Persistent Graphs in Python with Neo4j

Tobias Ivarsson

Hacker @ Neo Technology

twitter: @thobe / #neo4j
email: tobias@neotechnology.com
web: http://www.neo4j.org/
web: http://www.thobe.org/

We all know the relational model.

It has been predominant for a long time.

Attendees

username	fullname	registration	tutorials	
guido	Guido van Rossum	null	yes	0
thobe	Tobias Ivarsson	2009-12-12	no	300
joe	John Doe	2010-02-05	yes	700
...	...	...	...	...

Attendees

username	fullname	registration	tutorials	payment
guido	Guido van Rossum	null	yes	0
thobe	Tobias Ivarsson	2009-12-12	no	300
joe	John Doe	2010-02-05	yes	700
...	...	...	...	...

Location

username	latitude	longitude	title	publish
thobe	55°36'47.70"N	12°58'34.50"E	Malmö	yes
joe	37°49'36.00"N	122°25'22.00"W	San Francisco	no
...	...	...	...	...

The relational model has a few problems, such as:
- poor support for sparse data
- modifying the data model is almost exclusively done through adding tables

Attendees

username	fullname	registration	tutorials	payment
guido	Guido van Rossum	null	yes	0
thobe	Tobias Ivarsson	2009-12-12	no	300
joe	John Doe	2010-02-05	yes	700
...	...	...	...	...

Sessions

id	title	time	room	...
...	...	...	...	...
...	...	...	...	...

Session attendance

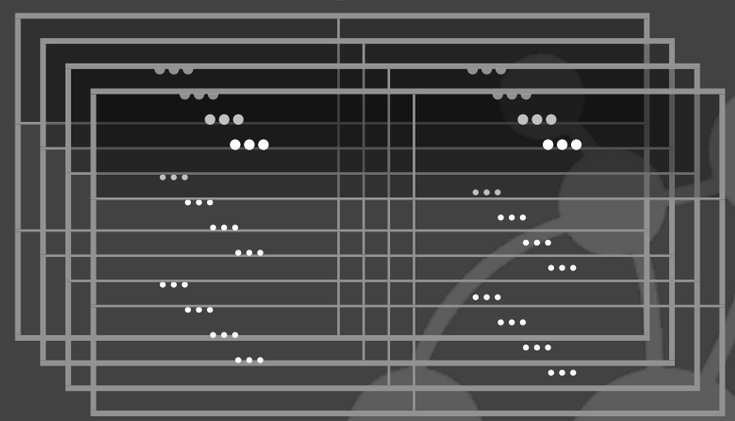
session	user
...	...
...	...

Location

username	latitude	longitude	title	publish
	6'47.70"N	12°58'34.50"E	Malmö	yes
	9'36.00"N	122°25'22.00"W	San Francisco	no
	...	...	...	...

After a while, modeling complex relationships leads to complicated schemas

More complication...



neo technology
value in relationships

mongoDB
{name: "mongo", type: "db"}

Google

basho

amazon.com

facebook

sones

CouchDB
relax

neo technology
value in relationships

twitter

hadoop

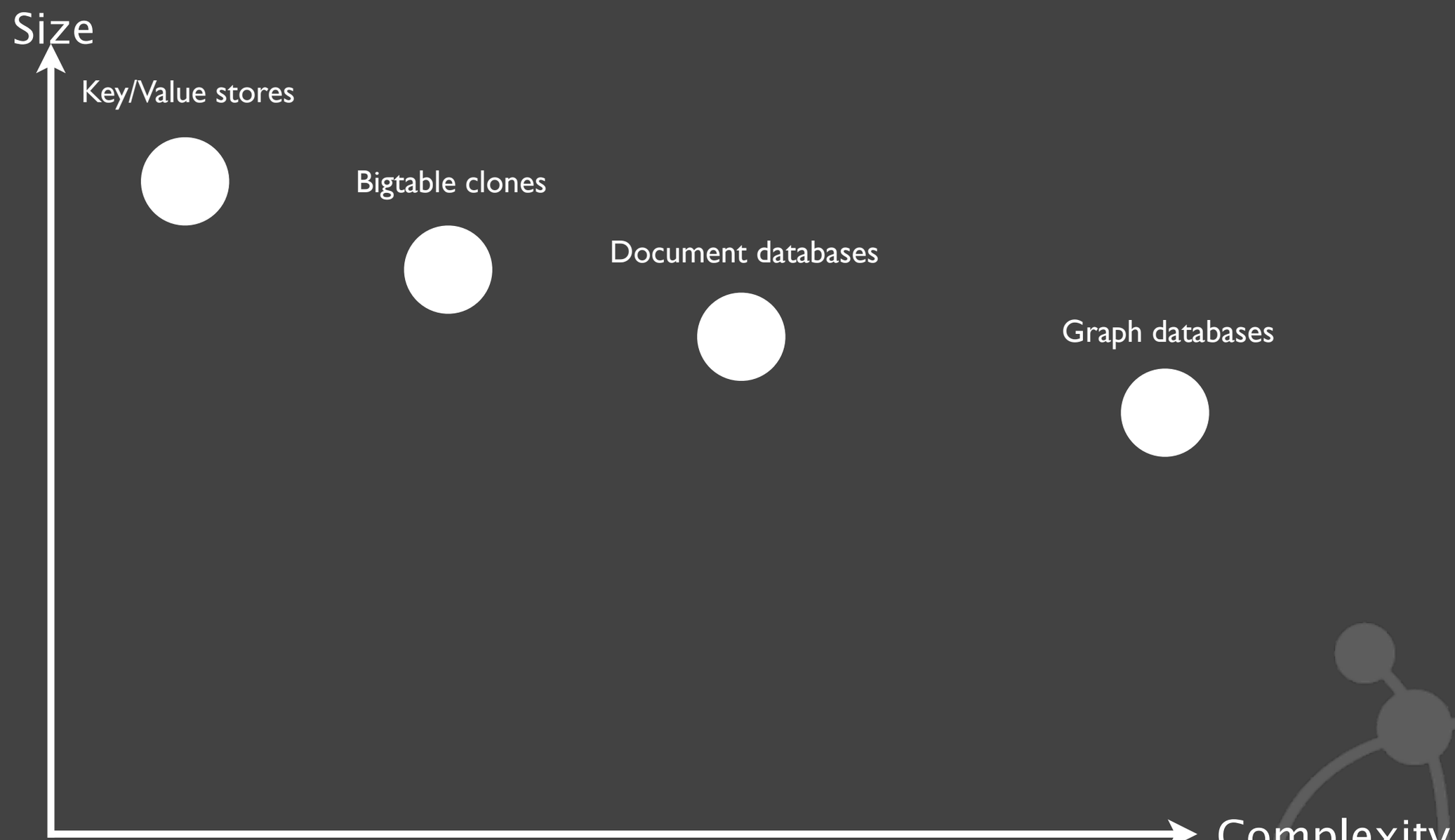
LinkedIn

A number of companies have realized that the relational model is insufficient and are working on alternative database solutions.

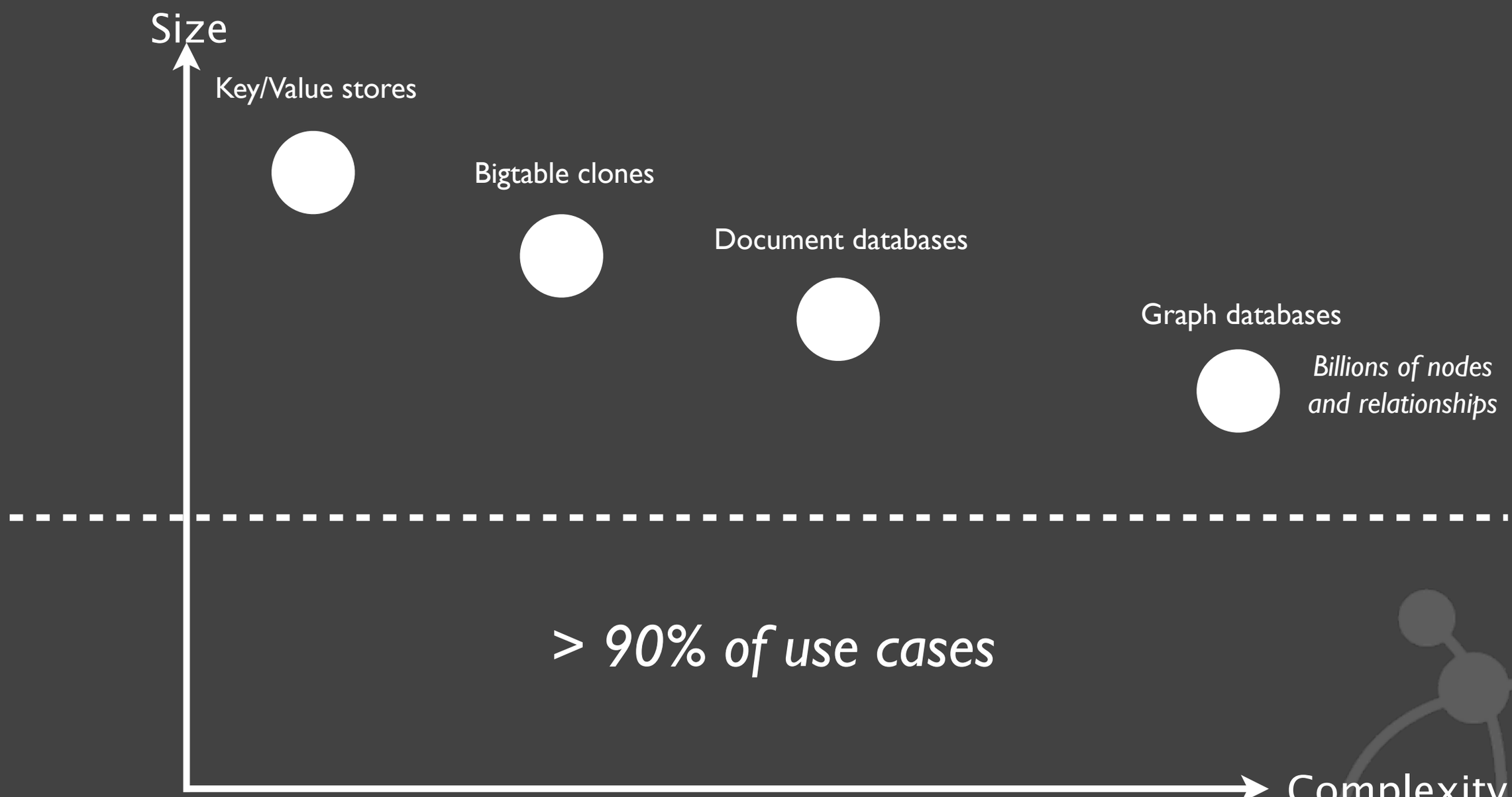
Most focus on scaling to large numbers

Graph Databases focuses on structure of data

Positioning w.r.t. other NOSQL DBs



Positioning w.r.t. other NOSQL DBs



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"
- Neo4j is Open Source / Free (as in speech) Software
  - AGPLv3
  - Commercial ("dual license") license available
    - Free (as in beer) for "small" installations
    - Inexpensive (as in startup-friendly) when you grow

Prices are available at http://neotechnology.com/

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

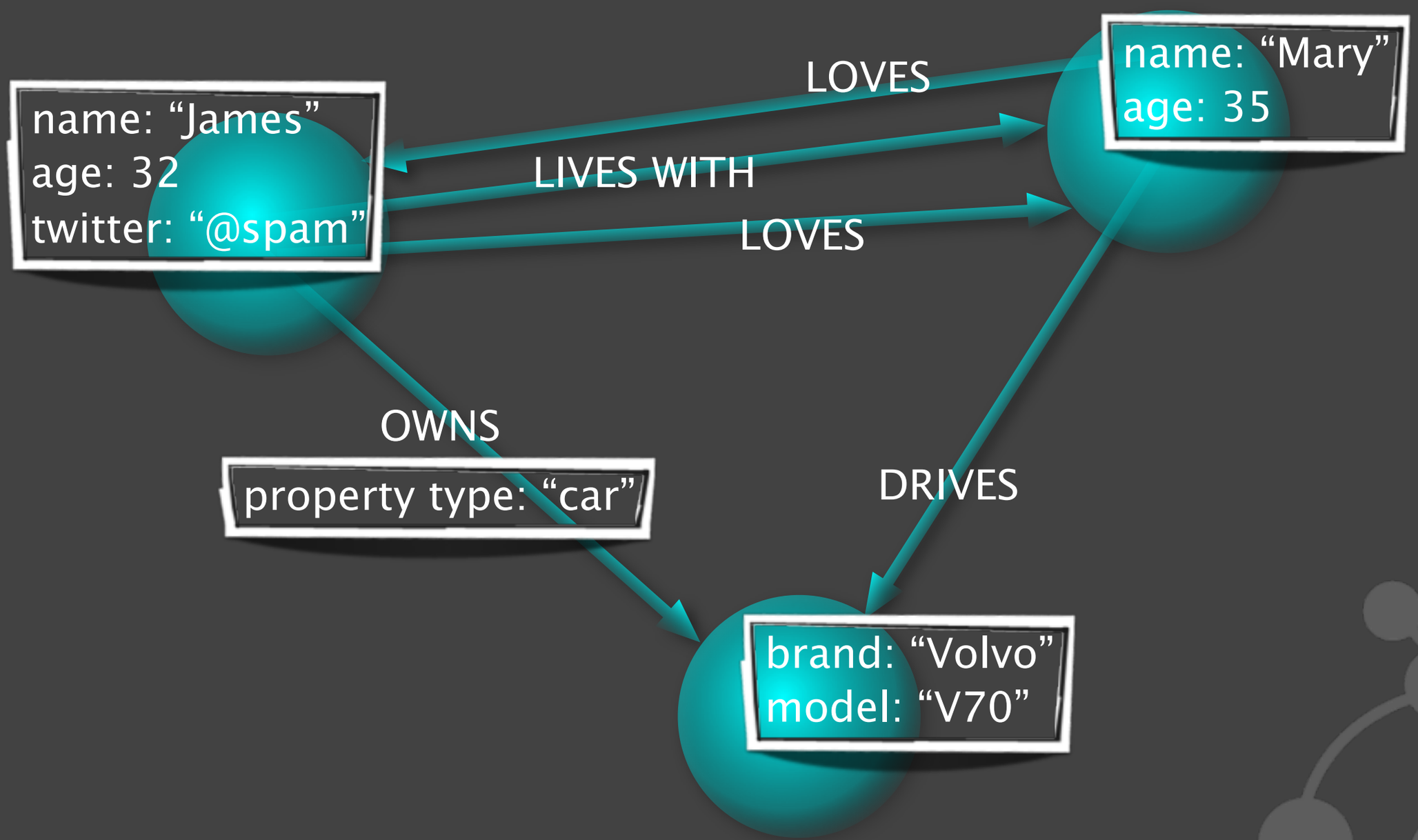
More about Neo4j

- Neo4j is stable
  - In 24/7 operation since 2003
- Neo4j is in active development
  - Neo Technology got VC funding October 2009
- Neo4j delivers high performance graph operations
  - traverses 1'000'000+ relationships / second on commodity hardware

The Neo4j Graph data model

- Nodes are connected to one another through relationships
- A Relationship is a connection between two nodes
  - Relationships have types
  - Relationships have a direction
  - Relationships are traversed equally fast in either direction
- Properties are mappings from a string key to a primitive value
  - Both Nodes and Relationships have properties
  - Primitive values are any of these (or an array of these):
    - String
    - Numbers: float, double, integers (1-8 byte)

The Neo4j Graph data model



Graphs are all around us

	A	B	C	D	...
1	17	3.14	3	17.79333333333	
2	42	10.11	14	30.33	
3	316	6.66	1	2104.56	
4	32	9.11	592	0.492432432432	
5				2153.175765766	
...					

Even if this spread sheet looks like it could be a fit for a RDBMS it isn't:

- RDBMSes have problems with extending indefinitely on both rows and collumns
- Formulas and data dependencies would quickly lead to heavy join operations

Graphs are all around us

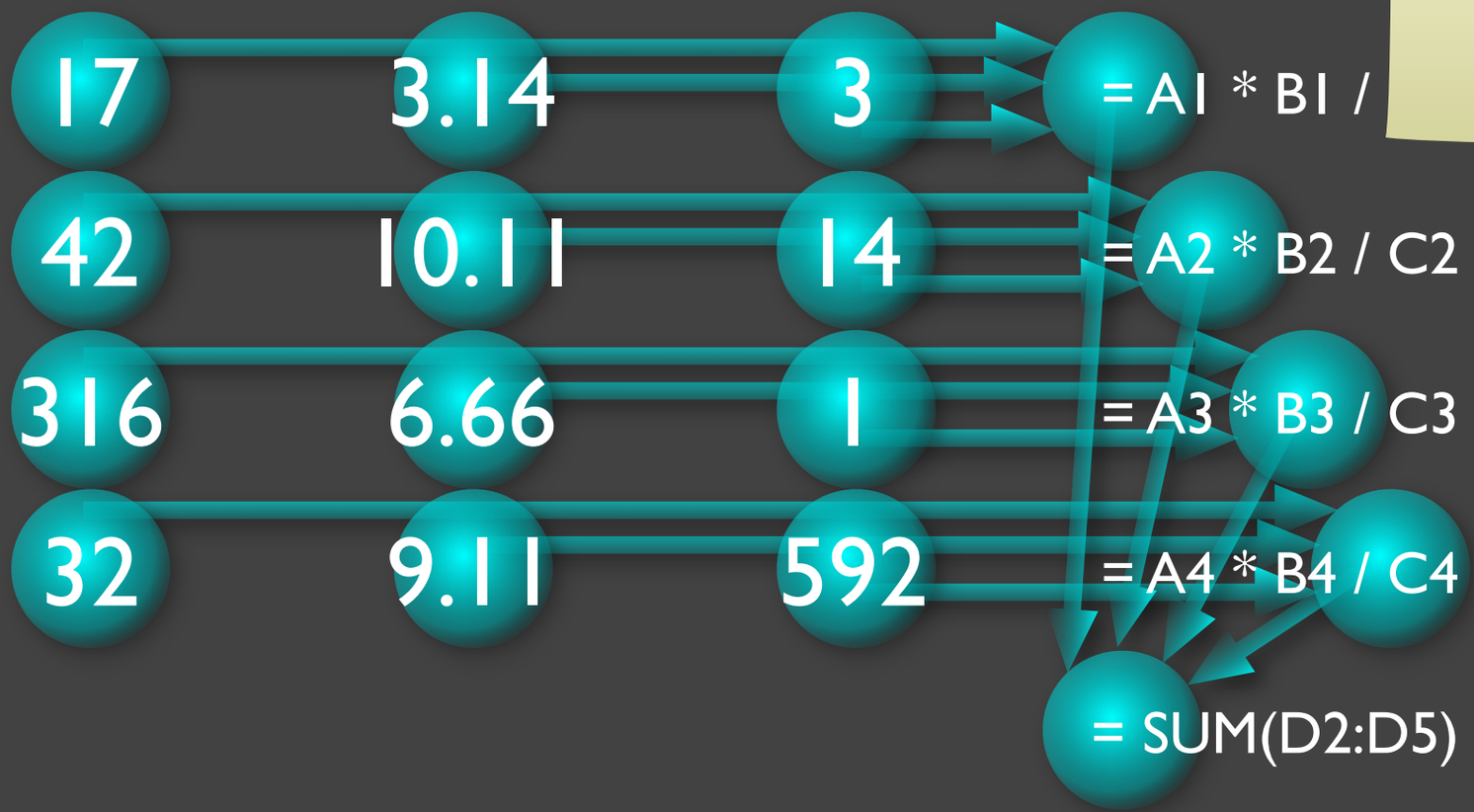
	A	B	C	D	...
1	17	3.14	3	= A1 * B1 / C1	
2	42	10.11	14	= A2 * B2 / C2	
3	316	6.66	1	= A3 * B3 / C3	
4	32	9.11	592	= A4 * B4 / C4	
5				= SUM(D2:D5)	
...					

Graphs are all around us

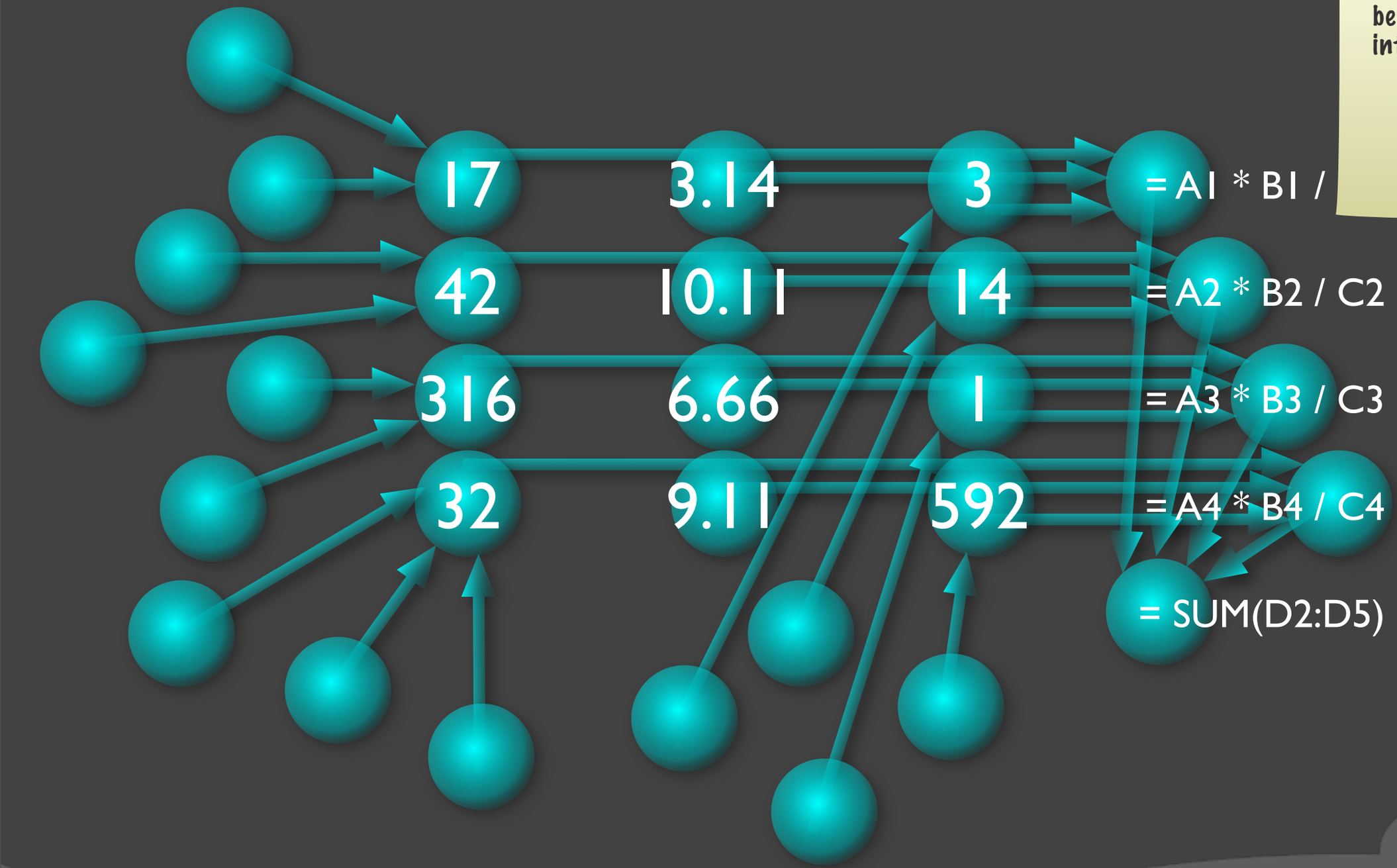
	A	B	C	D	...
1	17	3.14	3	= A1 * B1 / C1	
2	42	10.11	14	= A2 * B2 / C2	
3	316	6.66	1	= A3 * B3 / C3	
4	32	9.11	592	= A4 * B4 / C4	
5				= SUM(D2:D5)	
...					

Graphs are all around us

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

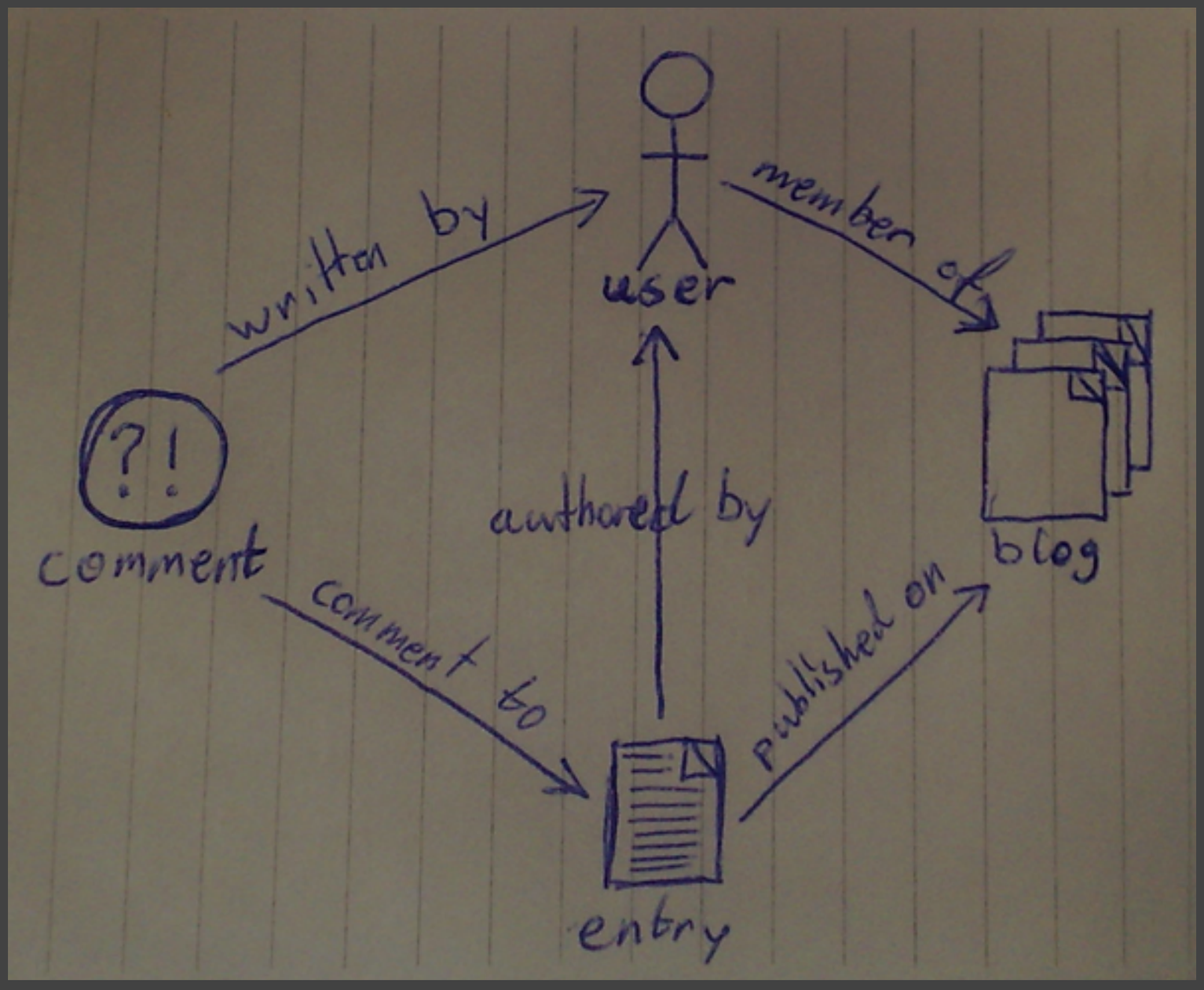


Graphs are all around us

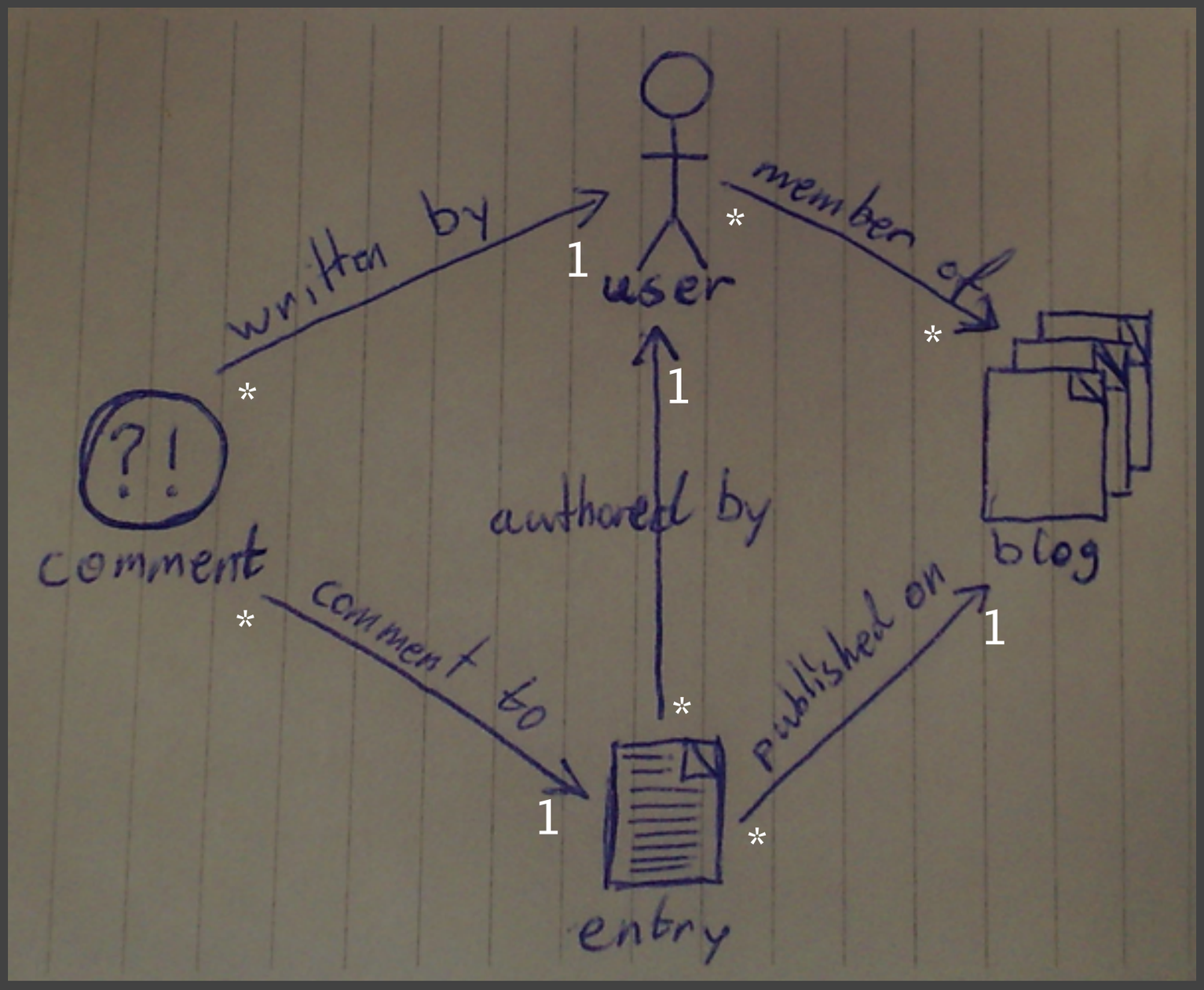


If we add external data sources the problem becomes even more interesting...

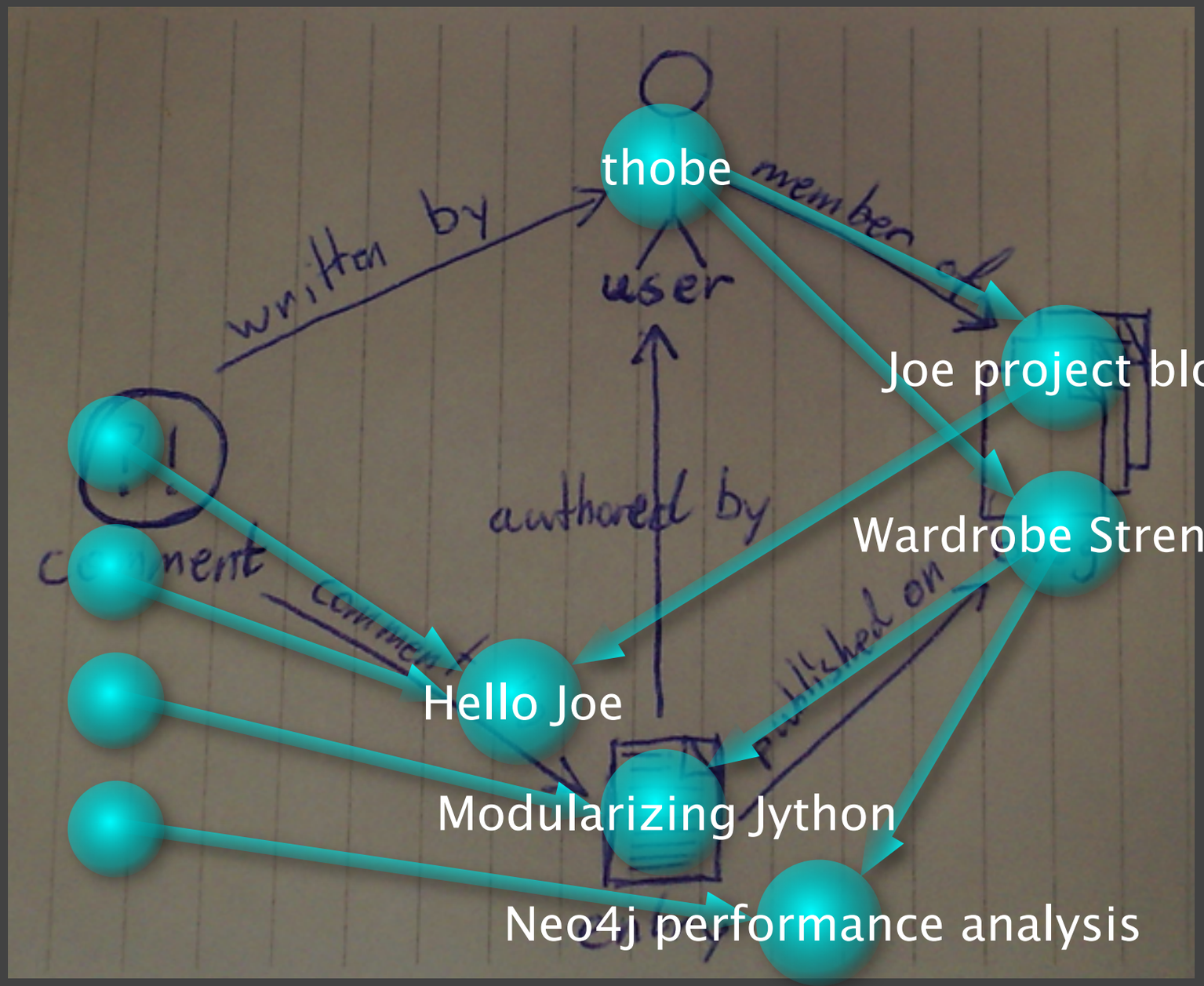
Graphs are whiteboard friendly



Graphs are whiteboard friendly



Graphs are whiteboard friendly



Query Languages

- Traversal API
- Sparql - "SQL for linked data"

```
SELECT ?person WHERE {
    ?person neo4j:KNOWS ?friend .
    ?friend neo4j:KNOWS ?foe .
    ?foe neo4j:name "Larry Ellison" .
}
```

- Gremlin - "perl for graphs"

```
./outE[@label='KNOWS']/inV[@age > 30]/@name
```

Python integration for Neo4j

- Mapping of the core Neo4j API for Python
  - Making it feel "Pythonic"
- Available from the Neo4j repository (and soon from PyPI)
  - http://components.neo4j.org/neo4j.py
    - svn co http://svn.neo4j.org/components/neo4j.py/trunk neo4j-python
- Works with both Jython and CPython
  - The threading of Jython is a plus with an embedded db...
- Comes with Django empowering batteries included
  - Could have support for other frameworks in the future

Simple interaction

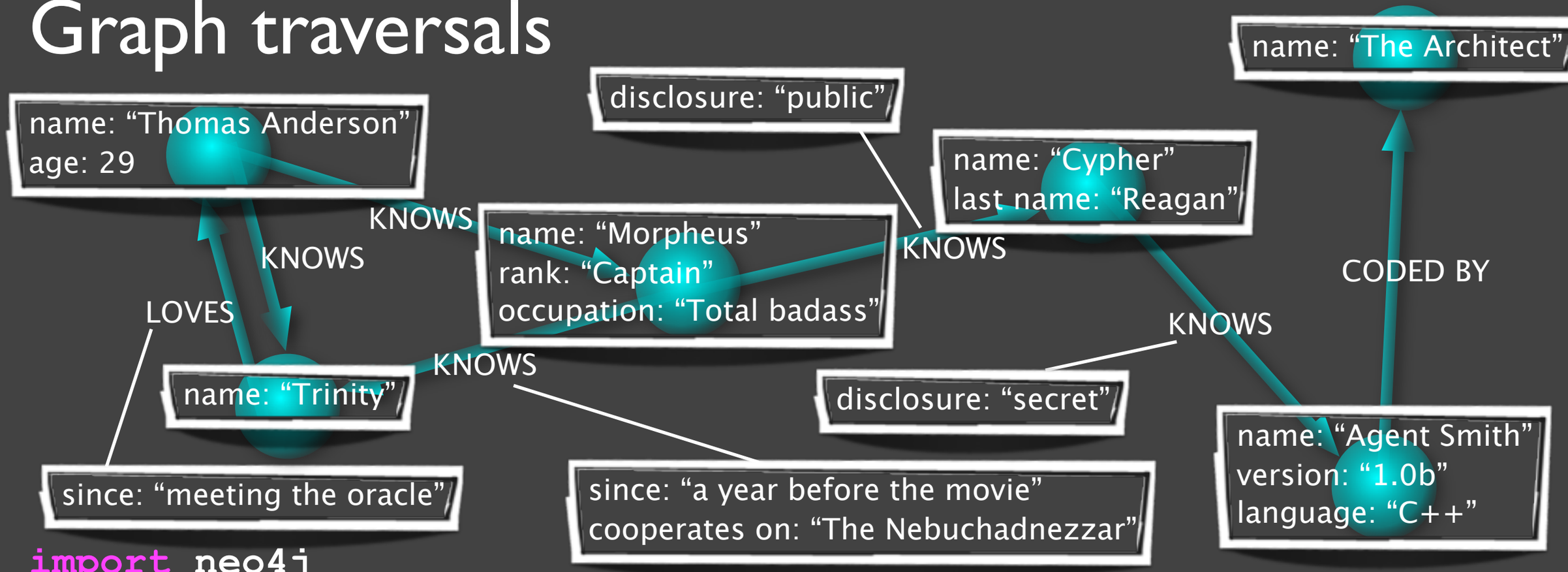
```
import neo4j
graphdb = neo4j.GraphDatabase("var/neo")

with graphdb.transaction:
    james = graphdb.node(name="James", age=32, twitter="@spam")
    mary = graphdb.node(name="Mary", age=35)
    the_car = graphdb.node(brand="Volvo", model="V70")
```

```
james.LOVES( mary )
mary.LOVES( james )
james.LIVES_WITH( mary )
james.OWNS( the_car, property_type="car" )
mary.DRIVES( the_car )
```

Creates the graph we saw in the first example.

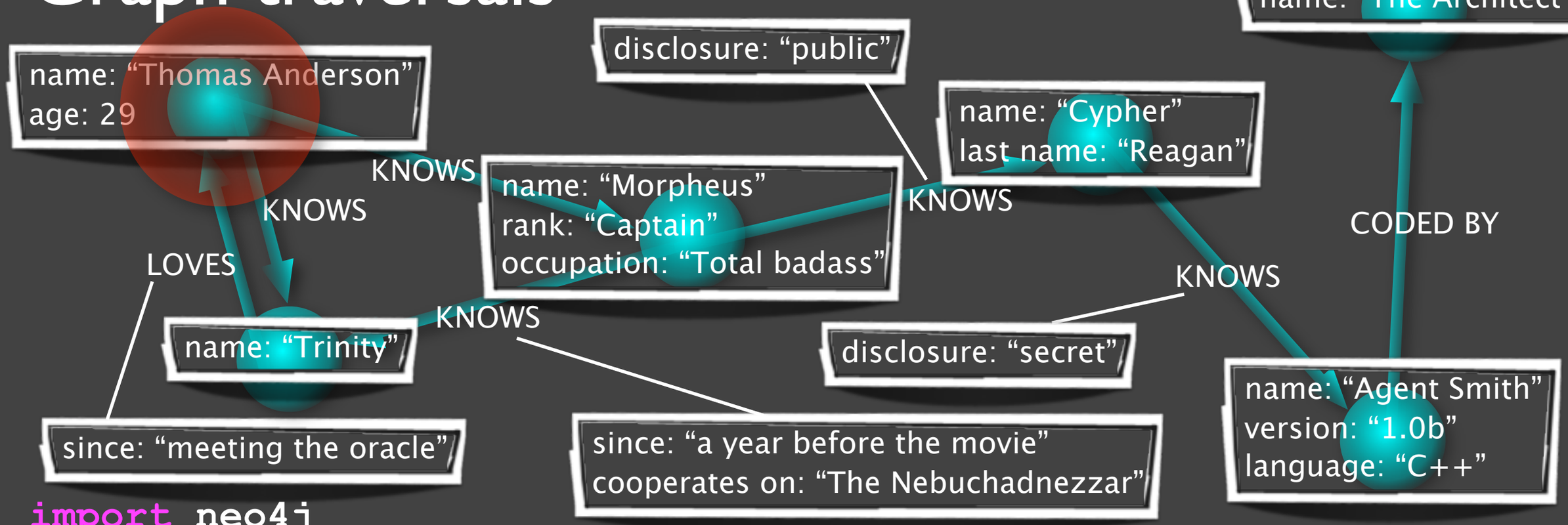
Graph traversals




```
import neo4j
class Friends(neo4j.Traversal): # Traversals ≈ queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREDTH_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 )
```

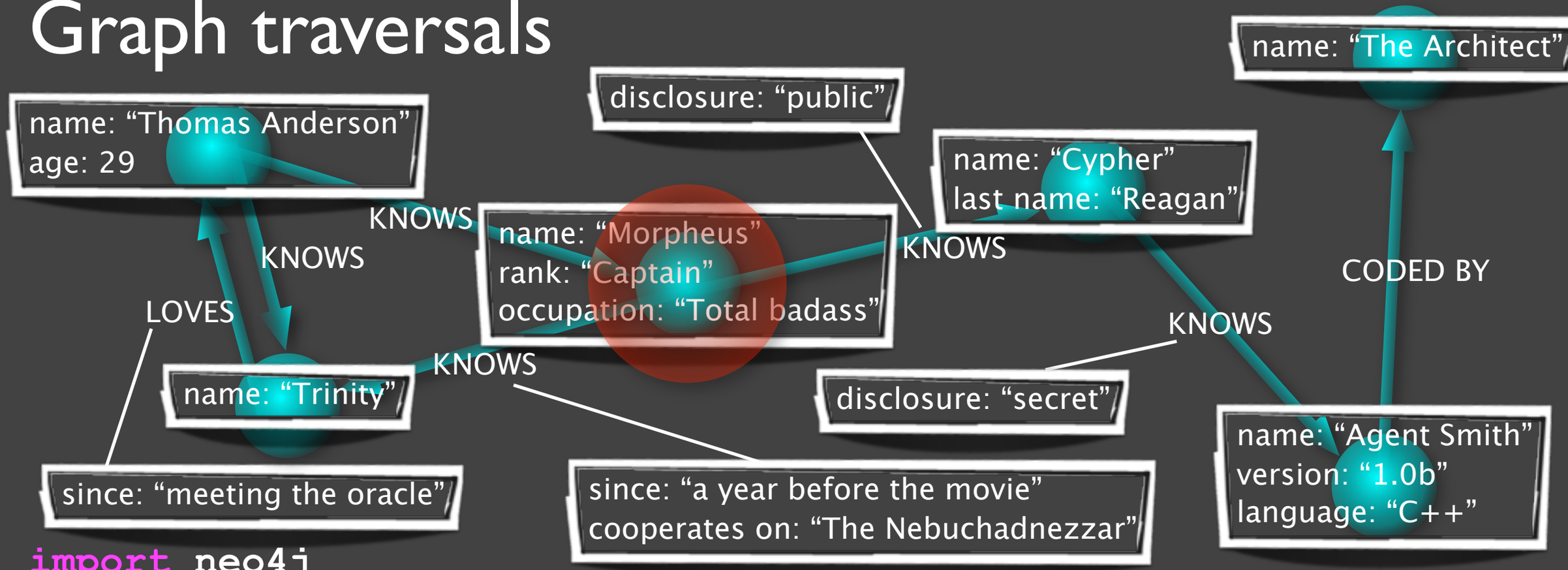
Graph traversals




```
import neo4j
class Friends(neo4j.Traversal): # Traversals ≈ queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREDTH_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

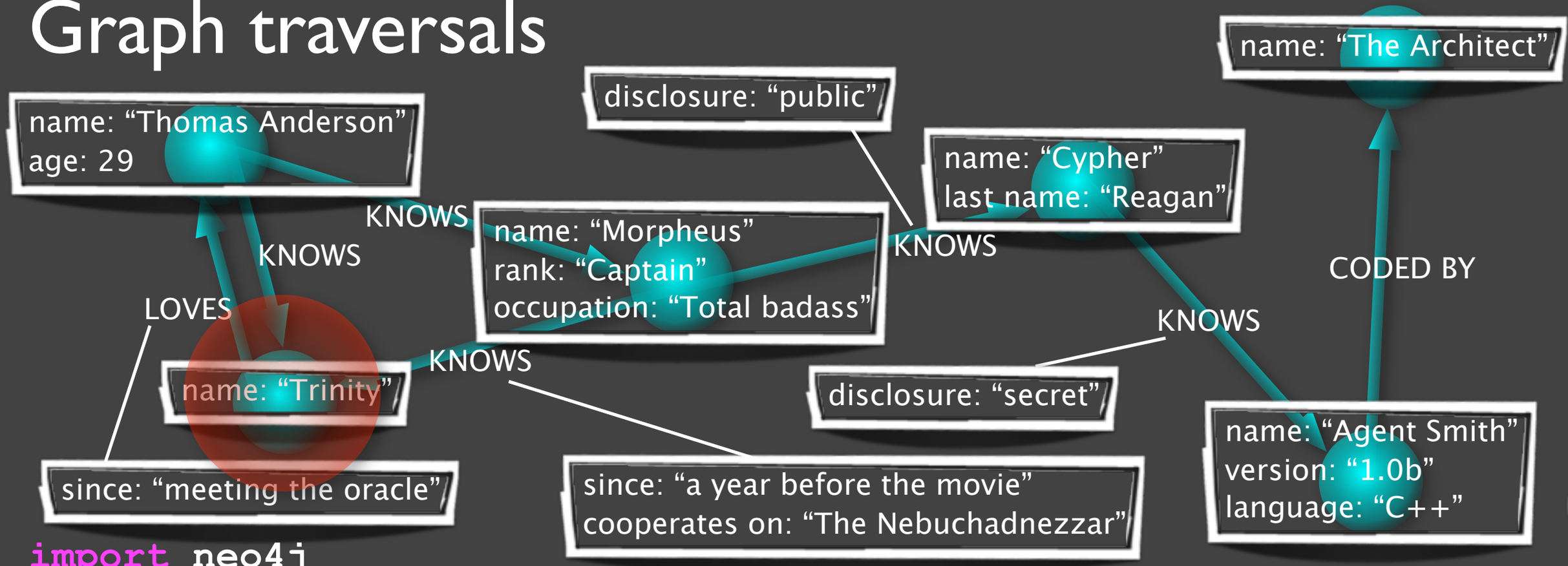



```
import neo4j
class Friends(neo4j.Traversal): # Traversals ≈ queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREDTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

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

```
Morpheus (@ depth=1)
```

Graph traversals

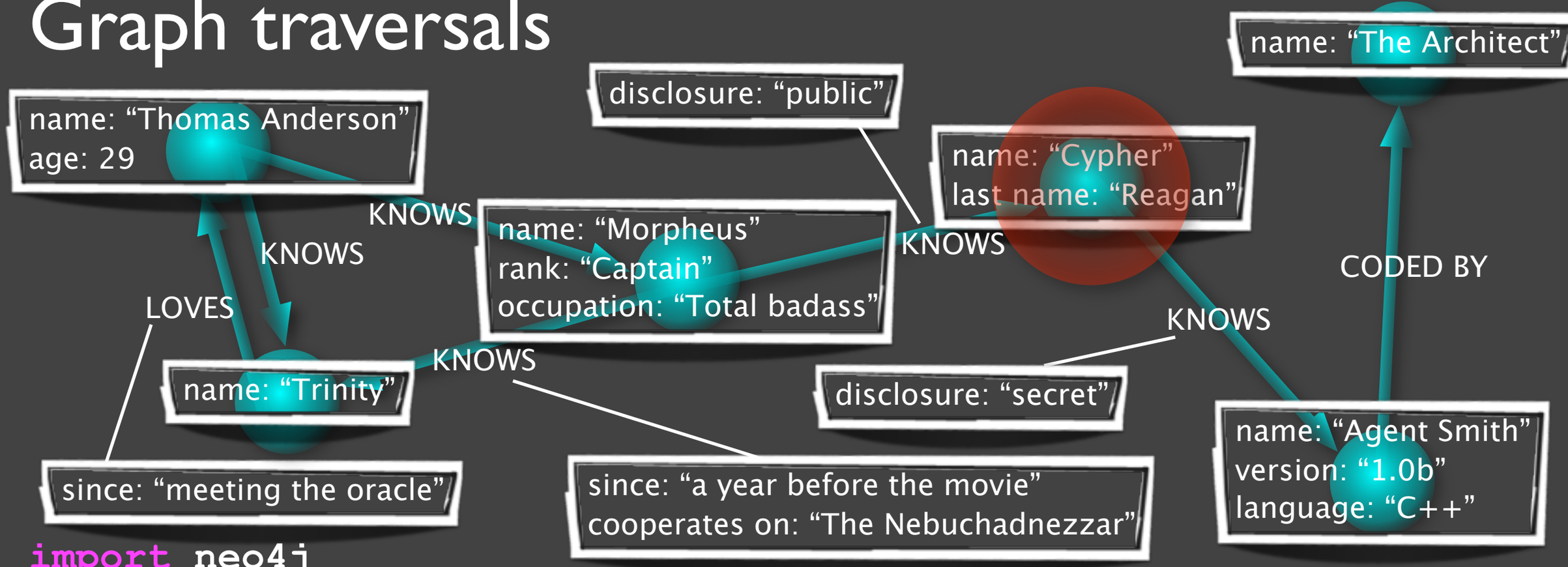



```
import neo4j
class Friends(neo4j.Traversal): # Traversals ≈ queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREDTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

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

```
Morpheus (@ depth=1)
Trinity (@ depth=1)
```

Graph traversals

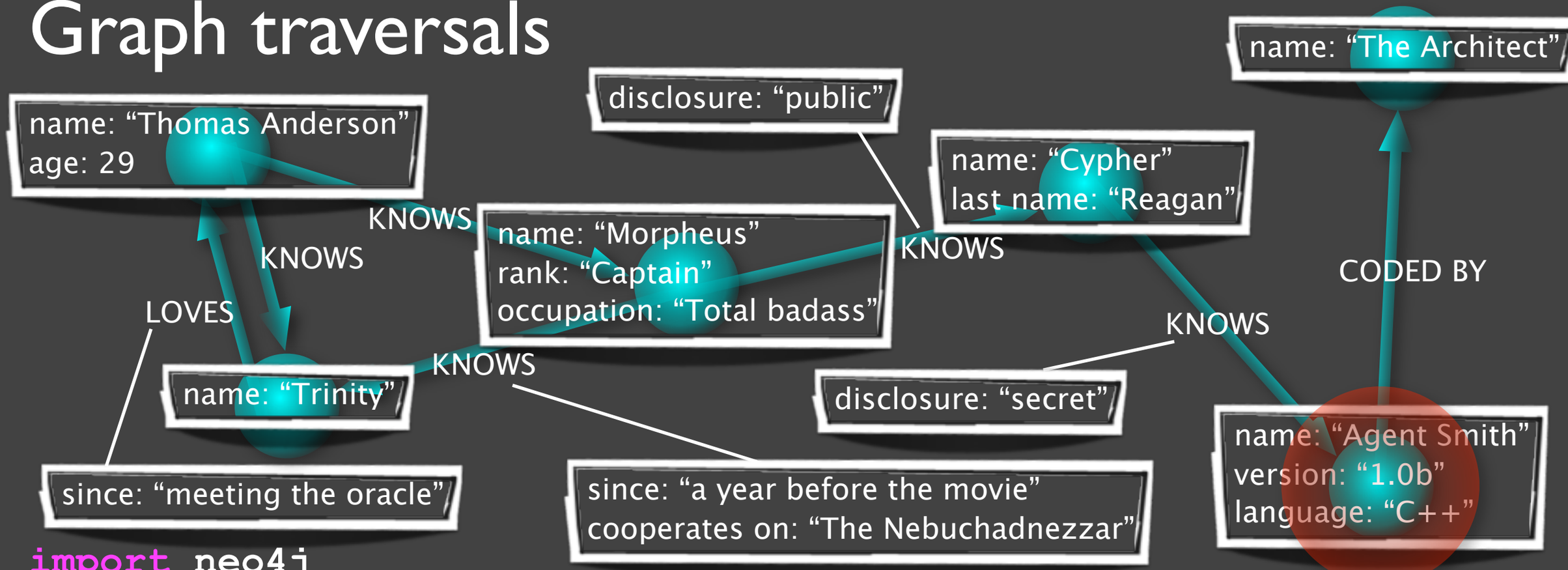



```
import neo4j
class Friends(neo4j.Traversal): # Traversals ≈ queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREDTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

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

```
Morpheus (@ depth=1)
Trinity (@ depth=1)
Cypher (@ depth=2)
```

Graph traversals

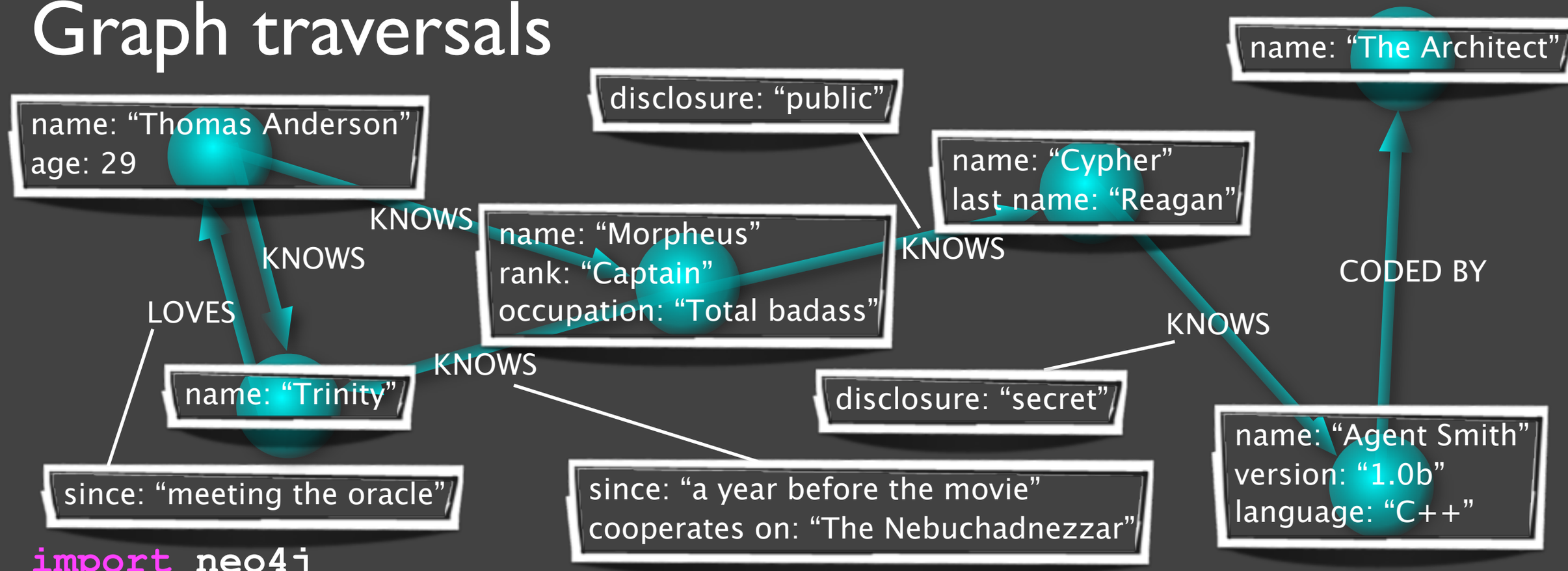



```
import neo4j
class Friends(neo4j.Traversal): # Traversals ≈ queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREDTH_FIRST
    stop = neo4j.STOP_AT_END_OF_GRAPH
    returnable = neo4j.RETURN_ALL_BUT_START_NODE

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

```
Morpheus (@ depth=1)
Trinity (@ depth=1)
Cypher (@ depth=2)
Agent Smith (@ depth=3)
```

Graph traversals




```
import neo4j
class Friends(neo4j.Traversal): # Traversals ≈ queries in Neo4j
    types = [ neo4j.Outgoing.KNOWS ]
    order = neo4j.BREDTH_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 )
```

```
Morpheus (@ depth=1)
Trinity (@ depth=1)
Cypher (@ depth=2)
Agent Smith (@ depth=3)
```

Batteries for Django

```
from neo4j.model import django_model as models

class Movie(models.NodeModel):
    title = models.Property(indexed=True)
    year = models.Property()
    href = property(lambda self: ('/movie/%s/' %
       (self.node.id,)))
    def __unicode__(self):
        return self.title

class Actor(models.NodeModel):
    name = models.Property(indexed=True)
    href = property(lambda self: ('/actor/%s/' %
       (self.node.id,)))
    def __unicode__(self):
        return self.name

# etc. ...
```

"My ORM already does this"

- ORMs and model evolution is a hard problem
  - virtually unsupported in Django
- SQL is a "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 strange auto-generated
- Object/Graph Mapping is *always* done the same
  - allows you to keep your data through application changes
  - or share data between multiple implementations

What your ORM doesn't do

- Drop down to underlying graph model
  - Traversals
  - Graph algorithms
  - Shortest path(s)
  - etc.

Buzzword summary

http://neo4j.org/

AGPLv3

SPARQL

Open Source

ACID

Object mapping

Shortest path

NOSQL

startup friendly

whiteboard friendly

Traversal

Embedded

Query language

Beer

Software Transactional Memory

polyglot persistence

Free Software

Scaling to complexity

http://neotechnology.com