node['name'], friend_node.depth)
import neo4j

class Friends(neo4j.Traversal): # Traversals \(\approx\) queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREADTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

    for friend_node in Friends(mr_anderson):
        print "%s (@ depth=%s)" % (friend_node["name"], friend_node_node.depth)
import neo4j

class Friends(neo4j.Traversal):
    types = [neo4j.Outgoing.KNOWS]
    order = neo4j.BREADTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

    for friend_node in Friends(mr_anderson):
        print("%s (@ depth=%s)" % (friend_node["name"], friend_node_node.depth))
import neo4j

class Friends(neo4j.Traversal):
    # Traversals ≈ queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREADTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

for friend_node in Friends(mr_anderson):
    print "%s (@ depth=%s)" % ( friend_node["name"],
                               friend_node_node.depth )
Finding a place to start

- Traversals need a Node to start from
  - QUESTION: How do I find the start Node?
  - ANSWER: You use an Index

- Indexes in Neo4j are different from Indexes in Relational Databases
  - RDBMSes use them for Joining
  - Neo4j use them for simple lookup

```java
index = graphDb.index["name"]

mr_anderson = index["Thomas Anderson"]

performTraversalFrom( mrAnderson )
```
Indexes in Neo4j

ㅇ The Graph *is* the main index
  • Use relationship labels for navigation
  • Build index structures *in the graph*
    ‣ Search trees, tag clouds, geospatial indexes, et.c.
    ‣ Linked/skip lists or other data structures in the graph
    ‣ We have utility libraries for this

ㅇ External indexes used *for lookup*
  • Finding a (number of) points to start traversals from
  • Major difference from RDBMS that use indexes for everything
Django integration does all of this for you!
Implementing the domain
from neo4j.model import django_model as models
from neo4j.model import django_model as models

class User(models.NodeModel):
    username = models.Property(indexed=True)
    name = models.Property()
    blogs = models.Relationship(Blog,
        type=models.Outgoing.member_of,
        related_name="users")
    def __unicode__(self):
        return self.name
from neo4j.model import django_model as models

class Blog(models.NodeModel):
    identifier = models.Property(indexed=True)
    title = models.Property()
    def __unicode__(self):
        return self.title
from neo4j.model import django_model as models

class Entry(models.NodeModel):
    title = models.Property()
    text = models.Property()
    date = models.Property()
    blog = models.Relationship(Blog,
        type=models.Outgoing.posted_on,
        single=True, optional=False,
        related_name="articles")
    author = models.Relationship(User,
        type=models.Outgoing.authored_by,
        single=True, optional=False,
        related_name="articles")
from neo4j.model import django_model as models

class Blog(models.NodeModel):
    identifier = models.Property(indexed=True)
    title = models.Property()

class User(models.NodeModel):
    username = models.Property(indexed=True)
    name = models.Property()
    blogs = models.Relationship(Blog,
        type=models.Outgoing.member_of,
        related_name="users")

class Entry(models.NodeModel):
    title = models.Property()
    text = models.Property()
    date = models.Property()
    blog = models.Relationship(Blog,
        type=models.Outgoing.posted_on,
        single=True, optional=False,
        related_name="articles")
    author = models.Relationship(User,
        type=models.Outgoing.authored_by,
        single=True, optional=False,
        related_name="articles")

The rest of the code for working with the domain objects is (mostly) the same as you are used to in Django.
Why not use an O/R mapper?

- Model evolution in ORMs is a hard problem
  - virtually unsupported in most ORM systems
- SQL is “compatible” across many RDBMSs
  - data is still locked in
- Each ORM maps object models differently
  - Moving to another ORM == legacy schema support
    - except your legacy schema is a strange auto-generated one
- Object/Graph Mapping is *always* done the same way
  - allows you to keep your data through application changes
  - or share data between multiple implementations
What an ORM doesn’t do

- Deep traversals
- Graph algorithms
- Shortest path(s)
- Routing
- etc.
Path exists in social network

- Each person has on average 50 friends

<table>
<thead>
<tr>
<th>Database</th>
<th># persons</th>
<th>query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational database</td>
<td>1 000</td>
<td>2 000 ms</td>
</tr>
<tr>
<td>Neo4j Graph Database</td>
<td>1 000</td>
<td>2 ms</td>
</tr>
</tbody>
</table>

The performance impact in Neo4j depends only on the degree of each node. In an RDBMS, it depends on the number of entries in the tables involved in the join(s).
Path exists in social network

- Each person has on average 50 friends

<table>
<thead>
<tr>
<th>Database</th>
<th># persons</th>
<th>query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational database</td>
<td>1 000</td>
<td>2 000 ms</td>
</tr>
<tr>
<td>Neo4j Graph Database</td>
<td>1 000</td>
<td>2 ms</td>
</tr>
<tr>
<td>Neo4j Graph Database</td>
<td>1 000 000</td>
<td>2 ms</td>
</tr>
<tr>
<td>Relational database</td>
<td>1 000 000</td>
<td>way too long...</td>
</tr>
</tbody>
</table>

The performance impact in Neo4j depends only on the degree of each node, in an RDBMS it depends on the number of entries in the tables involved in the join(s).
On-line real time routing with Neo4j

- 20 million Nodes - represents places
- 62 million Edges - represents direct roads between places
  - These edges have a length property, for the length of the road
- Average optimal route, 100 separate roads, found in 100ms
- Worst case route we could find:
  - Optimal route is 5500 separate roads
  - Total length ~770km
  - Found in less than 3 seconds
- Uses A* “best first” search

There’s a difference between least number of hops and least cost.
Jython vs. CPython

- Neo4j with the Python bindings work in both
  - Requires *no code modification* in your code
- Neo4j at its core is an Embedded (in-process) database
  - CPython manages concurrency by forking multiple processes
  - Jython has full concurrency support in the same JVM
  - Stand-alone Neo4j server-process with (C)Python client is being worked on
- Neo4j has a RESTful interface
  - There are Python clients
  - The API differs slightly (no transactions)
Finding out more

- [http://neo4j.org/](http://neo4j.org/) - project website - main place for getting started
  - Contains screen casts, download links, et.c.
    - Specifically [http://components.neo4j.org/neo4j.py/](http://components.neo4j.org/neo4j.py/)
  - [http://github.com/neo4j-examples](http://github.com/neo4j-examples) - small example applications

- [https://lists.neo4j.org/](https://lists.neo4j.org/) - community mailing list

- [http://twitter.com/neo4j/team](http://twitter.com/neo4j/team) - follow the Neo4j team

Helping out!

- Neo4j and the Python integration is all Open Source
- The Python bindings in particular would benefit from more devs...
  - Integrate more of the Neo4j components
    - Neo4j Spatial
    - The Graph Algorithms package
    - The Graph Matching component
  - Trimming off the rough edges in the Django integration
  - Native client for CPython
Buzzword summary

AGPLv3
SPARQL
Object mapping
TRaversal
whiteboard friendly
A* routing
Scaling to complexity
Polyglot persistence

Semi structured
Open Source
ACID transactions
Gremlin
Software Transactional Memory
Gremlin

RESTful
NOSQL
Query language

Beer

Embedded
Schema free
Free Software