NoSQL Databases – Graph Databases & Neo4j

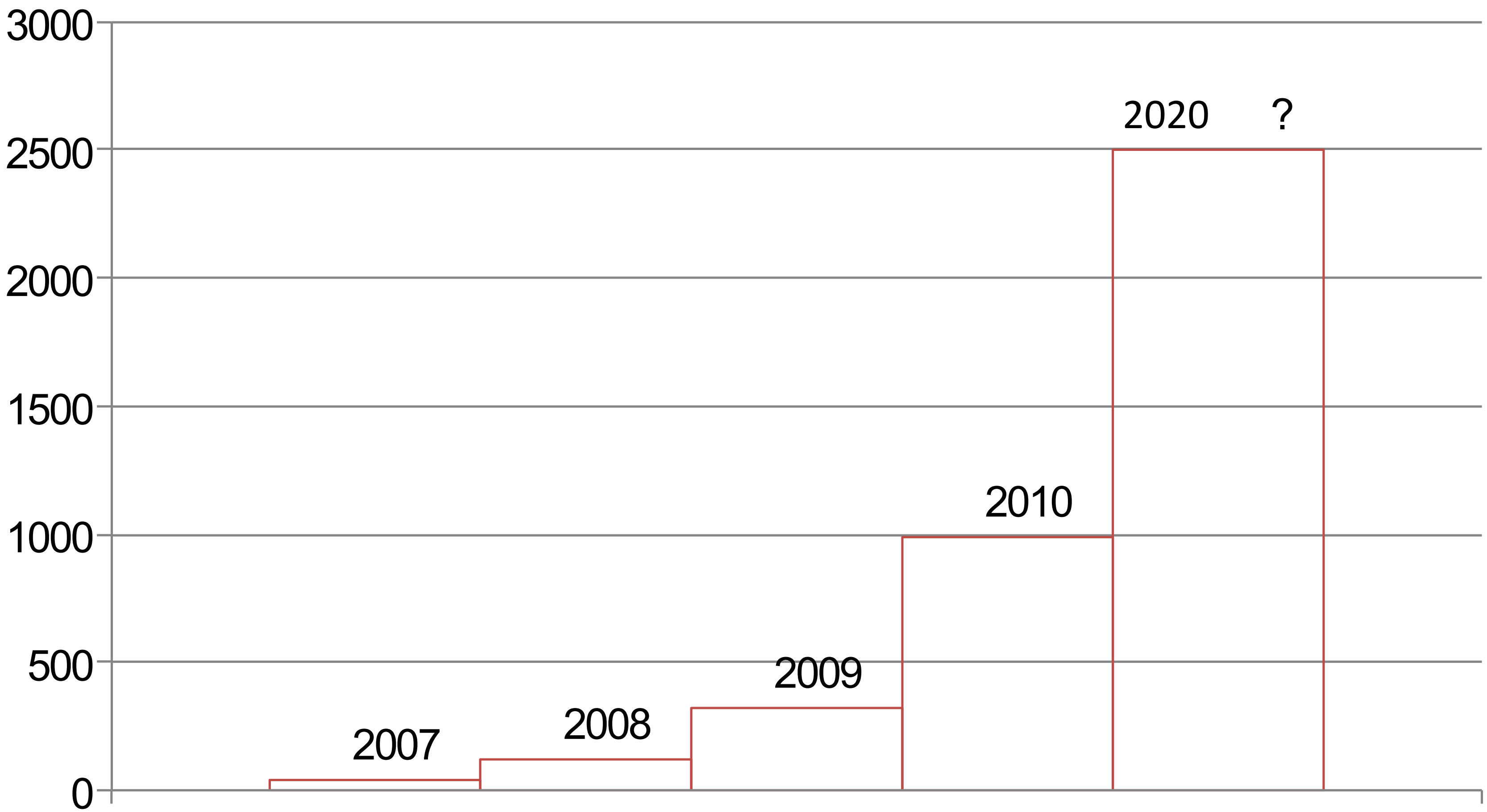
Advanced Databases

Enric Biosca Trias ebiosca@maia.ub.es

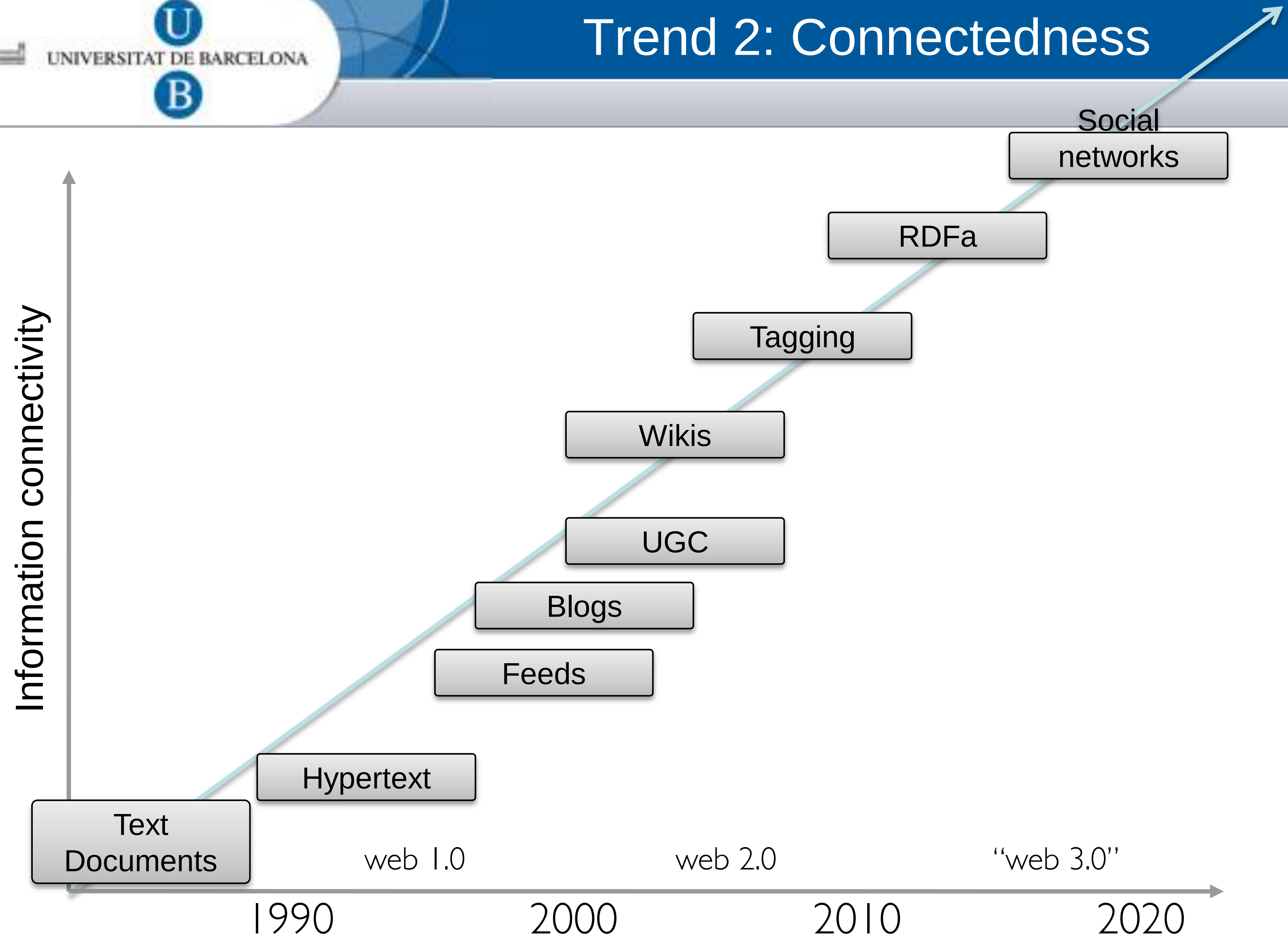
Dept. Matemàtica Aplicada i Anàlisi.

Universitat de Barcelona

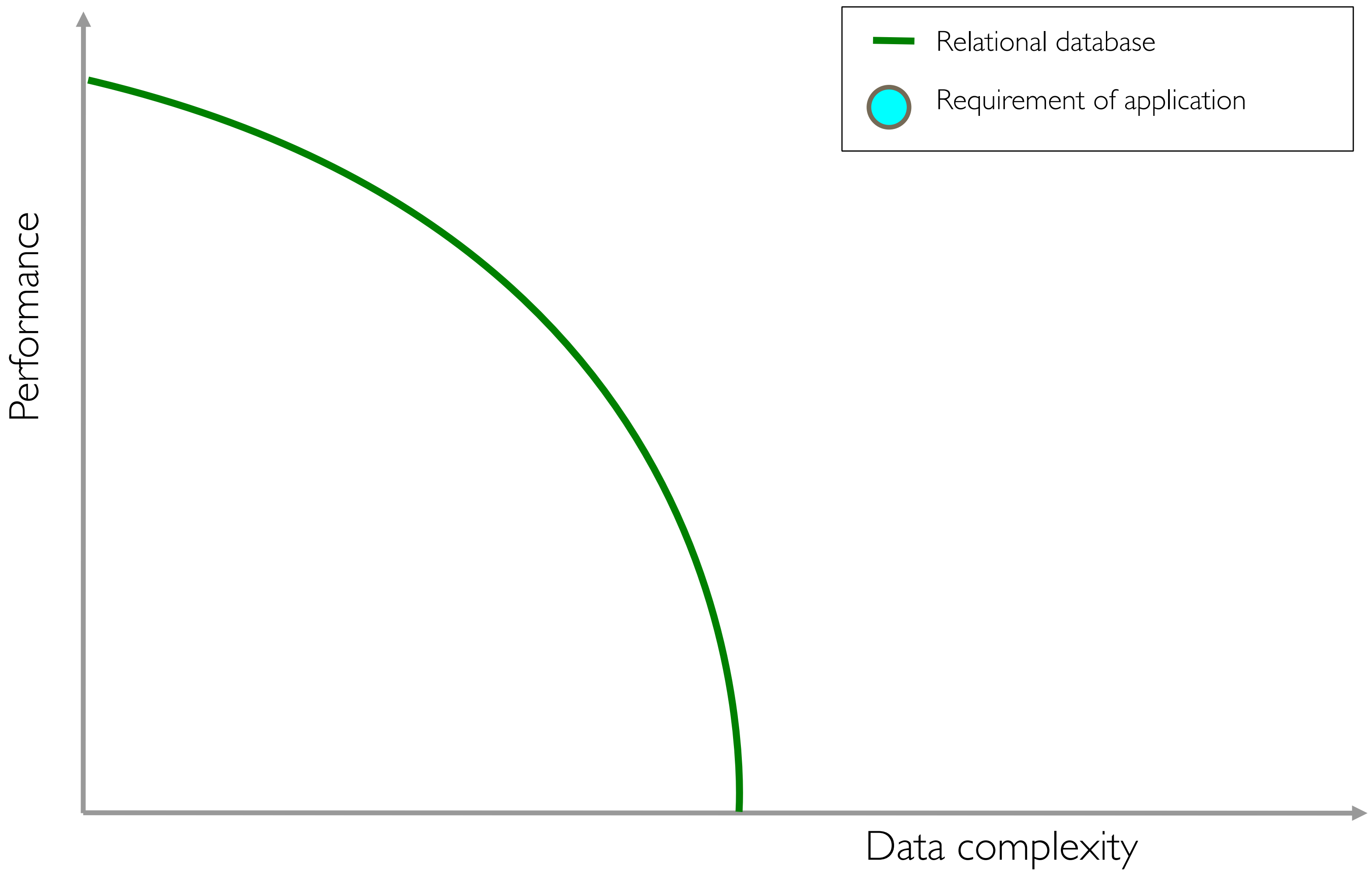
Trend 1: Data Size



Trend 2: Connectedness

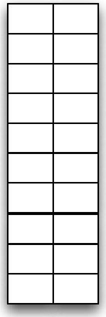


Side note: RDBMS performance

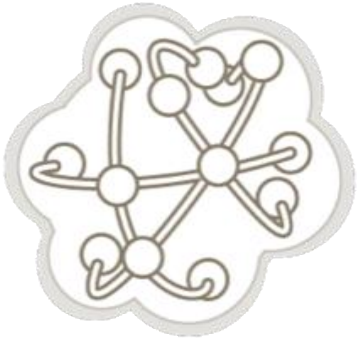


NoSQL Databases

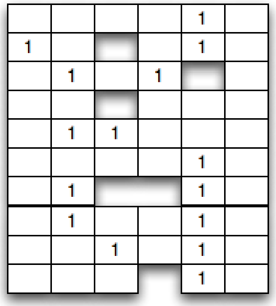
Key-Value



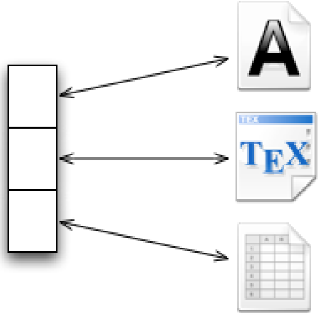
Graph DB



BigTable



Document



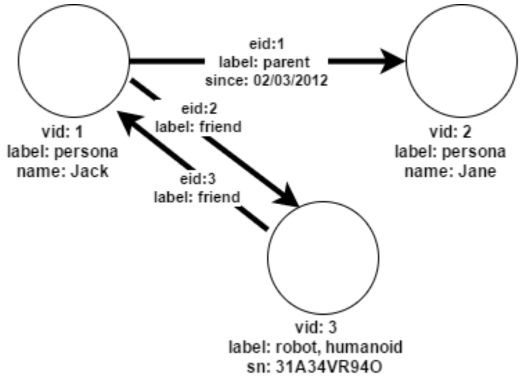
Pros and Cons

- Strengths
  - Powerful data model
  - Fast
    - For connected data, can be many orders of magnitude faster than RDBMS
- Weaknesses:
  - Sharding
    - Though they *can* scale reasonably well
    - And for some domains you can shard too!

Graphs overview

- A graph data structure consists of a finite set of **nodes** (or nodes), together with a set of pairs of these vertices. These pairs are known as **edges**, arcs, or lines for an undirected graph and as arrows, directed edges, directed arcs, or directed lines for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

- **Set of vertices**
  - **Unique id**
  - **Edges**
  - **Labels**
  - **Properties**
- **Set of edges**
  - **Unique id**
  - **2 vertices**
  - **Label**
  - **Collection of properties**



Capabilities

- # As Storage or Database

- Graph databases can be used instead of RDBMS.
- **Focused in relationships between entities** rather than attributes.
- **All data can be stored in a single graph.** No need to split it across multiple tables.
- Using pointers it **avoids doing multiple joins.**

Use cases

SQL-like graph queries

- **Operatinal false positive detection.**
- **speed. NIF-NIF person connections**

Native graph storage

- **Access management system** (metadata)
- **360º view of your data** (all data in a single graph)
- **Non structured data.** Can evolve across time

Capabilities

- Batch analítics engine
- **Computationally intensive tasks.**
- **Complex queries.**
- **Multiple level deep searches.**
- They benefit from parallel and distributed processing.
- **Graph algorithms are more efficient in a graph-native platform.** In RDBMS those algorithms do not always finish.

Use cases
Client clustering (target group marketing) **Ring detection** for fraud analysis **Product recommendation** **Node centrality, Page Rank** to find the most influencer node in a network **Shortest path** for traffic optimization system

Capabilities

- Discovery data

- It allows the data scientist to **work with all the data at one place.** No need to split the data across different tables.
- **Free schema**. No fixed relationships.
- **Graph visualization** tools are discovery layer's **core component.**
- **Intuitive and easy to use** for nonscientists.

Use cases
Discovery of implicit or latent relationships in data • **New business opportunities** • **Custom fraud detection algorithms** • **Exploration by non tech-focused users.** Example: Panama Papers **Network and IT operations** (maintenance, cyber security)

Graph solutions

Database

neo4j

Oracle Spatial and Graph

ORACLE DATABASE 12c

Analytics

APACHE GIRAPH

APACHE Spark™

Oracle Labs PGX

neo4j

Pros	Cons
Very popular Graph pattern-matching language for queries (CYPHER) Graphical developer interface	Non distributed solution No built-in analytics Slow data ingestion

Connectivity:

Oracle Labs PGX

Oracle Labs
PGX

Pros	Cons
Can be distributed Analytics-focused framework Built-in parallel graph algorithms Custom algorithms easily parallelizable SQL-like graph querying language Multiple languages (Java, groovy, scala, R)	Expertise required for low-level access Homogeneous graph RAM limited in single-node option

Connectivity:

Spark GraphX

Pros	Cons
Distributed Use already existing spark clusters Very popular Apache License Built-in algorithms Multiple languages (scala, python, java,R)	Slower(it is a general-purpose framework) Need to find optimized algorithms

Connectivity:

Graph exploration

Discovery tools
- Notebooks (Zeppelin, jupyter)

Ad-hoc applications
- Java

Through APIs

Visualization

Desktop applications
- Already developed, only need to be supplied with data
- Custom functionality limited to plugins

Web libraries
- Compatible with any modern browser
- Must be developed specifically for our needs

Graph exploration

- Discovery t

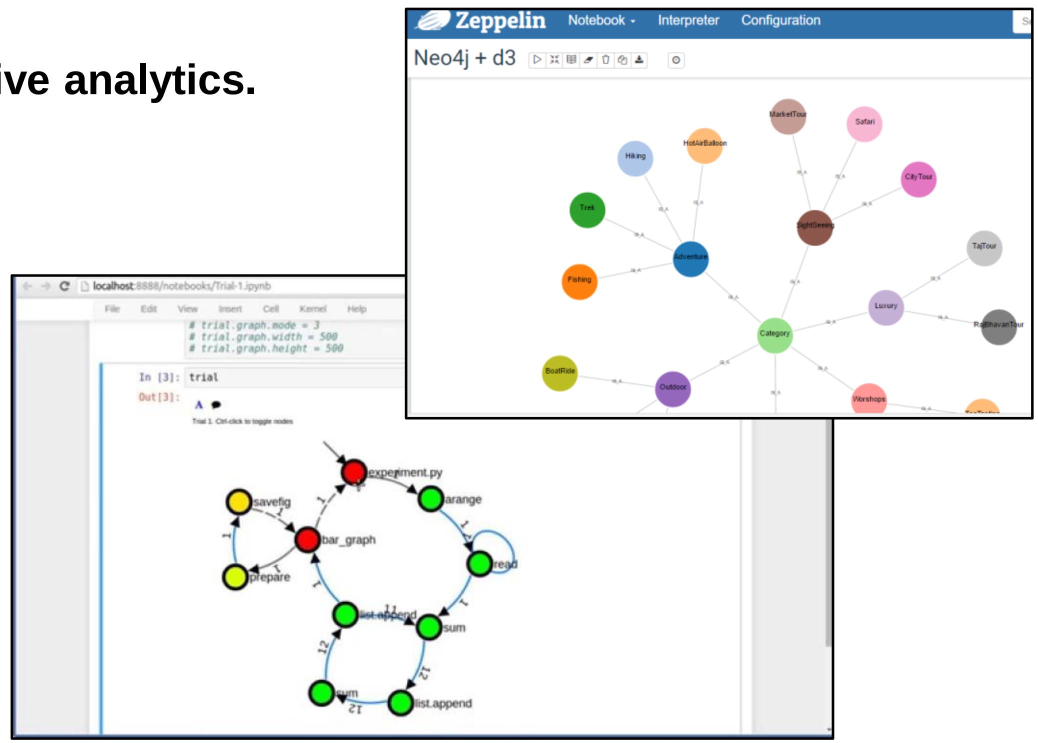
Apache Zeppelin

jupyter

- **Web-based notebooks enables interative analytics.**
- **Open-source**

Common features:

- Pluggable visualization frameworks
- Interactive and collaborative
- Allows any language / data-processing backend.
- Supports many interpreters (Java, Scala, Python…)
- REST API



Graph visualization

- Desktop applications

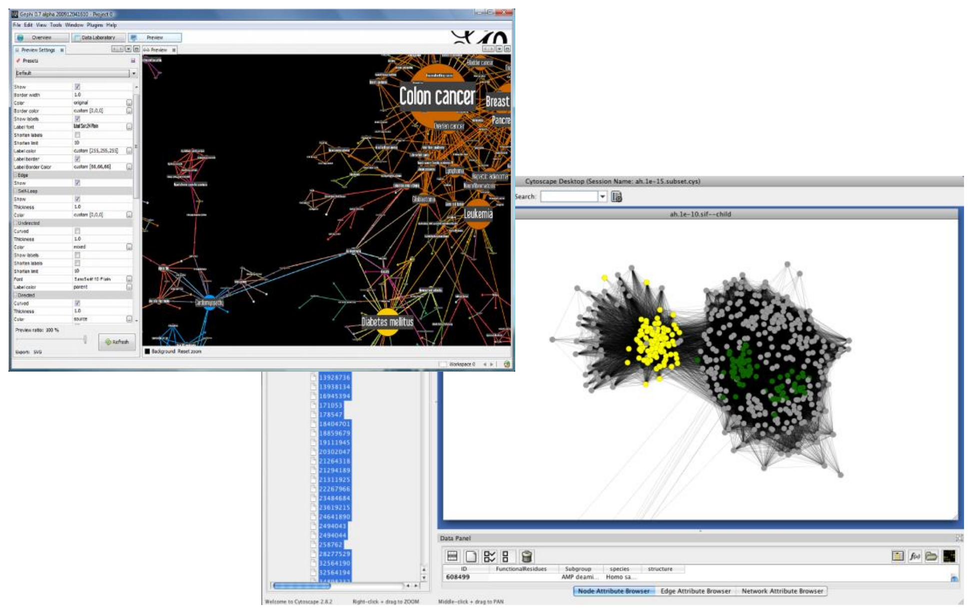
Gephi

Cytoscape

- **Network analysis and visualization software**
- **Open-source**

Common features:

- Customizable visual format
- Built-in layout algorithms
- Statistics and metrics computation
- Node/edge filtering
- Support plugins



• Web libraries

- **There are numerous web libraries capable of graph representation and manipulation.**
- **Customized interaction (such as layout algorithms or specific graph operations) must often be developed.**
- **Most notorious options are D3.js and Linkurious**

KeyLines NETWORK INSIGHT

sigmajs

LINKURIOUS VISUALIZE GRAPH DATA EASILY

D3 Data-Driven Documents

vis.js

plotly.js

arbor.js

Graph visualization

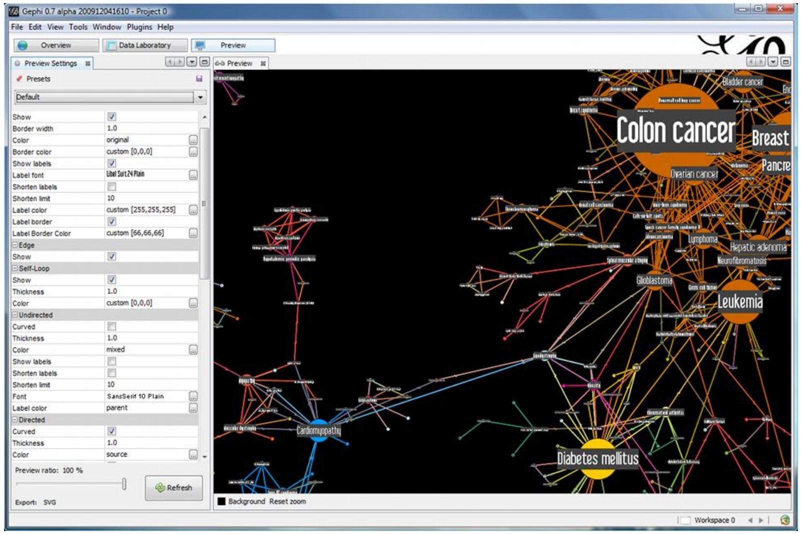
- Desktop applications

Gephi

- **Network analysis and visualization software**
- **Open-source**
- **Developed by *The Gephi Consortium*, a non-profit corporation.**

Main features:

- Real-time visualization (up to 1M nodes)
- Built-in layout algorithms
- Statistics and metrics computation
- Dynamic node/edge filtering
- Supports plugins



UNIVERSITAT DE BARCELONA

Graph visualization

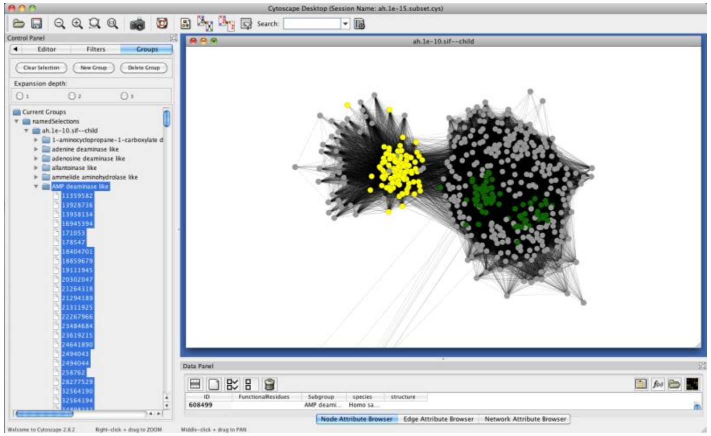
- Desktop applications

Cytoscape

- **Molecular network analysis software**
- **Open-source**
- **Can automatically deploy web-based visualizations of graphs loaded in the desktop application**

Main features:

- Supports many standard network file formats
- Built-in layout algorithms
- Node/egde filtering
- Supports plugins
- Available as both the desktop application (Cytoscape) and web libraries (Cytoscape.js)



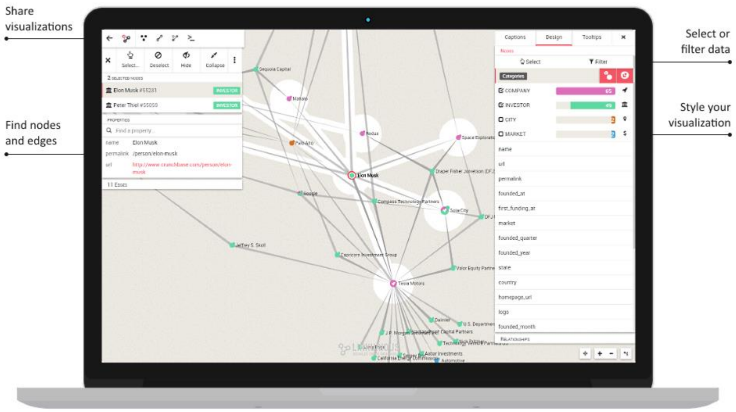
- Web libraries

LINKURIOUS
VISUALIZE GRAPH DATA EASILY

- **Commercial JavaScript library based on Sigma.js**
- **Focuses on providing a business intelligence approach to graphs**
- **Provides business support and integrations for companies**

Main features:

- Rendering for up to 20K nodes
- Supports data import/export operations with numerous formats
- Plugins available
- Built-in search engine
- Cypher language support to query the visualized graph
- Built-in graph data editor



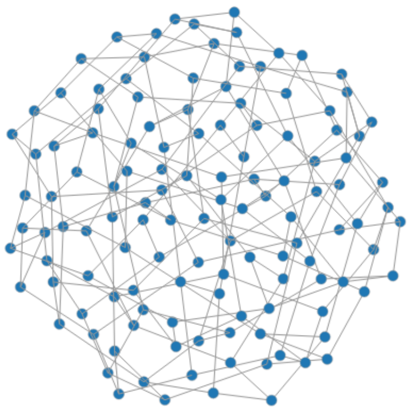
- Web libraries

D3 Data-Driven Documents

- **JavaScript library to create graphic representations of data**
- **Not a graph-specific library, but can display graphs with great user interaction and dynamic modifications.**

Main features:

- Excels at graphic customization
- Supports animated transitions
- Large community support



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

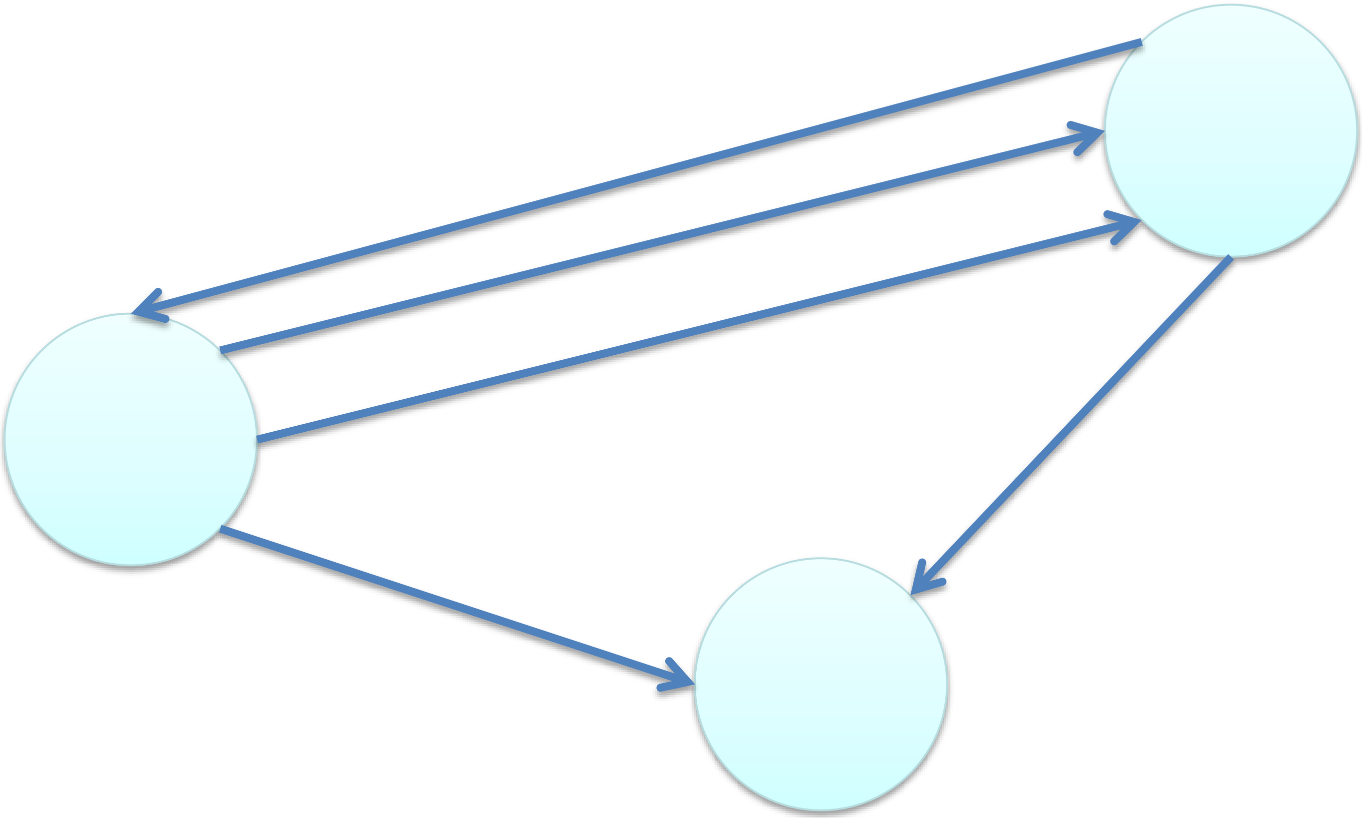
	# persons	query time
Relational database	1000	2000ms
Neo4j	1000	2ms

Social Network “path exists” Performance

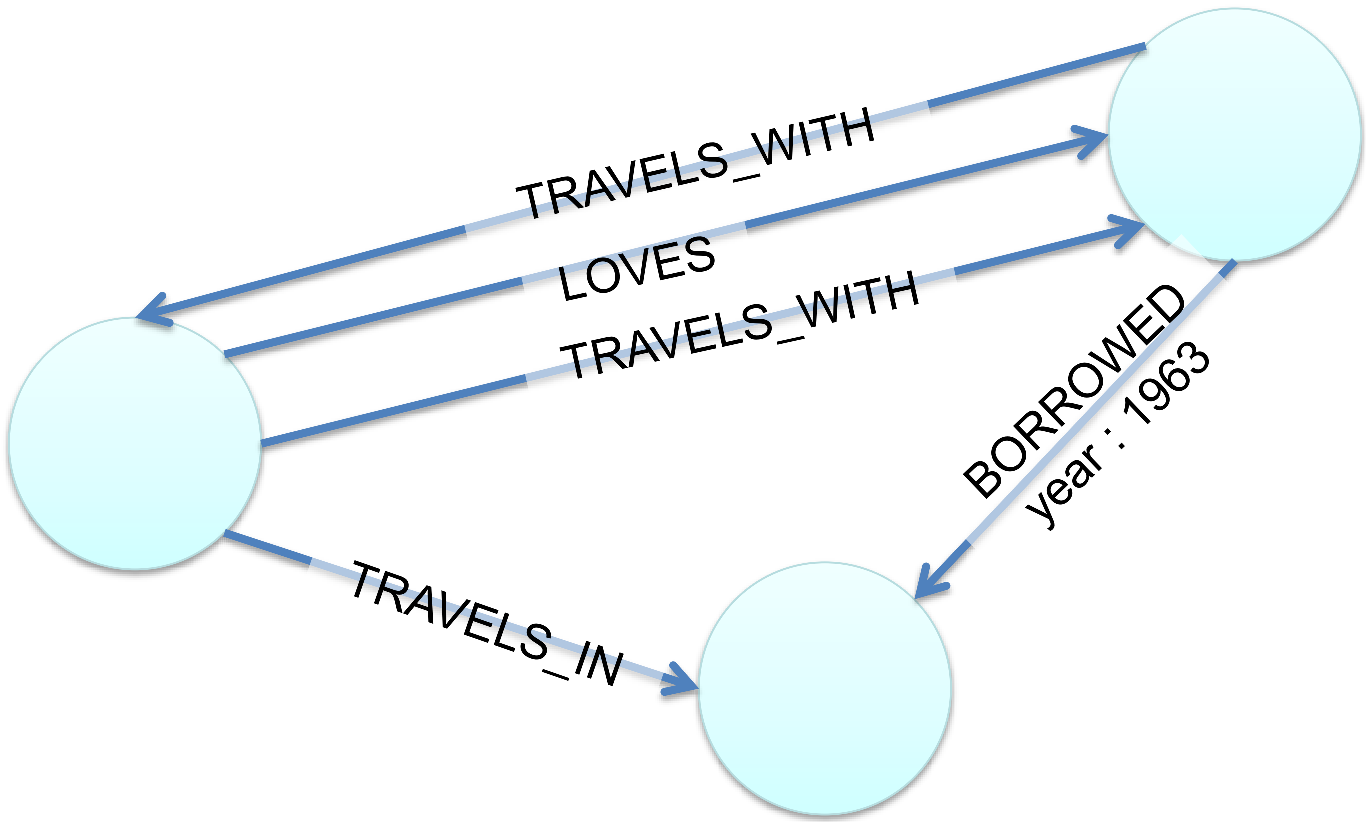
- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4
  - Caches warm to eliminate disk IO

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

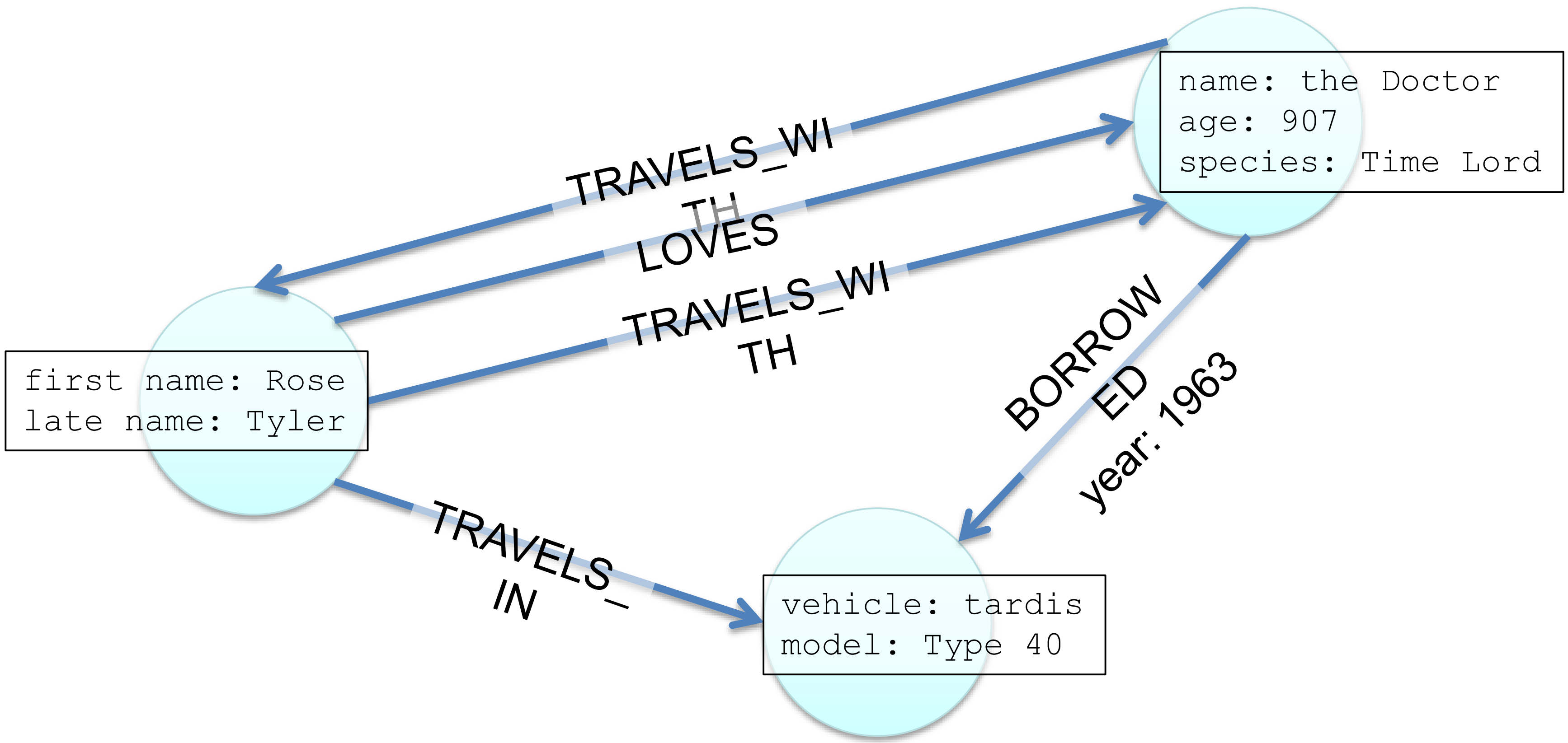
Property Graph Model



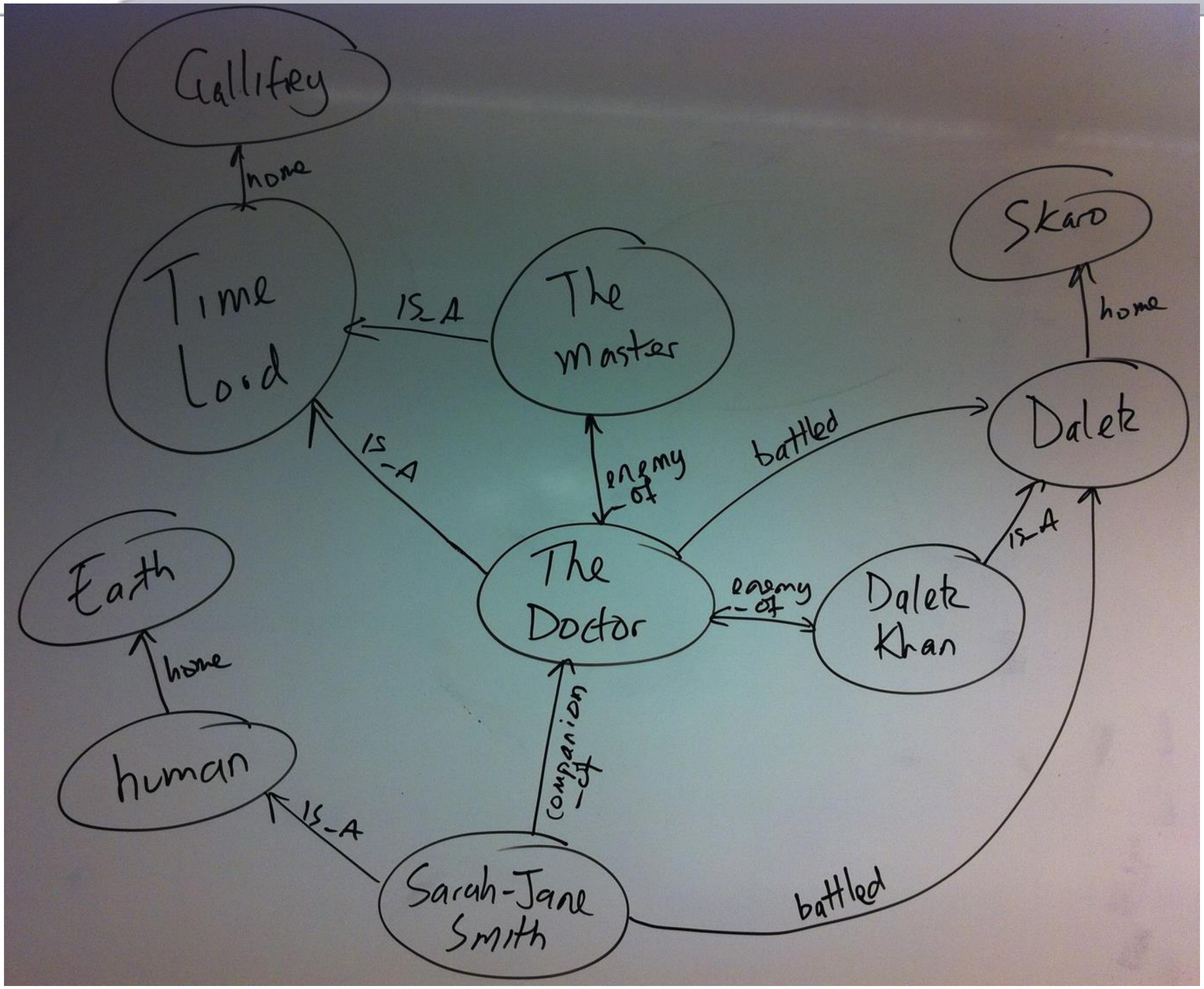
Property Graph Model



Property Graph Model



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

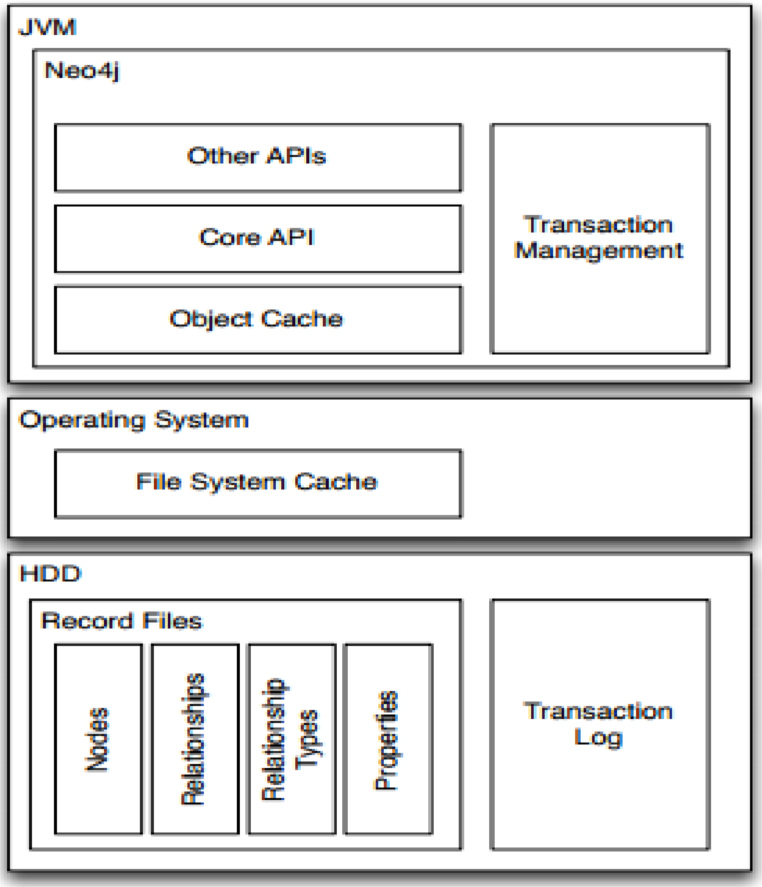
What's Neo4j?

- It's is a Graph Database
- Embeddable and server
- Full ACID transactions
  - don't mess around with durability, ever.
- Schema free, bottom-up data model design

More on Neo4j

- Neo4j is stable
  - In 24/7 operation since 2003
- Neo4j is under active development
- High performance graph operations
  - Traverses 1,000,000+ relationships / second on commodity hardware

Neo4j Logical Architecture



Remember, there’s NOSQL

So how do we query it?

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!

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

UNIVERSITAT DE BARCELONA

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
```

Cypher Query

```
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
```

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

UNIVERSITAT DE BARCELONA

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);
```

UNIVERSITAT DE BARCELONA

Path finder code

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

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


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

algo

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());
```

REST API

- The only way to access the server today
  - Binary protocol part of the product roadmap
- JSON is the default format
  - Remember to include these headers:
    - Accept:application/json
    - Content-Type:application/json

Example

curl http://localhost:7474/db/data/node/1


```
{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/1/relationships/out",
  "data" : {
    "character" : "Doctor"
  },
  "traverse" : "http://localhost:7474/db/data/node/1/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/1/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/1",
  "properties" : "http://localhost:7474/db/data/node/1/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/1/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/1/relationships",
  "all_relationships" : "http://localhost:7474/db/data/node/1/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/in/{-list|&|types}"
}
```