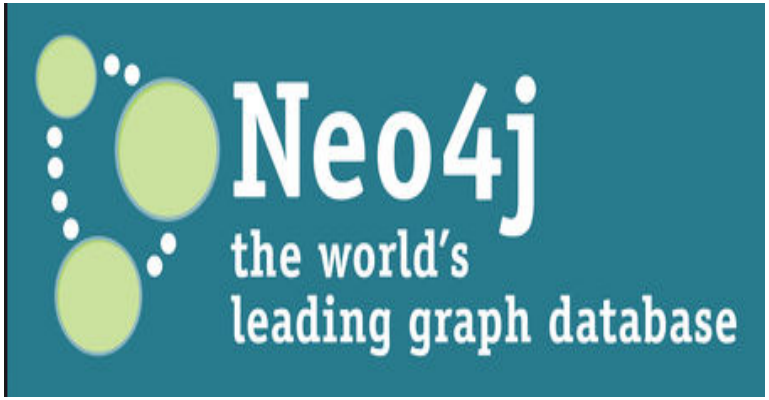
# Introducing Neo4j

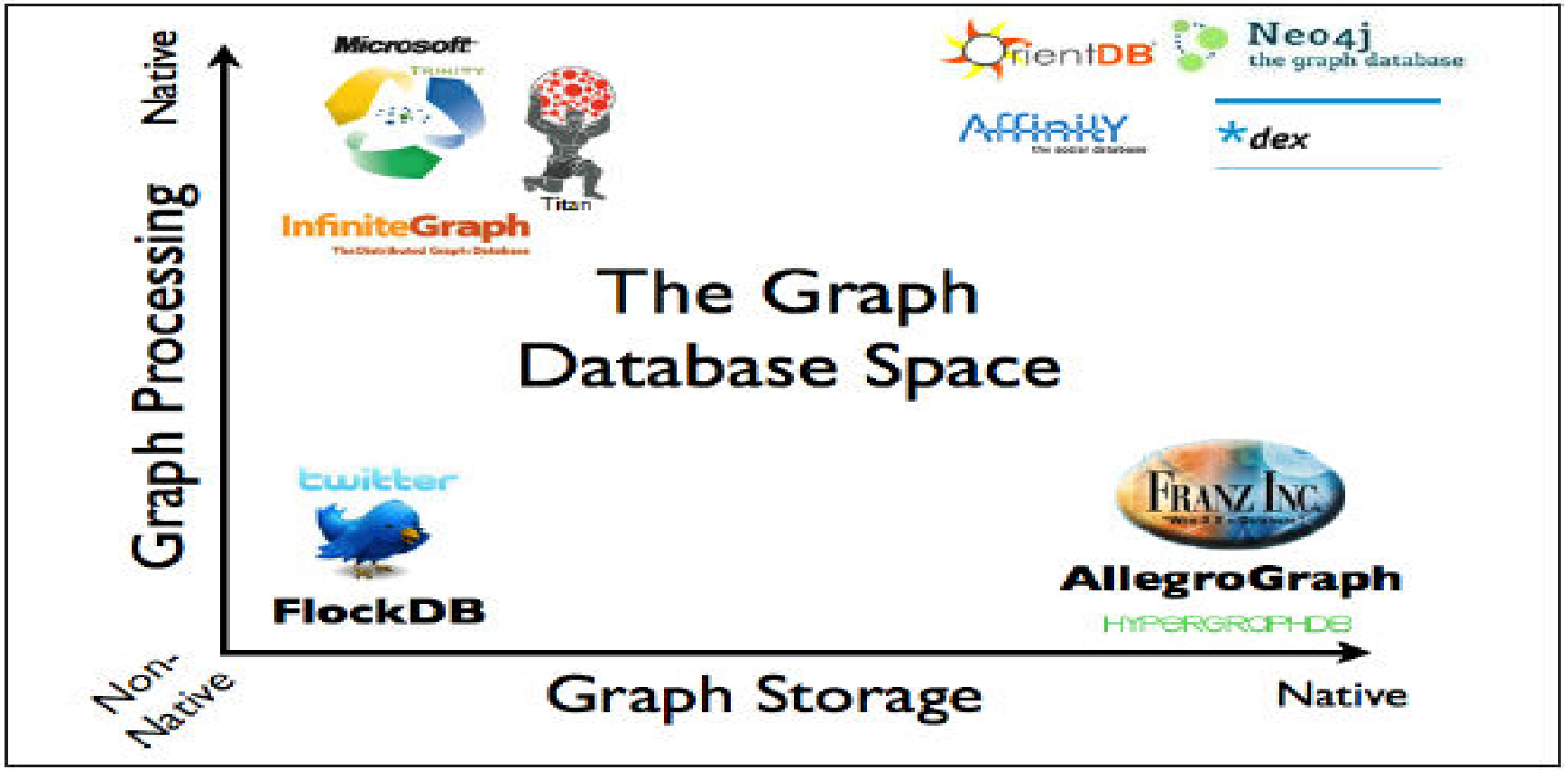
- **Introduced in 2010**
- **Open source**
- **Java-based**
- **NoSQL Graph Database**


# What is Neo4j

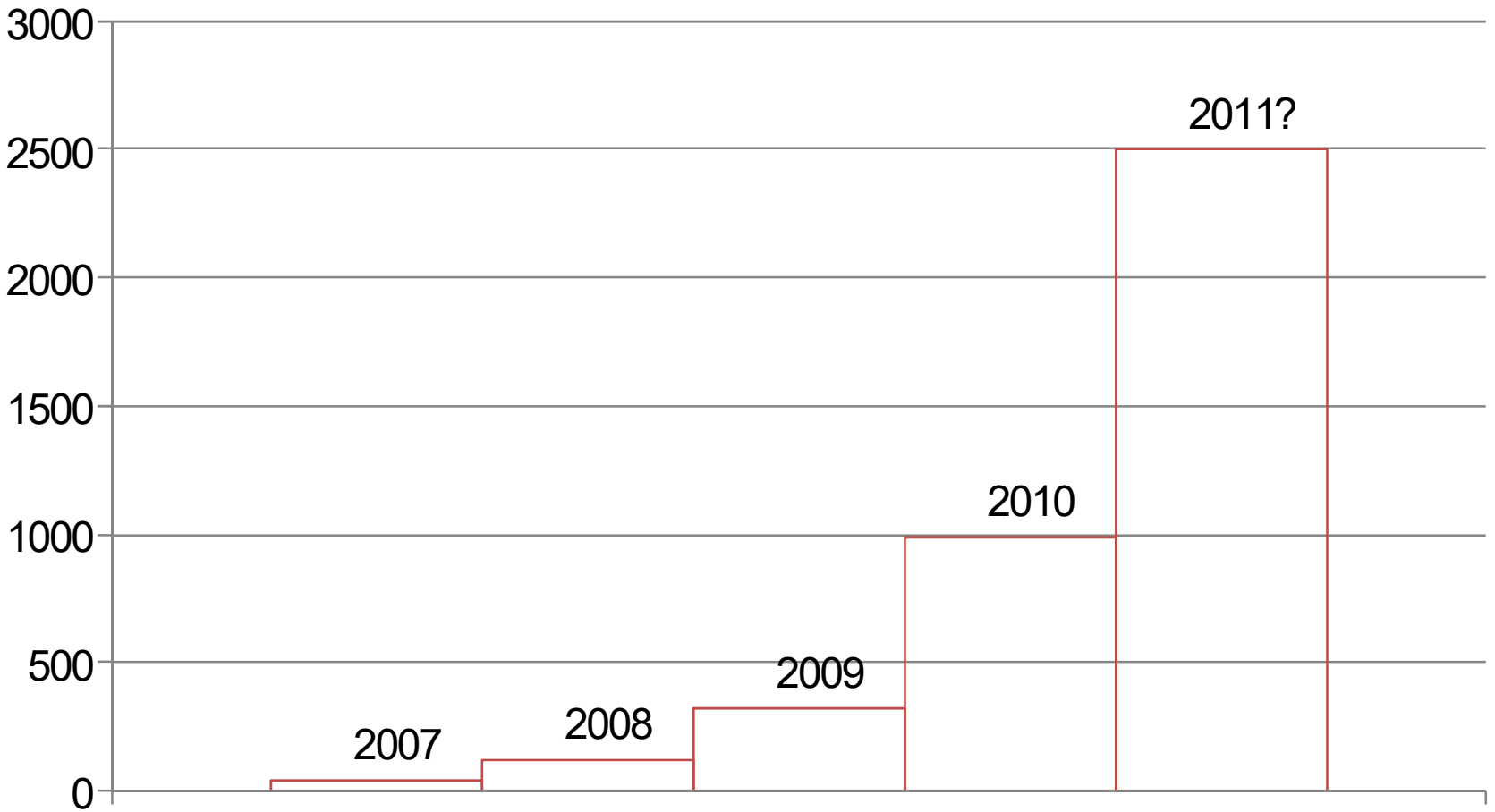
- Developed by Neo Technologies
- Most Popular Graph Database
- Implemented in Java
- Open Source

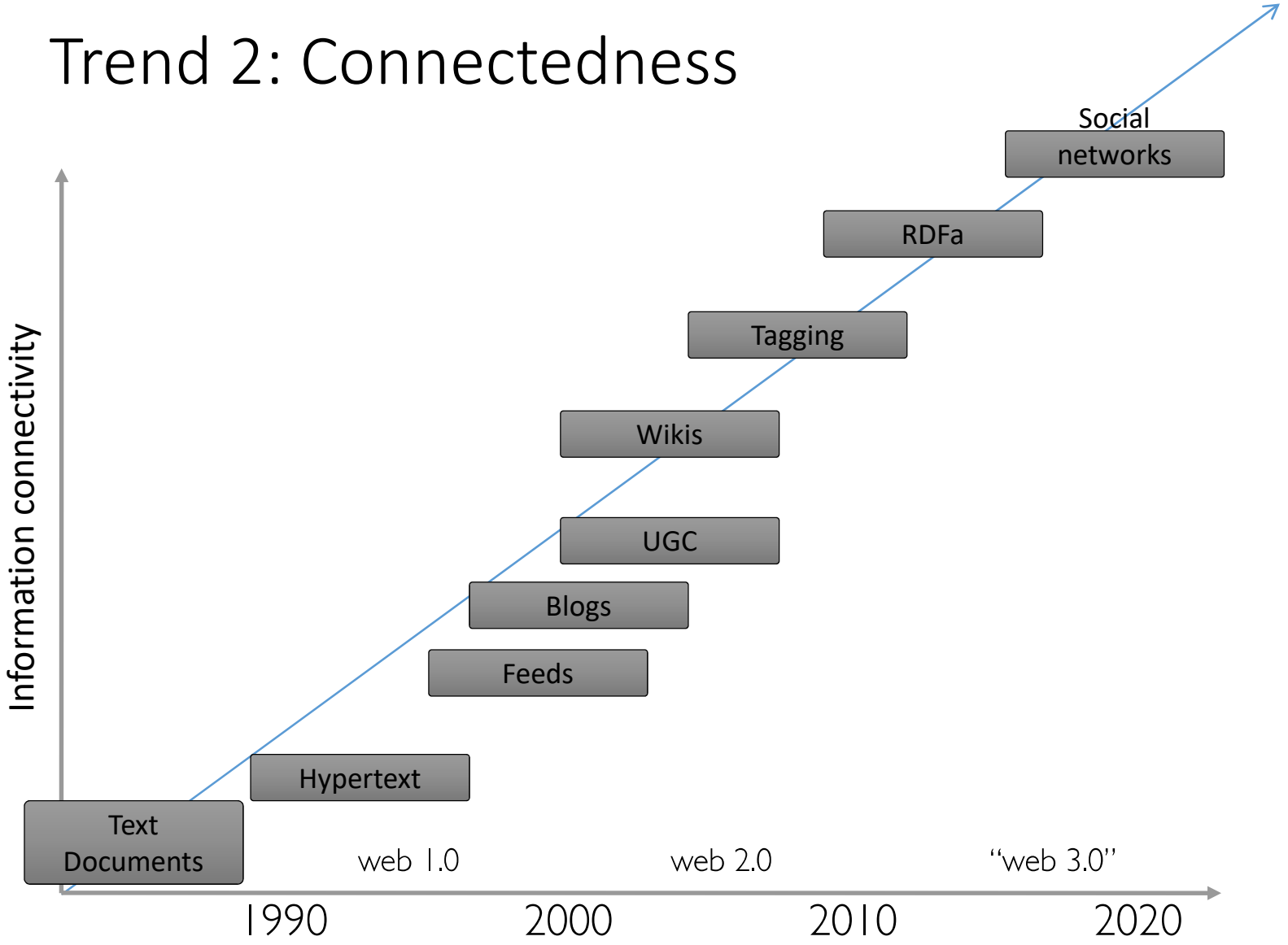
# The Graph Database Space



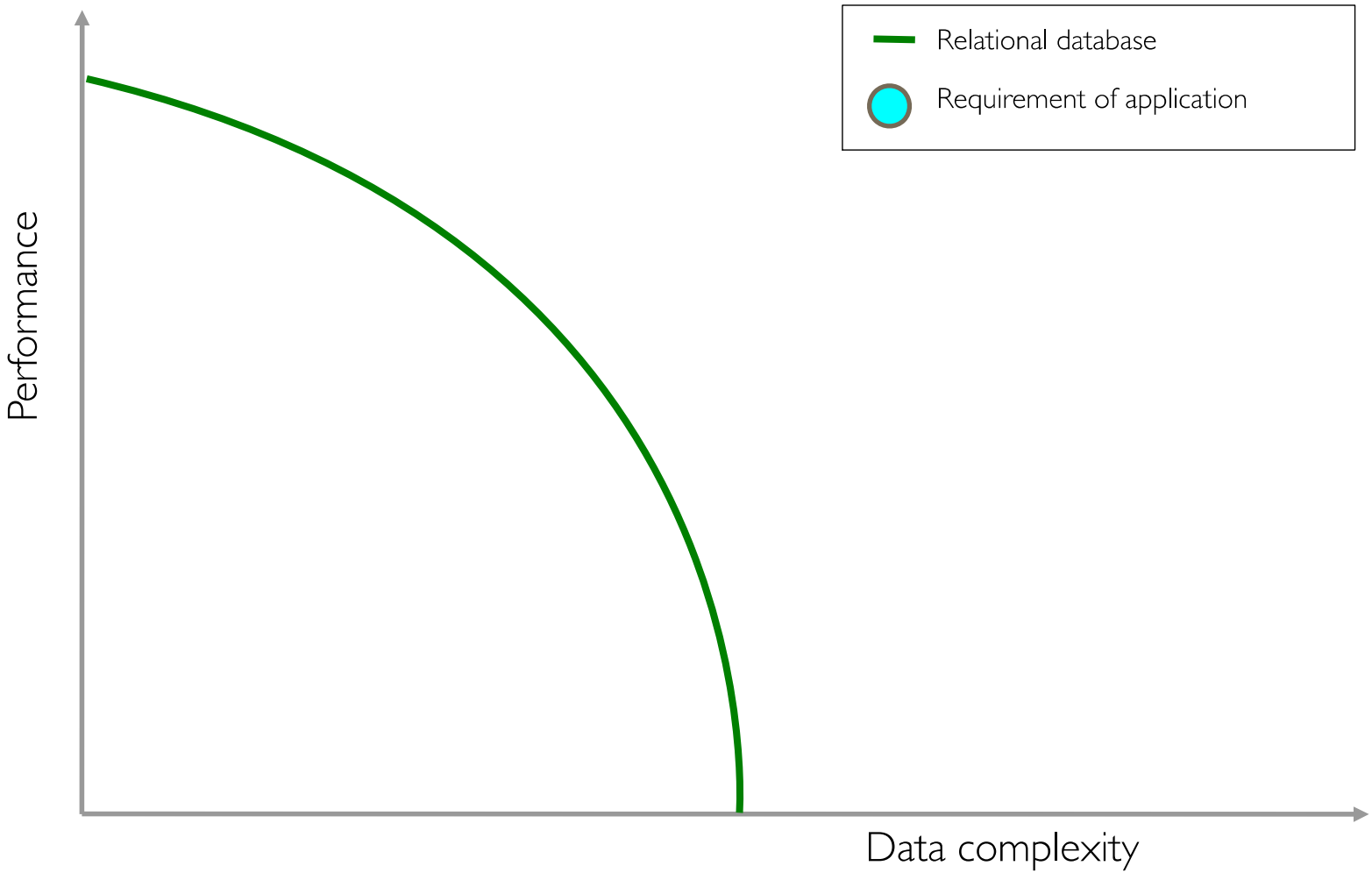
# Trend 1: Data Size



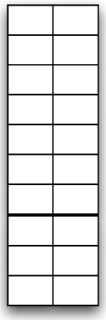
# Trend 2: Connectedness



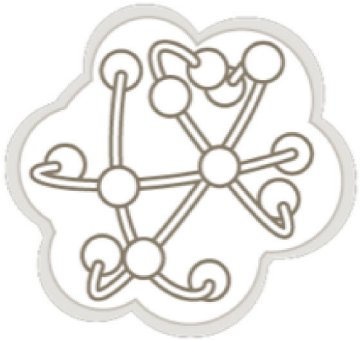
# Side note: RDBMS performance



**Key-Value**

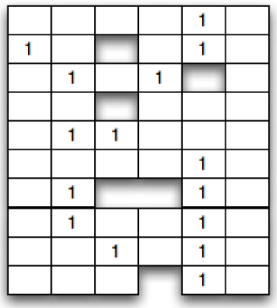


**Graph DB**

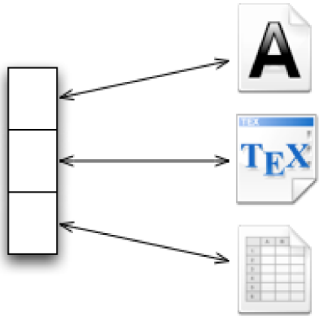


# Four NOSQL Categories

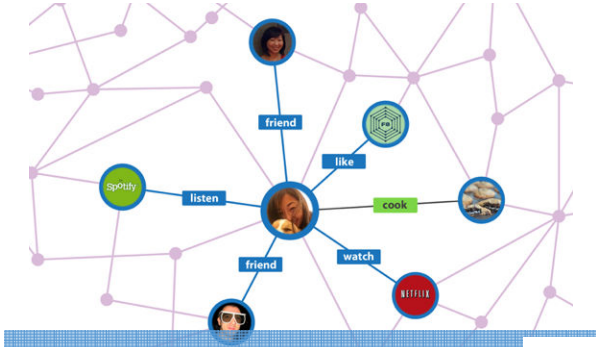
**BigTable**



**Document**



# Everyone is talking about graphs...




Facebook Open Graph

**Google Just Got A Whole Lot Smarter, Launches Its Knowledge Graph**

FREDERIC LARDINOIS

Bing one-ups knowledge graph, hires Encyclopaedia Britannica to supply results

*By Daniel Cooper posted Jun 8th 2012 3:02PM*


Neo4j
The World's Leading Graph Database

**Why the Interest Graph Is a Marketer's Best Friend**

1 day ago by Nadim Hossain

5

**Introducing Graph Search**

# But why?

- Knowledge graph: beyond links, search is smarter when considering how things are related
- Facebook graph search: people are most interested in finding things in their part of the world
- Bing+Britannica: wait a second, we've always thought this way, referencing and cross-referencing
- You: have relationships to people, to organizations, to places, to things -- your personal graph

# A graph...

- you know the common data structures
  - linked lists, trees, object "graphs"
- a graph is the general purpose data structure
  - suitable for any data that is related
- well-understood patterns and algorithms
  - studied since Leonard Euler's 7 Bridges (1736)
  - Codd's Relational Model (1970)
  - not a new idea, just an idea who's time is now

# A graph database…

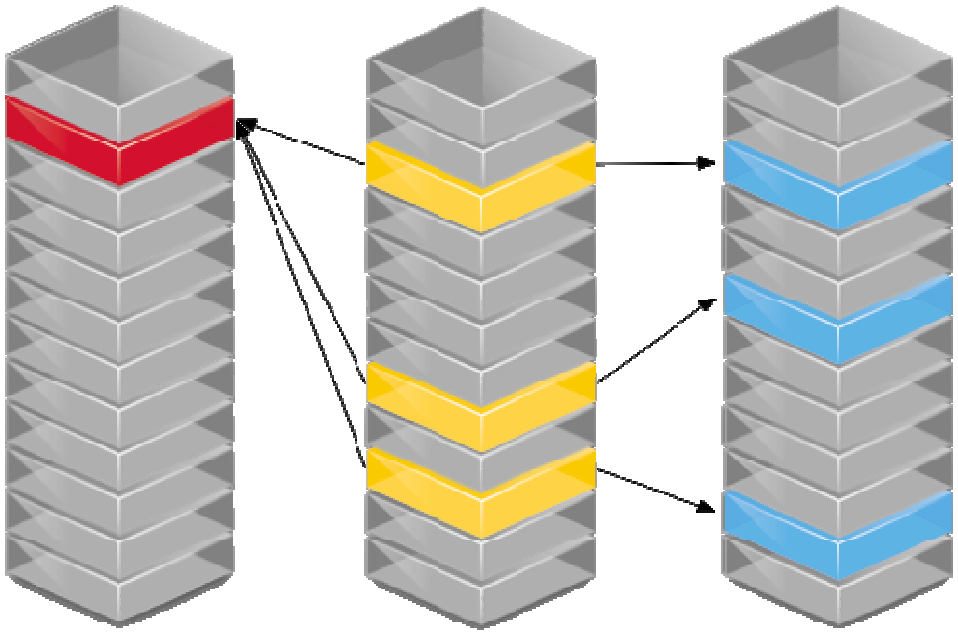
- optimized for the connections between records
- really, really fast at querying across records
- a database: transactional with the usual operations
- “A relational database may tell you how many books you sold last quarter,
- but a graph database will tell your customer which book they should buy next.”

# Predictive Analysis with Graph Theory

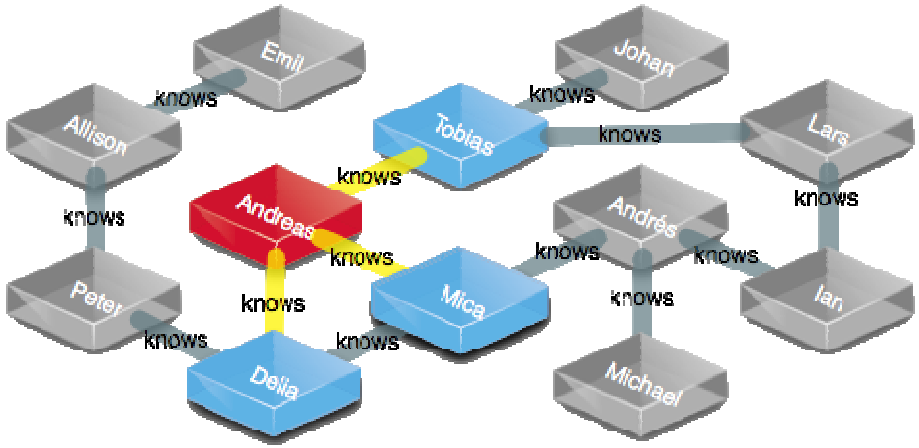
- Depth- and Breadth- First Search
- Path-Finding with Dijkstra's Algorithm
- The A* Algorithm
- Graph Theory and Predictive Modeling
- Local Bridges
- Summary: Graphs are truly remarkable structures. Our understanding of them is rooted in several hundred years of mathematical and scientific study. And yet we're only just beginning to understand how to apply them to our personal, social, and business lives. The technology is here, open and available to all in the form of the modern graph database; the opportunities are endless.
- Graph theory algorithms and analytical techniques are not demanding: we need only understand how to apply them to achieve our goals.

# You know relational

## now consider relationships...



# We're talking about a Property Graph



# Google "neo4j"

- neo4j.org
- neotechnology.com
- github.com/neo4j
- neo4j.meetup.com
- graphconnect.com

# Neo4j is a Graph Database

- A Graph Database:
  - a Property Graph containing Nodes, Relationships
  - with Properties on both
  - perfect for complex, highly connected data
- A Graph Database:
  - reliable with real ACID Transactions
  - scalable: tons and tons of records
  - Server with REST API, or Embeddable on the JVM
  - high-performance with High-Availability (read scaling)

# Cypher - a graph query language

- a pattern-matching query language
- declarative grammar with clauses (like SQL)
- aggregation, ordering, limits
- create, read, update, delete

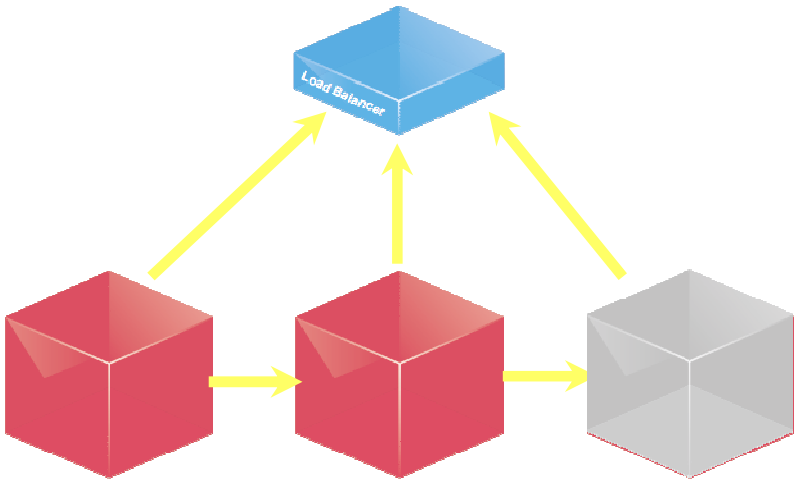
```
// get node from an index named "foo"
start foo=node:people(name='Andreas') return foo

// find "bar" nodes related to Andreas
start foo=node:people(name='Andreas')
match (foo)-->(bar) return bar

// create a node
create (me {name:'Andreas'})
```

# Neo4j HA - High Availability Cluster

- master-slave replication
  - read-scaling
- single datacenter, or global zones
  - tolerance for high-latency
- redundancy provides improved uptime
  - automatic failover



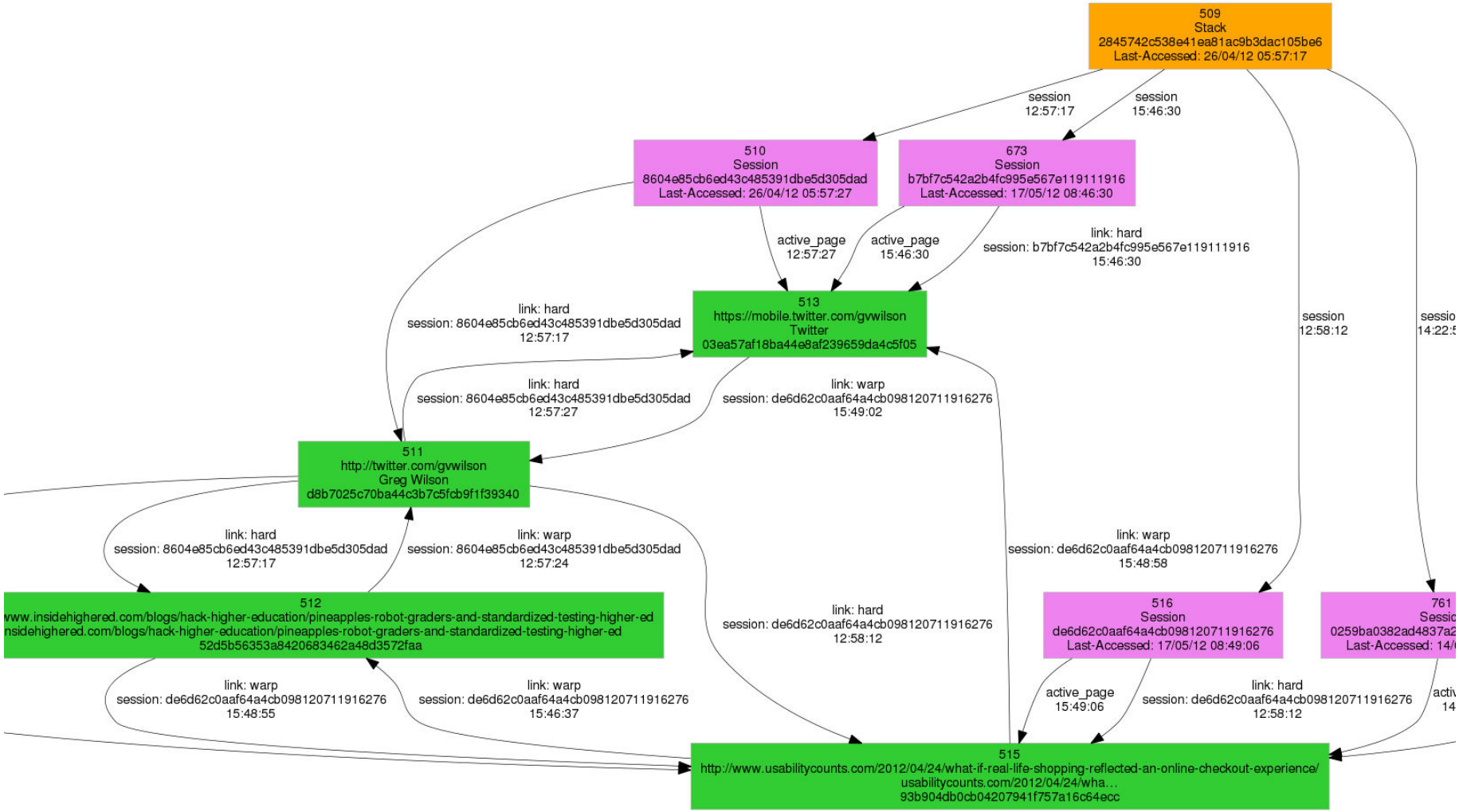
# [A] Mozilla Pancake

- Experimental cloud-based browser
- Built to improve how users Discover, Collect, Share & Organize things on the web
- Goal: help users better access & curate information on the net, on any device

# Why Neo4J?

- The internet is a network of pages connected to each other. What better way to model that than in graphs?
- No time lost fighting with less expressive datastores
- Easy to implement experimental features

# Cute meta + data

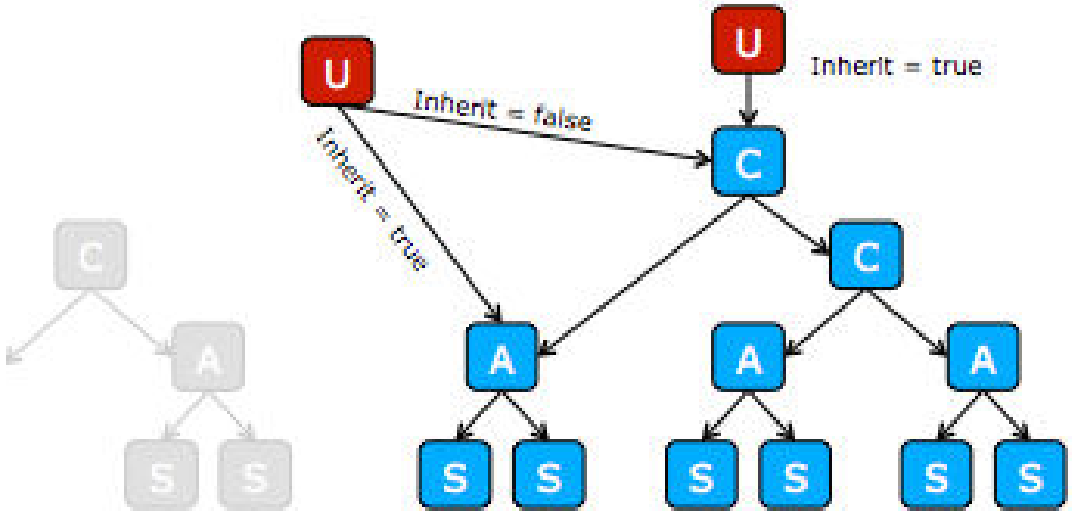


# Neo4J Co-Existence

- Node uuids as refs in external ElasticSearch also in internal Lucene
- Custom search ranking for user history based on node relationship data
- MySQL for user data, Redis for metrics

# Example Access Authorization

Access may be given directly or by inheritance



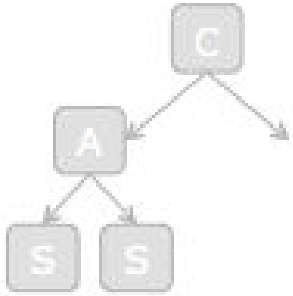

User

Customer

Account

Subscription

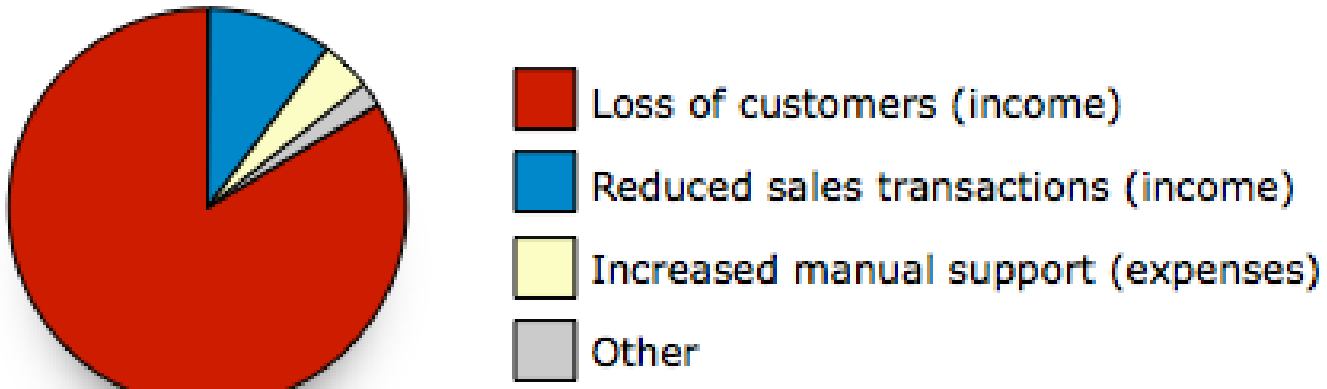



telenor

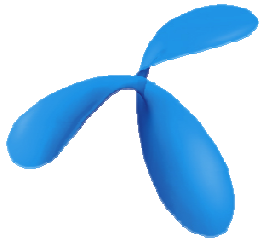
# Business Case

The business case is built on the negative consequence of NOT addressing the problem.



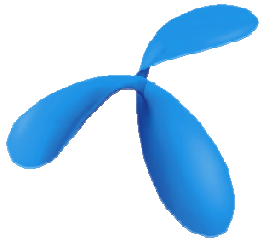
# Current ACL Service

- Stored procedure in DB calculating all access
  - cached results for up to 24 hours
  - minutes to calculate for large customers
  - extremely complex to understand (1500 lines)
  - depends on temporary tables
  - joins across multiple tables



# ACL With Neo4j

- Faster than current solution
- Simpler to understand the logic
  - a dozen or so lines of code
- Avoid large temporary tables
- Tailored for service (resource authorization)



# [C] Master of your Domain

# [C] MDM within Cisco

*master data management, sales compensation management, online customer support*

## Description

Real-time conflict detection in sales compensation management. Business-critical “P1” system. Neo4j allows Cisco to model complex algorithms, which still maintaining high performance over a large dataset.

## Background

Neo4j replaces Oracle RAC, which was not performant enough for the use case.

## Architecture

3-node Enterprise cluster with mirrored disaster recovery cluster

Dedicated hardware in own datacenter

Embedded in custom webapp

## Sizing

35 million nodes

50 million relationships

600 million properties

## Benefits

***Performance : “Minutes to Milliseconds”***

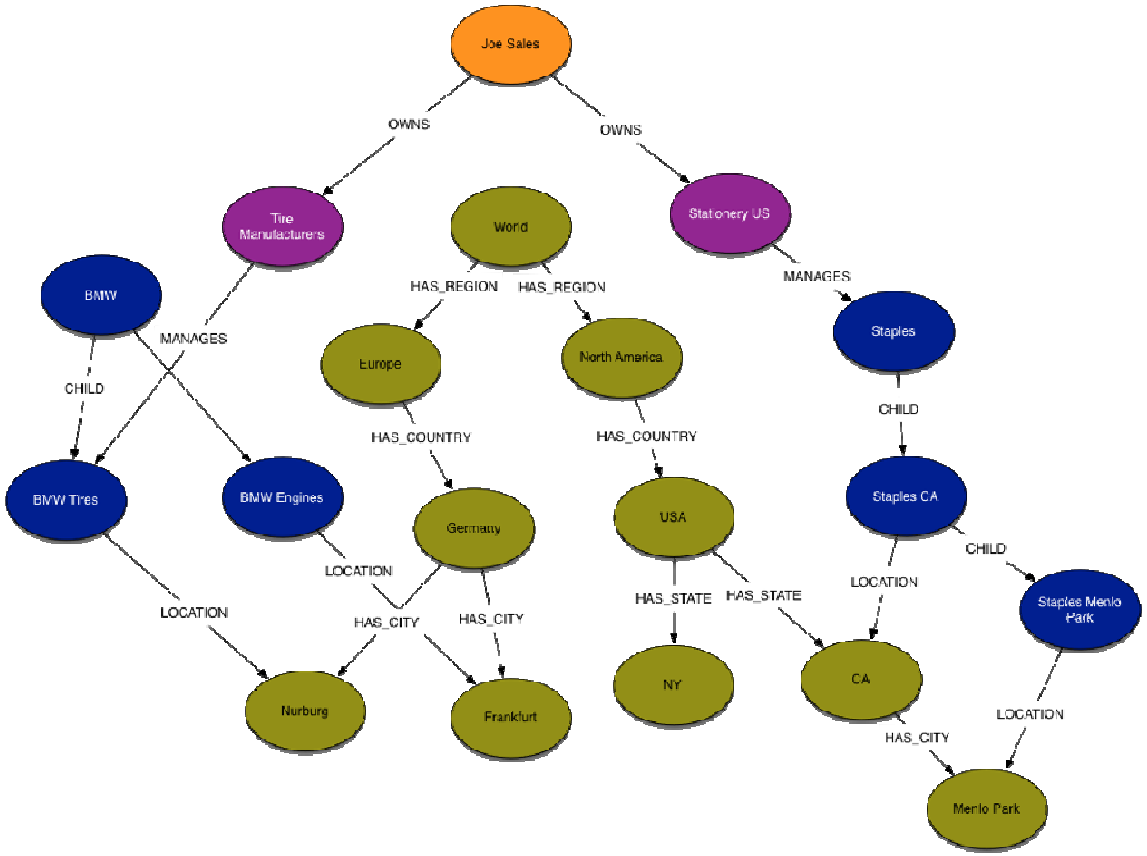
Outperforms Oracle RAC, serving complex queries in real time

***Flexibility***

Allows for Cisco to model interconnected data and complex queries with ease

***Robustness***

With 9+ years of production experience, Neo4j brings a solid product.



Really, once you start
thinking in graphs
it's hard to stop

# What will *you* build?

Business intelligence

Geospatial

catalogs

Systems Management

access control

Social computing

your brain

Biotechnology

routing

genealogy

linguistics

Making Sense of all that data

compensation

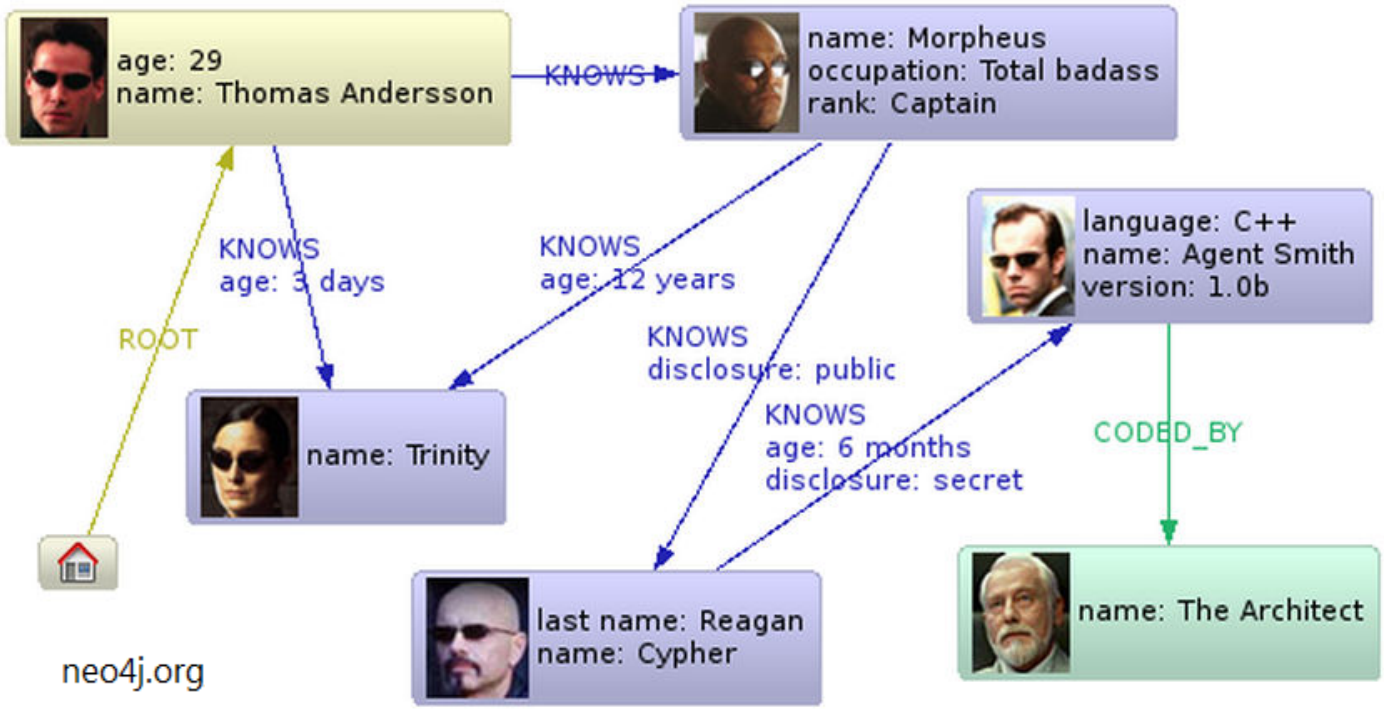
market vectors

# Graph Databases

- Database that uses graph structures with nodes, edges and properties to store data
- Provides index-free adjacency
  - Every node is a pointer to its adjacent element
- Edges hold most of the important information and connect
  - nodes to other nodes
  - nodes to properties
- Nodes represent entities
- Edges represent relationships
- Connections between data are explored
- Faster for associative data sets
- Intuitive
- Optimal for searching social network data

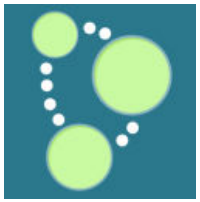
# Graph Database

- Neo4j is a graph database
- Database full of linked nodes
- Stores data as nodes and relationships



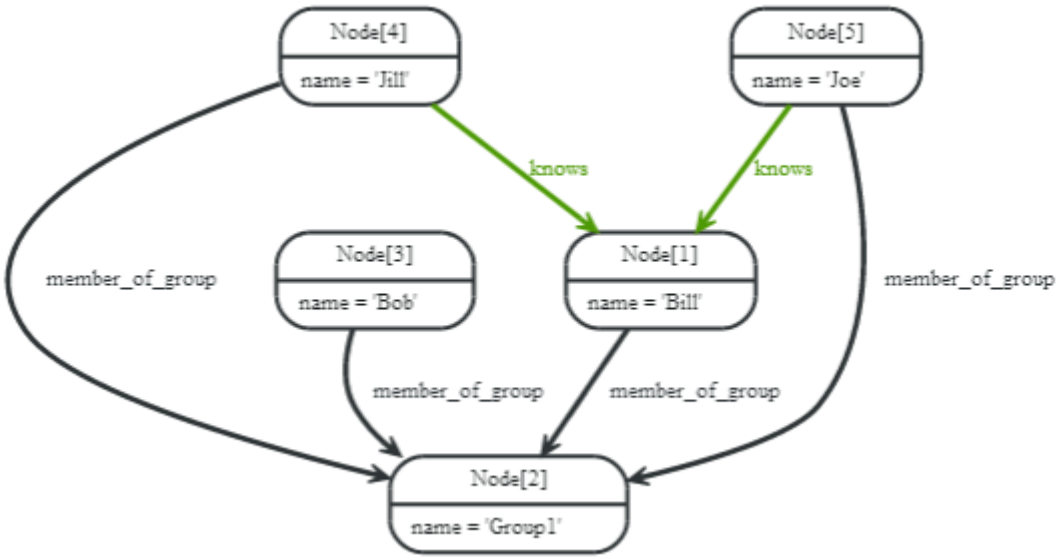
# Graph Database (cont.)

- Nodes represent entities
- Edges represent relationships
- Connections between data are explored
- Faster for associative data sets
- Intuitive
- Optimal for searching social network data



# How Neo4j Works (cont.)

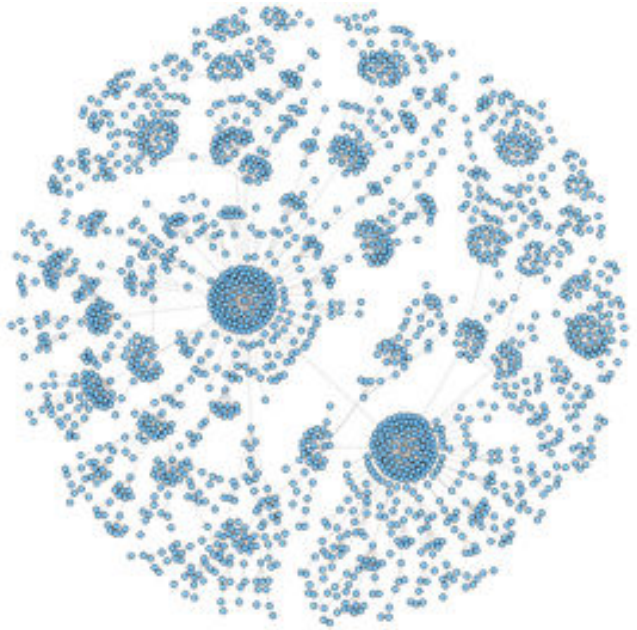
- Dealing with complexity
  - Simple domains can be complex
  - Mutual Friend example




| name | mutualGroups | mutualFriends |
|---|---|---|
| "Jill" | 1 | 1 |
| "Bob" | 1 | 0 |
| 2 rows | | |
| 1 ms | | |

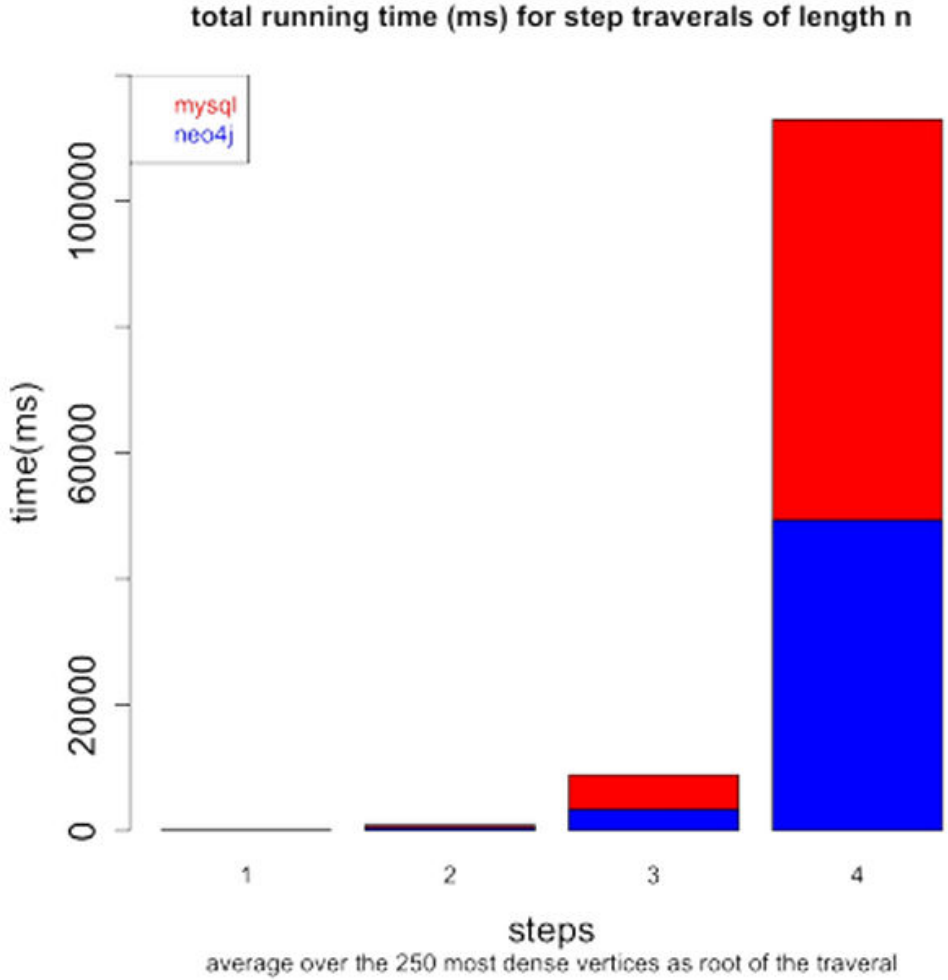
# Case Study

- A side by side comparison of a relational database and Neo4j
- Artificially generated graph dataset with natural statistics
- 1 million vertices and 4 million edges



# Case Study Results

- First 250 vertices used as roots

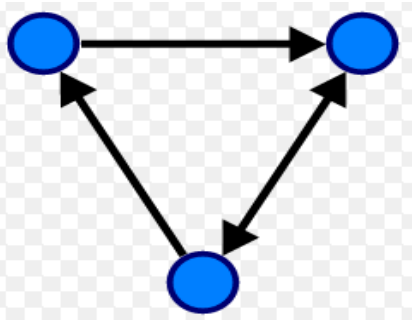


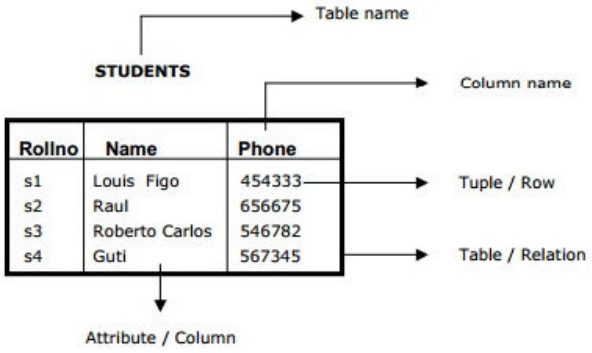
# Advantage of Graph Databases

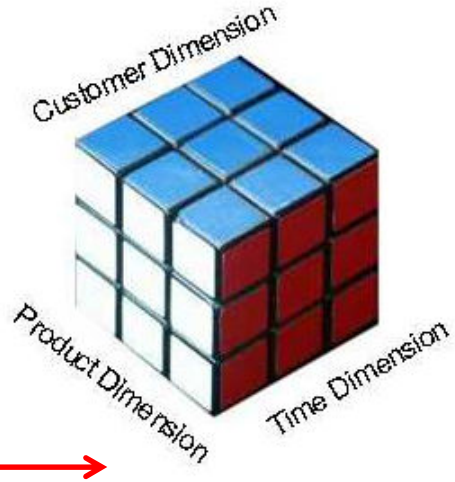
- When there are relationships that you want to analyze Graph databases become a very nice fit because of the data structure
- Graph databases are very fast for associative data sets
  - Like social networks
- Map more directly to object oriented applications
  - Object classification and Parent->Child relationships

# Disadvantages

- If data is just tabular with not much relationship between the data, graph databases do not fare well
- OLAP support for graph databases is not well developed
  - Lots of research happening in this area

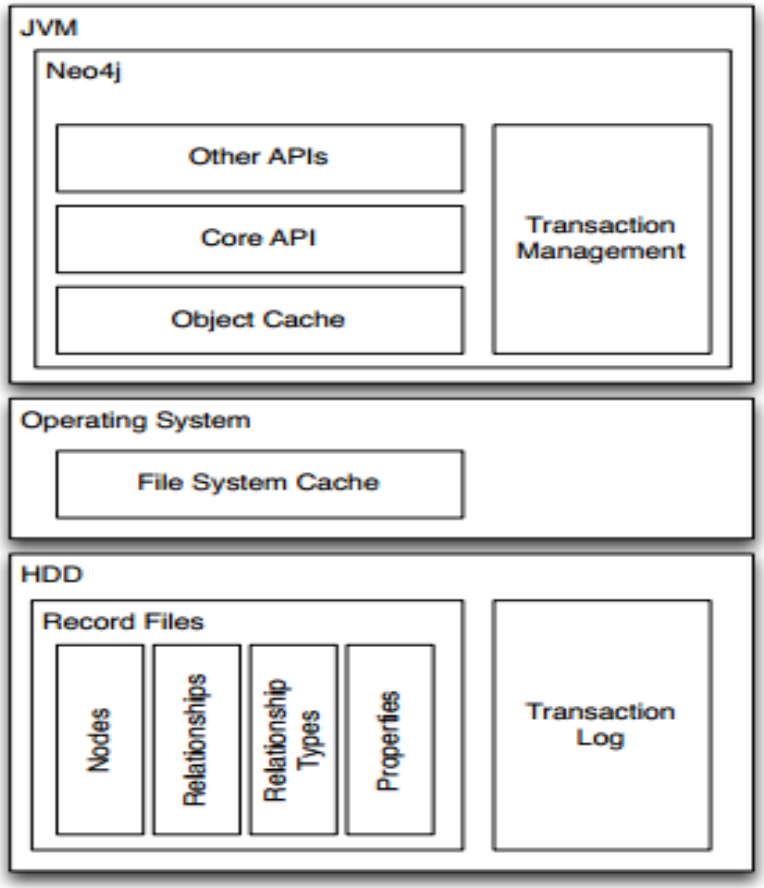





Ease of aggregation

# Neo4j Software Architecture



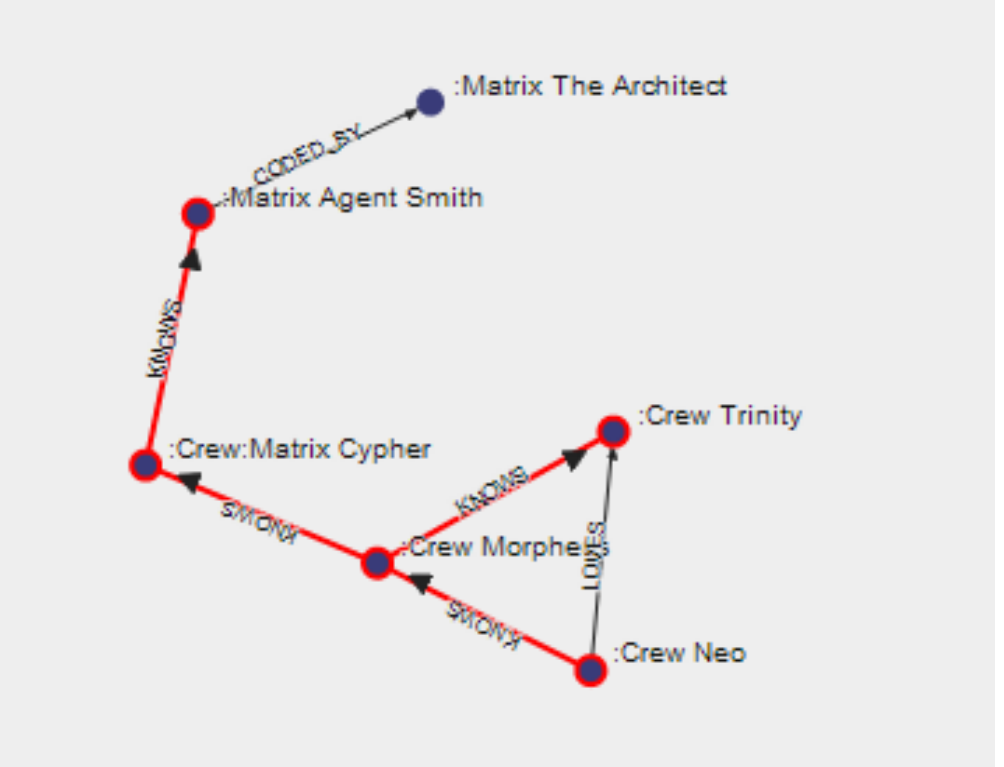
# Cypher

- Query Language for Neo4j
- Easy to formulate queries based on relationships
- Many features stem from improving on pain points with SQL such as join tables

# Cypher

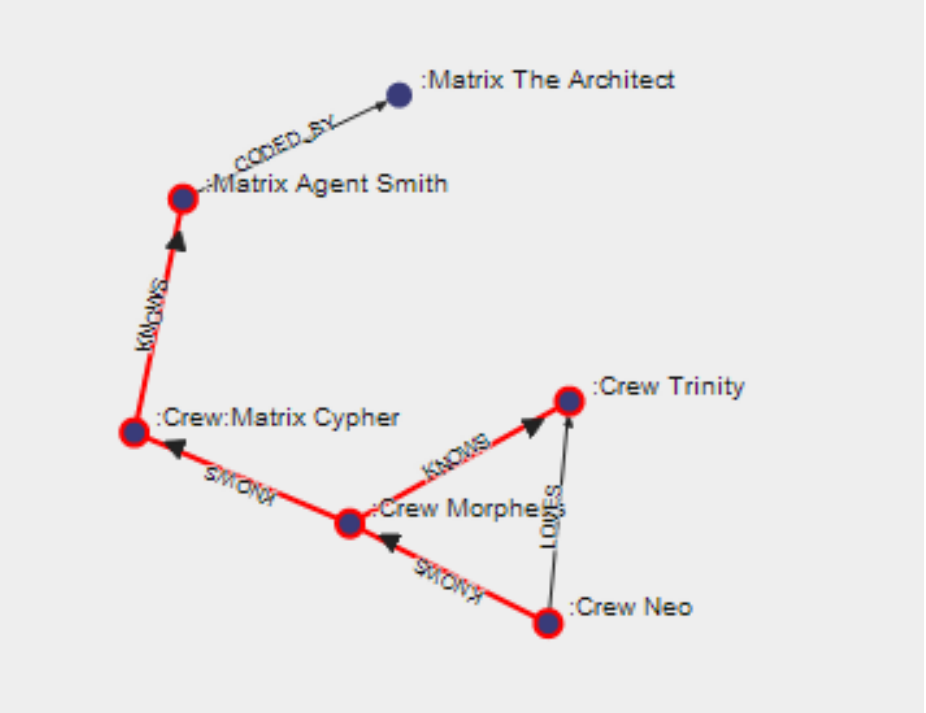
```
CREATE (Neo:Crew { name:'Neo' })
```

```
(Neo)-[:KNOWS]->(Morpheus)
```



# Cypher

```
Query:
MATCH (n:Crew)-[r:KNOWS*]-m
WHERE n.name='Neo'
RETURN n AS Neo,r,m
```




| Neo | r | m |
|---|---|---|
| {name:"Neo"} | [(0)-[0:KNOWS]->(1)] | (1:Crew {name:"Morpheus"}) |
| {name:"Neo"} | [(0)-[0:KNOWS]->(1), (1)-[2:KNOWS]->(2)] | (2:Crew {name:"Trinity"}) |
| {name:"Neo"} | [(0)-[0:KNOWS]->(1), (1)-[3:KNOWS]->(3)] | (3:Crew:Matrix {name:"Cypher"}) |
| {name:"Neo"} | [(0)-[0:KNOWS]->(1), (1)-[3:KNOWS]->(3), (3)-[4:KNOWS]->(4)] | (4:Matrix {name:"Agent Smith"}) |

# Application Domains


(www.neo4j.org)

# Key questions to ask yourself

- Is my data going to have a lot of relationships?
- What sort of questions would I like to ask my database?

# Social Network “path exists” Performance

- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4
  - Caches warm to eliminate disk IO

| | # persons | query time |
|---|---|---|
| Relational database | 1000 | 2000ms |

# Social Network “path exists” Performance

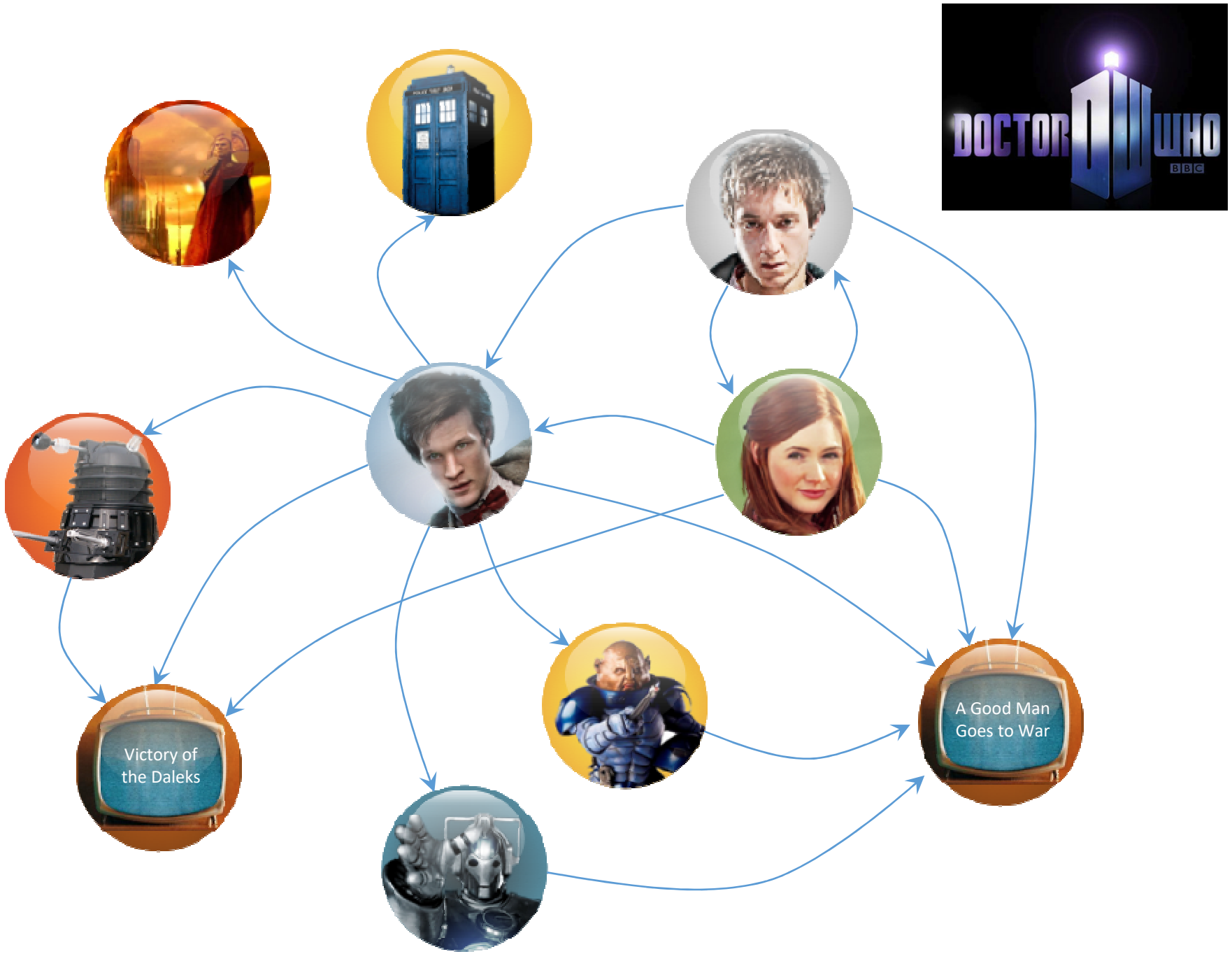
- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4
  - Caches warm to eliminate disk IO

| | # persons | query time |
|---|---|---|
| Relational database | 1000 | 2000ms |
| Neo4j | 1000 | 2ms |

# Social Network “path exists” Performance

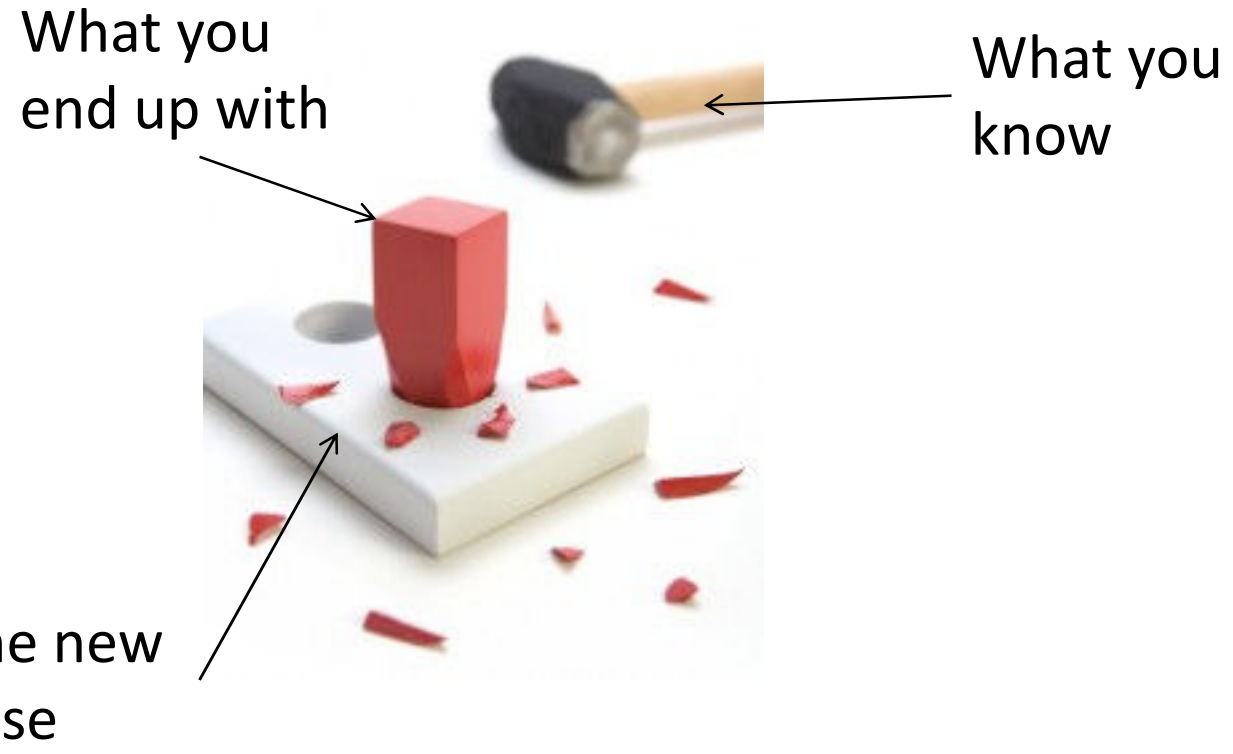
- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4
  - Caches warm to eliminate disk IO

| | # persons | query time |
|---|---|---|
| Relational database | 1000 | 2000ms |
| Neo4j | 1000 | 2ms |
| Neo4j | 1000000 | 2ms |

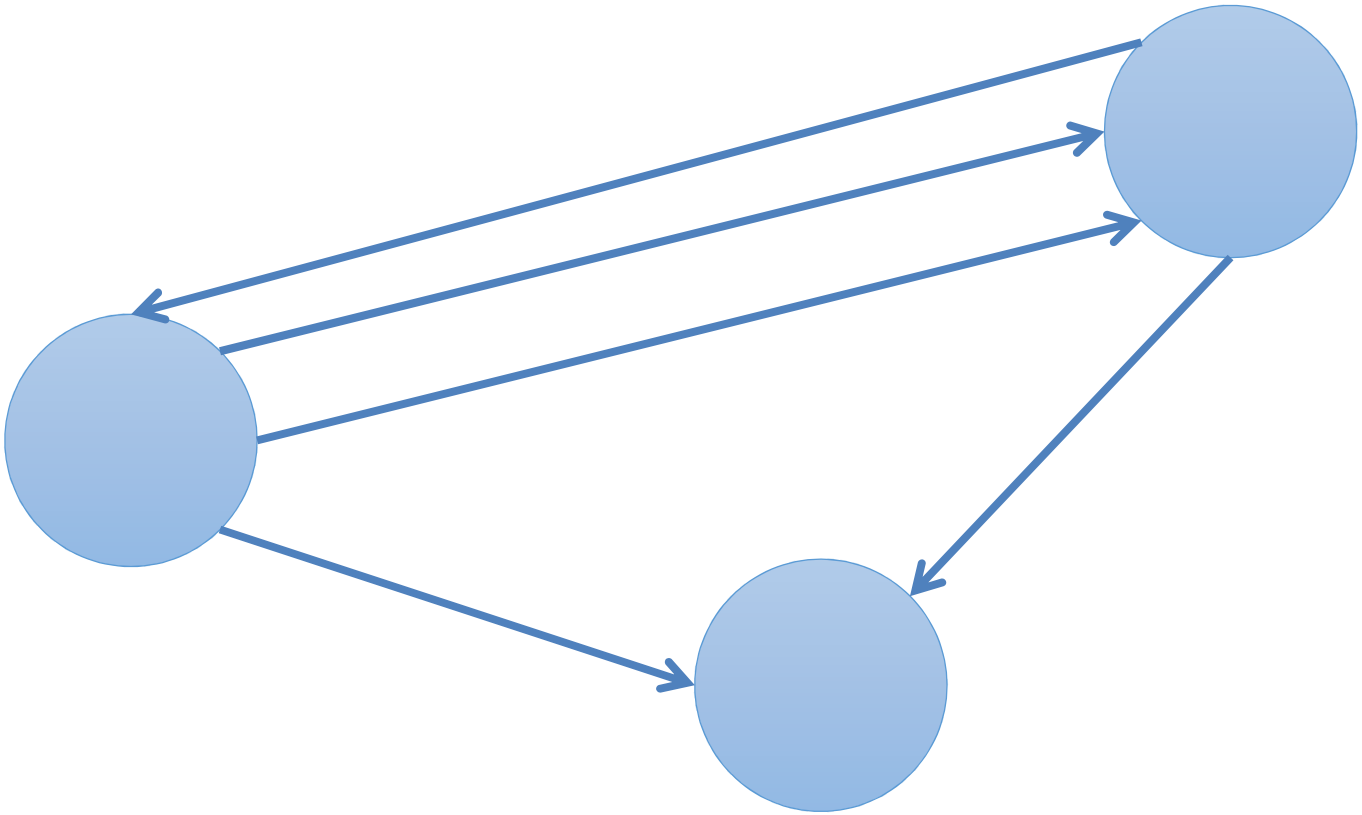

DOCTOR WHO
BBC
Victory of the Daleks
A Good Man Goes to War

Common design misunderstanding is to impose a relational model onto the graph DB – effectively taking a RDBMS dump and importing it.

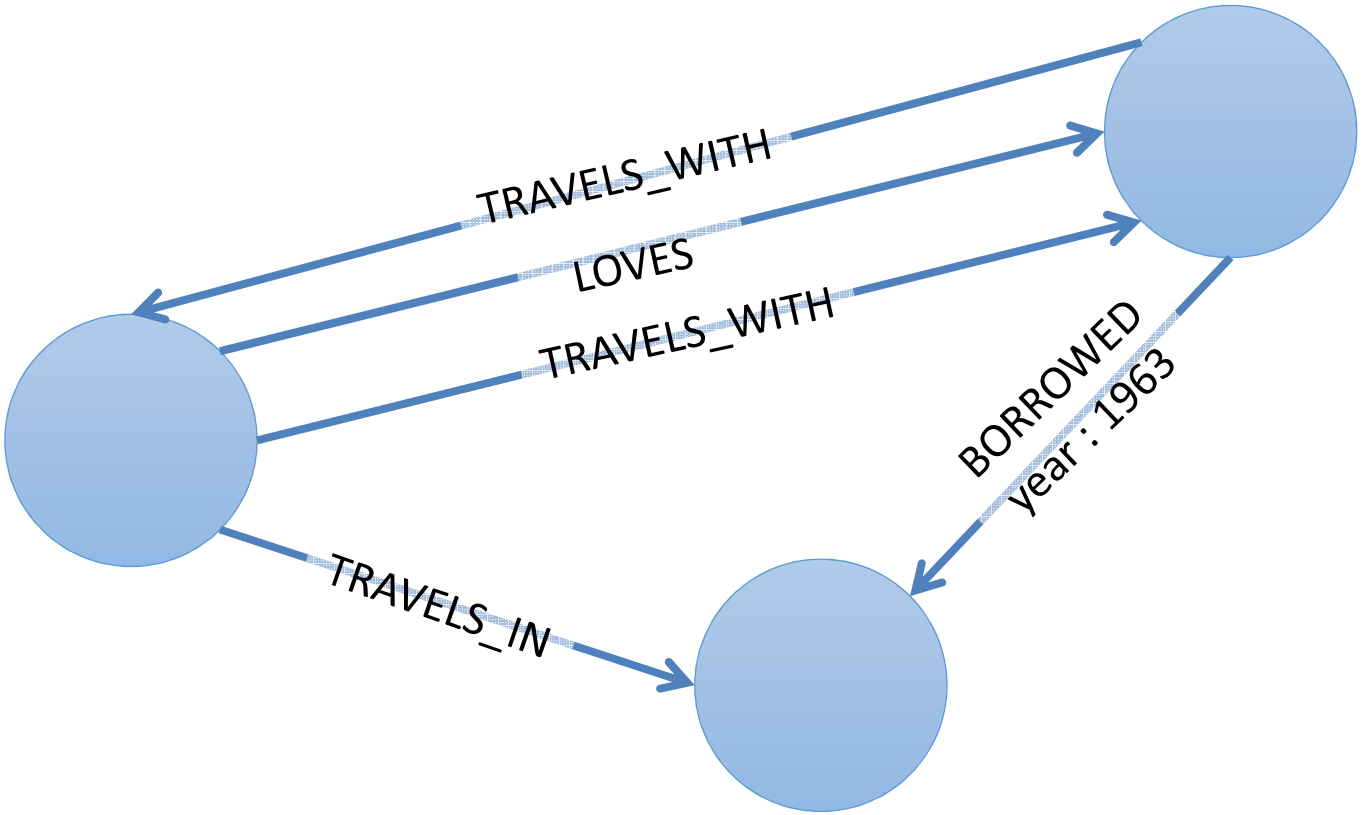
- Won't be good



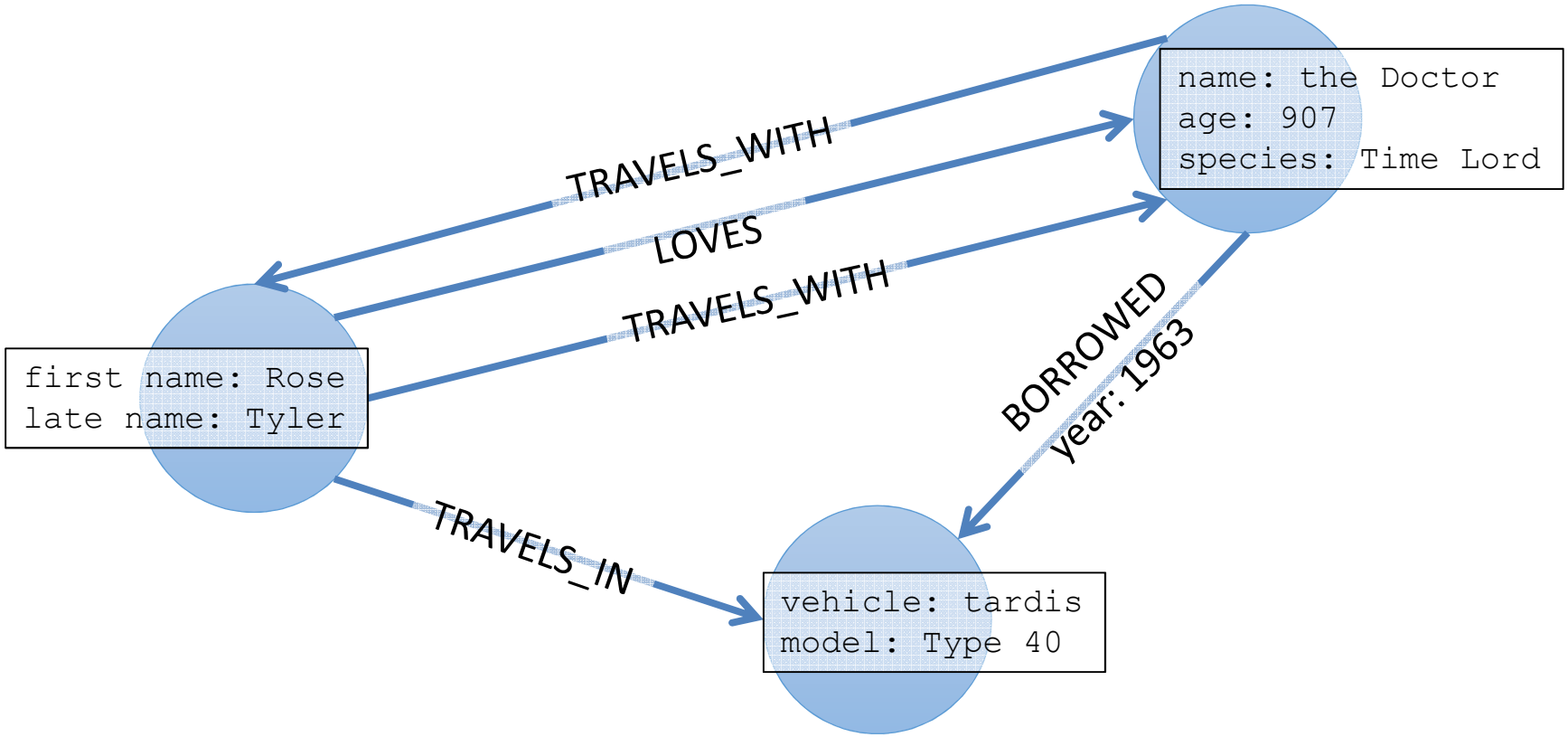
# Property Graph Model



# Property Graph Model

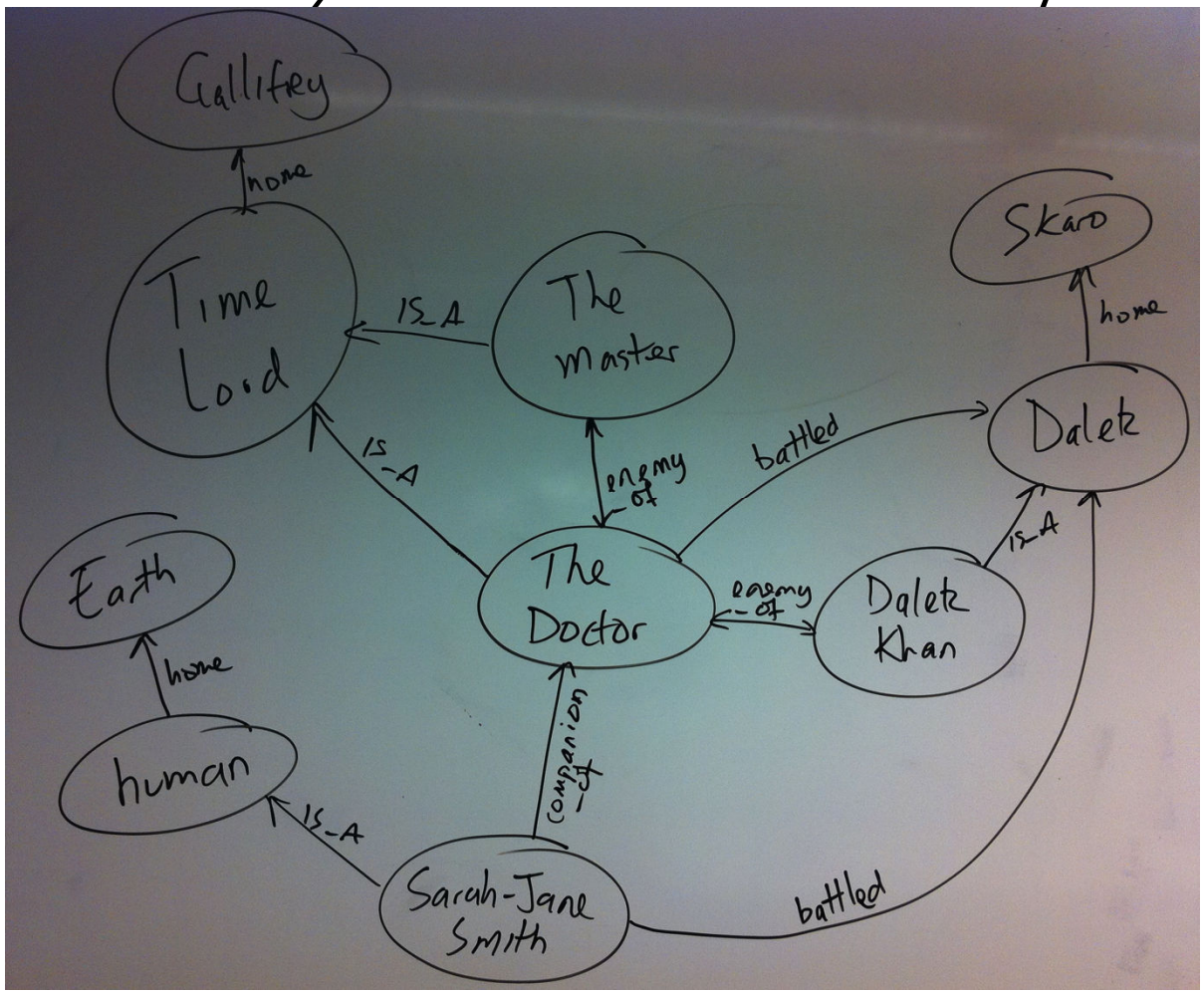


# Property Graph Model



Start off sketching the domain. That's your model done – we see this when we revisit databases months after they're been designed and put into production
No decomposition, ER design, normalisation/denormalisation as you need with RDBMS.

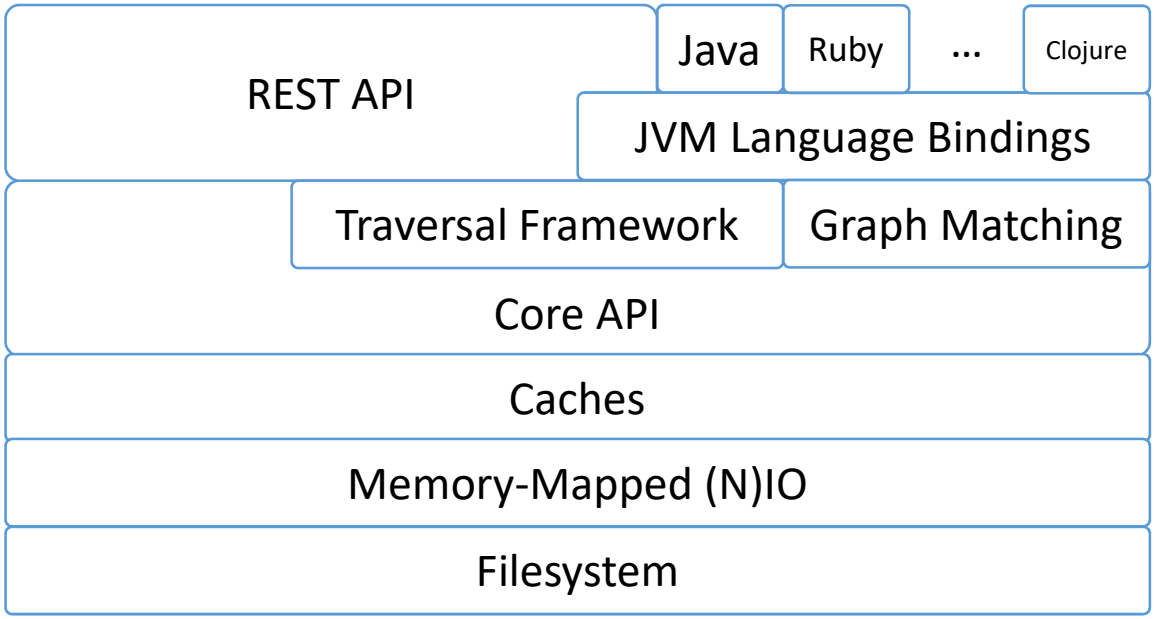
# Graphs are *very* whiteboard-friendly



# Schema-less Databases

- Graph databases don’t excuse you from design
  - Any more than dynamically typed languages excuse you from design
- Good design still requires effort
- But less difficult than RDBMS because you don’t need to normalise
  - And then de-normalise!

# Neo4j Logical Architecture



# Data access is *programmatic*

- Through the Java APIs
  - JVM languages have bindings to the same APIs
    - JRuby, Jython, Clojure, Scala…
- Managing nodes and relationships
- Indexing
- Traversing
- Path finding
- Pattern matching

# Core API

- Deals with graphs in terms of their fundamentals:
  - Nodes
    - Properties
      - KV Pairs
  - Relationships
    - Start node
    - End node
    - Properties
      - KV Pairs

# Creating Nodes

```
GraphDatabaseService db = new
    EmbeddedGraphDatabase("/tmp/neo");
Transaction tx = db.beginTx();
try {
  Node theDoctor = db.createNode();
  theDoctor.setProperty("character", "the
Doctor");
  tx.success();
} finally {
  tx.finish();
}
```

# Creating Relationships

```
Transaction tx = db.beginTx();
try {
  Node theDoctor = db.createNode();
  theDoctor.setProperty("character", "The Doctor");

  Node susan = db.createNode();
  susan.setProperty("firstname", "Susan");
  susan.setProperty("lastname", "Campbell");

  susan.createRelationshipTo(theDoctor,
DynamicRelationshipType.withName("COMPANION_OF"));

  tx.success();
} finally {
  tx.finish();
}
```

# Indexing a Graph?

- Graphs are their own indexes!
- But sometimes we want short-cuts to well-known nodes
- Can do this in our own code
  - Just keep a reference to any interesting nodes
- Indexes offer more flexibility in what constitutes an “interesting node”

# No free lunch

Indexes trade read performance for write cost

Just like any database, even RDBMS

# Don’t index every node!
## (or relationship)

# Lucene

- The default index implementation for Neo4j
  - Default implementation for `IndexManager`
- Supports many indexes per database
- Each index supports nodes *or* relationships
- Supports exact and regex-based matching
- Supports scoring
  - Number of hits in the index for a given item
  - Great for recommendations!

# Creating a Node Index

```
GraphDatabaseService db = …
Index<Node> planets = db.index().forNodes("planets");
```

# Creating a Relationship Index

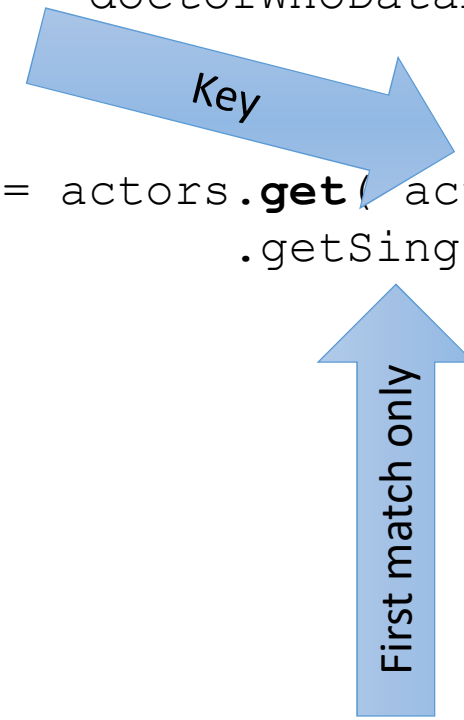
```
GraphDatabaseService db = …
Index<Relationship> enemies = db.index().forRelationships("enemies");
```

# Exact Matches

```
GraphDatabaseService db = …

Index<Node> actors = doctorWhoDatabase.index()
                                      .forNodes("actors");

Node rogerDelgado = actors.get("actor", "Roger Delgado")
                          .getSingle();
```

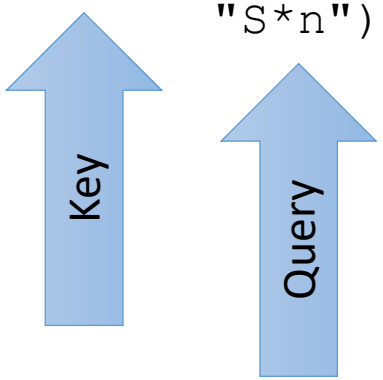


# Query Matches

```
GraphDatabaseService db = …

Index<Node> species = doctorWhoDatabase.index()
                                       .forNodes("species");

IndexHits<Node> speciesHits = species.query("species",
                                            "S*n");
```



# Must use transactions to mutate indexes

- Mutating access is still protected by transactions
  - Which cover both index and graph

```
GraphDatabaseService db = …
Transaction tx = db.beginTx();
  try {
    Node nixon= db.createNode();
    nixon("character", "Richard Nixon");
    db.index().forNodes("characters").add(nixon,
                         "character",
                       nixon.getProperty("character"));
    tx.success();
  } finally {
    tx.finish();
}
```

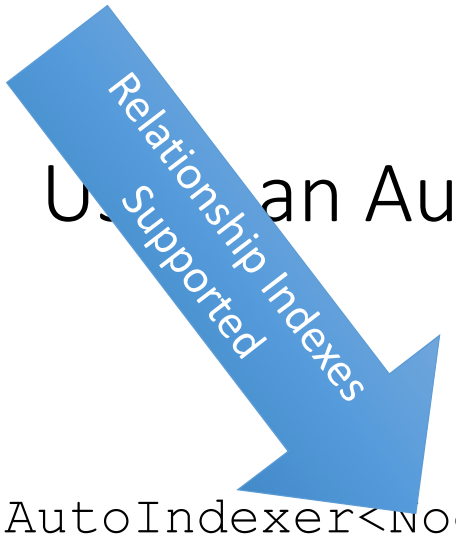
# What’s an auto index?

- It’s an index that stays consistent with the graph data

# Auto index lifecycle

- On creation, specify the property name that will be indexed
- If node/relationship or property is removed from the graph, it will be removed from the index
- Stop indexing a property
  - Any node/rel you touch which has that property will be removed from the auto index
- Re-indexing is a manual activity (1.4)
  - Existing properties won't be indexed unless you touch them

# Using an Auto-Index




```
AutoIndexer<Node> nodeAutoIndex =
  graphDb.index().getNodeAutoIndexer();

nodeAutoIndex.startAutoIndexingProperty
  ("species");

nodeAutoIndex.setEnabled( true );

ReadableIndex<Node> autoNodeIndex =
 graphDb.index().getNodeAutoIndexer()
  .getAutoIndex();
```

# Mixing Core API and Indexes

- Indexes are typically used only to provide starting points
- Then the heavy work is done by traversing the graph
- Can happily mix index operations with graph operations to great effect

# Comparing APIs

**Core**

- Basic (nodes, relationships)
- Fast
- Imperative
- Flexible
  - Can easily intermix mutating operations

**Traversers**

- Expressive
- Fast
- Declarative (mostly)
- Opinionated

# (At least) Two Traverser APIs

- Neo4j has declarative traversal frameworks
  - What, not how
- There are two of these
  - And development is active
  - No “one framework to rule them all” yet

# Simple Traverser API

- Mature
- Designed for the 80% case
- In the **`org.neo4j.graphdb`** package

```
Node daleks = …
Traverser t = daleks.traverse(
  Order.DEPTH_FIRST,
  StopEvaluator.DEPTH_ONE,
  ReturnableEvaluator.ALL_BUT_START_NODE,
  DoctorWhoRelations.ENEMY_OF,
  Direction.OUTGOING);
```

# Custom ReturnableEvaluator

```
Traverser t = theDoctor.traverse(Order.DEPTH_FIRST,
  StopEvaluator.DEPTH_ONE,
  new ReturnableEvaluator() {
    public boolean isReturnableNode(TraversalPosition pos)
    {
      return pos.currentNode().hasProperty("actor");
    }
  },
  DoctorWhoRelations.PLAYED, Direction.INCOMING);
```

# “New” Traverser API

- Newer (obviously!)
- Designed for the 95% use case
- In the **`org.neo4j.graphdb.traversal`** package
- http://wiki.neo4j.org/content/Traversal_Framework

```
Traverser traverser = Traversal.description()
    .relationships(DoctorWhoRelationships§.ENEMY_OF, Direction.OUTGOING)
    .depthFirst()
    .uniqueness(Uniqueness.NODE_GLOBAL)
    .evaluator(new Evaluator() {
      public Evaluation evaluate(Path path) {
        // Only include if we're at depth 2, for enemy-of-enemy
        if(path.length() == 2) {
          return Evaluation.INCLUDE_AND_PRUNE;
        } else if(path.length() > 2){
          return Evaluation.EXCLUDE_AND_PRUNE;
        } else {
          return Evaluation.EXCLUDE_AND_CONTINUE;
        }
      }
    })
    .traverse(theMaster);
```

# What is Cypher?

- Declarative graph pattern matching language
  - “SQL for graphs”
  - Tabular results
- Cypher is evolving steadily
  - Syntax changes between releases
- Supports queries
  - Including aggregation, ordering and limits
  - Mutating operations in product roadmap

# Example Query

- The top 5 most frequently appearing companions:

```
start doctor=node:characters(name = 'Doctor')
match (doctor)<-[:COMPANION_OF]-(companion)
      -[:APPEARED_IN]->(episode)
return companion.name, count(episode)
order by count(episode) desc
limit 5
```

Start node from index

Subgraph pattern

Accumulates rows by episode

Limit returned rows

# Results

| companion.name | count(episode) |
|---|---|
| Rose Tyler | 30 |
| Sarah Jane Smith | 22 |
| Jamie McCrimmon | 21 |
| Amy Pond | 21 |
| Tegan Jovanka | 20 |
| 5 rows, 49 ms | |

# Code

```
ExecutionEngine engine = new ExecutionEngine(database);

String cql = "start doctor=node:characters(name='Doctor')"
                + " match (doctor)<-[:COMPANION_OF]-
(companion)"
           + "-[:APPEARED_IN]->(episode)"
           + " return companion.name, count(episode)"
                + " order by count(episode) desc limit 5";

ExecutionResult result = engine.execute(cql);
```

# Code

```
ExecutionEngine engine = new ExecutionEngine(database);

String cql = "start doctor=node:characters(name='Doctor')"
                + " match (doctor)<-[:COMPANION_OF]-
(companion)"
           + "-[:APPEARED_IN]->(episode)"
           + " return companion.name, count(episode)"
                + " order by count(episode) desc limit 5";

ExecutionResult result = engine.execute(cql);
```

**Top tip:**

`ExecutionResult.dumpToString()`

is your best friend

# Where and Aggregation

- Aggregation:

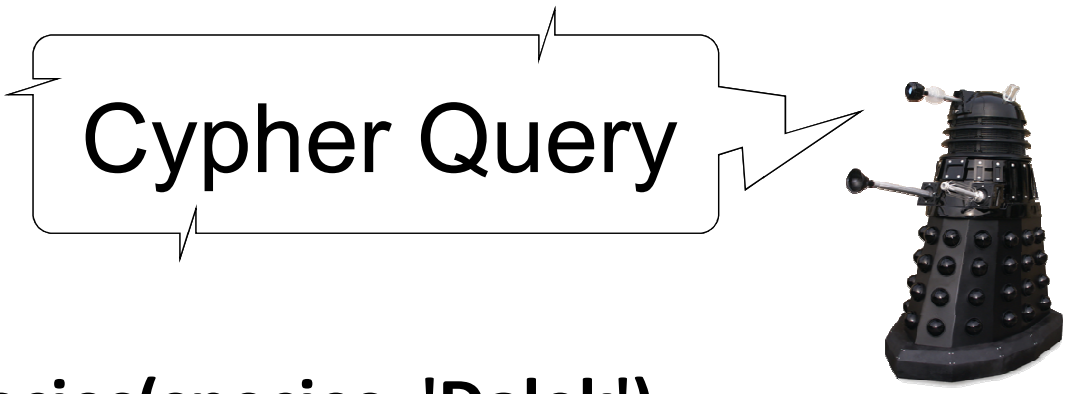
```
COUNT, SUM, AVG, MAX, MIN, COLLECT
```

- Where clauses:

```
start doctor=node:characters(name = 'Doctor')
match (doctor)<-[:PLAYED]-(actor)-[:APPEARED_IN]->(episode)
where actor.actor = 'Tom Baker'
      and episode.title =~ /.*Dalek.*/
return episode.title
```

- Ordering:

```
order by <property>
order by <property> desc
```




```
start daleks=node:species(species='Dalek')
match daleks-[:APPEARED_IN]->episode<-[:USED_IN]-
        ()<-[:MEMBER_OF]-()-[:COMPOSED_OF]->
        part-[:ORIGINAL_PROP]->originalprop
return originalprop.name, part.type, count(episode)
order by count(episode) desc
limit 1
```

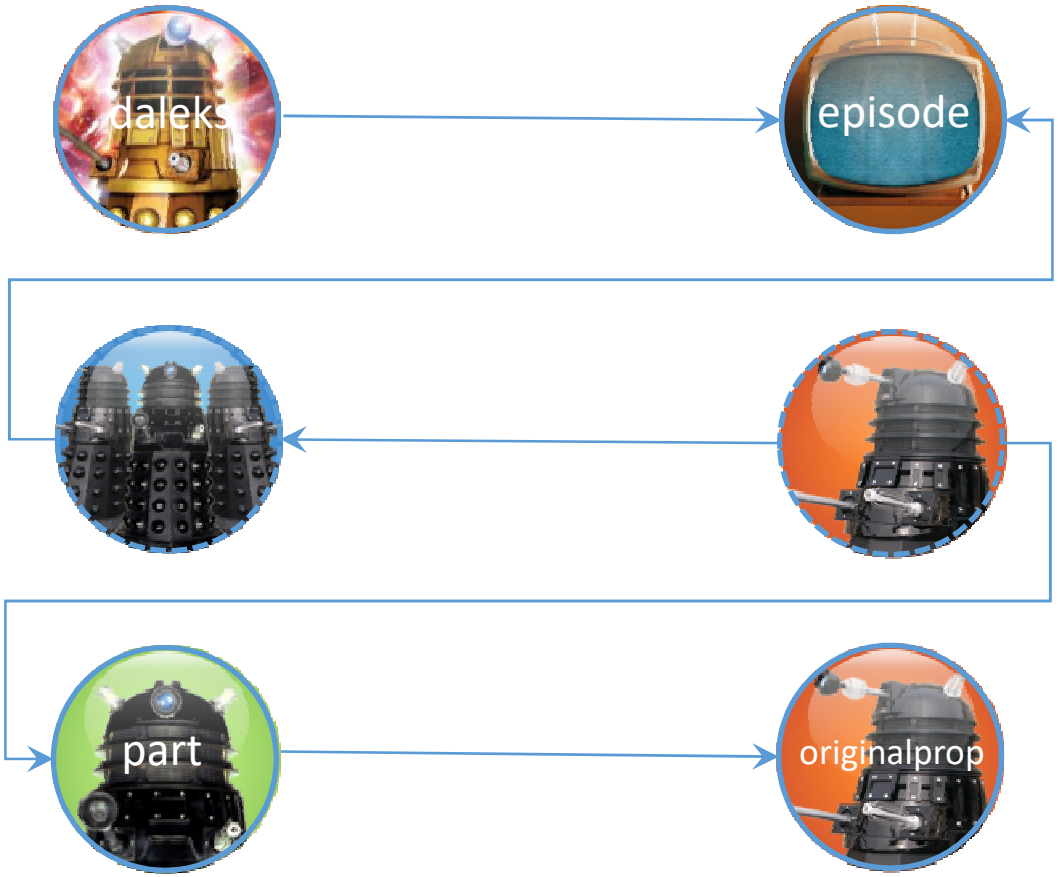
Index Lookup

```
start daleks=node:species(species='Dalek')
match daleks-[:APPEARED_IN]->episode<-[:USED_IN]-
        ()<-[:MEMBER_OF]-()-[:COMPOSED_OF]->
        part-[:ORIGINAL_PROP]->originalprop
return originalprop.name, part.type, count(episode)
order by count(episode) desc
limit 1
```

# Match Nodes & Relationships

```
start daleks=node:species(species='Dalek')
match daleks-[:APPEARED_IN]->episode<-[:USED_IN]-
      ()<-[:MEMBER_OF]-()-[:COMPOSED_OF]->
      part-[:ORIGINAL_PROP]->originalprop
return originalprop.name, part.type, count(episode)
order by count(episode) desc
limit 1
```

**match daleks-[:APPEARED_IN]->episode<-[:USED_IN]-()<-[:MEMBER_OF]-()-[:COMPOSED_OF]->part-[:ORIGINAL_PROP]->originalprop**



# Return Values

```
start daleks=node:species(species='Dalek')
match daleks-[:APPEARED_IN]->episode<-[:USED_IN]-
        ()<-[:MEMBER_OF]-()-[:COMPOSED_OF]->
        part-[:ORIGINAL_PROP]->originalprop
return originalprop.name, part.type, count(episode)
order by count(episode) desc
limit 1
```

# Graph Algos

- Payback time for Algorithms 101
- Graph algos are a higher level of abstraction
  - You do less work!
- The database comes with a set of useful algorithms built-in
  - Time to get some payback from Algorithms 101

# The Doctor versus The Master

- The Doctor and the Master been around for a while
- But what's the key feature of their relationship?
  - They're both timelords, they both come from Gallifrey, they've fought

# Try the Shortest Path!

What's the most direct path between the Doctor and the Master?

```
Node theMaster = …
Node theDoctor = …

int maxDepth = 5;
PathFinder<Path> shortestPathFinder =
                          GraphAlgoFactory.shortestPath(
                            Traversal.expanderForAllTypes(),
                            maxDepth);

Path shortestPath =
    shortestPathFinder.findSinglePath(theDoctor, theMaster);
```

# Path finder code


```
Node rose = ...
Node daleks = ...

PathFinder<Path> pathFinder = GraphAlgoFactory.pathsWithLength(
    Traversal.expanderForTypes(
    DoctorWhoRelationships.KNOWS,
    Direction.BOTH), 2);

Iterable<Path> paths = pathFinder.findAllPaths(rose, daleks);
```

constraints

fixed path length

# Why graph matching?

- It’s super-powerful for looking for patterns in a data set
  - E.g. retail analytics
- Higher-level abstraction than raw traversers
  - You do less work!

# Setting up and matching a pattern

```
final PatternNode theDoctor = new PatternNode();
theDoctor.setAssociation(universe.theDoctor());

final PatternNode anEpisode = new PatternNode();
anEpisode.addPropertyConstraint("title", CommonValueMatchers.has());
anEpisode.addPropertyConstraint("episode", CommonValueMatchers.has());

final PatternNode aDoctorActor = new PatternNode();
aDoctorActor.createRelationshipTo(theDoctor, DoctorWhoUniverse.PLAYED);
aDoctorActor.createRelationshipTo(anEpisode, DoctorWhoUniverse.APPEARED_IN);
aDoctorActor.addPropertyConstraint("actor", CommonValueMatchers.has());

final PatternNode theCybermen = new PatternNode();
theCybermen.setAssociation(universe.speciesIndex.get("species", "Cyberman").getSingle());
theCybermen.createRelationshipTo(anEpisode, DoctorWhoUniverse.APPEARED_IN);
theCybermen.createRelationshipTo(theDoctor, DoctorWhoUniverse.ENEMY_OF);

PatternMatcher matcher = PatternMatcher.getMatcher();
final Iterable<PatternMatch> matches = matcher.match(theDoctor, universe.theDoctor());
```

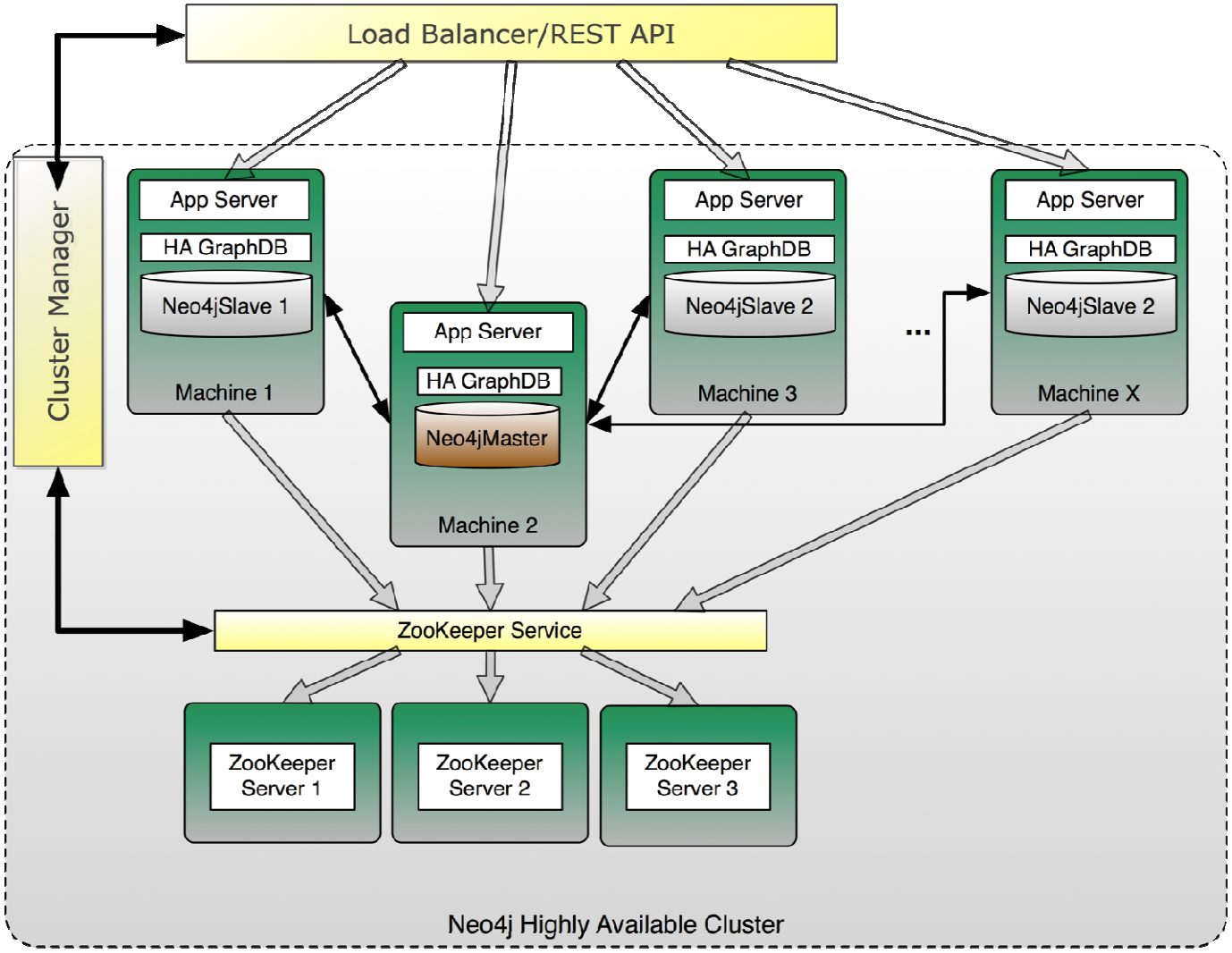
# Two Challenges

- Operational considerations
  - Data should always be available
- Scale
  - Large dataset support

# Neo4j HA

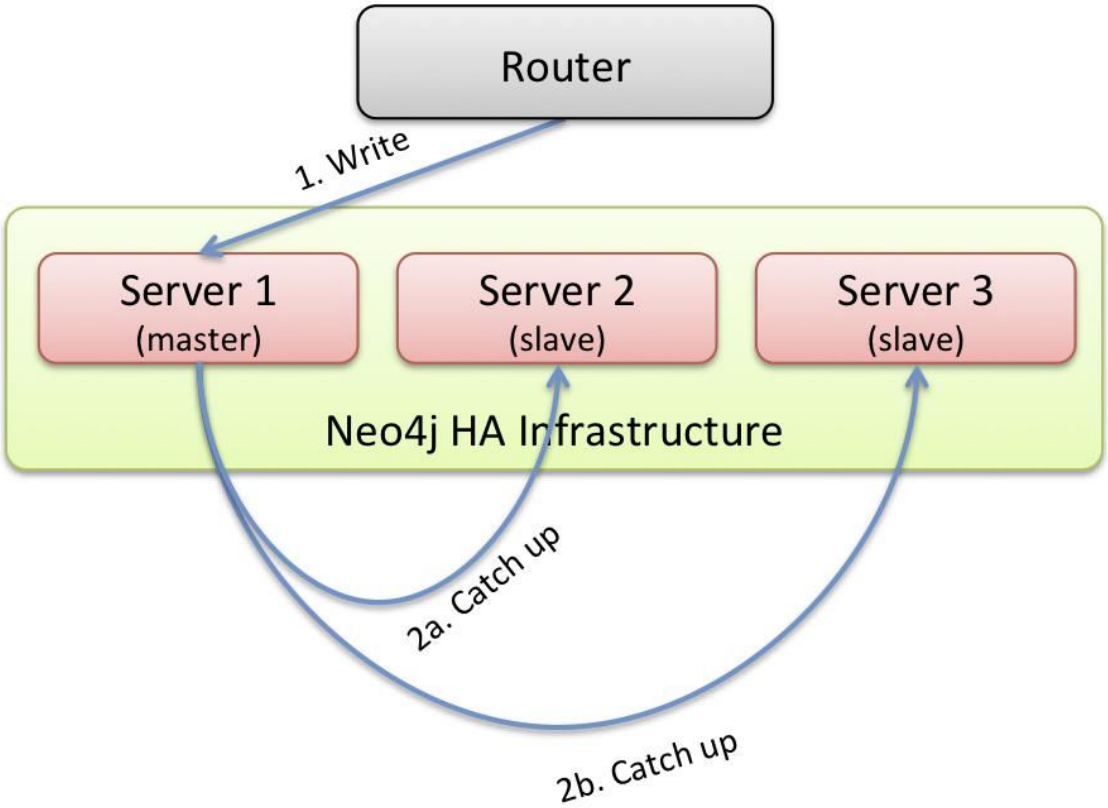
- The HA component supports master-slave replication
  - For clustering
  - For DR across sites

# HA Architecture



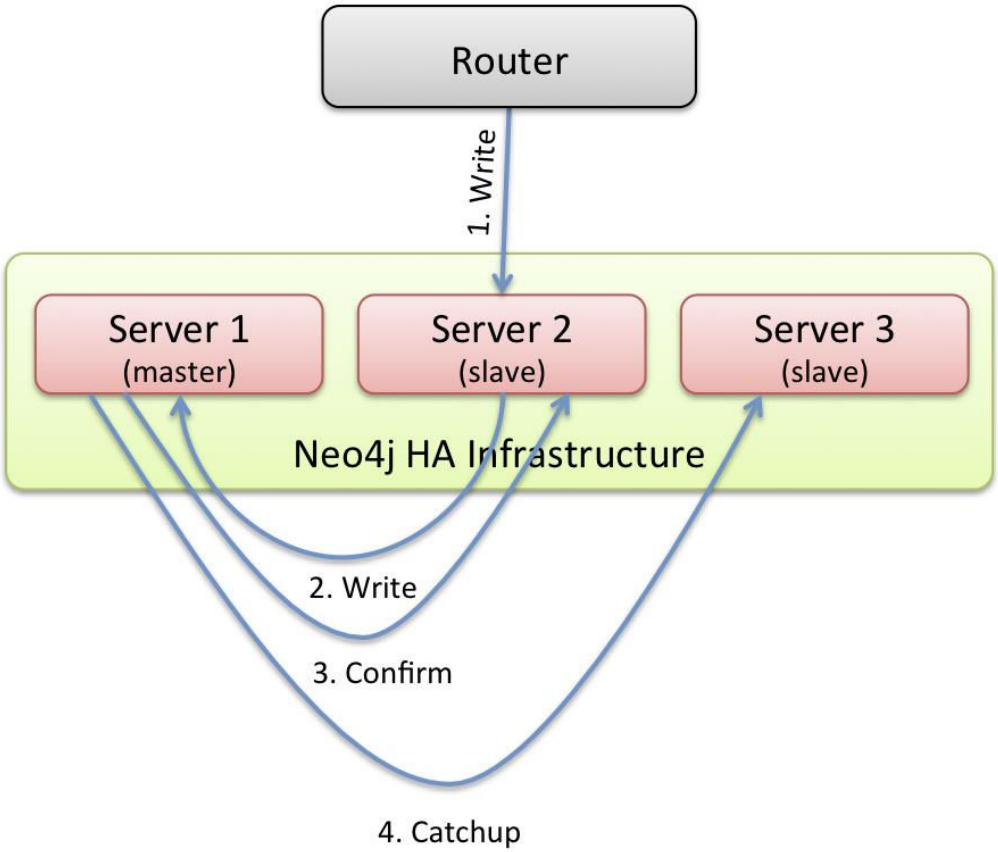
# Write to a Master

- Writes to the master are fast
  - And slaves eventually catch up



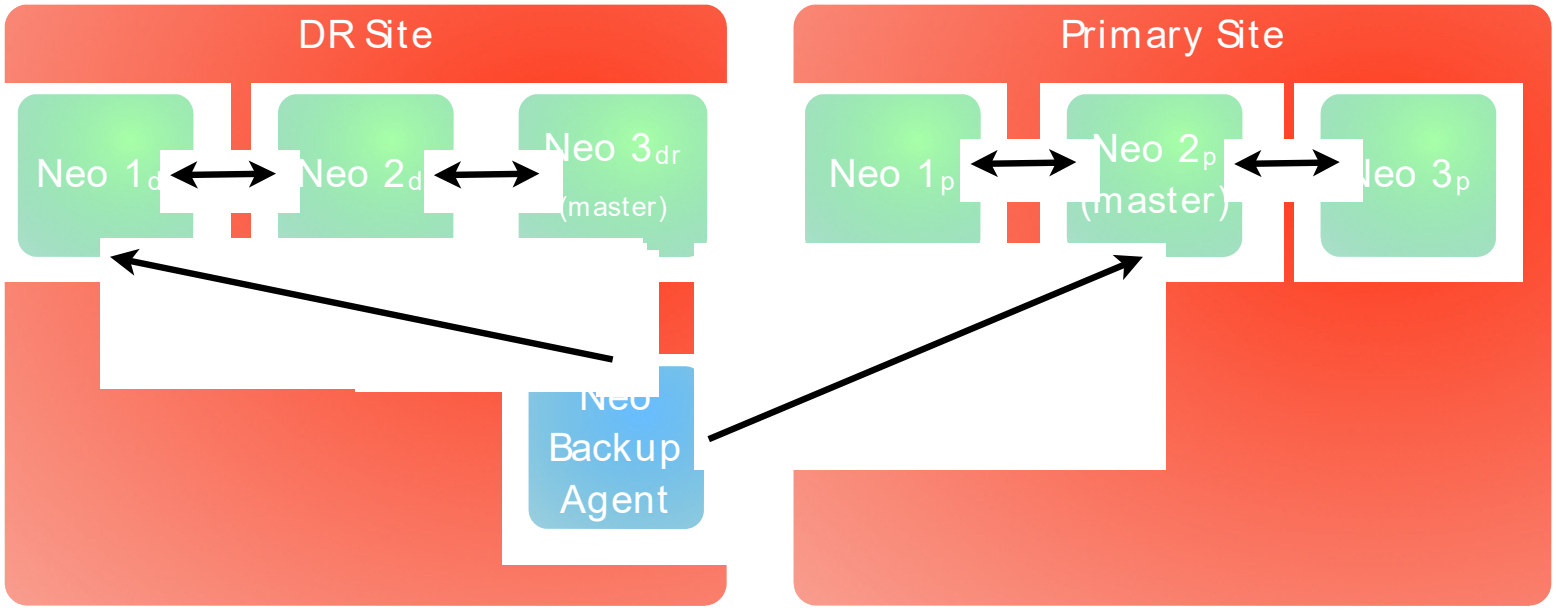
# Write to a Slave

- Writes to a slave cause a synchronous transaction with the master
  - And the other slaves eventually catch up



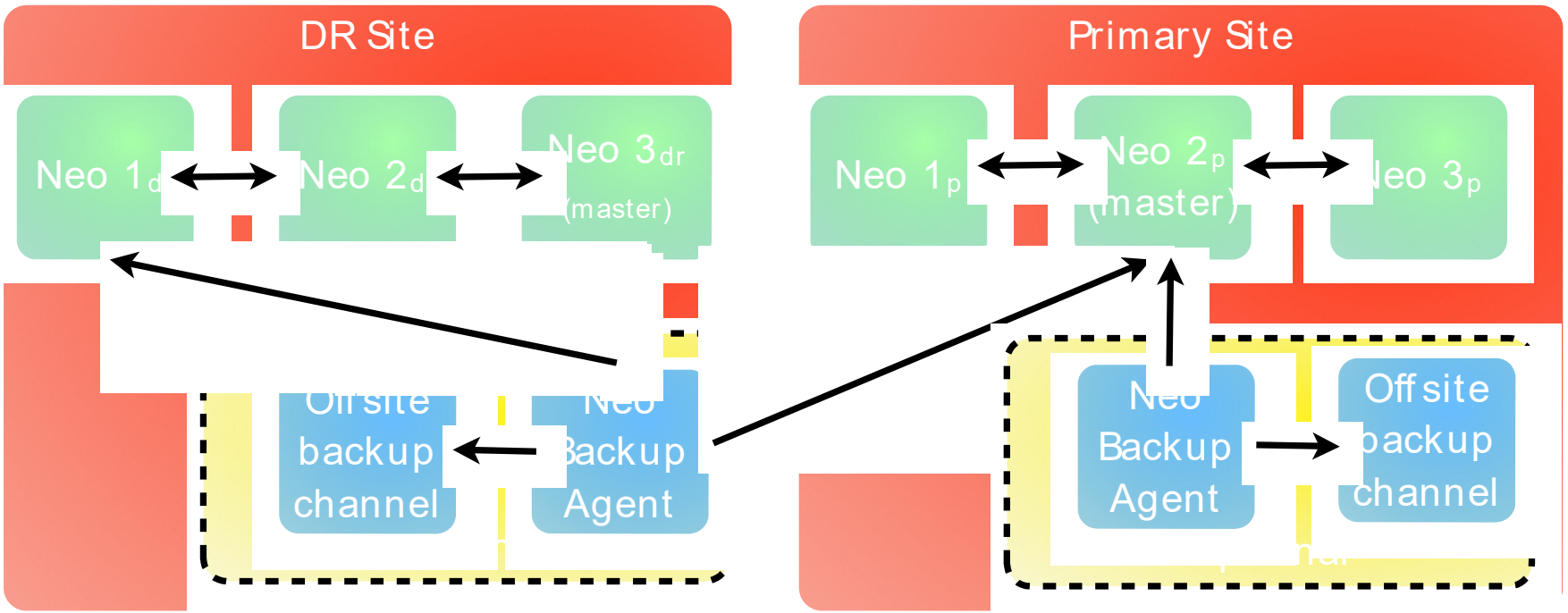
So that’s local availability
sorted,
now we need Disaster Recovery

# HA Locally, Backup Agent for DR



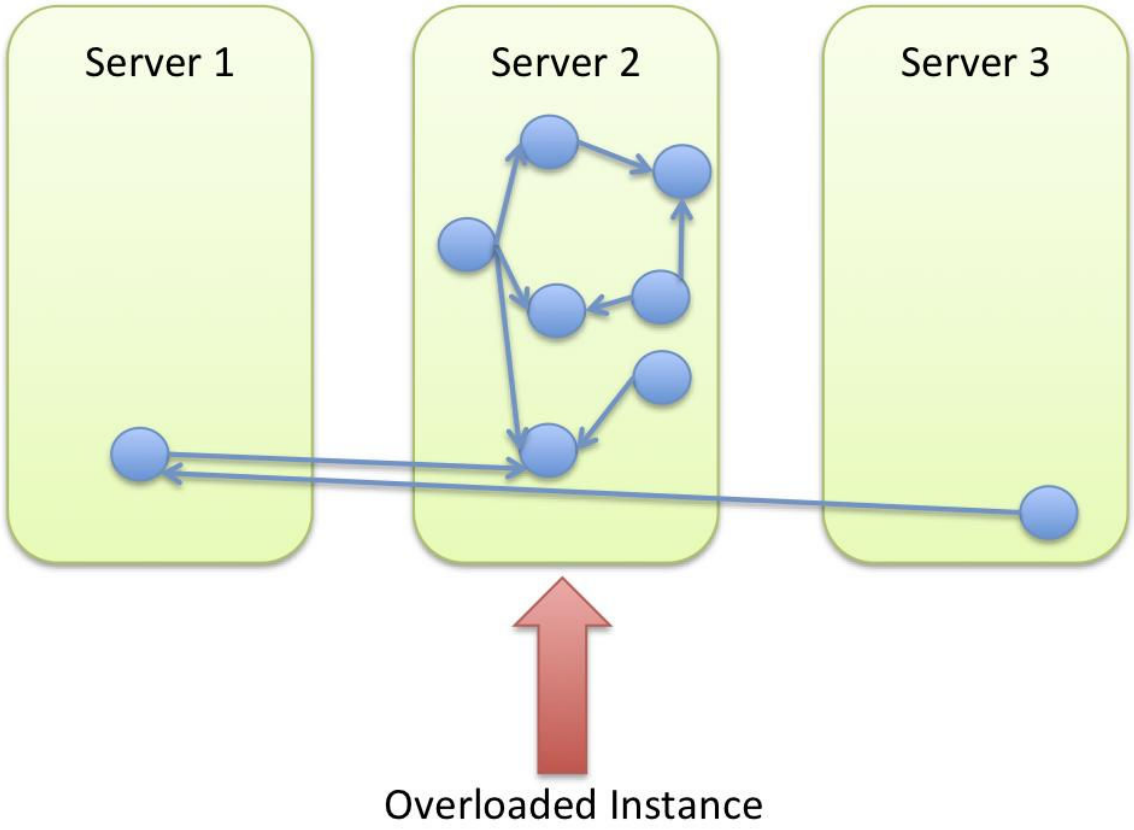
# But what about backup?

# Same Story!

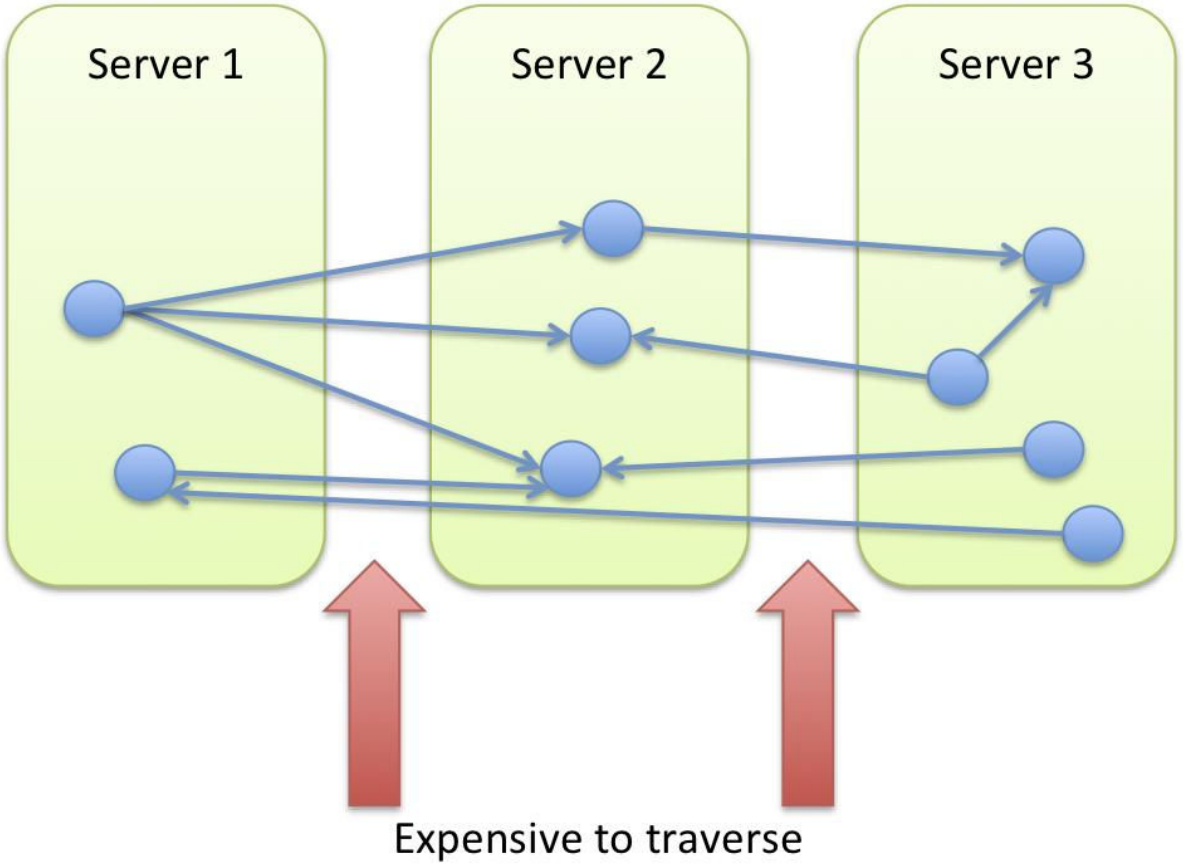

DR Site
Neo 1d
Neo 2d
Neo 3dr
(master)
Offsite
backup
channel
Backup
Agent
Primary Site
Neo 1p
Neo 2p
(master)
Neo 3p
Backup
Agent
Offsite
backup
channel

Scaling graphs is hard

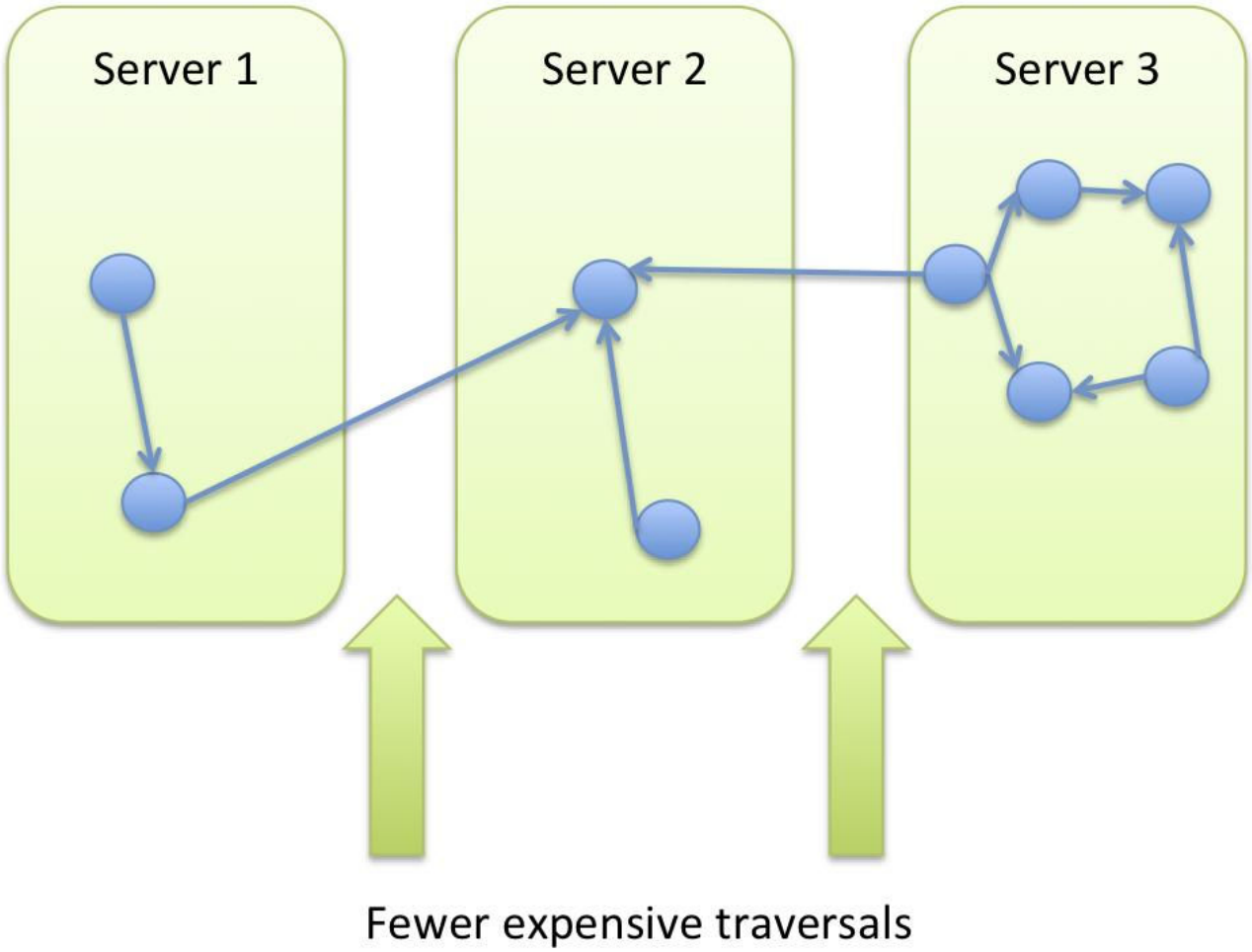
# “Black Hole” server



# Chatty Network



# Minimal Point Cut



# So how do we scale Neo4j?

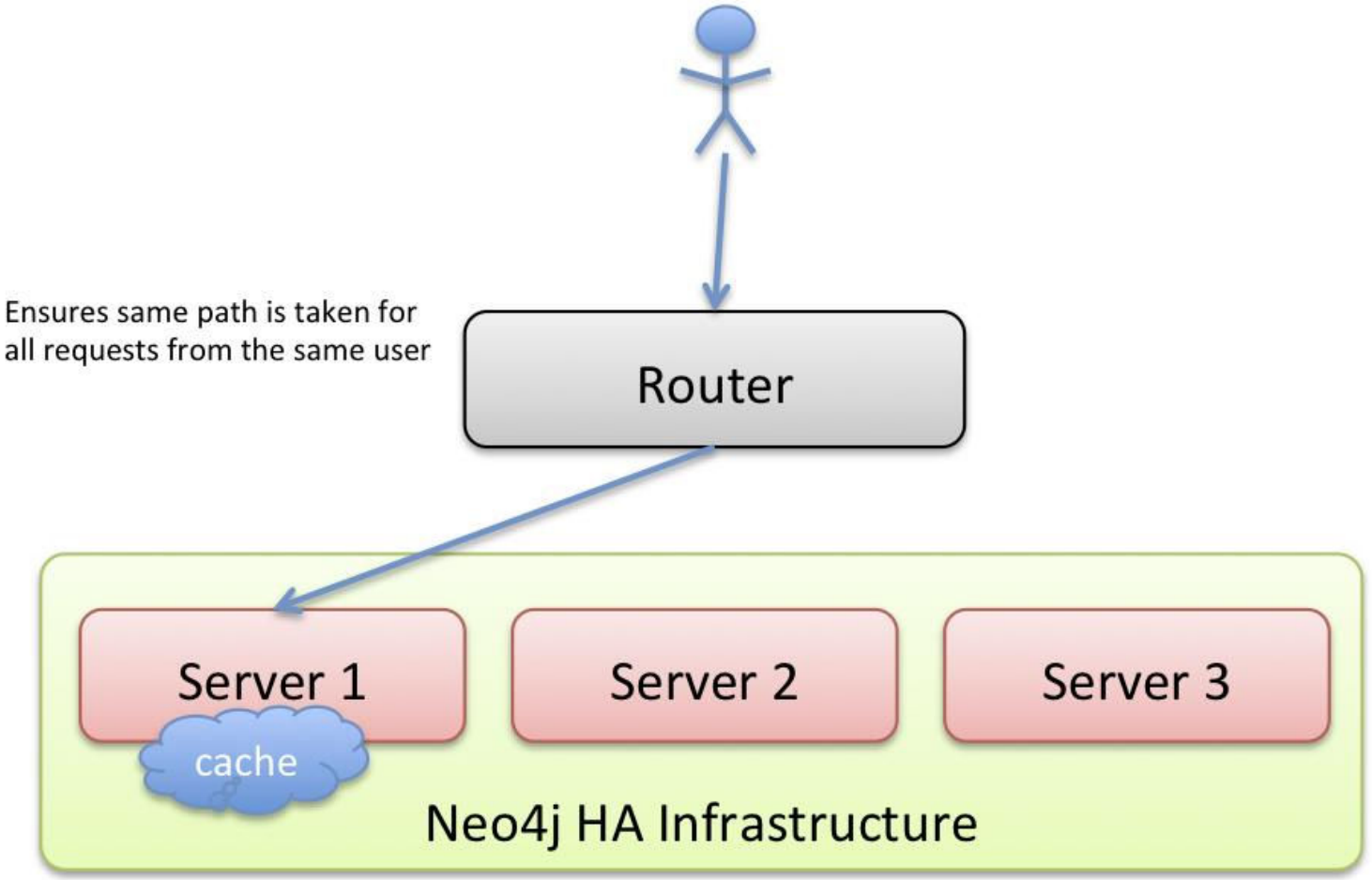
# A Humble Blade

- Blades are powerful!
- A typical blade will contain 128GB memory
  - We can use most of that
- If O(dataset) ≈ O(memory) then we're going to be very fast
  - Remember we can do millions of traversals per second if the caches are warm

# Cache Sharding

- A strategy for coping with large data sets
  - Terabyte scale
- Too big to hold all in RAM on a single server
  - Not too big to worry about replicating it on disk
- Use each blade's main memory to cache part of the dataset
  - Try to keep caches warm
  - Full data is replicated on each rack

# Consistent Routing



# Domain-specific sharding

- Eventually (Petabyte) level data cannot be replicated practically
- Need to shard data across machines
- **Remember: no perfect algorithm exists**

- But we humans sometimes have *domain insight*