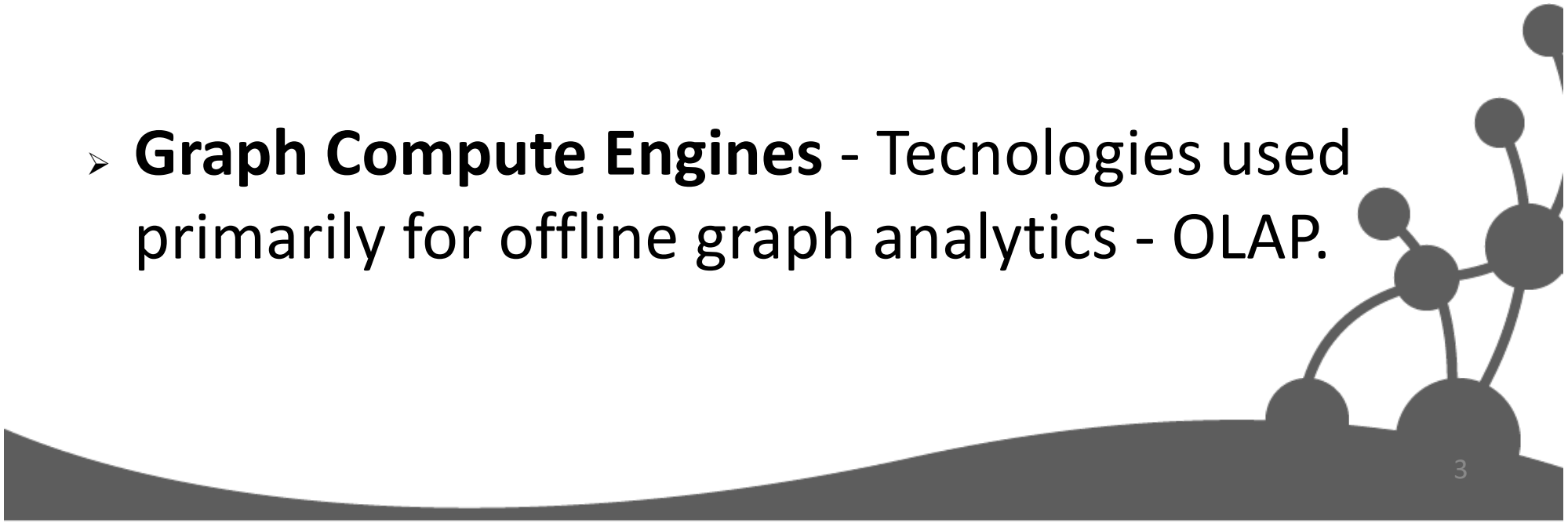
Graph databases

High Level view of Graph Space

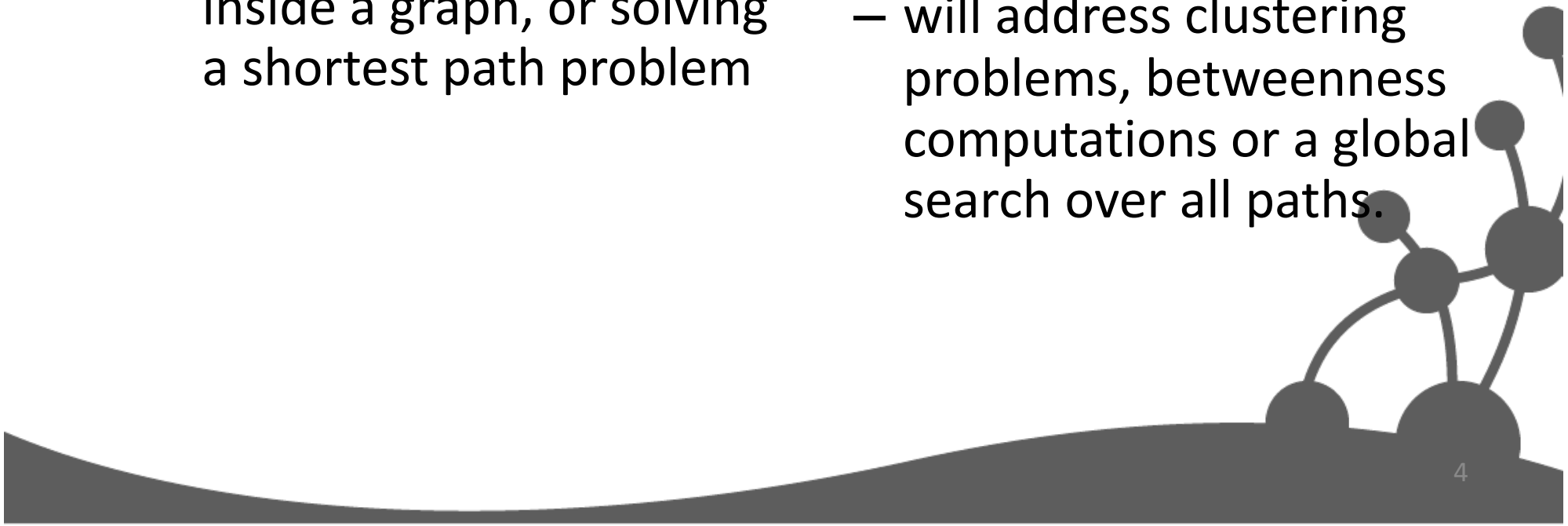
- **Graph Databases** - Technologies used primarily for transactional online graph persistence – OLTP.

- **Graph Compute Engines** - Tecnologies used primarily for offline graph analytics - OLAP.



Use cases

- Graph database
  - Low latency (1 ms),
  - high volumes are typically handled with vertical scaling on a single server.
  - well fitted for navigation inside a graph, or solving a shortest path problem
- Graph processing framework
  - Scaling by distributing the processing across several commodity servers,
  - lower latency (secs)
  - will address clustering problems, betweenness computations or a global search over all paths.

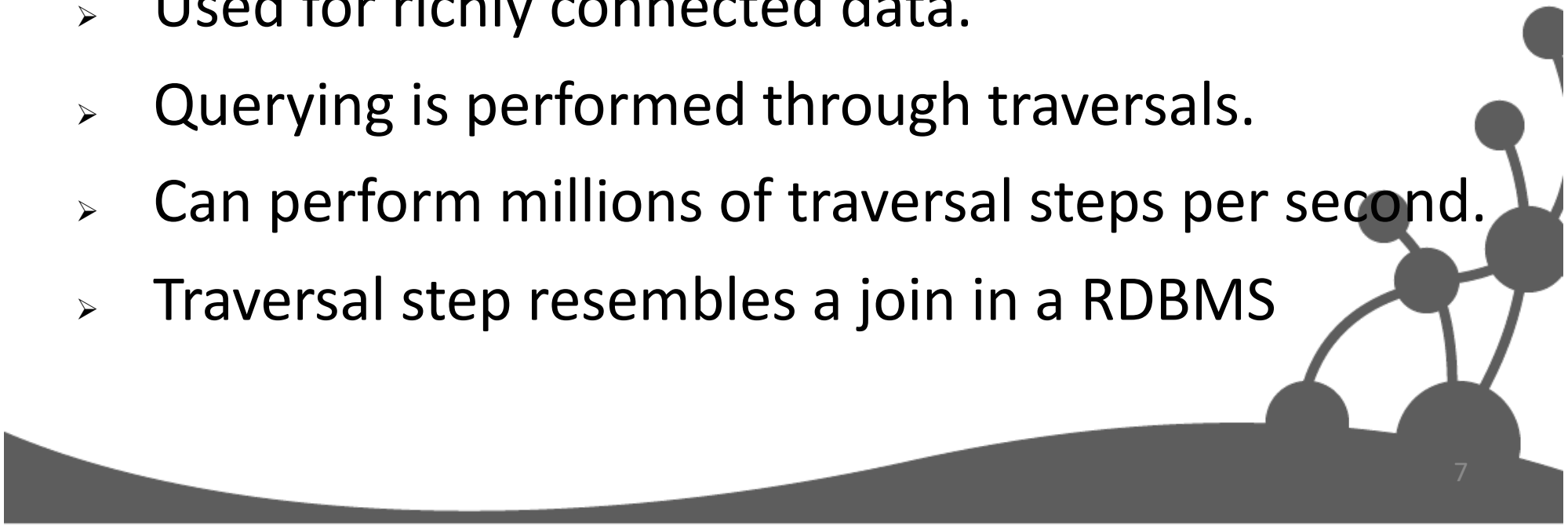


Graph Databases

.

- Online database management system with -
  - **C**reate, **R**ead, **U**pdate, **D**elete

  methods that expose a graph data model.
- Built for use with transactional (OLTP) systems.
- Used for richly connected data.
- Querying is performed through traversals.
- Can perform millions of traversal steps per second.
- Traversal step resembles a join in a RDBMS



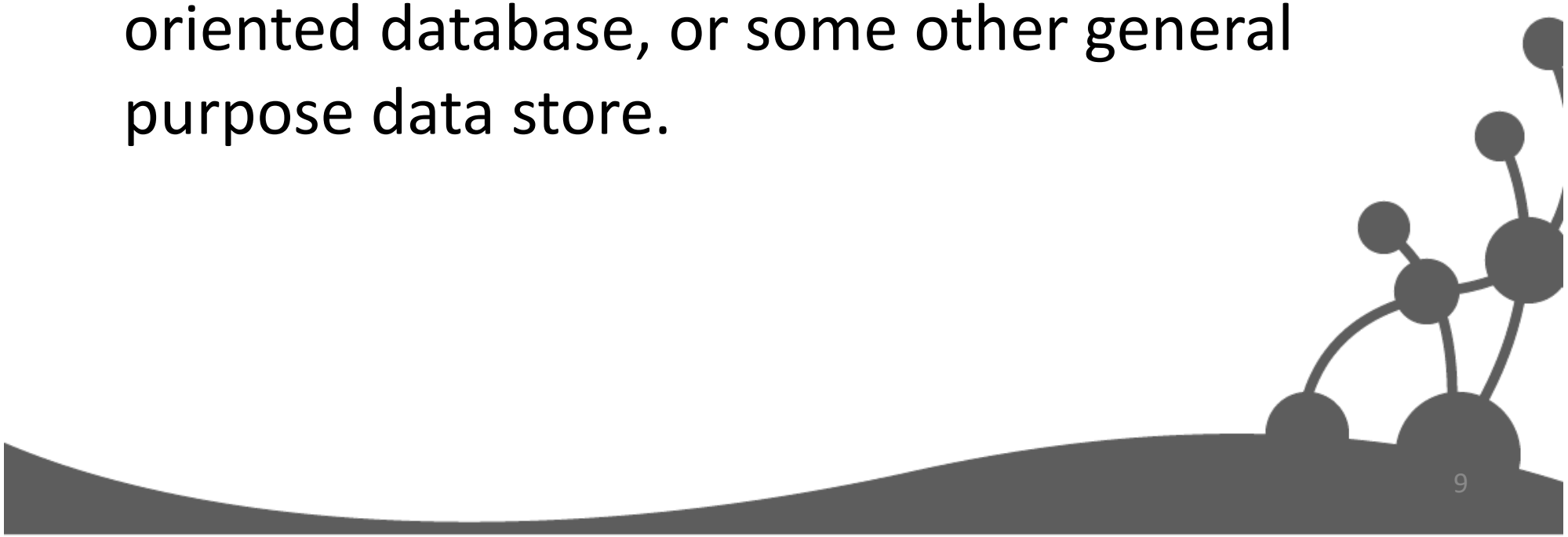
Graph Database Properties

- The Underlying Storage : Native / Non-Native
- The Processing Engine : Native / Non-Native

Graph DB – The Underlying Storage

- Native Graph Storage – Optimized and designed for storing and managing graphs.

- Non-Native Graph Storage – Serialize the graph data into a relational database, an object oriented database, or some other general purpose data store.



Graph DB – The processing Engine

- Native graph DB :
  - Index free adjacency – Connected Nodes physically point to each other in the database

Power of Graph Databases

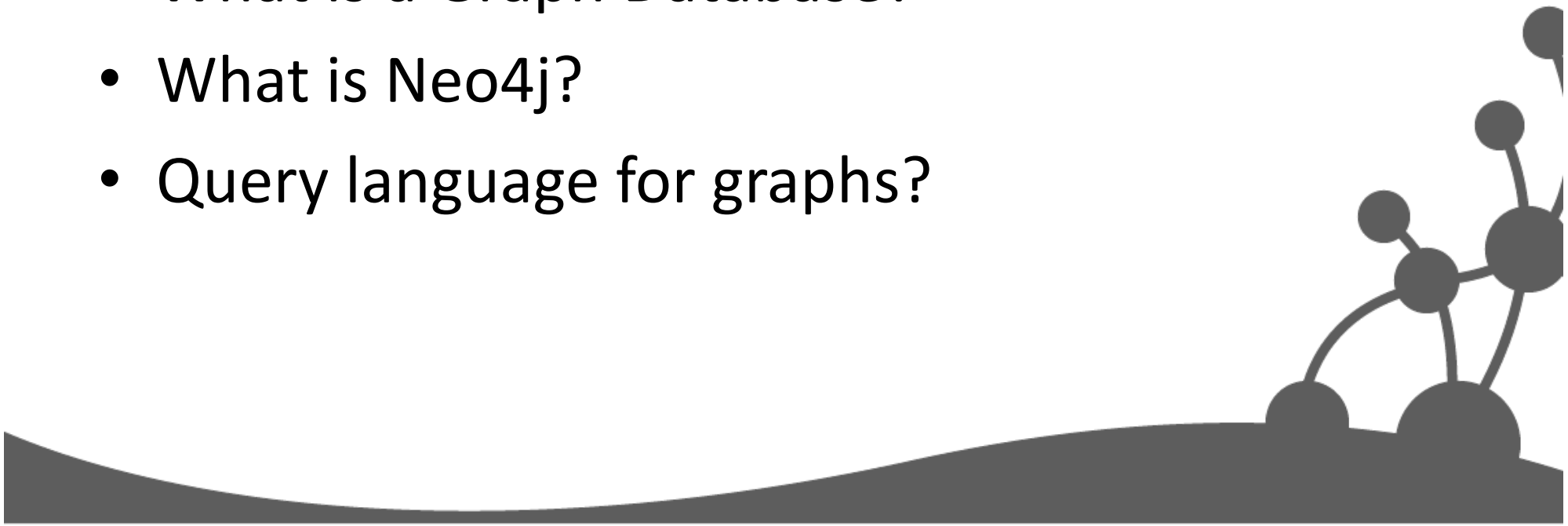
- Performance
- Flexibility
- Agility

Comparison

- Relational Databases
- NoSQL Databases
- Graph Databases

Agenda

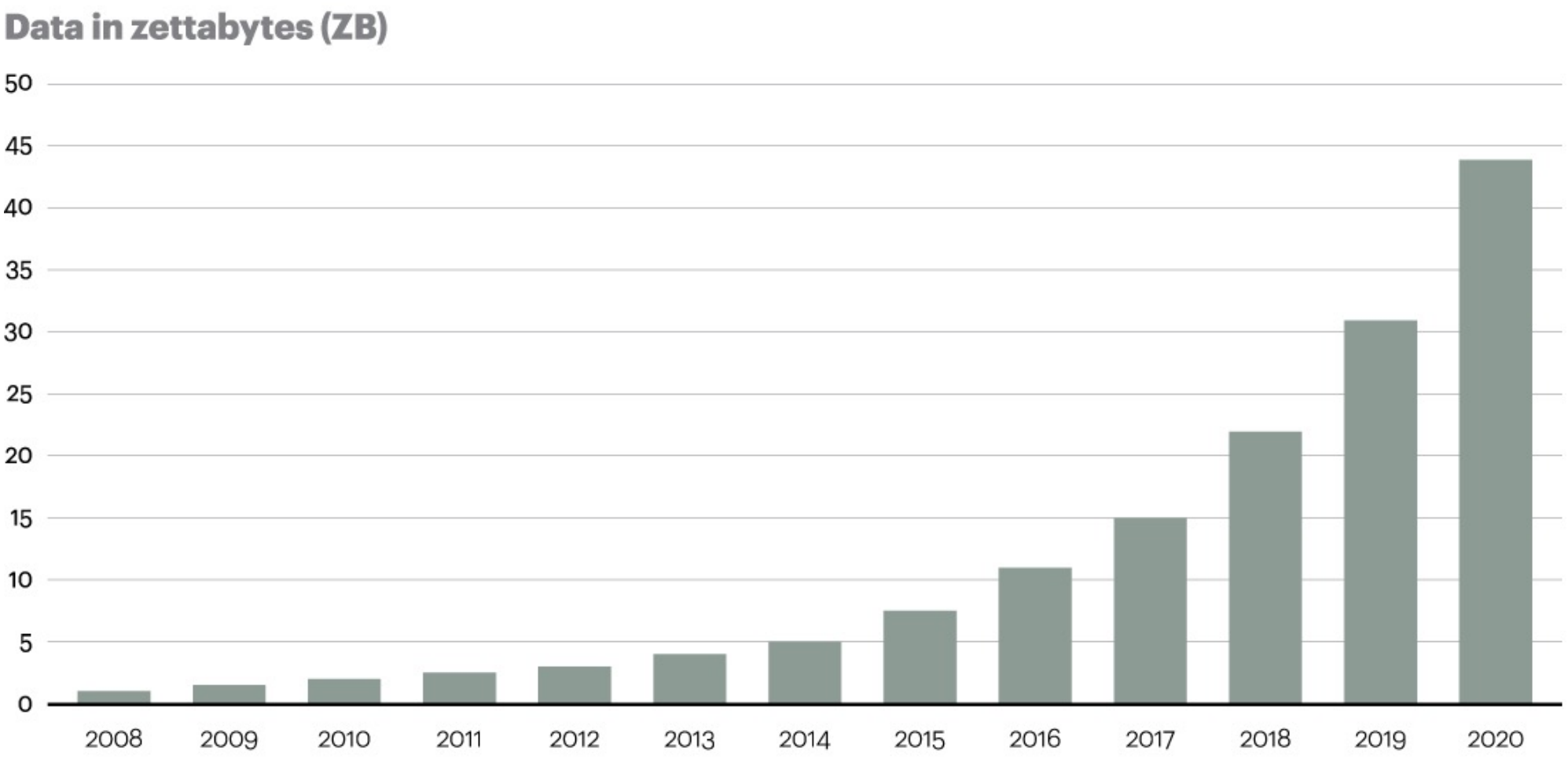
- Trends in Data
- NOSQL
- What is a Graph?
- What is a Graph Database?
- What is Neo4j?
- Query language for graphs?



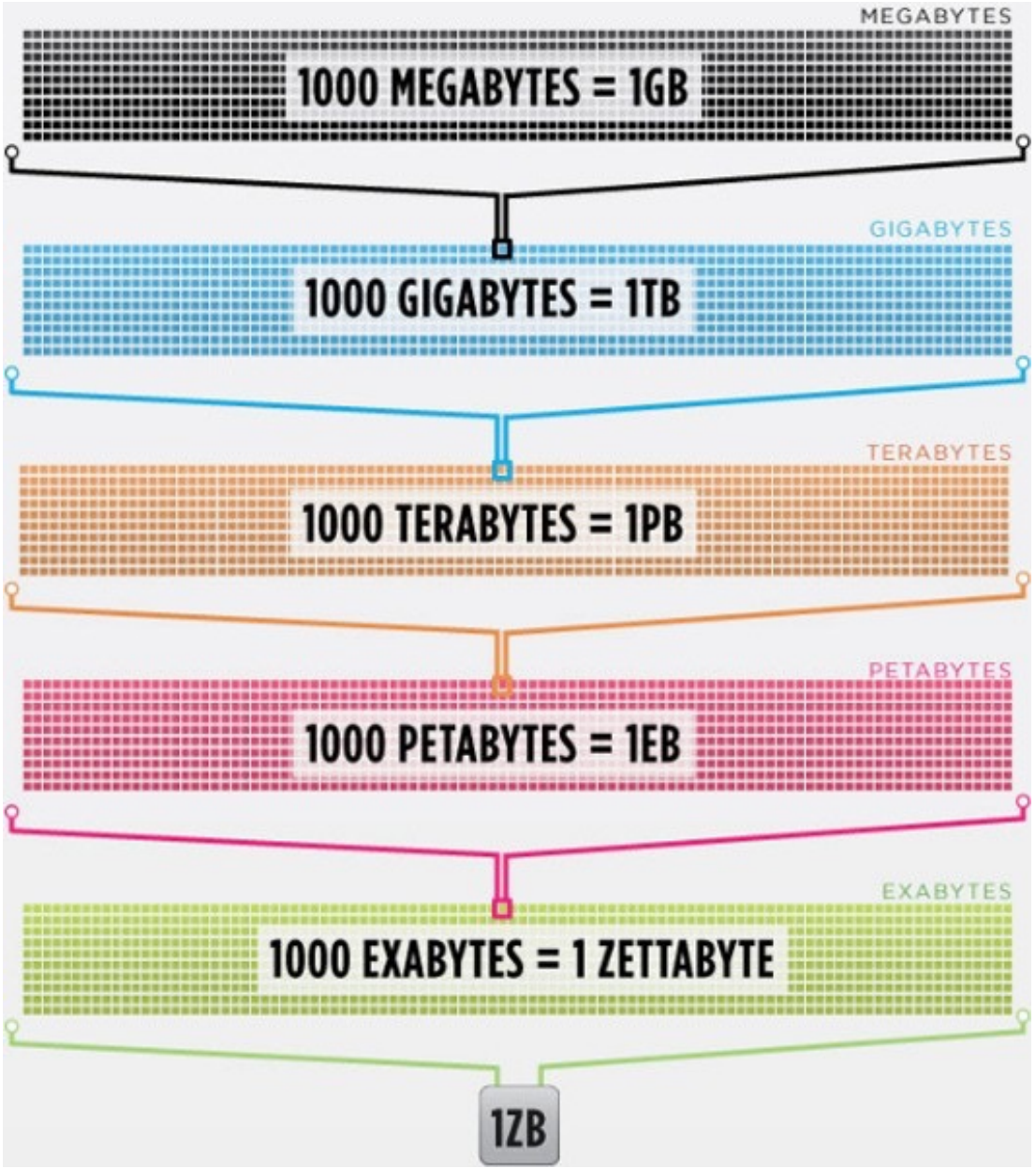
Trends in Data

Figure 1

Data is growing at a 40 percent compound annual rate, reaching nearly 45 ZB by 2020



Source: Oracle, 2012

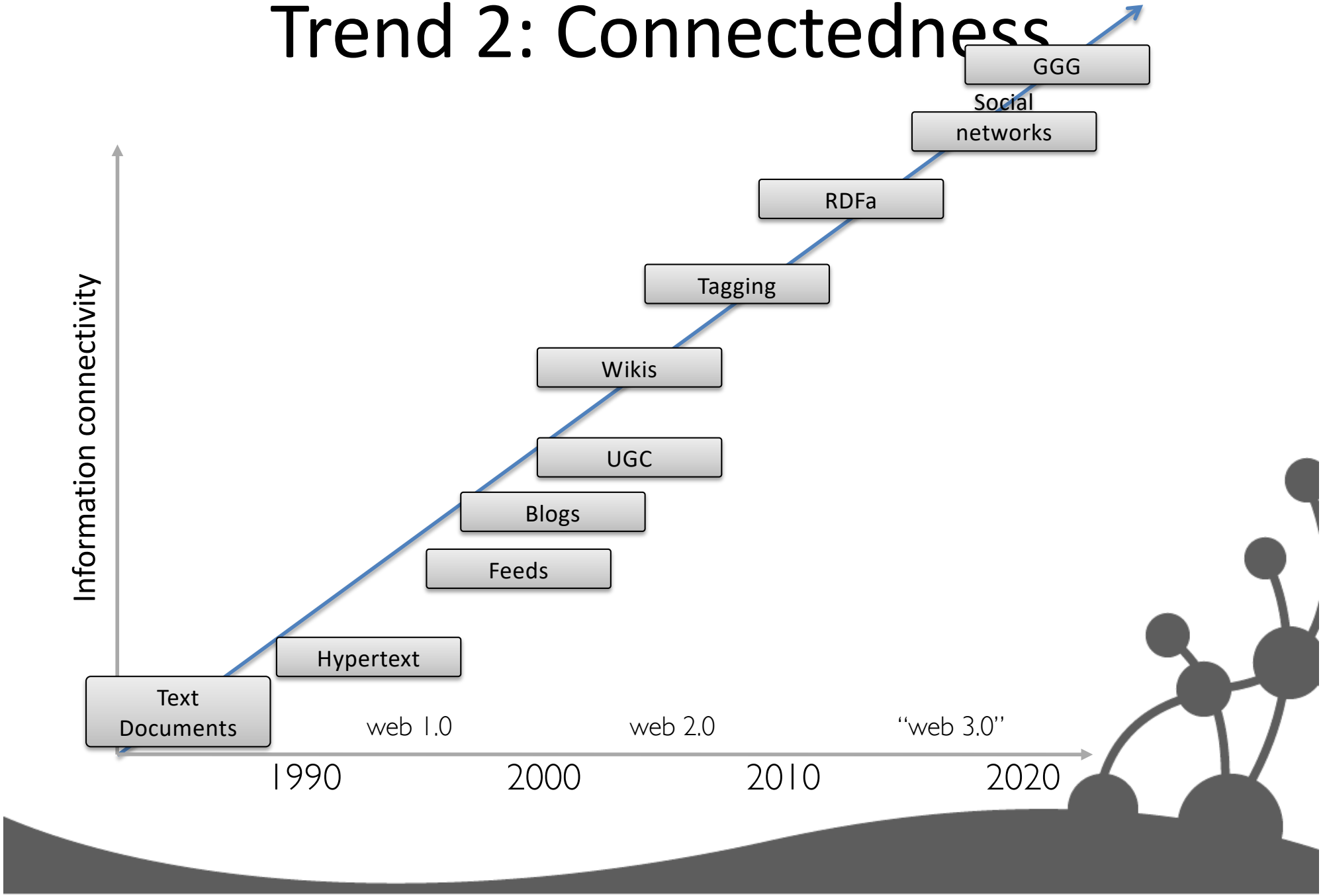


Data is getting bigger:

"Every 2 days we create as much information as we did up to 2003"

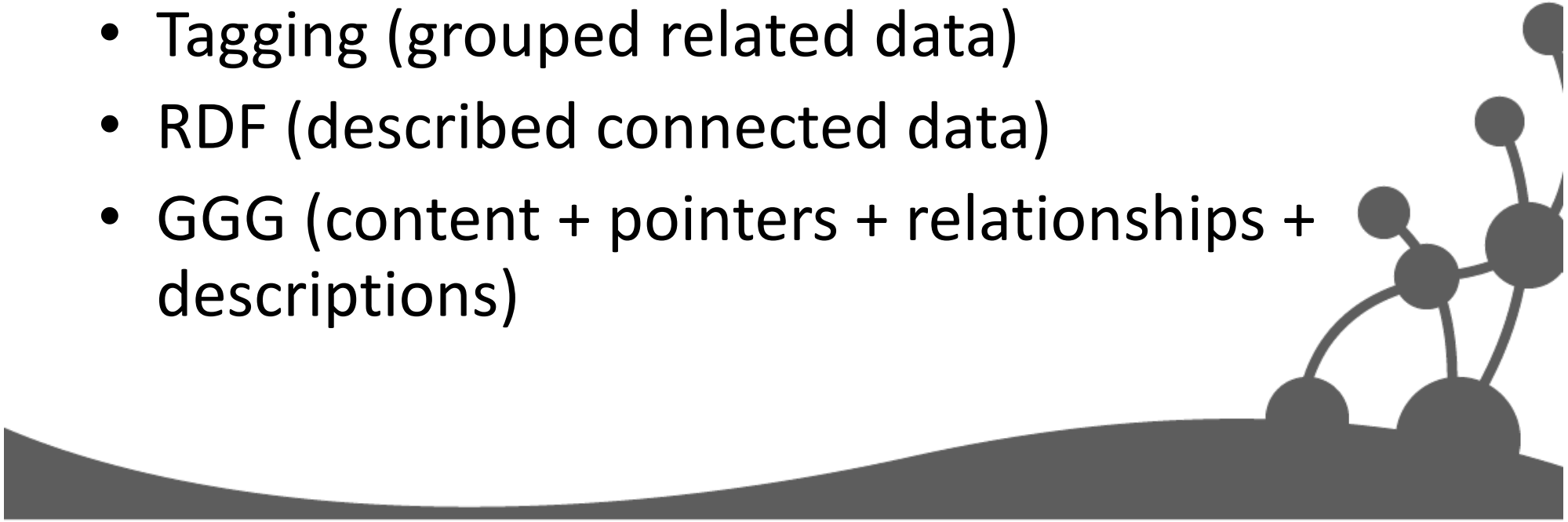
– Eric Schmidt, Google

Trend 2: Connectedness



Data is more connected:

- Text (content)
- HyperText (added pointers)
- RSS (joined those pointers)
- Blogs (added pingbacks)
- Tagging (grouped related data)
- RDF (described connected data)
- GGG (content + pointers + relationships + descriptions)



Data is more Semi-Structured:

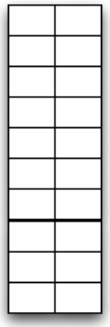
- If you tried to collect all the data of every movie ever made, how would you model it?
- Actors, Characters, Locations, Dates, Costs, Ratings, Showings, Ticket Sales, etc.

NOSQL

Not Only SQL

Less than 10% of the NOSQL Vendors

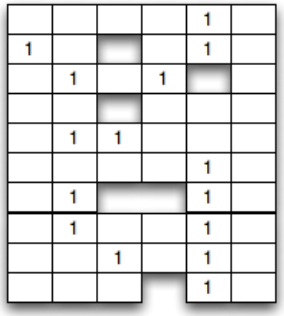
Key-Value



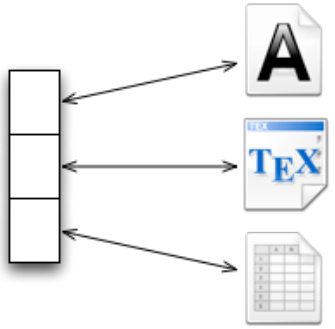
Graph DB

Four NOSQL Categories

BigTable

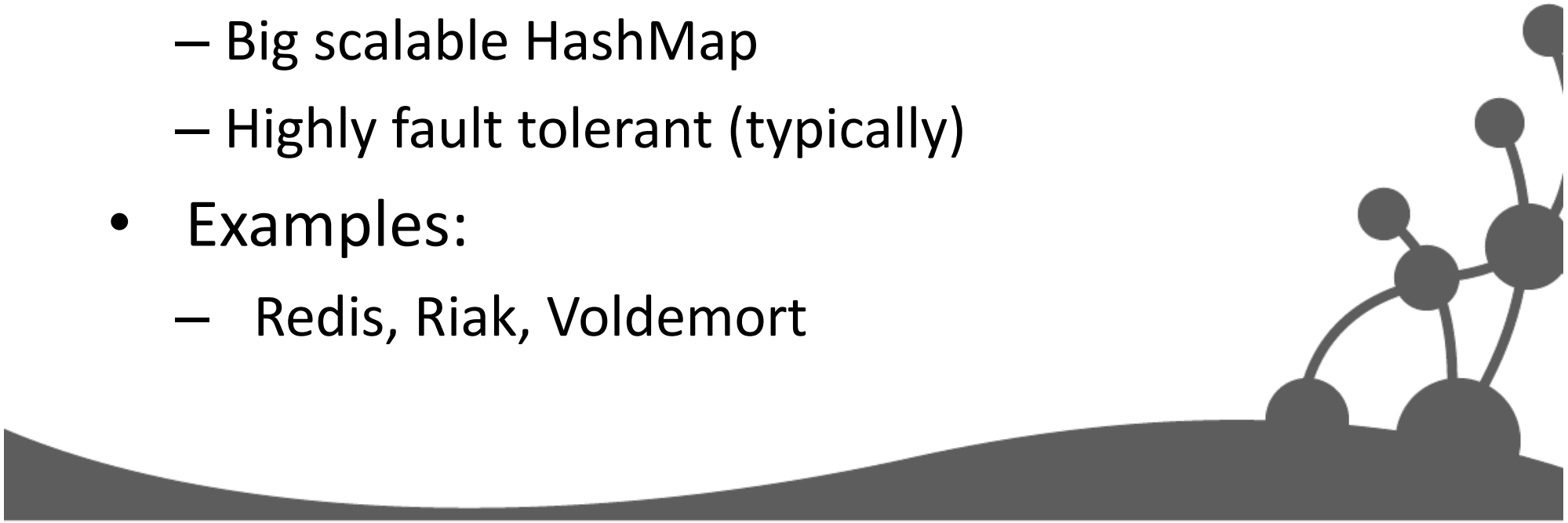


Document



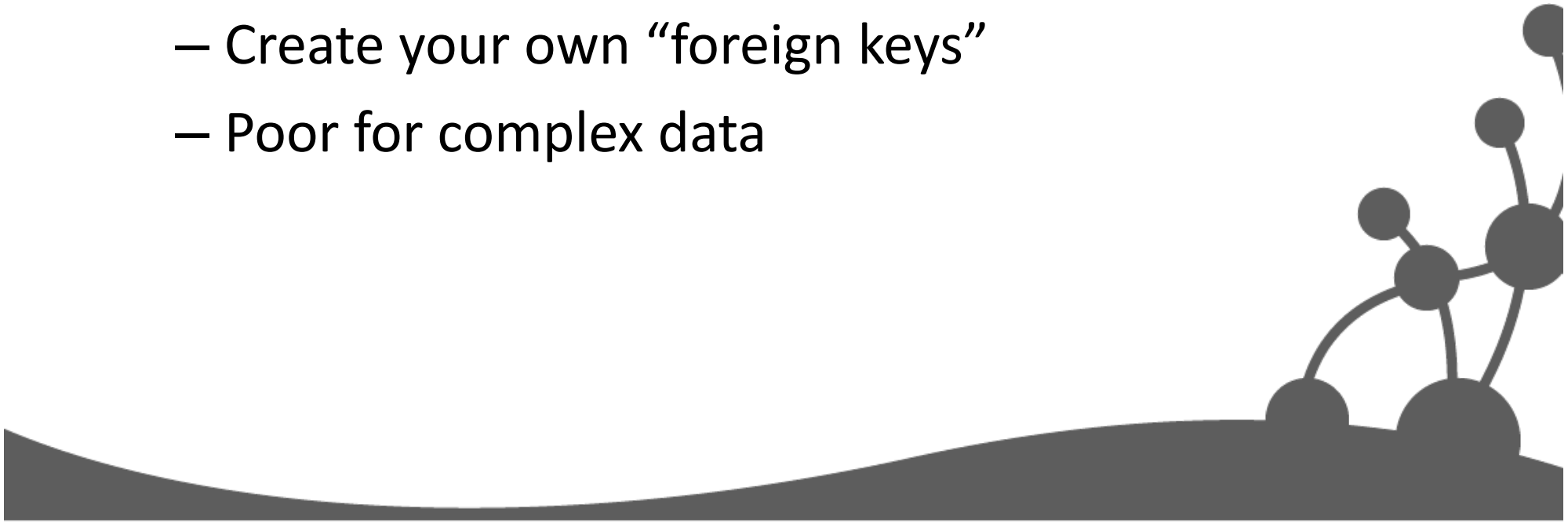
Key Value Stores

- Most Based on **Dynamo**: Amazon Highly Available Key-Value Store
- Data Model:
  - Global key-value mapping
  - Big scalable HashMap
  - Highly fault tolerant (typically)
- Examples:
  - Redis, Riak, Voldemort



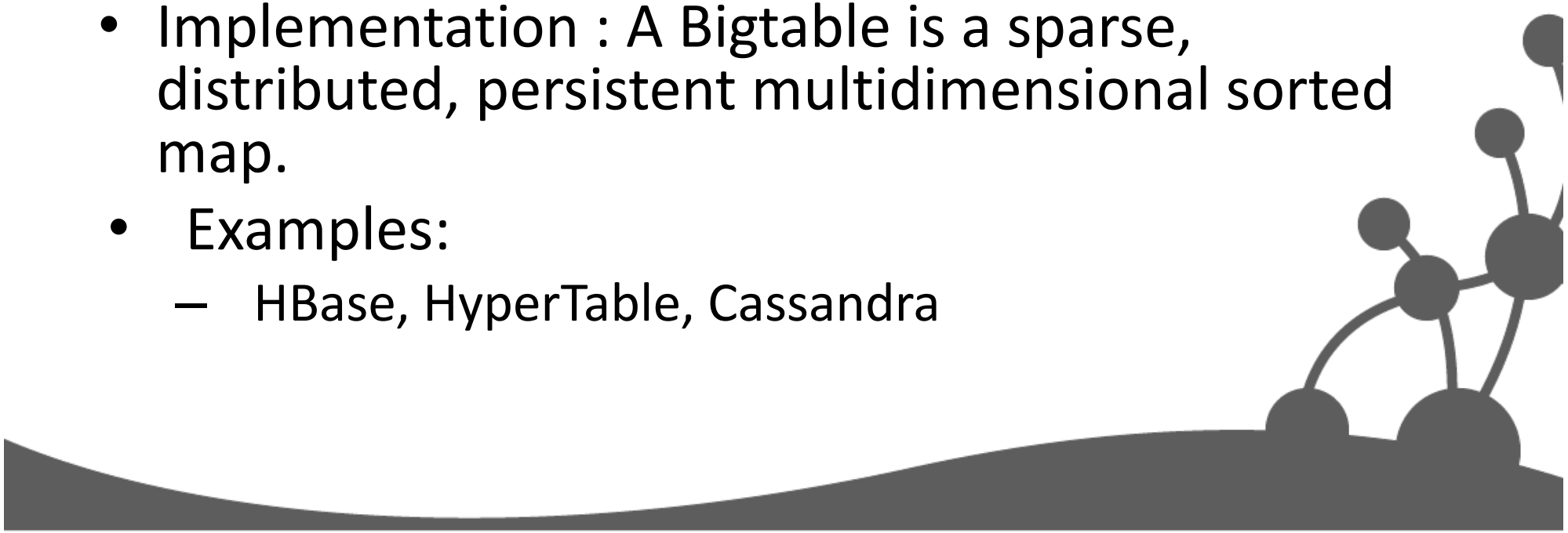
Key Value Stores: Pros and Cons

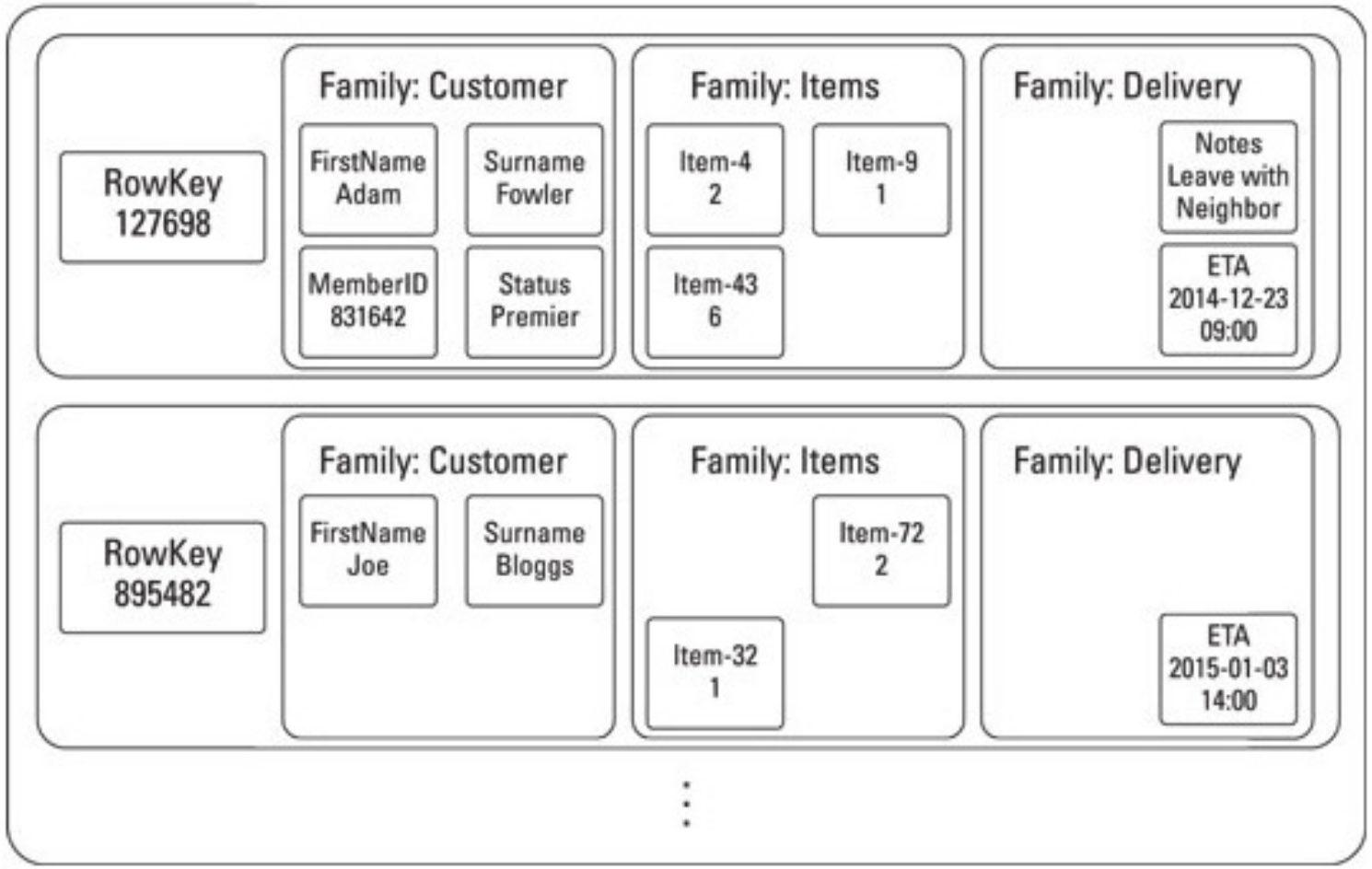
- Pros:
  - Simple data model
  - Scalable
- Cons
  - Create your own “foreign keys”
  - Poor for complex data



Column Family

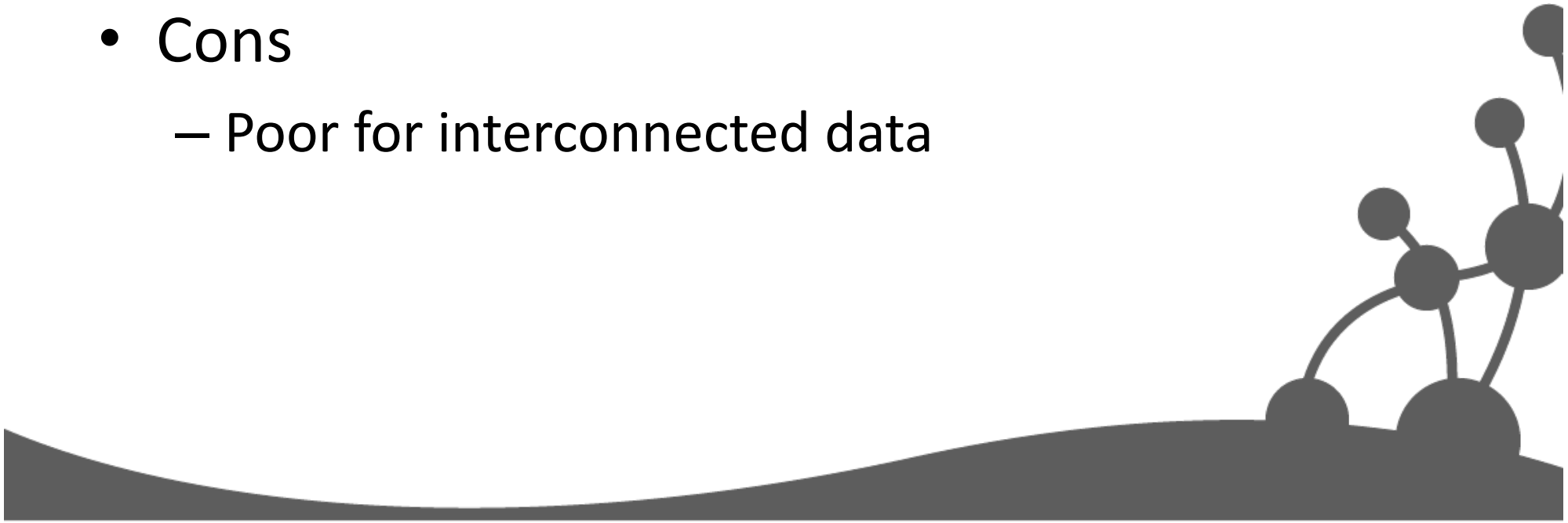
- Most Based on **BigTable**: Google's Distributed Storage System for Structured Data
- Data Model:
  - A big table, with column families
  - Map Reduce for querying/processing
- Implementation : A Bigtable is a sparse, distributed, persistent multidimensional sorted map.
- Examples:
  - HBase, HyperTable, Cassandra




RowKey
127698
Family: Customer
FirstName
Adam
Surname
Fowler
MemberID
831642
Status
Premier
Family: Items
Item-4
2
Item-9
1
Item-43
6
Family: Delivery
Notes
Leave with
Neighbor
ETA
2014-12-23
09:00
RowKey
895482
Family: Customer
FirstName
Joe
Surname
Bloggs
Family: Items
Item-72
2
Item-32
1
Family: Delivery
ETA
2015-01-03
14:00

Column Family: Pros and Cons

- Pros:
  - Supports Semi-Structured Data
  - Naturally Indexed (columns)
  - Scalable
- Cons
  - Poor for interconnected data



Document Databases

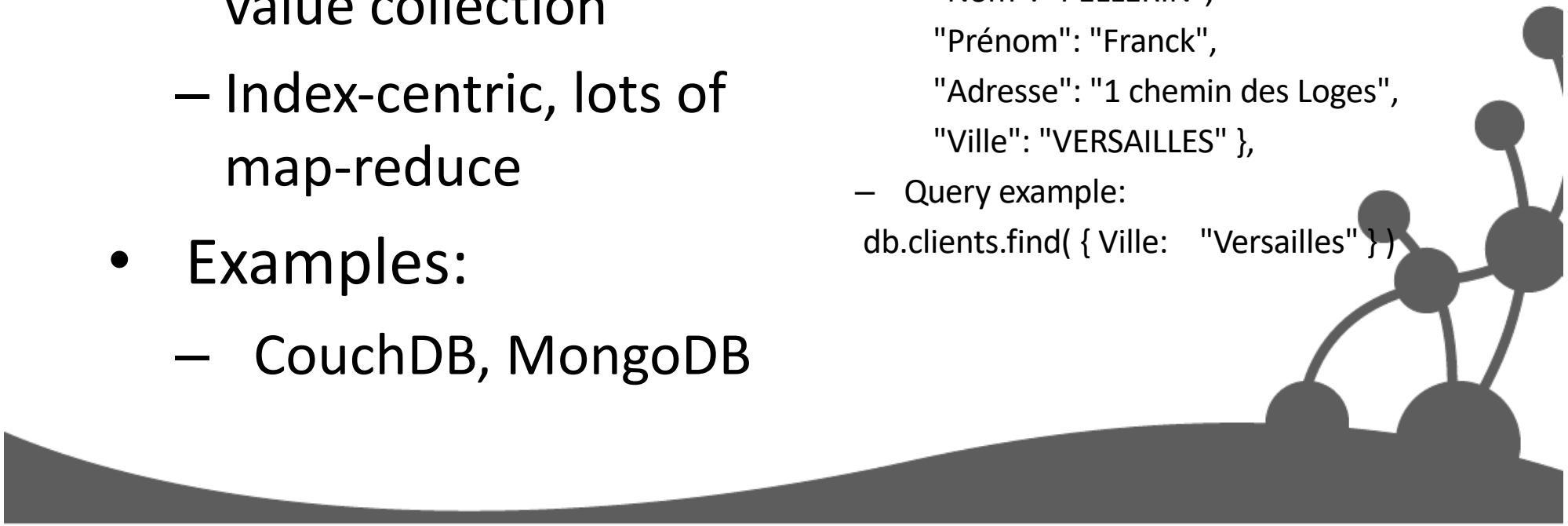
- Data Model:
  - A collection of documents
  - A document is a key value collection
  - Index-centric, lots of map-reduce
- Examples:
  - CouchDB, MongoDB

- Example :

```
{ "_id”:ObjectId("4efa8d2b7d284dad101e4bc7"),
        "Nom": "DUMOND",
        "Prénom": "Jean",
        "Âge": 43 },
{ "_id”:ObjectId("4efa8d2b7d284dad101e4bc8"),
        "Nom": "PELLERIN",
        "Prénom": "Franck",
        "Adresse": "1 chemin des Loges",
        "Ville": "VERSAILLES" },
```

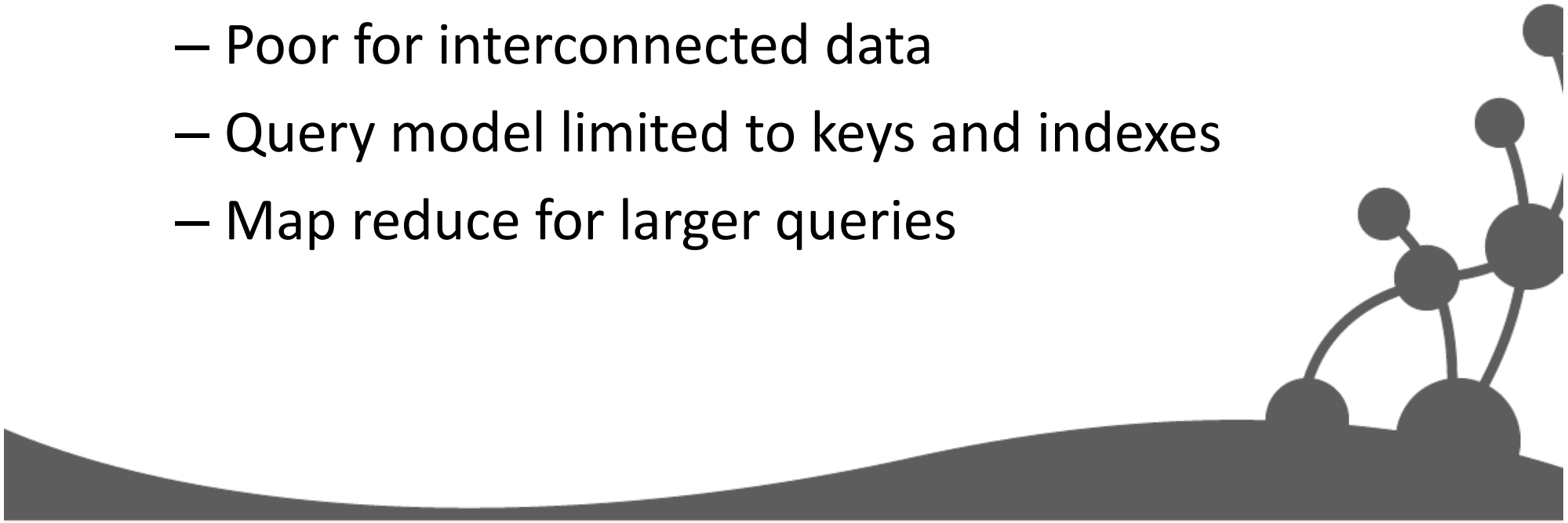
  - Query example:

```
db.clients.find( { Ville:    "Versailles" } )
```



Document Databases: Pros and Cons

- Pros:
  - Simple, powerful data model
  - Scalable
- Cons
  - Poor for interconnected data
  - Query model limited to keys and indexes
  - Map reduce for larger queries

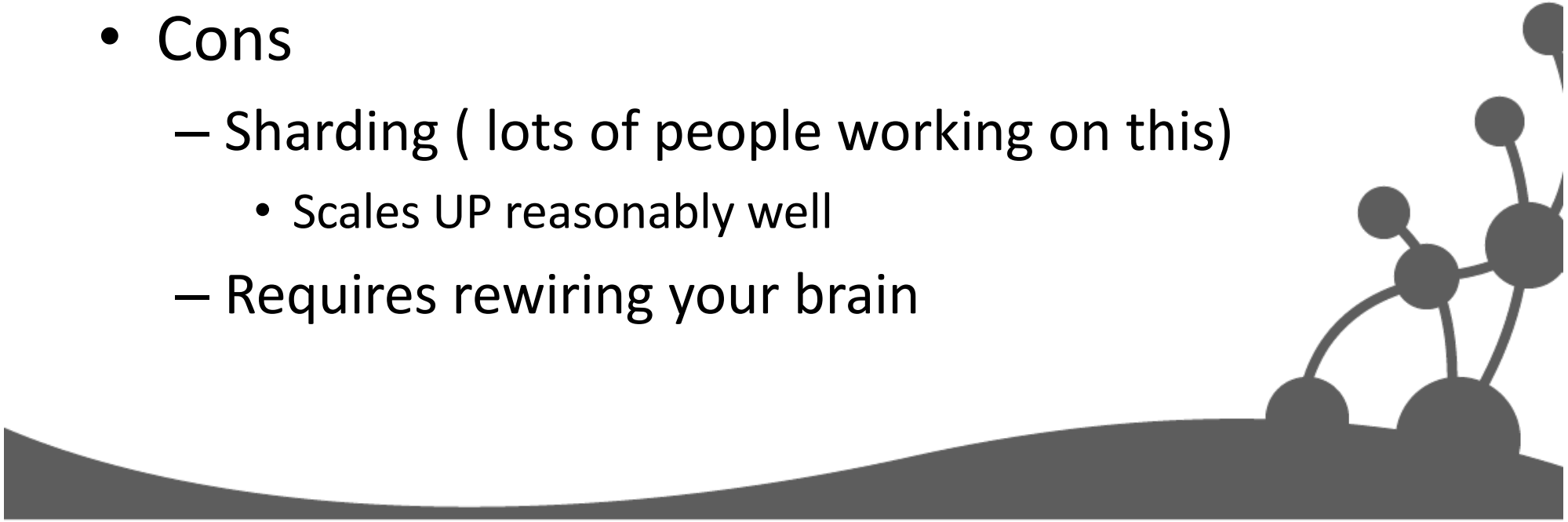


Graph Databases

- Data Model:
  - Nodes and Relationships
- Examples:
  - Neo4j, OrientDB, InfiniteGraph, AllegroGraph

Graph Databases: Pros and Cons

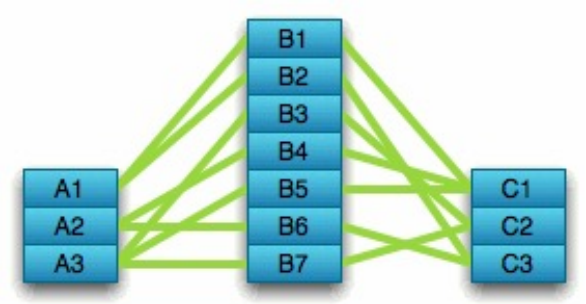
- Pros:
  - Powerful data model, as general as RDBMS
  - Connected data locally indexed
  - Easy to query
- Cons
  - Sharding (lots of people working on this)
    - Scales UP reasonably well
  - Requires rewiring your brain



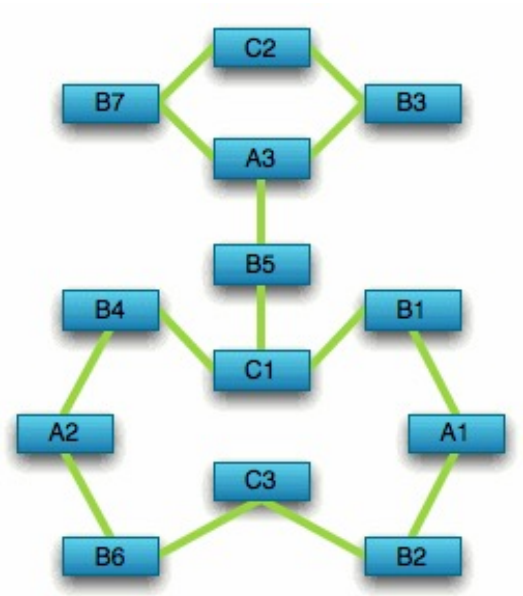
Compared to Relational Databases

Optimized for aggregation

```
SELECT T.M, S.N
FROM T, S
WHERE T.H=S.I ;
```



Optimized for connections

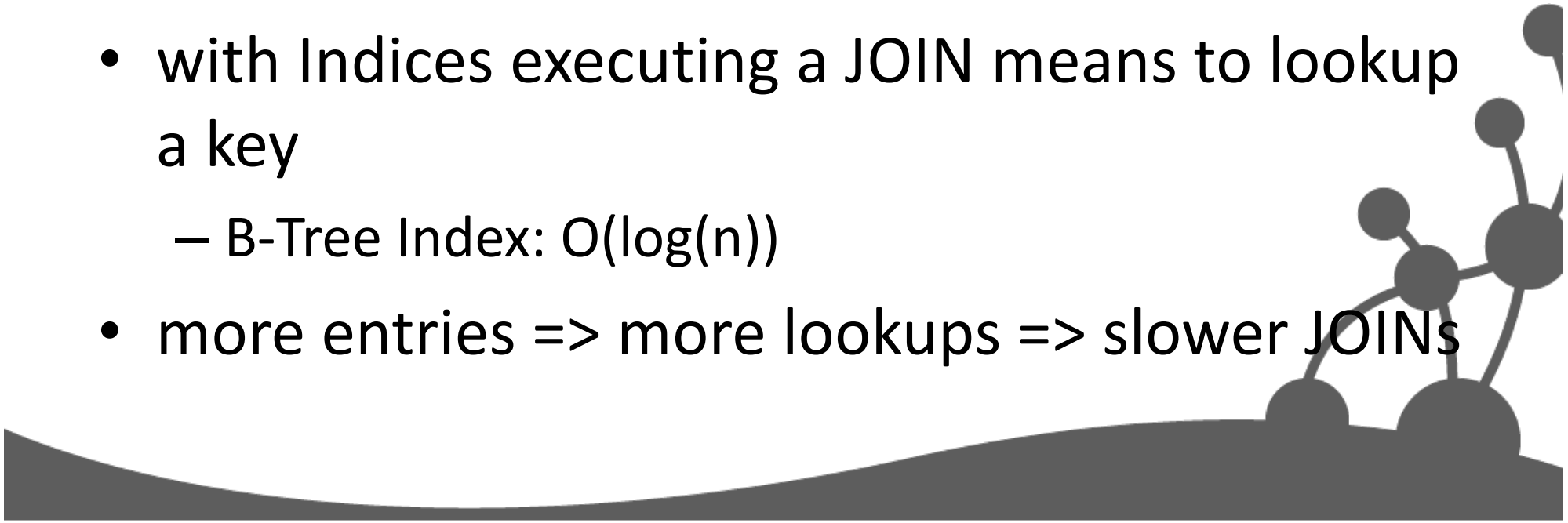


Join problem

- all JOINs are executed every time you query

(traverse) the relationship

- executing a JOIN means to search for a key in another table
- with Indices executing a JOIN means to lookup a key
  - B-Tree Index: O(log(n))
- more entries => more lookups => slower JOINs

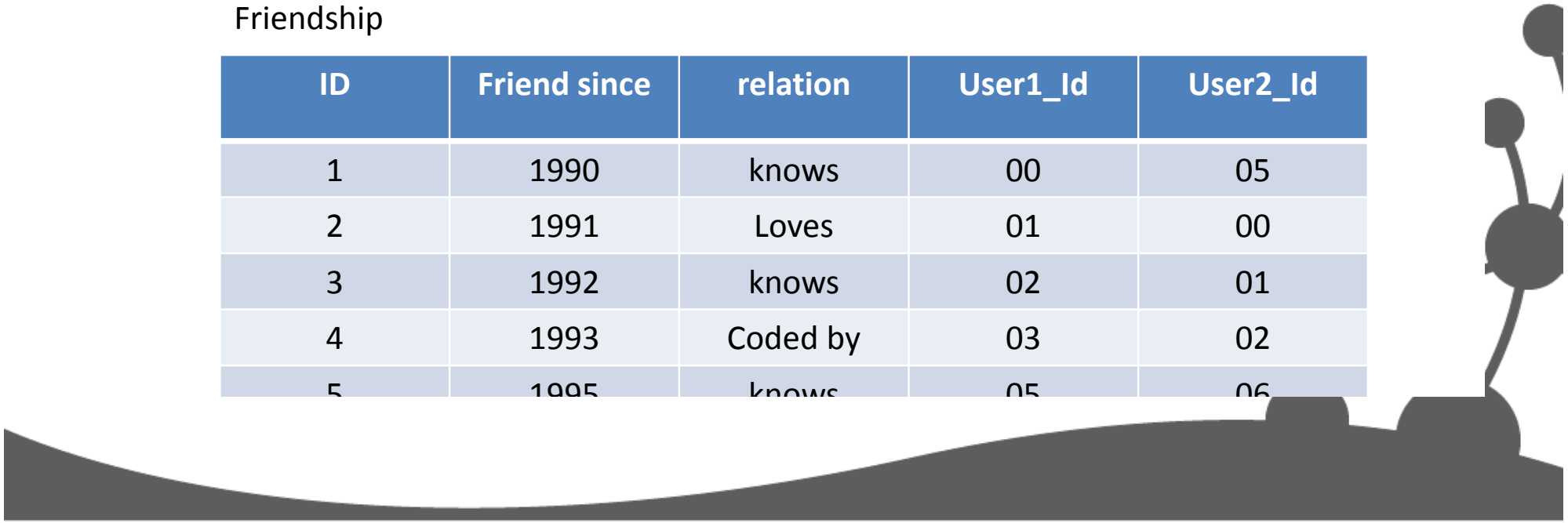


Relational model

00	Neo	M	29
01	Trinity	F	28
02	Reagan	M	32
03	Agent Smith	M	30
04	The Architect	M	35
05	Morpheus	M	69

Friendship

ID	Friend since	relation	User1_Id	User2_Id
1	1990	knows	00	05
2	1991	Loves	01	00
3	1992	knows	02	01
4	1993	Coded by	03	02
5	1995	knows	05	06



Graph model

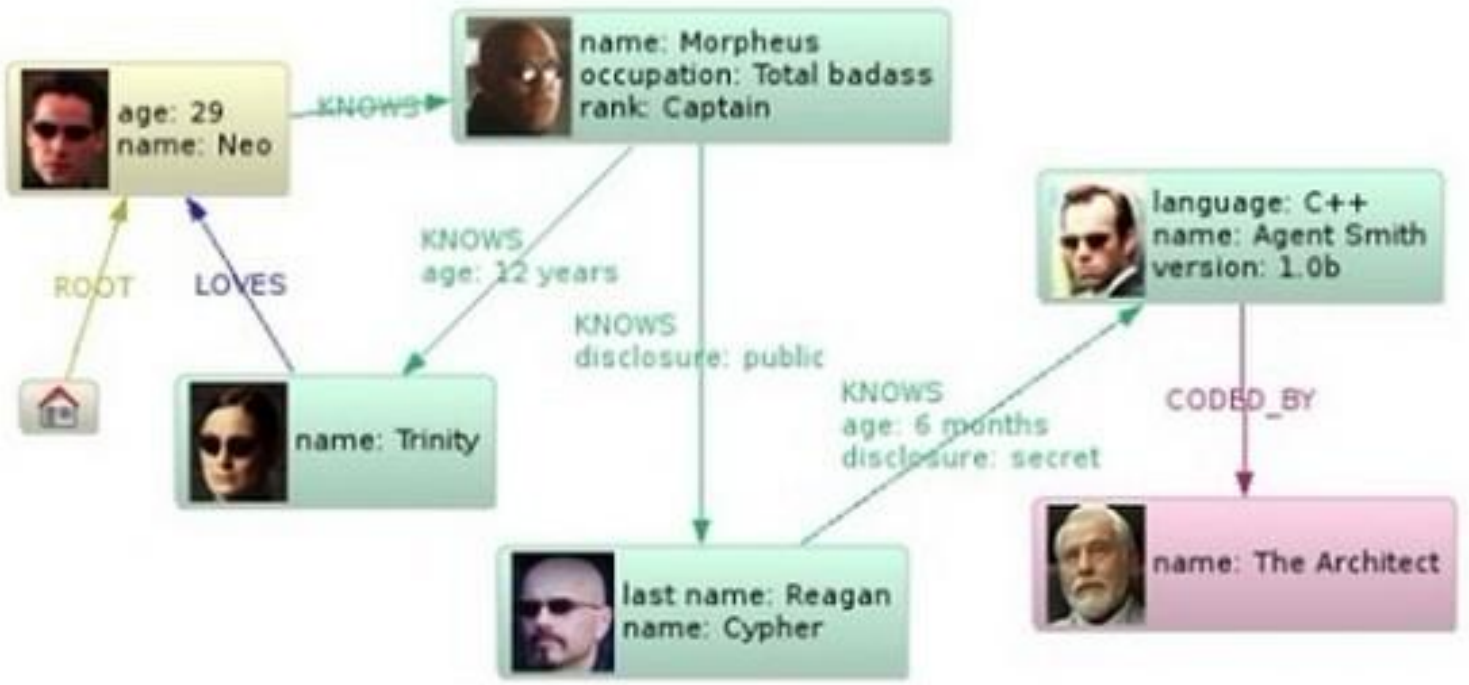
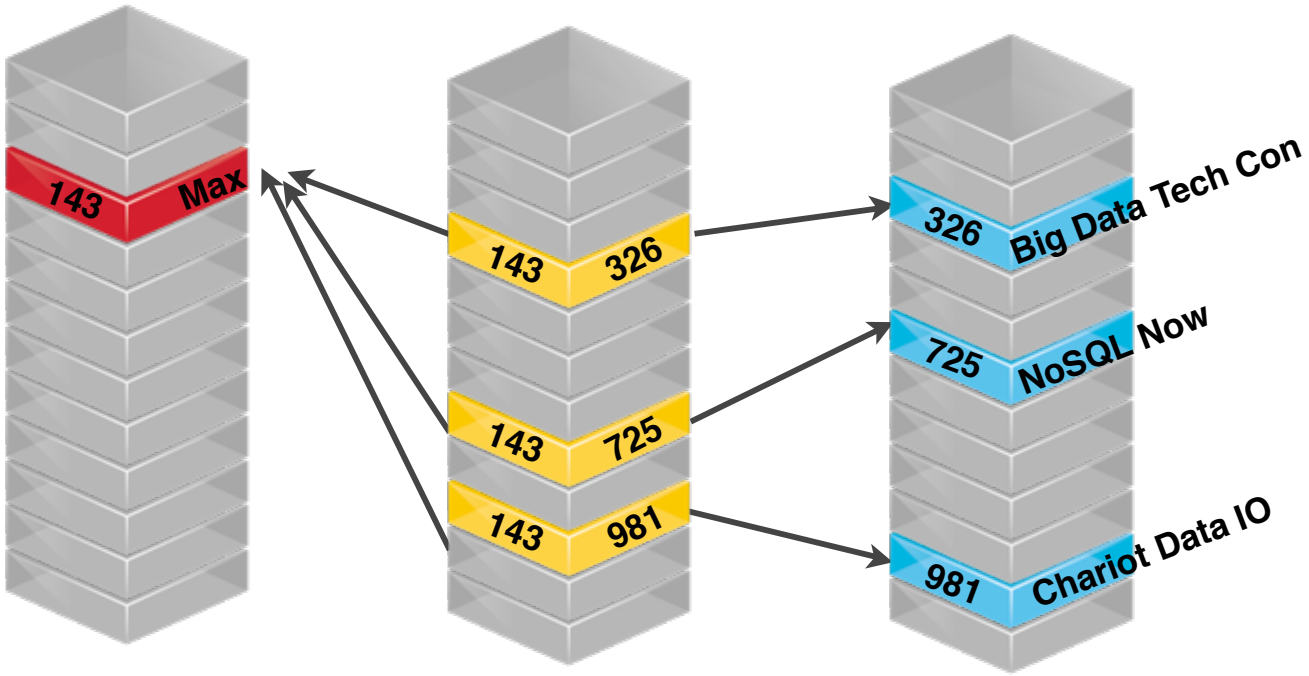
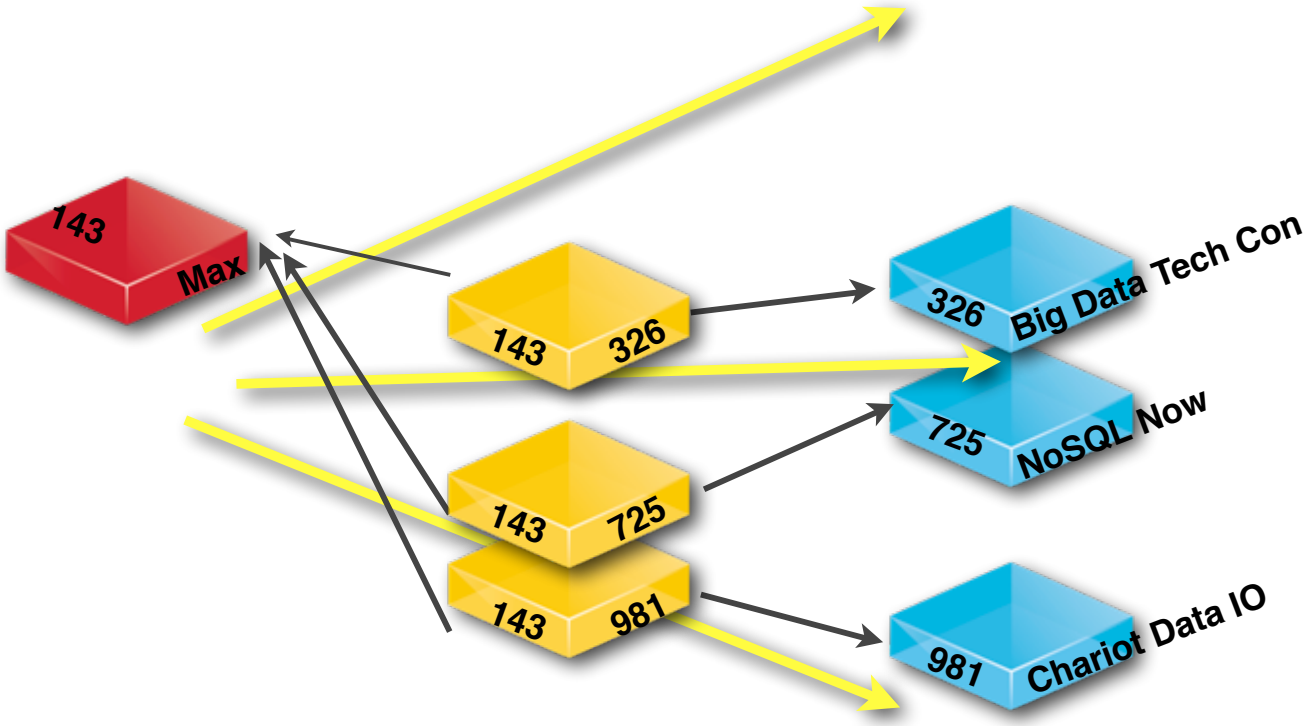


Image source: http://readwrite.com/2011/04/20/5-graph-databases-to-consider

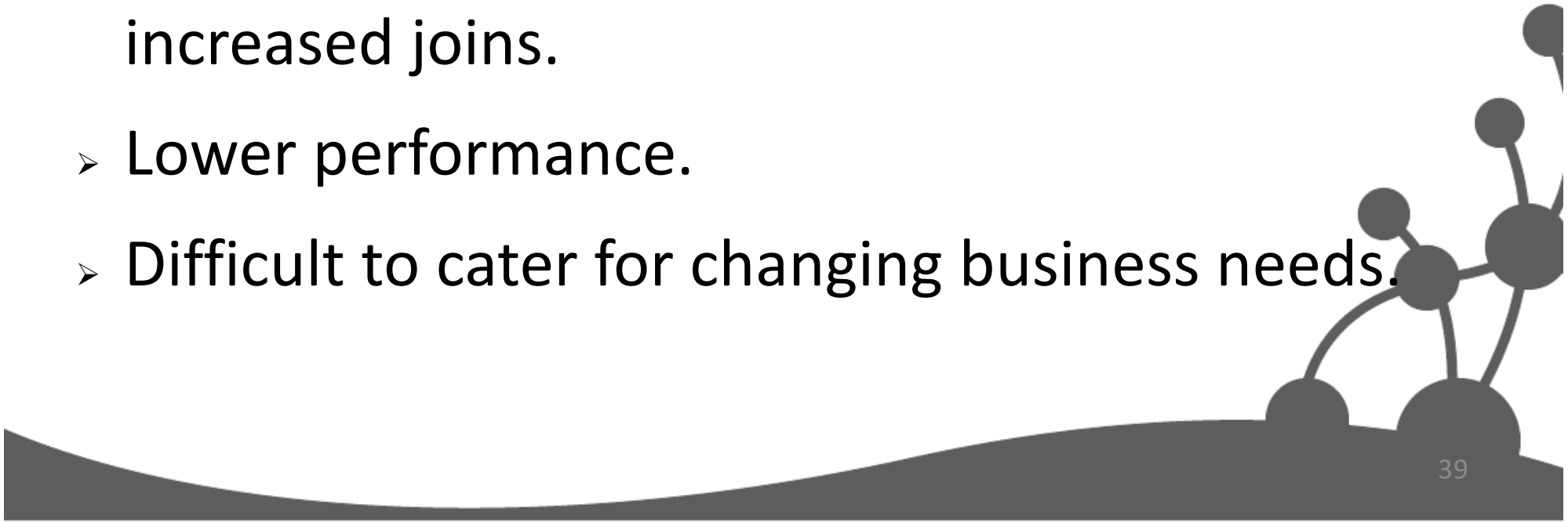


People Attend Conferences


143
Max
143
326
326
Big Data Tech Con
725
NoSQL Now
143
725
143
981
981
Chariot Data IO

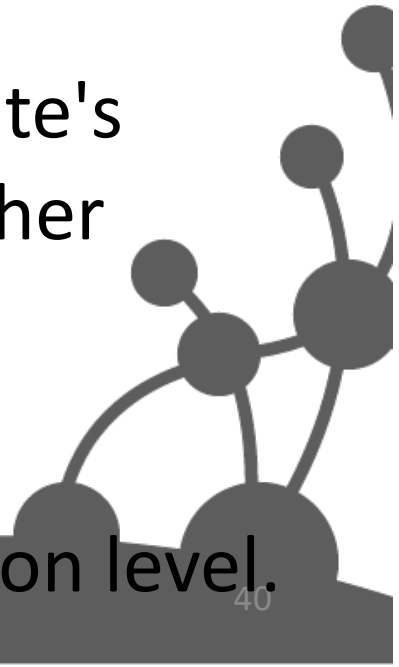
Relational Databases Lack Relationships

- Initially designed to codify paper forms and tabular structures.
- Deal poorly with relationships.
- The rise in connectedness translates into increased joins.
- Lower performance.
- Difficult to cater for changing business needs.



NoSQL Databases also lack Relationships

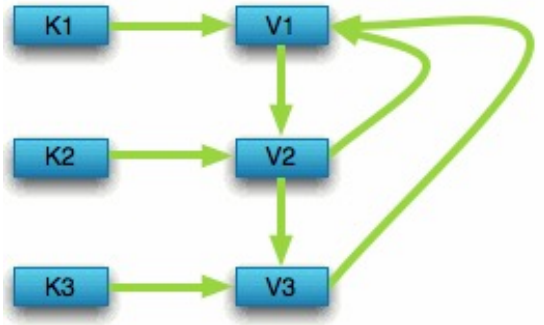
- NOSQL Databases e.g key-value, document or column oriented store sets of disconnected values/documents/columns.
- Makes it difficult to use them for connected data and graphs.
- One of the solution is to embed an aggregate's identifier inside the field belonging to another aggregate.
  - Effectively introducing foreign keys
- Requires joining aggregates at the application level.



Compared to Key Value Stores

Optimized for simple look-ups

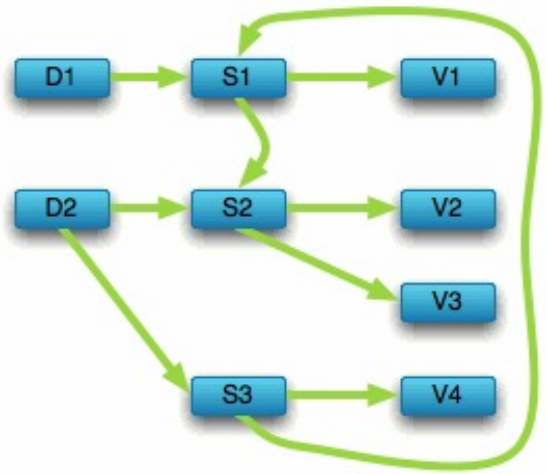
Optimized for traversing connected data



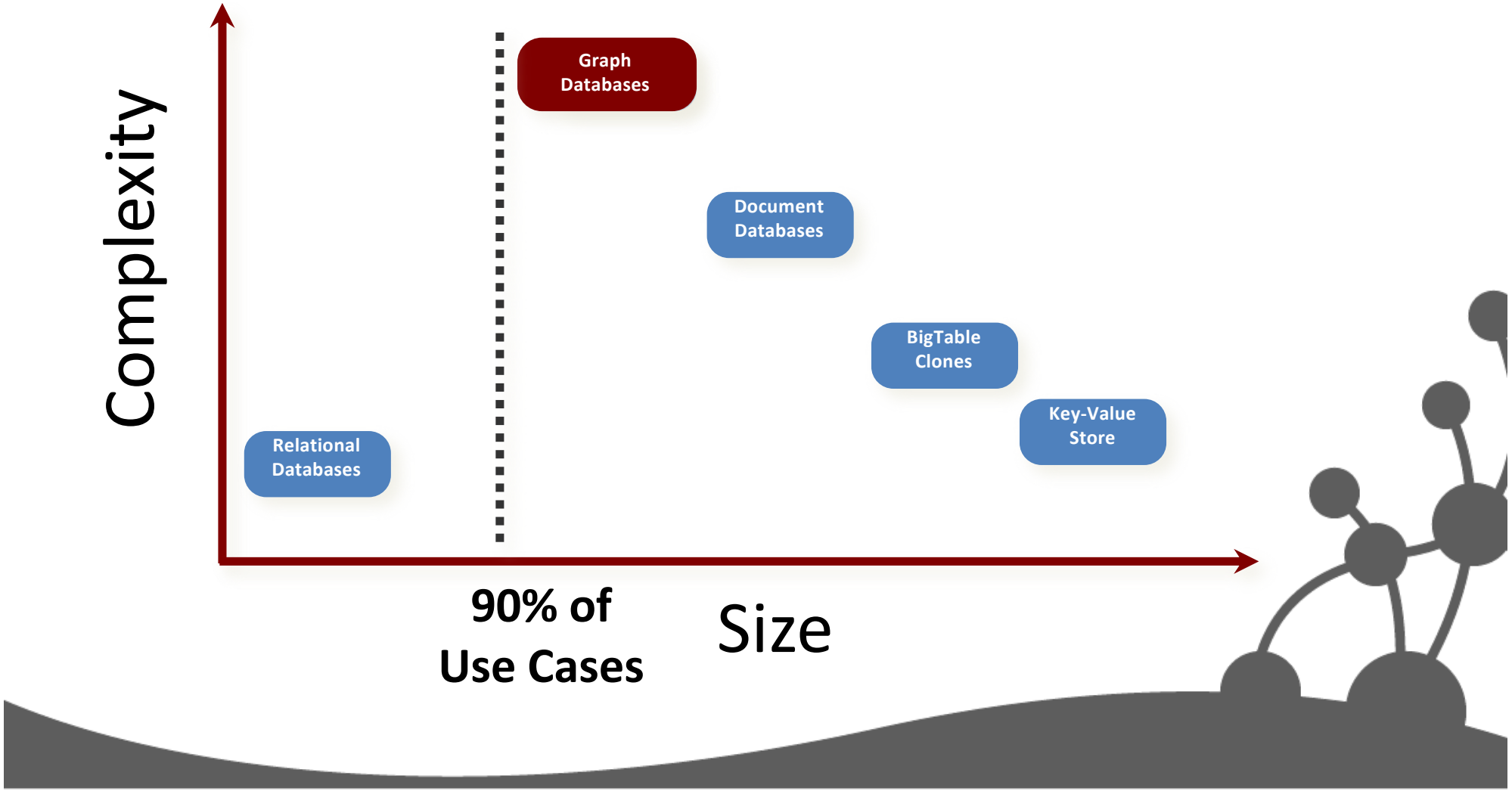
Compared to document Stores

Optimized for “trees” of data

Optimized for seeing the forest and the trees, and the branches, and the trunks



Living in a NOSQL World



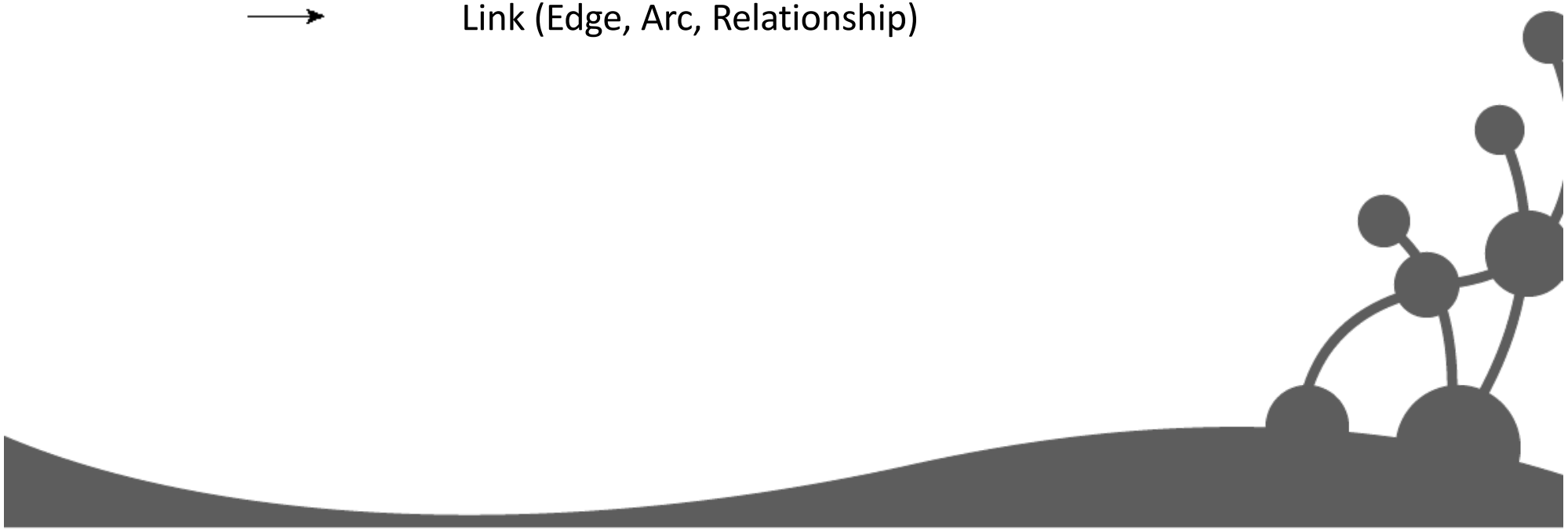
What is a Graph?

What is a Graph?

- An abstract representation of a set of objects where some pairs are connected by links.

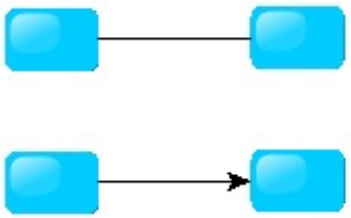
Object (Vertex, Node)

Link (Edge, Arc, Relationship)

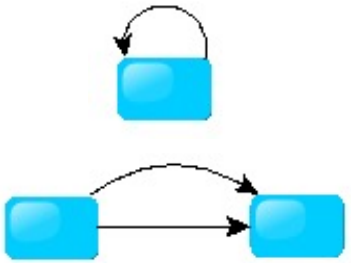


Different Kinds of Graphs

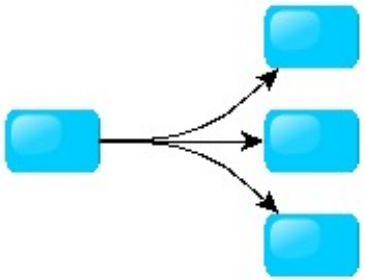
- Undirected Graph
- Directed Graph



- Pseudo Graph
- Multi Graph

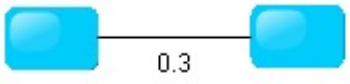


- Hyper Graph

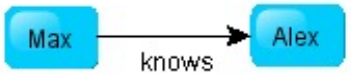


More Kinds of Graphs

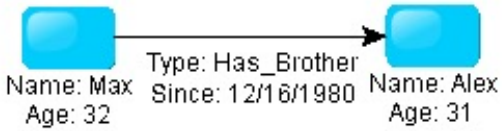
- Weighted Graph



- Labeled Graph

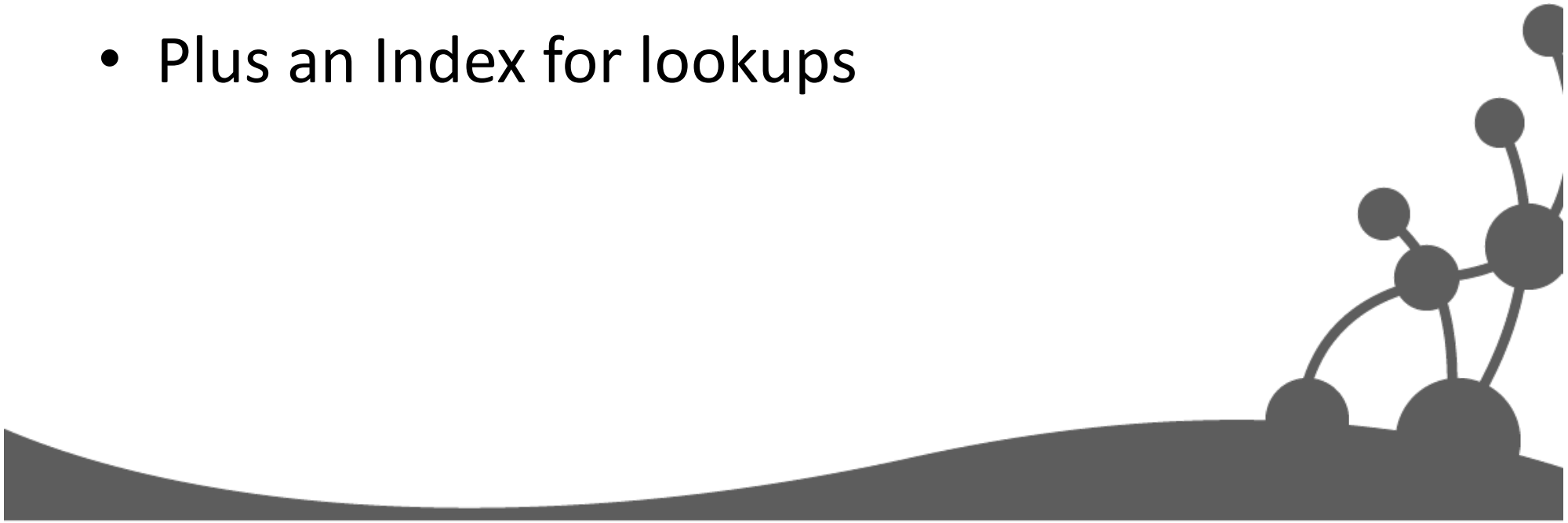


- Property Graph

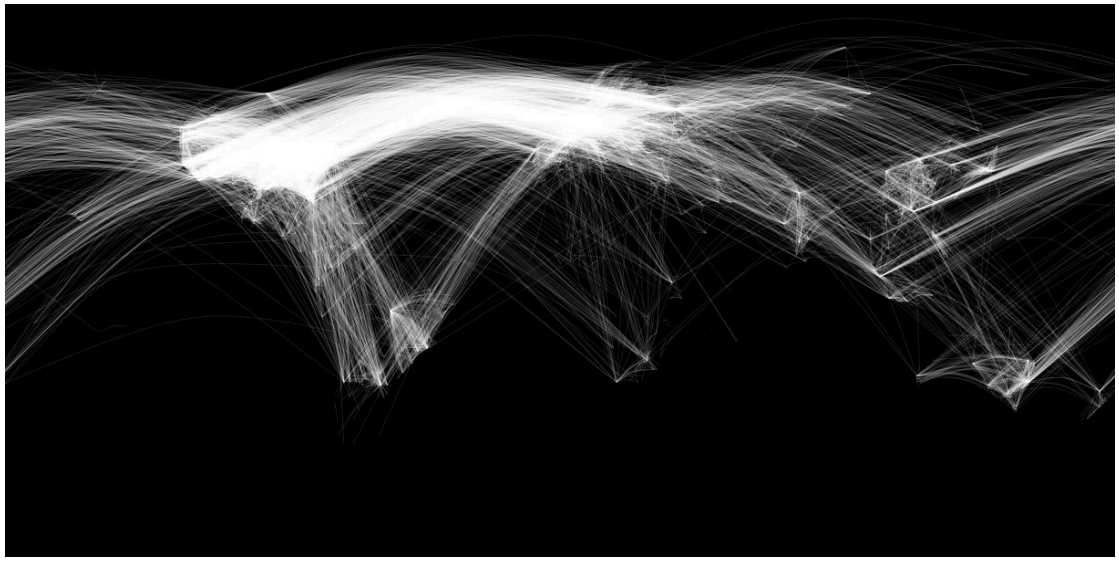


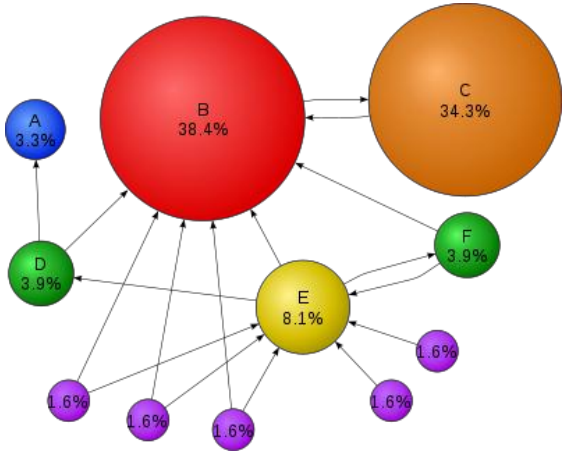
What is a Graph Database?

- A database with an explicit graph structure
- Each node knows its adjacent nodes
- As the number of nodes increases, the cost of a local step (or hop) remains the same
- Plus an Index for lookups



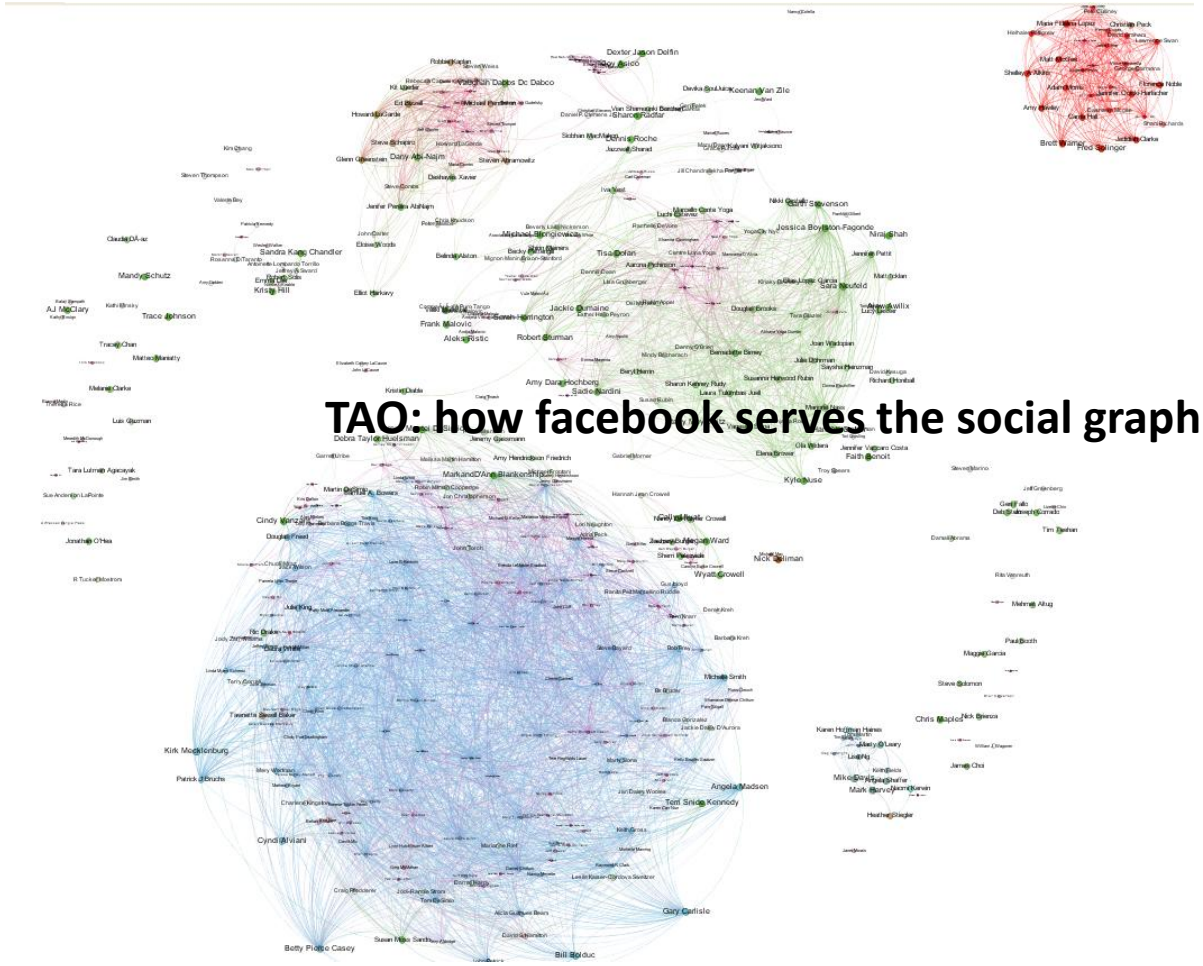
Example: Internet





Google™
PageRank

Example: Social Network



TAO: how facebook serves the social graph

facebook

LinkedIn

twitter

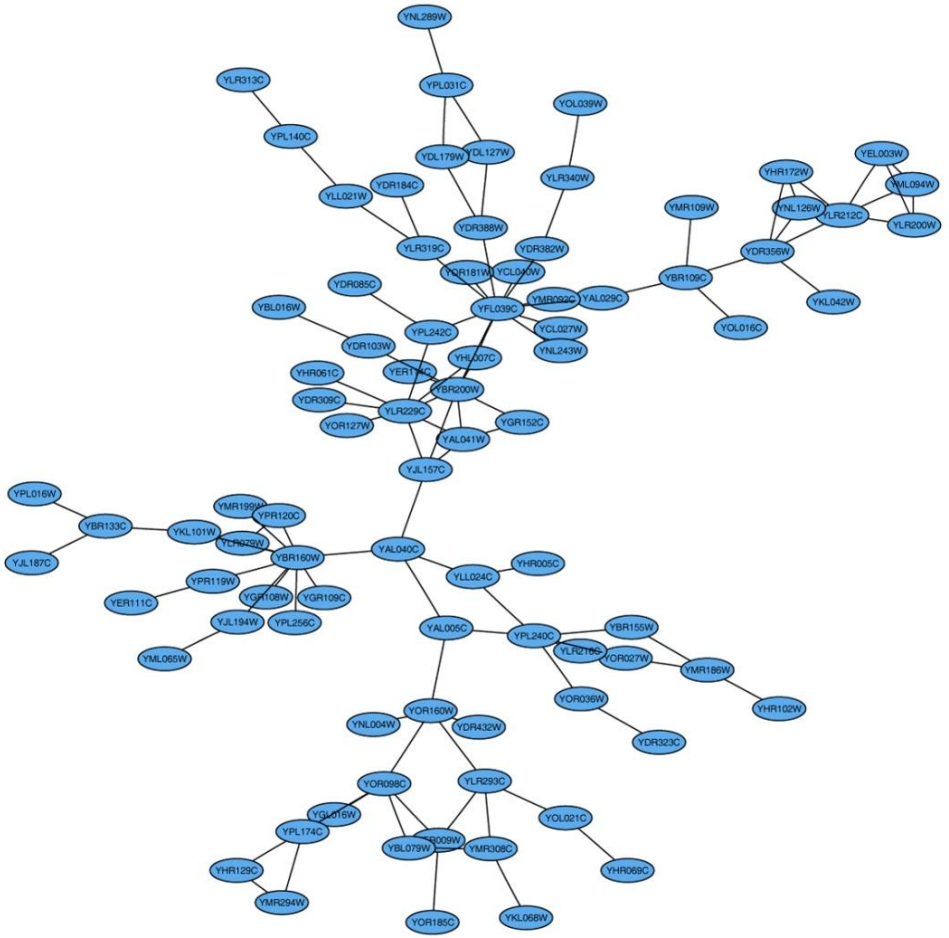
Recommendation Systems

amazon

Buy.com

YouTube

Computational Biology



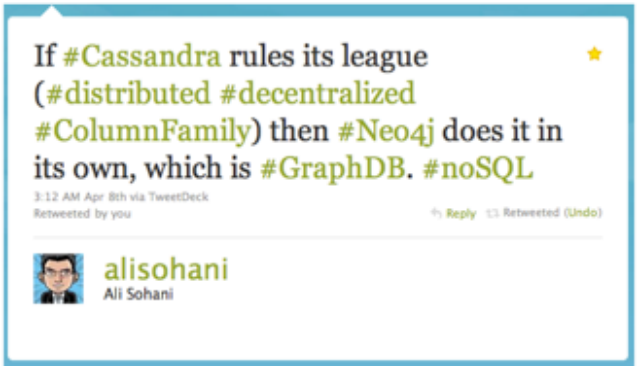
Protein Interaction

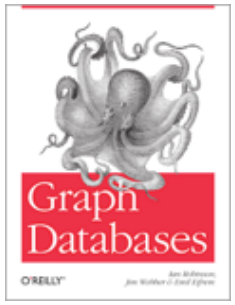
What is Neo4j?

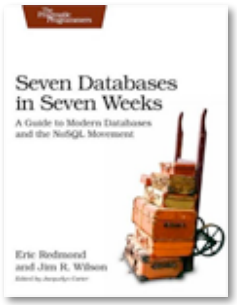
The Neo4j Graph Database

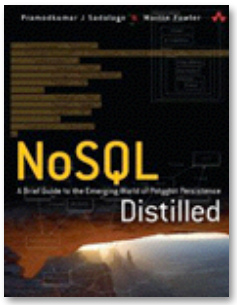
neo technology
graphs are everywhere

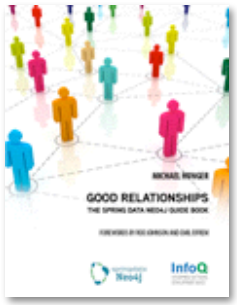
- Neo4j is the **leading graph database** in the world today
  - Most widely deployed: **500,000+ downloads**
  - **Largest ecosystem**: active forums, code contributions, etc
  - Most mature product: in development since 2000, **in 24/7 production since 2003**

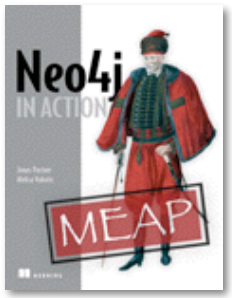












Neo4j

The World's Leading Graph Database

Neo4j is an open-source, high-performance, enterprise-grade NOSQL graph database.

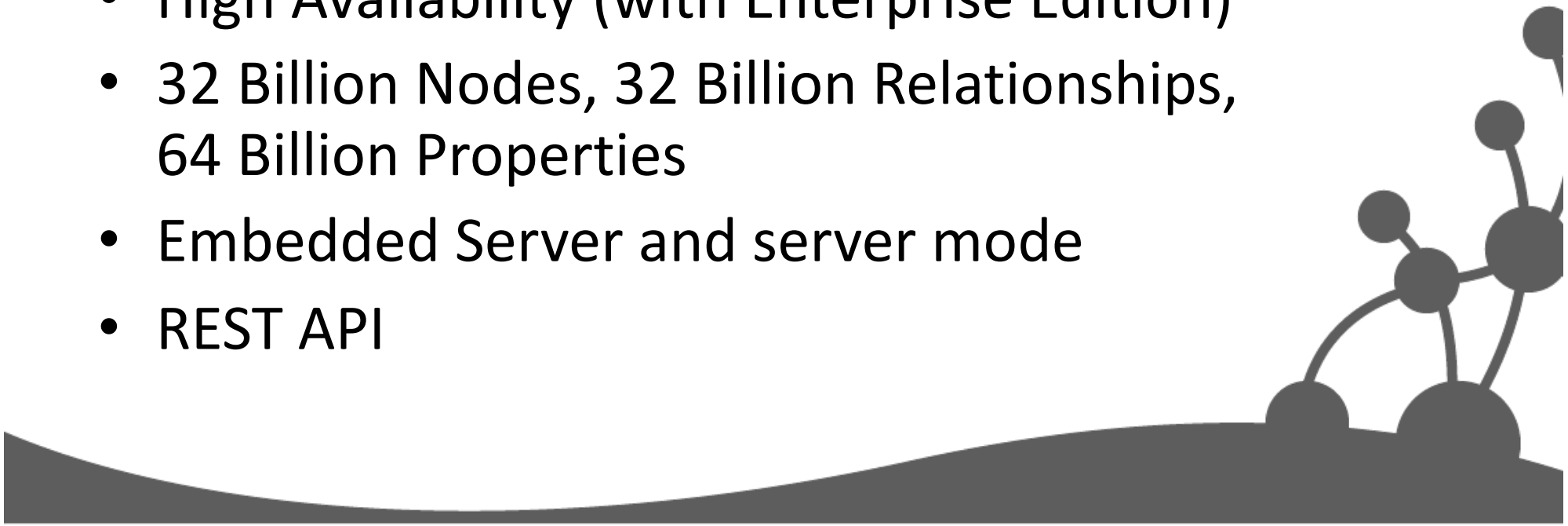
ACID

Java

http://www.neotechnology.com/customers/

What is Neo4j?

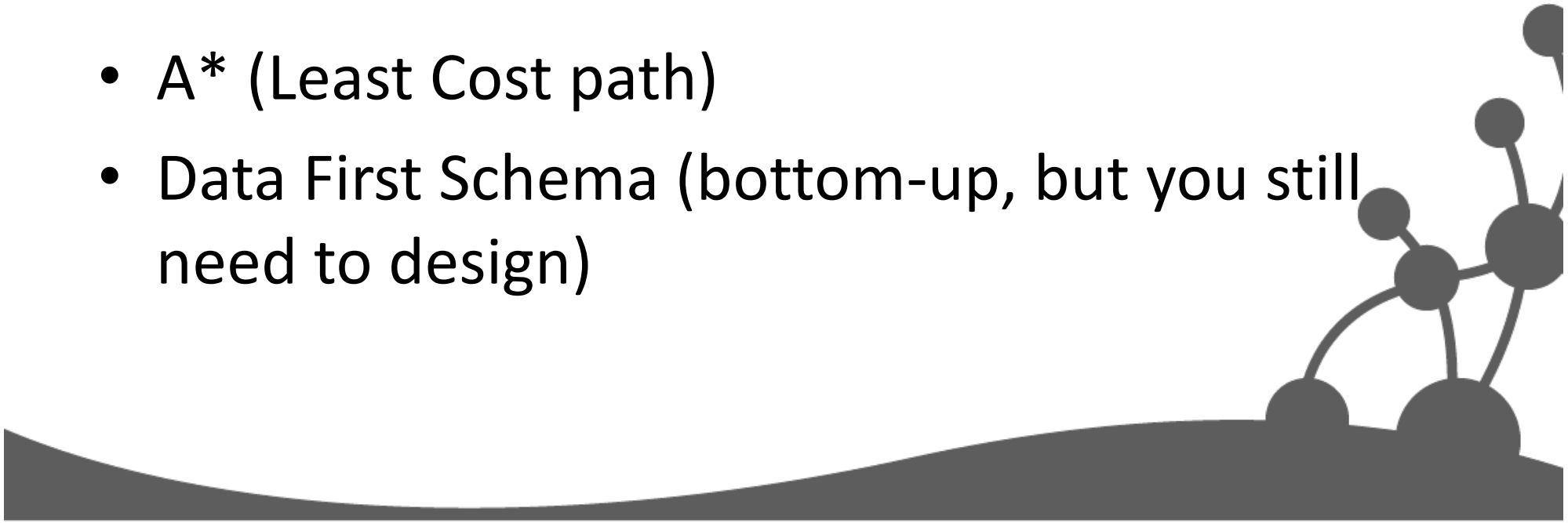
- A Graph Database + Lucene Index
- Property Graph
- Full ACID (atomicity, consistency, isolation, durability)
- High Availability (with Enterprise Edition)
- 32 Billion Nodes, 32 Billion Relationships, 64 Billion Properties
- Embedded Server and server mode
- REST API



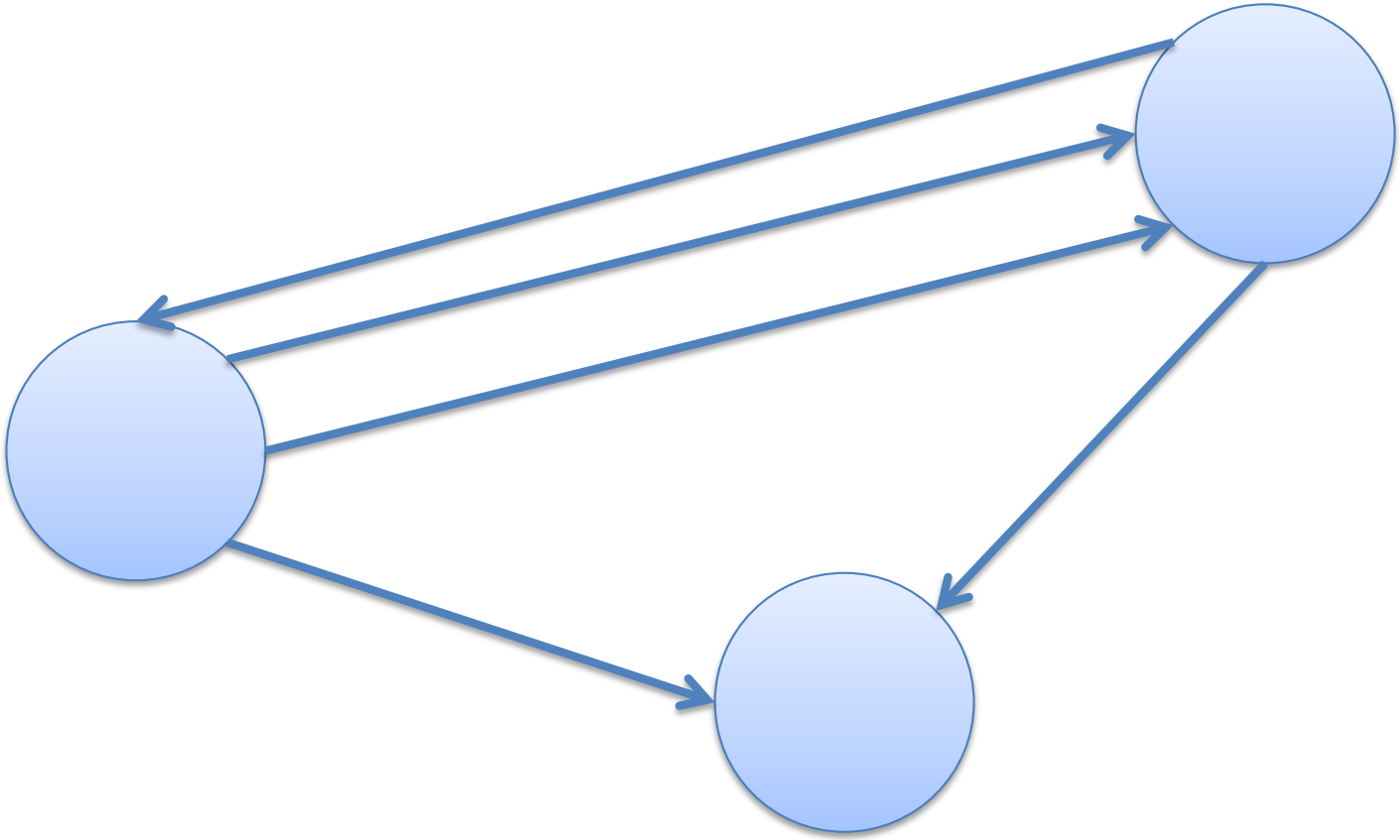
Good For

- Highly connected data (social networks)
- Recommendations (e-commerce)
- Path Finding (how do I know you?)

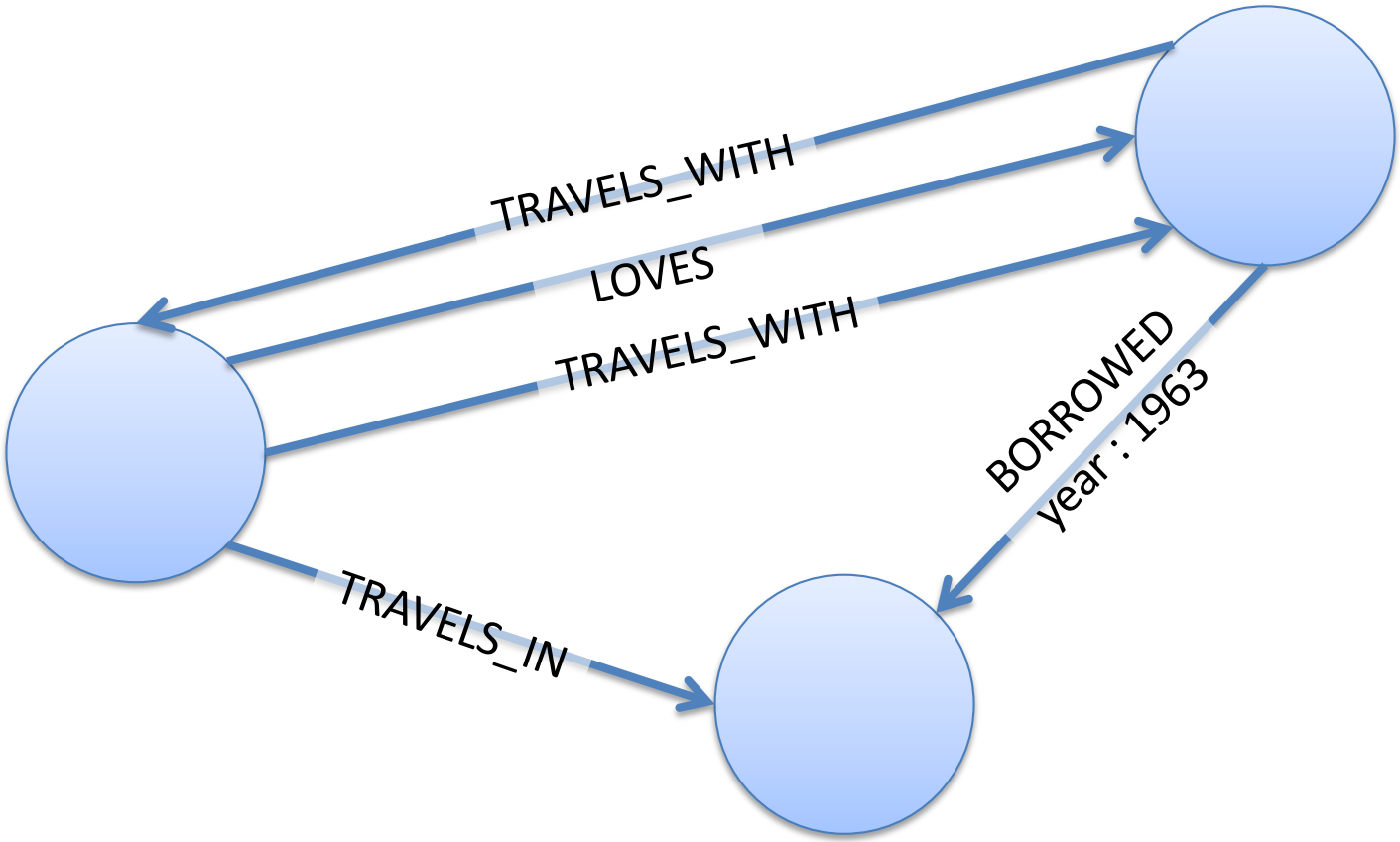
- A* (Least Cost path)
- Data First Schema (bottom-up, but you still need to design)



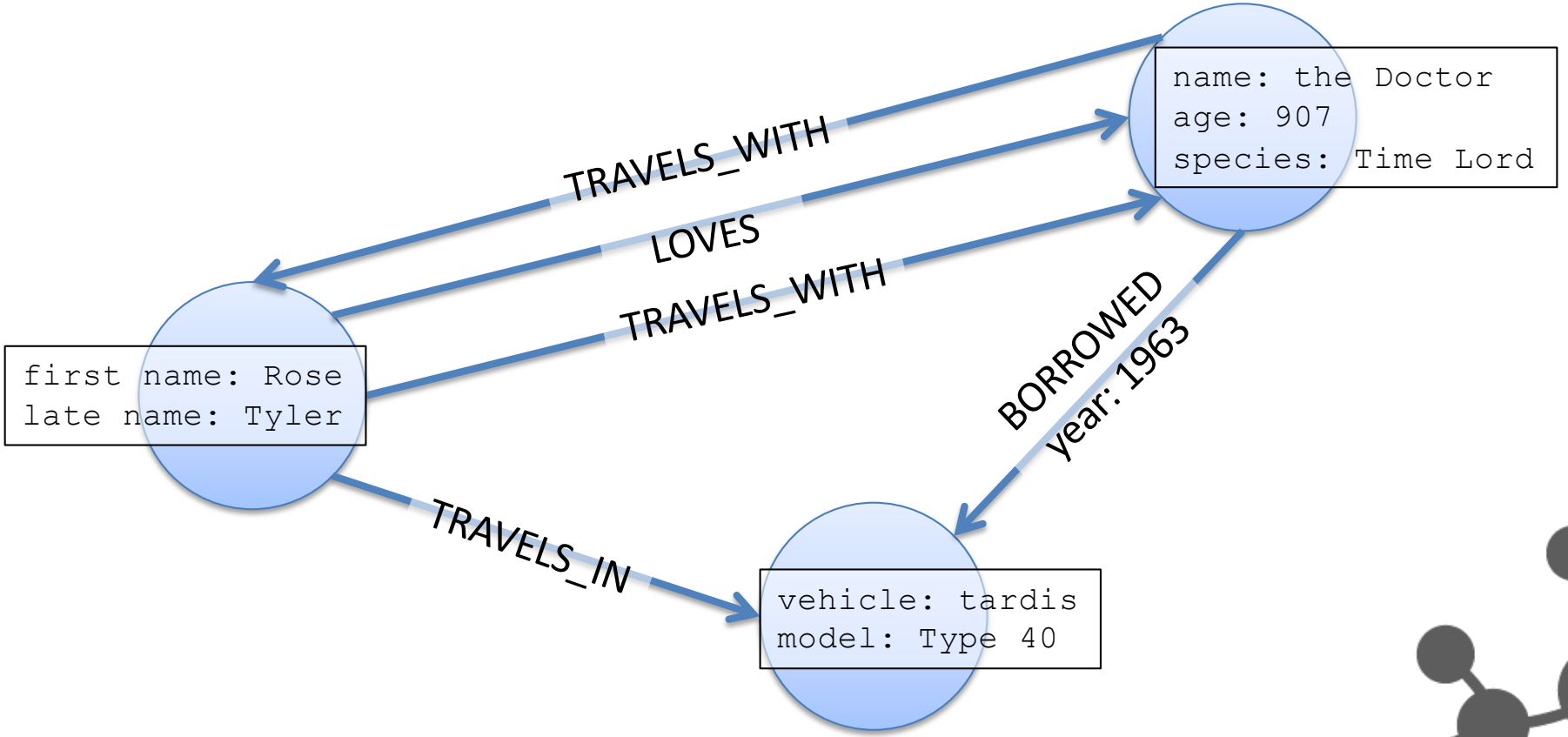
Property Graph Model



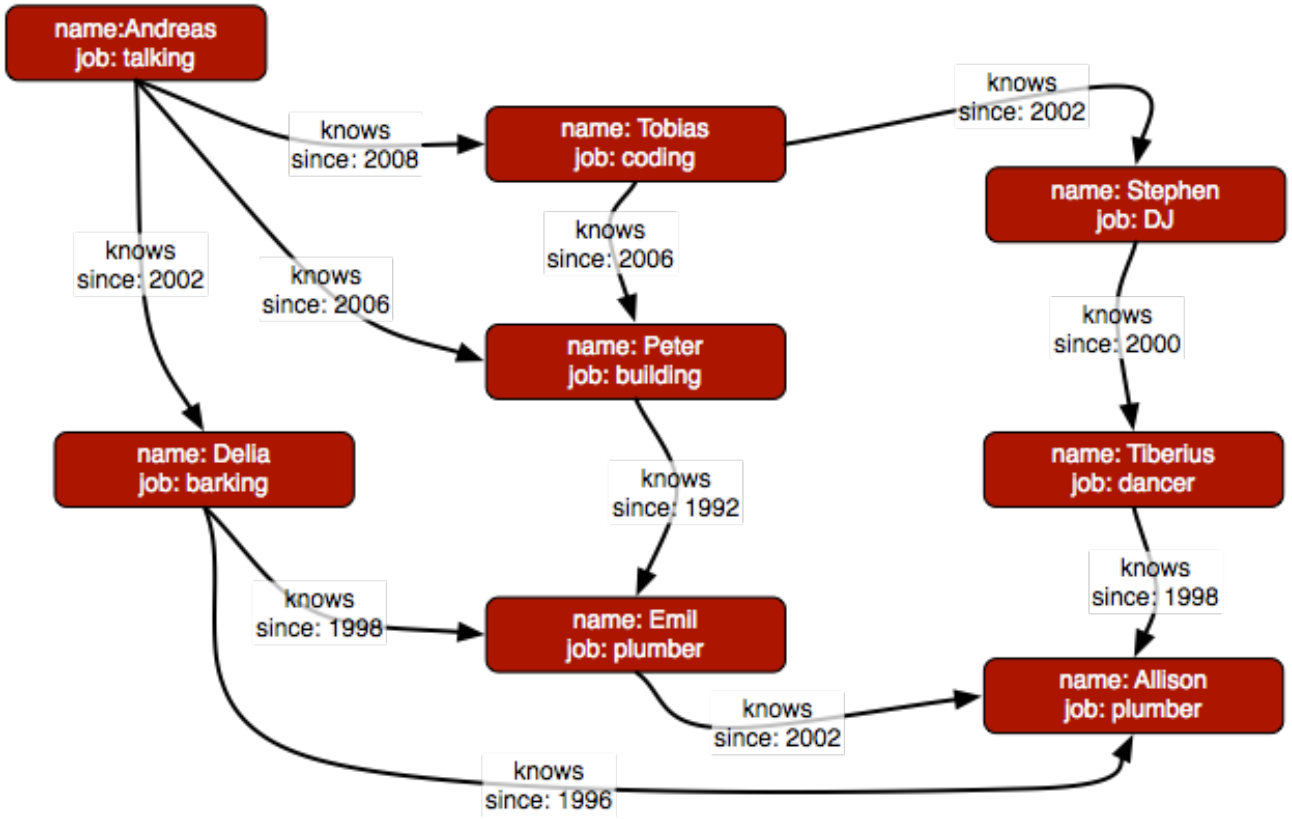
Property Graph Model



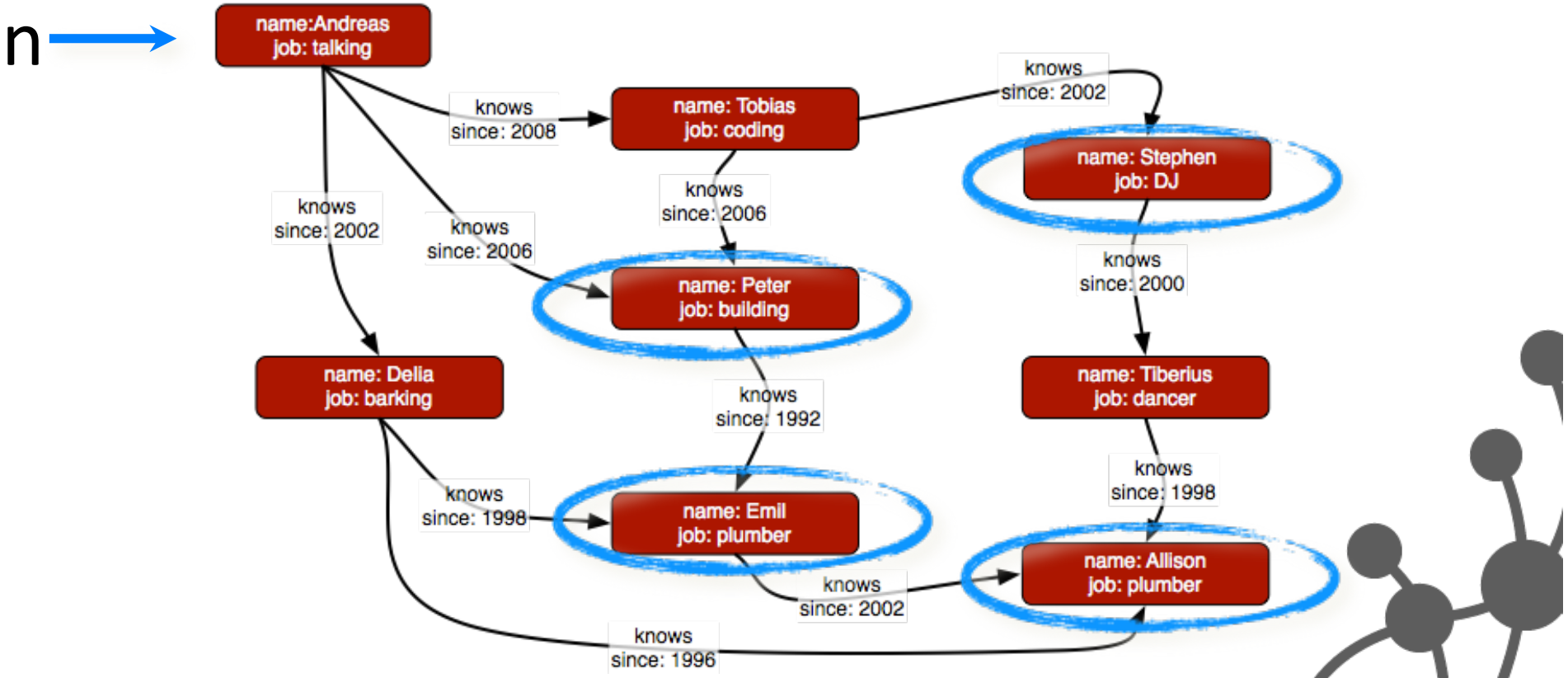
Property Graph Model



Property Graph



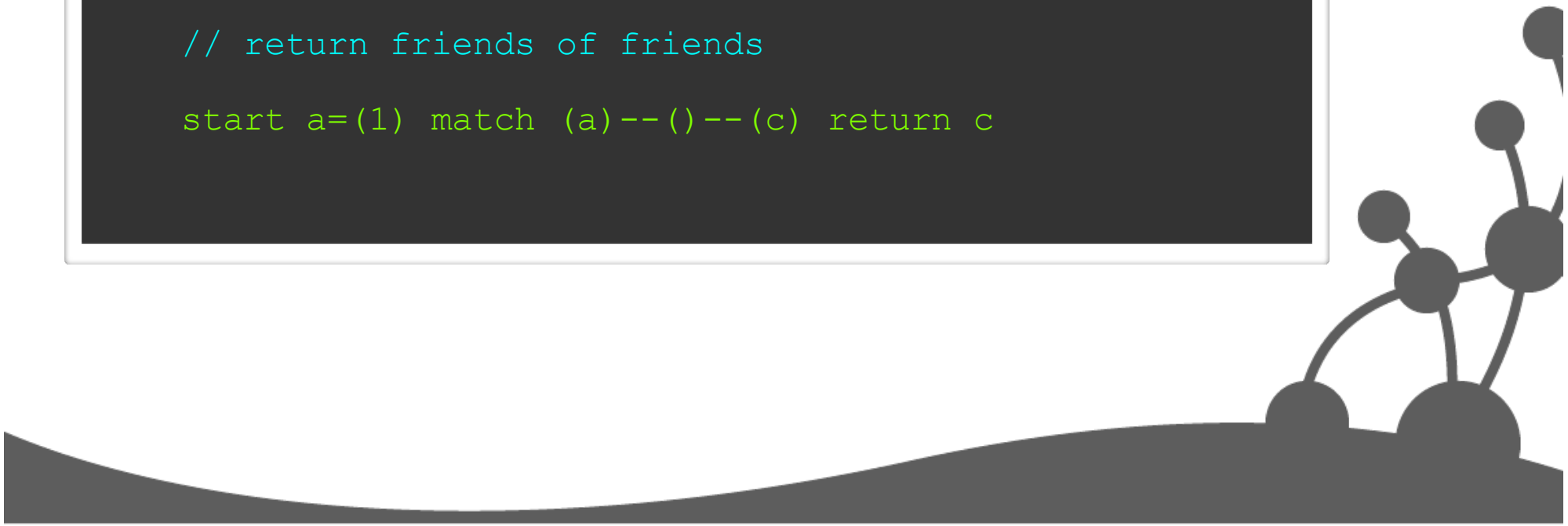
```
// then traverse to find results
start n=(people-index, name, "Andreas")
match (n)--()--(foaf) return foaf
```



Cypher

Pattern Matching Query Language (like SQL for graphs)

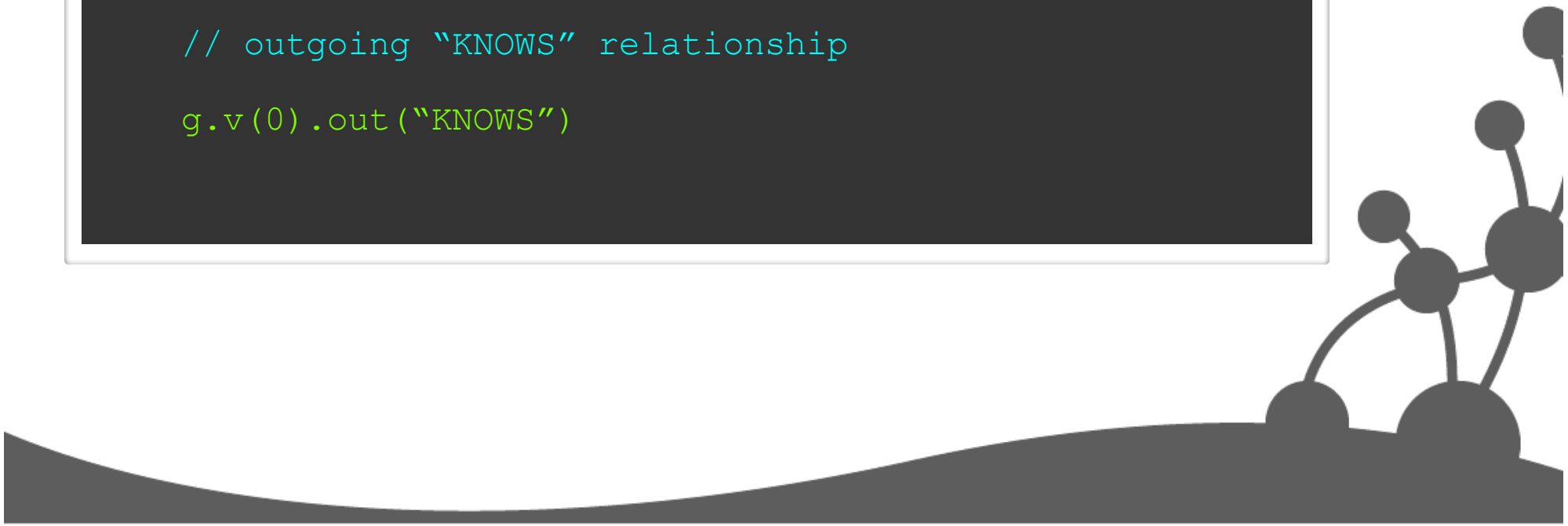
```
// get node 0
start a=(0) return a

// traverse from node 1
start a=(1) match (a)-->(b) return b

// return friends of friends
start a=(1) match (a)--()--(c) return c
```



Gremlin

A Graph Scripting DSL (groovy-based)

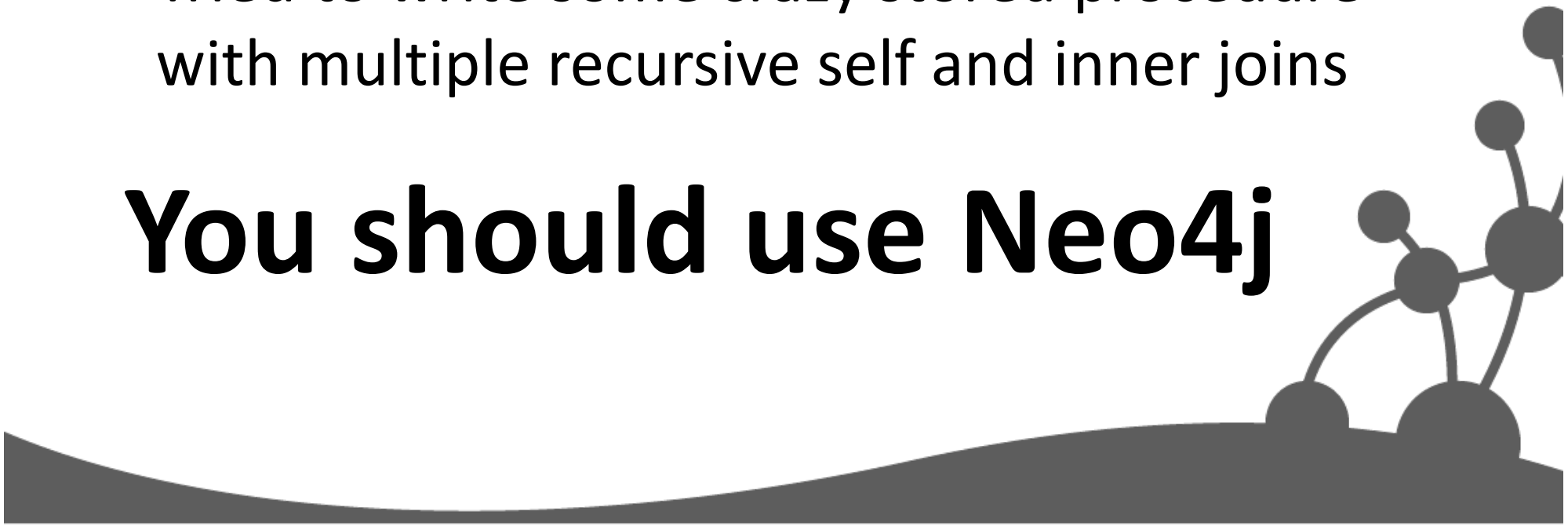
```
// get node 0

g.v(0)

// nodes with incoming relationship

g.v(0).in

// outgoing "KNOWS" relationship

g.v(0).out("KNOWS")
```



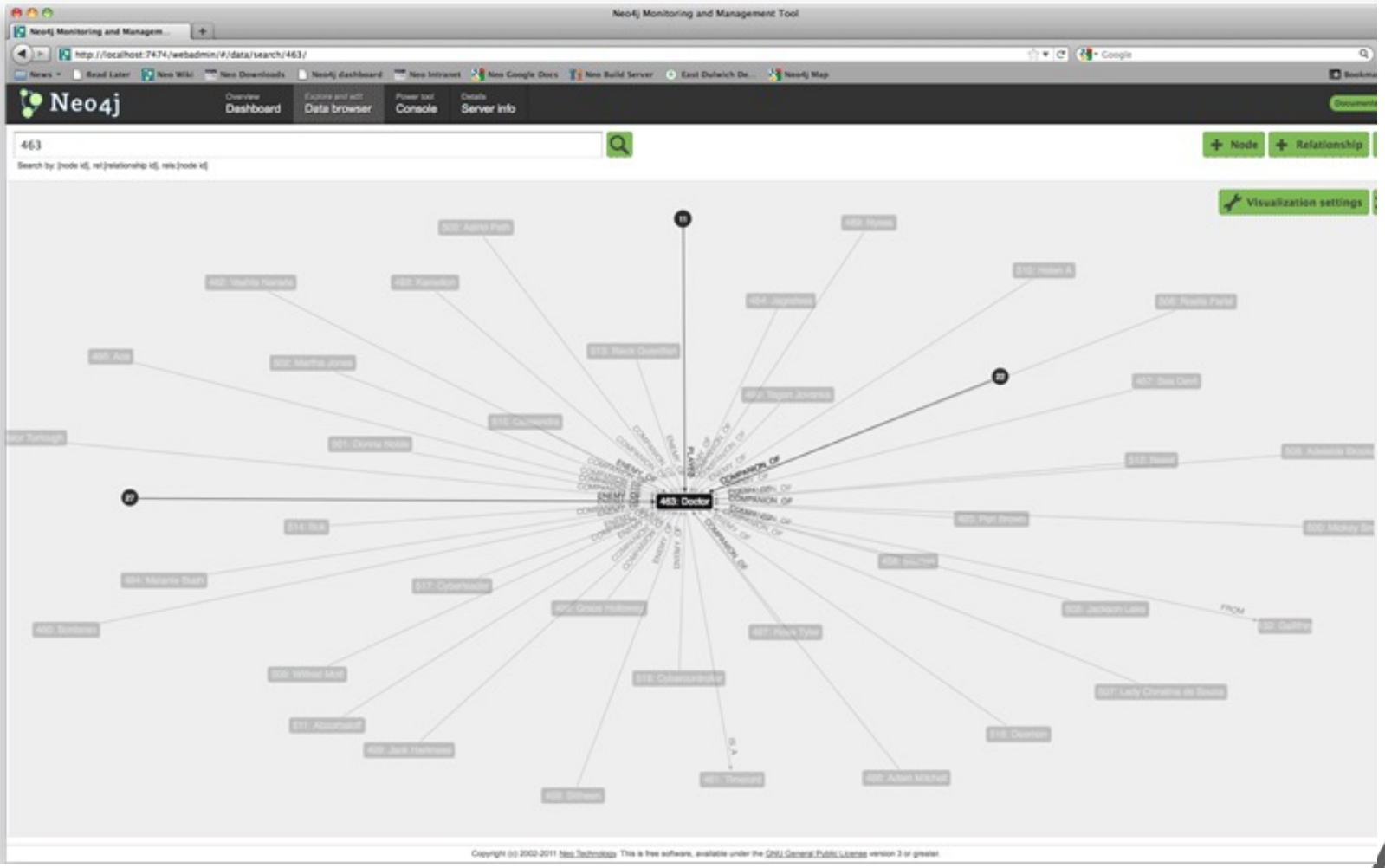
If you've ever

- Joined more than 7 tables together
- Modeled a graph in a table
- Written a recursive query
- Tried to write some crazy stored procedure with multiple recursive self and inner joins

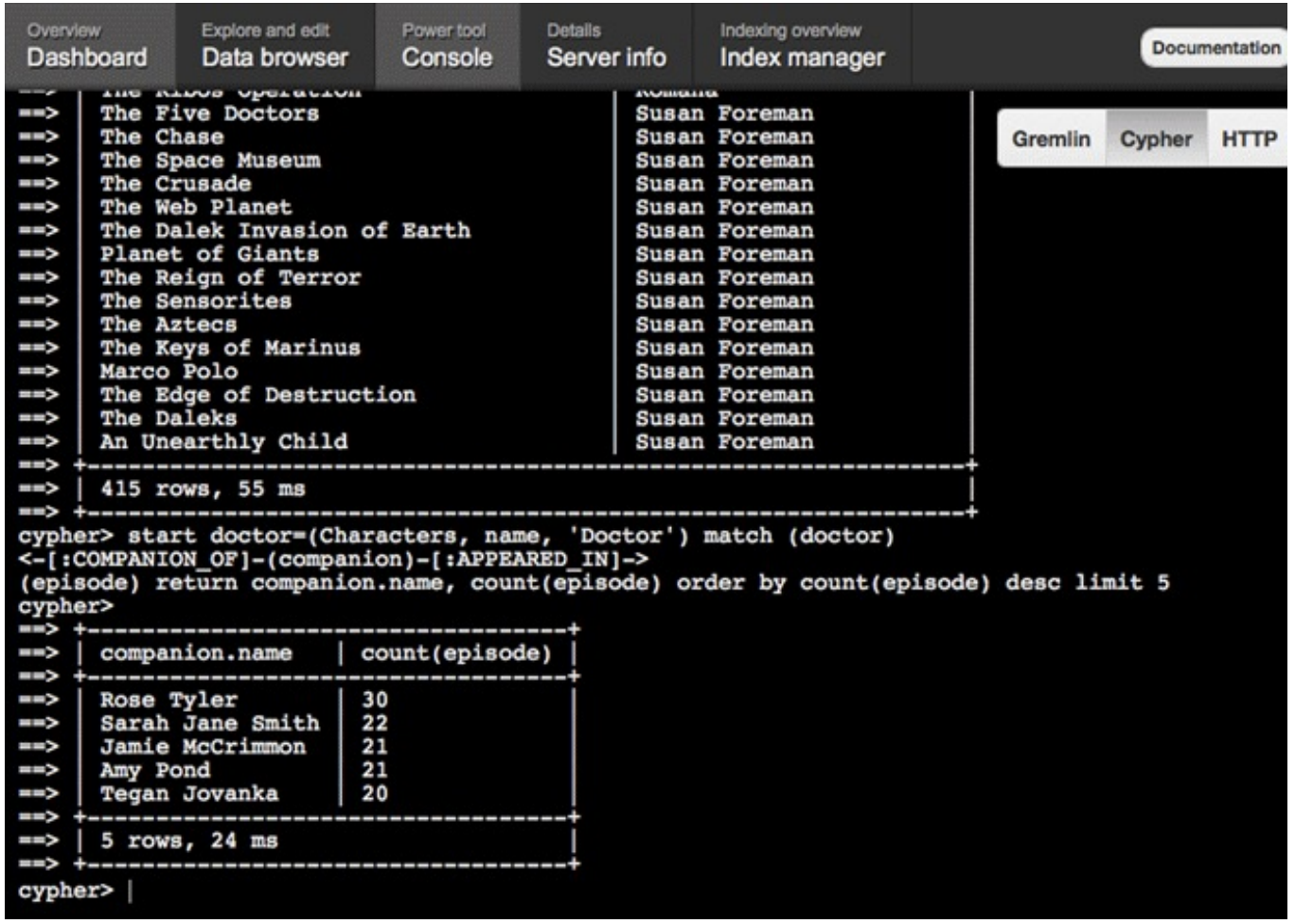
You should use Neo4j



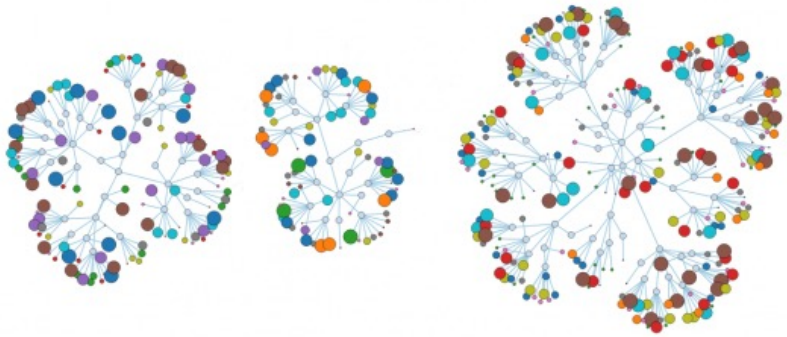
Neo4j Data Browser

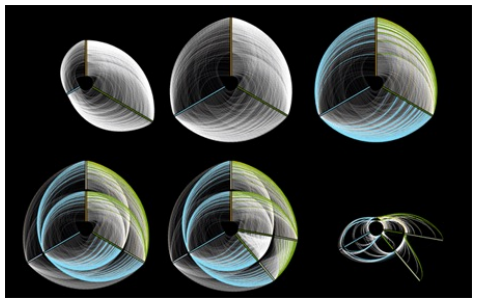


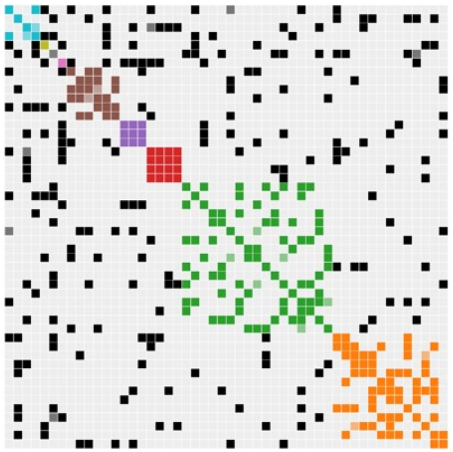
Neo4j Console

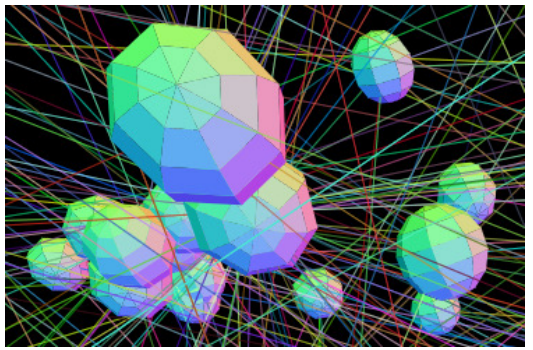


What does a Graph look like?









Neo4J 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!

Social Network “path exists” Performance

- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 5
  - Caches warm to eliminate disk IO

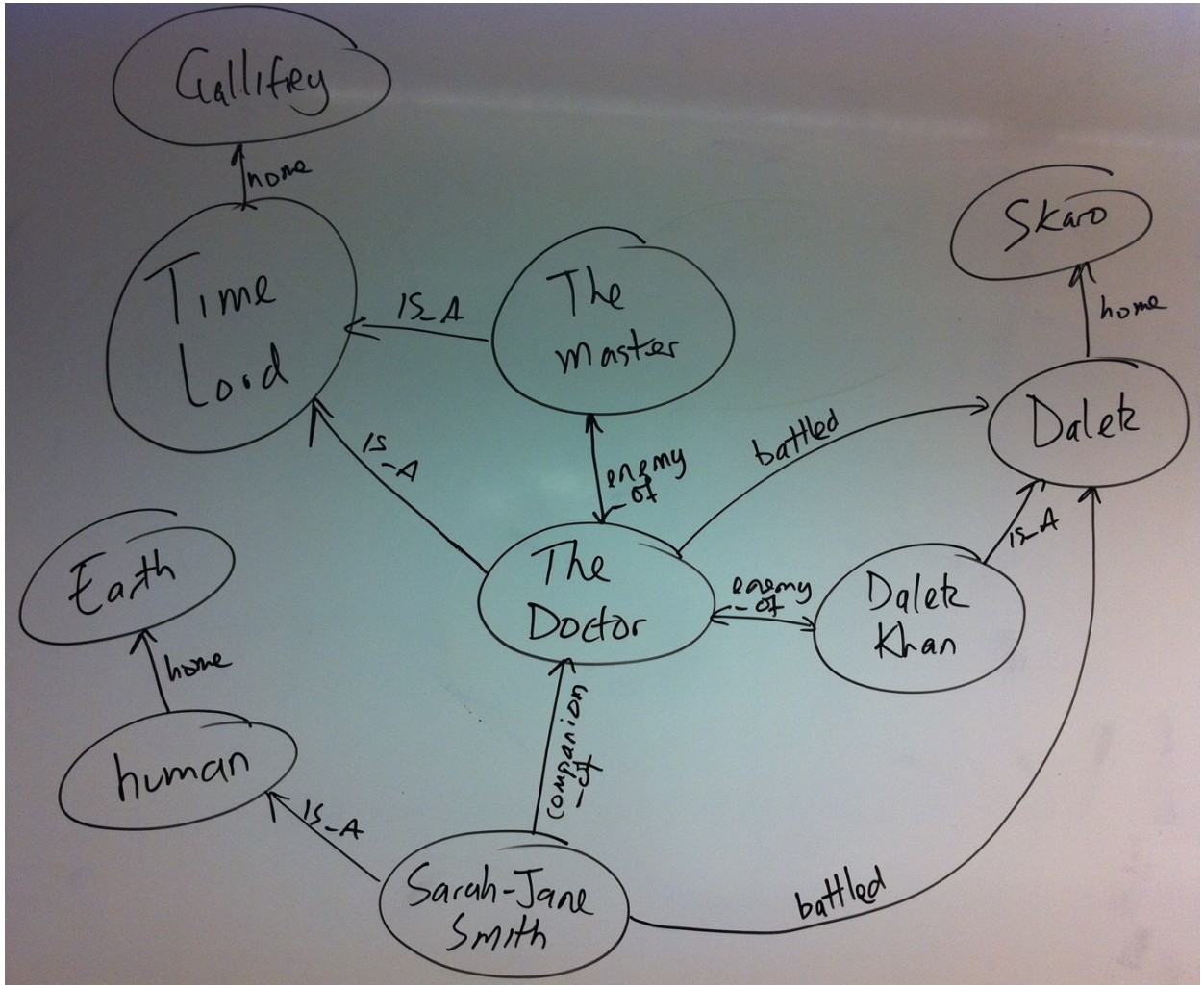
Depth	# persons	RDBMS execution time (s)	Neo4j execution time (s)
2	2500	0.016	0.01
3	110000	30.2	0.168
4	600000	1543	1.35
5	800000	unfinished	2.13

From « Neo4j in action », Partner and Vukotic

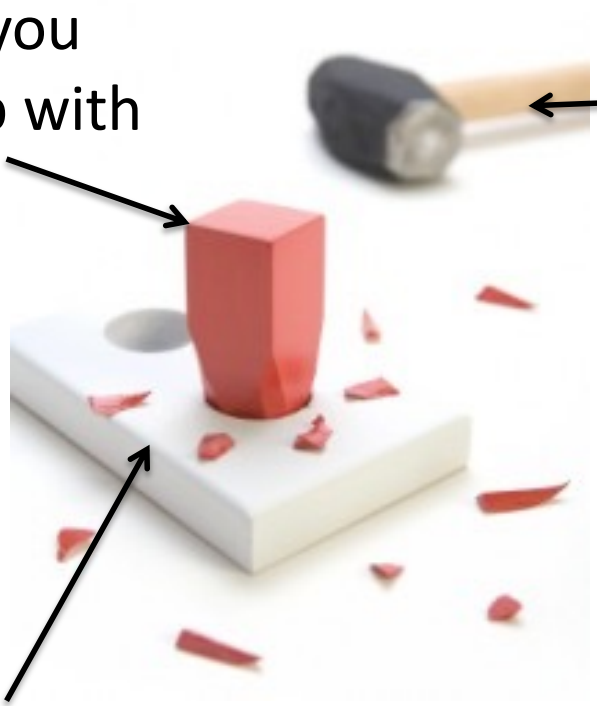
Schema-less Databases

- Graph databases don’t 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!

Graphs are *very* whiteboard-friendly



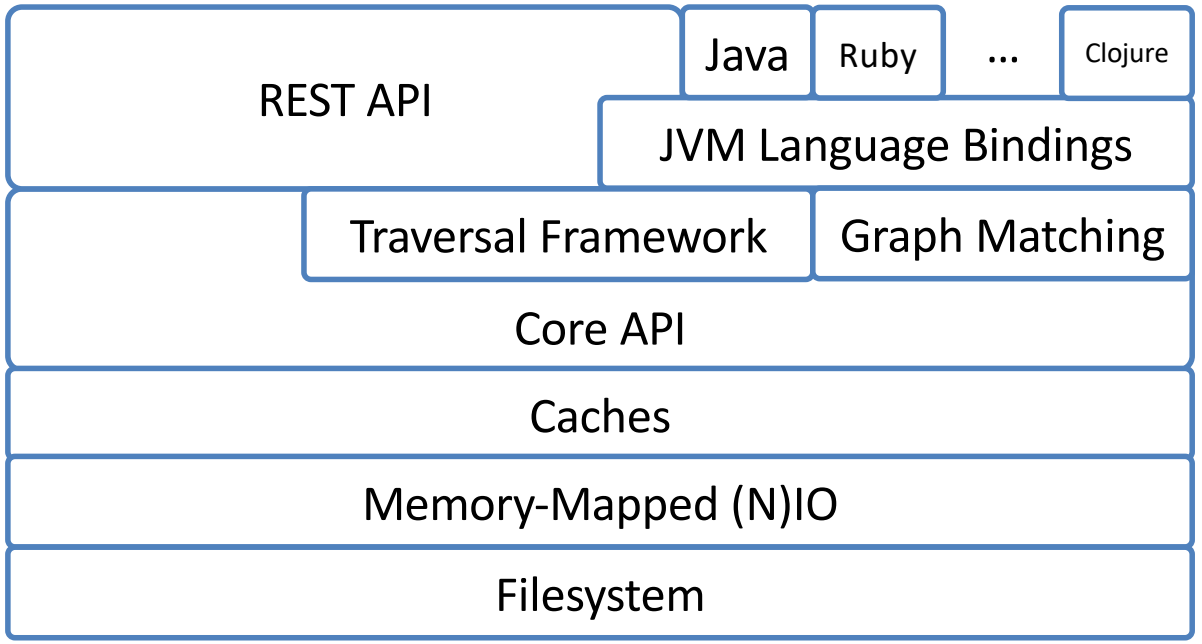
Design errors



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?

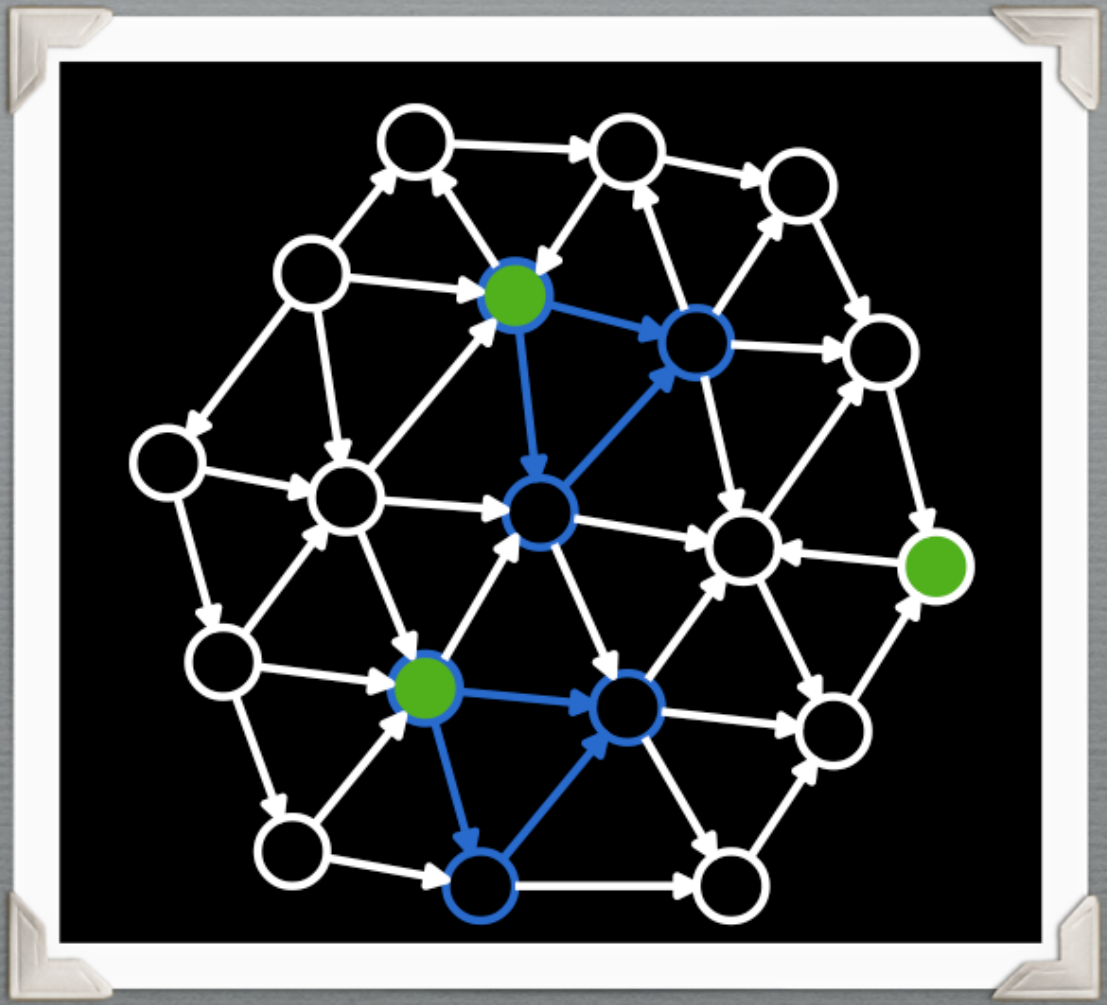
Data access is *programmatic*

- 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

Cypher

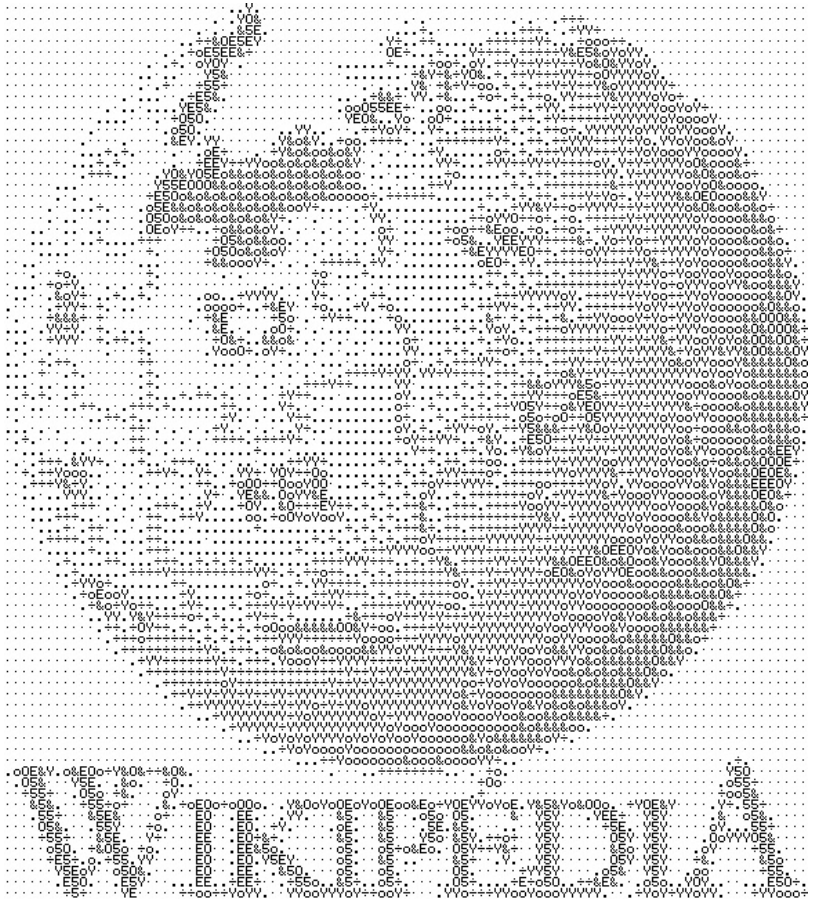
What is Cypher?

- 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



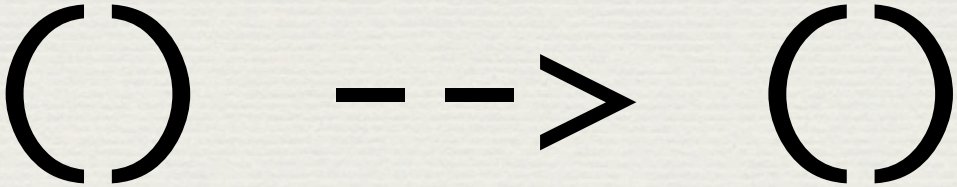
Matching Patterns

ASCII art



ASCII Art

ASCII Art

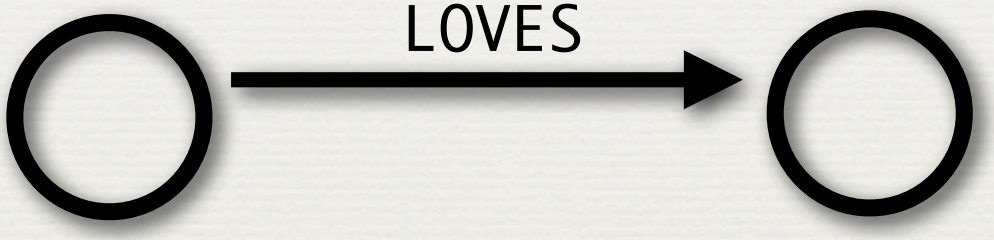


Named Nodes

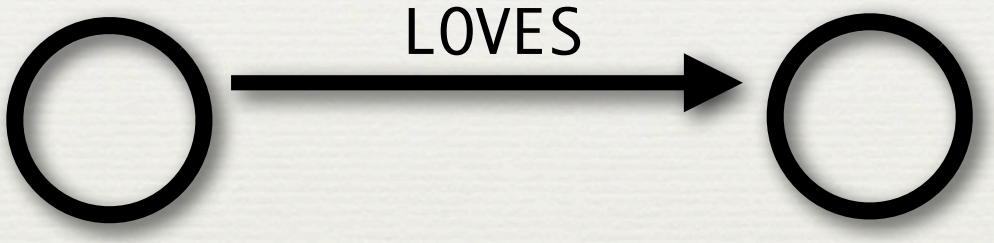
Named Nodes

```
(A) --> (B)
```

Typed Relationships



Typed Relationships



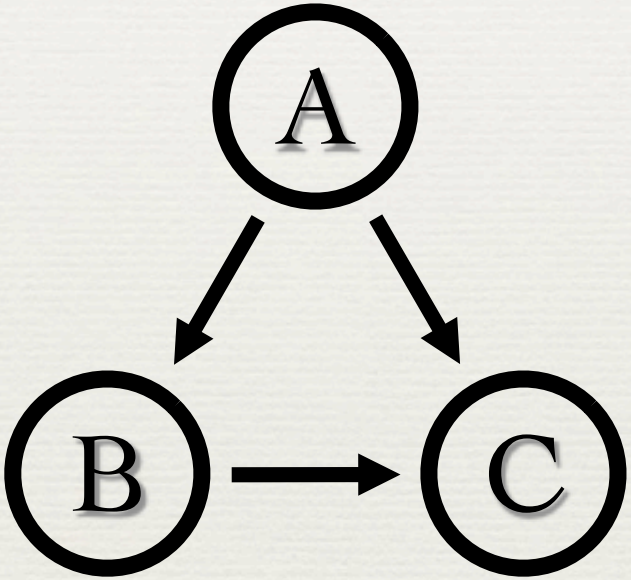

```
A -[:LOVES]-> B
```

Describing a Path

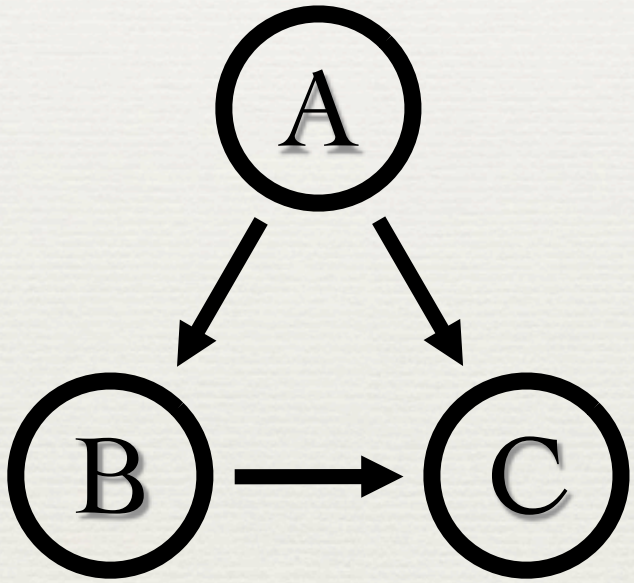
Describing a Path

A --> B --> C

Multiple Paths

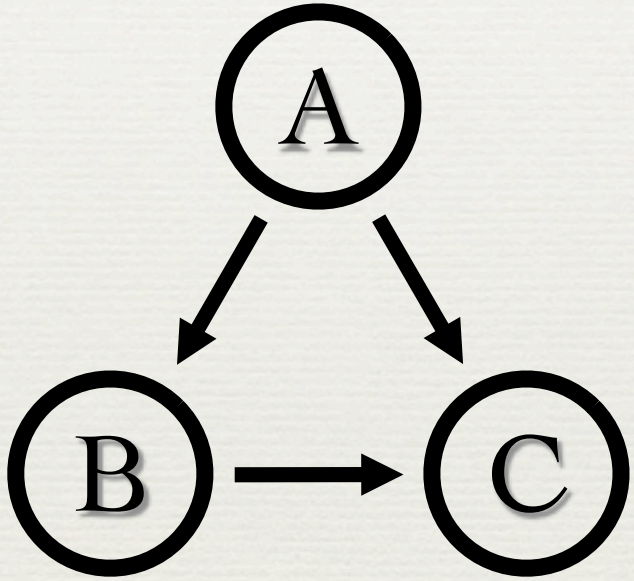


Multiple Paths




```
A --> B --> C, A --> C
```

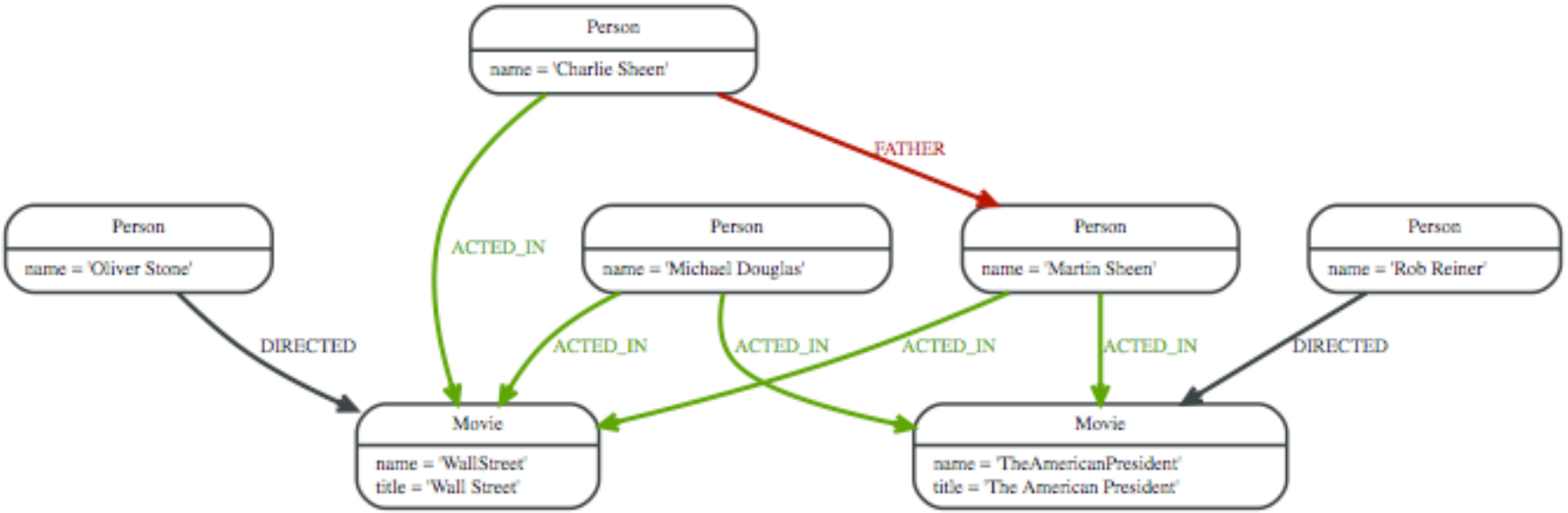
Multiple Paths




```
A --> B --> C, A --> C
```

```
A --> B --> C <-- A
```

Graph Example



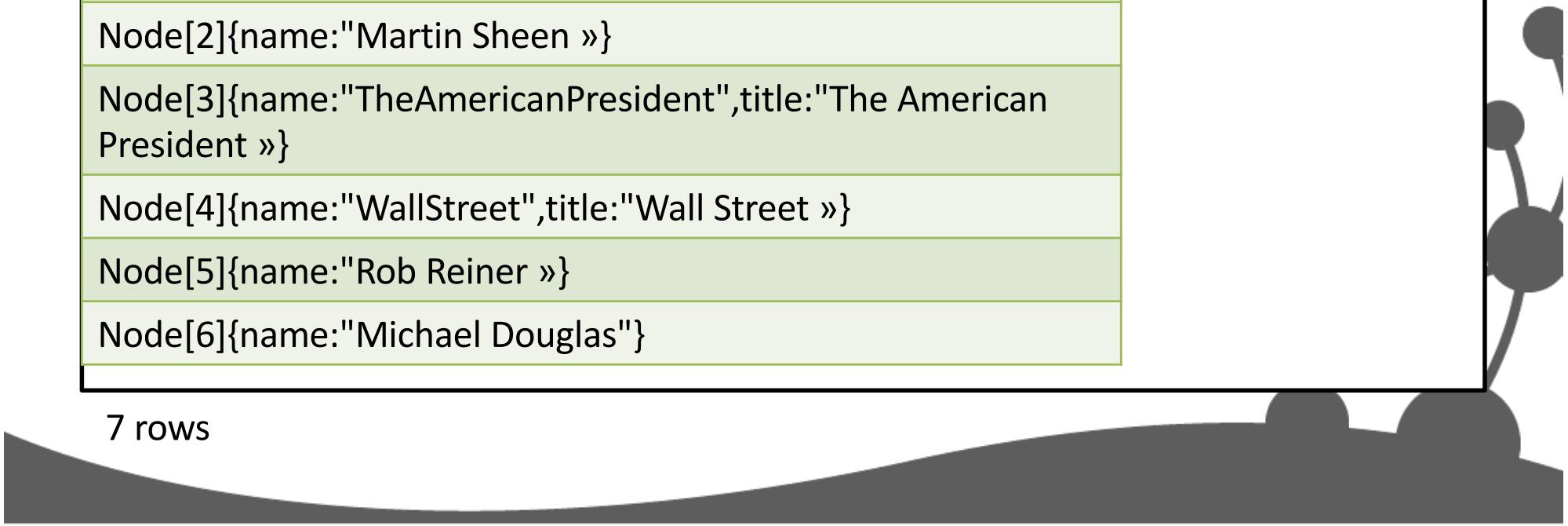
Example- **Get all nodes**

```
MATCH (n)
RETURN n
```

Result

Movie
Node[0]{name:"Oliver Stone »}
Node[1]{name:"Charlie Sheen »}
Node[2]{name:"Martin Sheen »}
Node[3]{name:"TheAmericanPresident",title:"The American President »}
Node[4]{name:"WallStreet",title:"Wall Street »}
Node[5]{name:"Rob Reiner »}
Node[6]{name:"Michael Douglas"}

7 rows



Examples

- **Get all nodes with a label**

```
MATCH (movie:Movie)
RETURN movie
```

Result

Movie

Node[3{name:"TheAmericanPresident",title:"The American President »}
Node[4]{name:"WallStreet",title:"Wall Street"}

2 rows

Example - related nodes

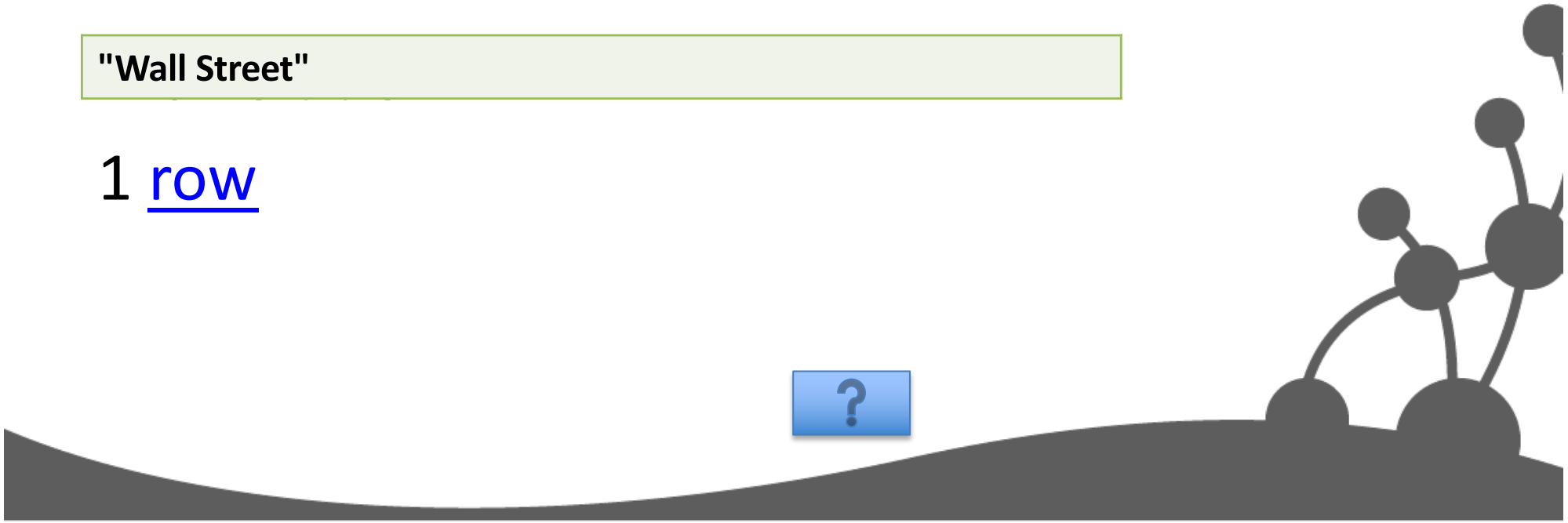
```
MATCH (director { name:'Oliver Stone' })--(movie)
RETURN movie.title
```

Returns all the movies directed by Oliver Stone.

Result

"Wall Street"

1 row



Example – match with labels

```
MATCH (charlie:Person { name:'Charlie Sheen' })--(movie:Movie)
RETURN movie
```

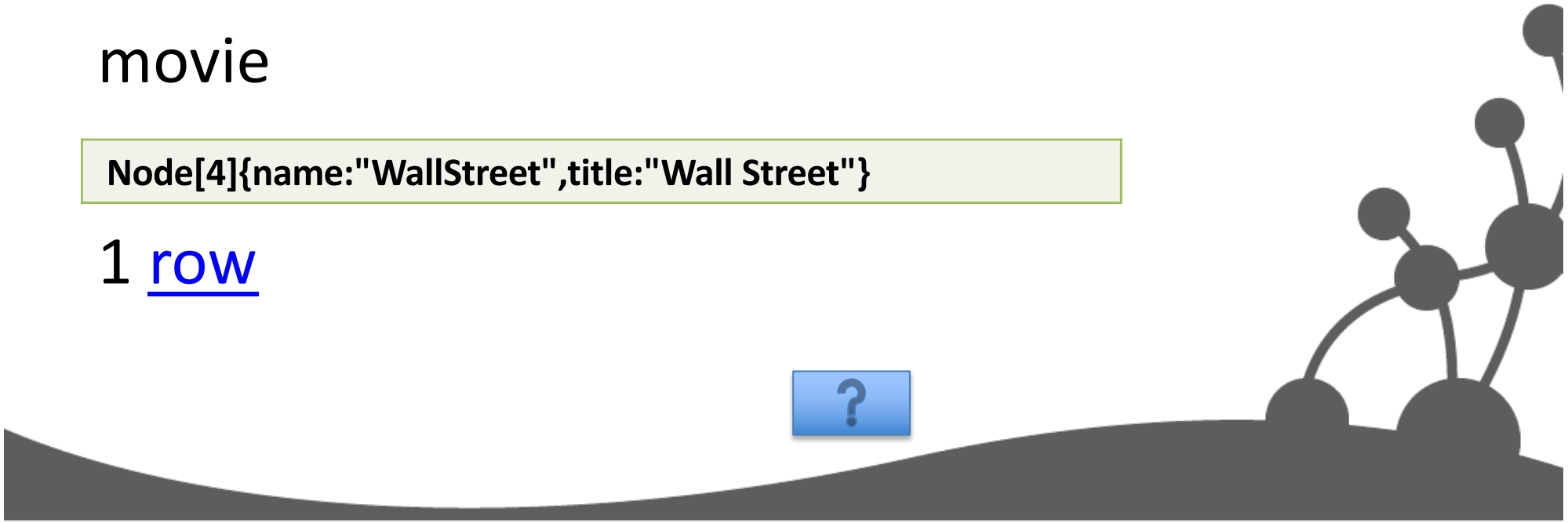
Return any nodes connected with the Person Charlie that are labeled Movie.

Result

movie

```
Node[4]{name:"WallStreet",title:"Wall Street"}
```

1 row



Directed relationships and identifier

```
MATCH (martin { name:'Martin Sheen' })-[r]->(movie)
RETURN r
```

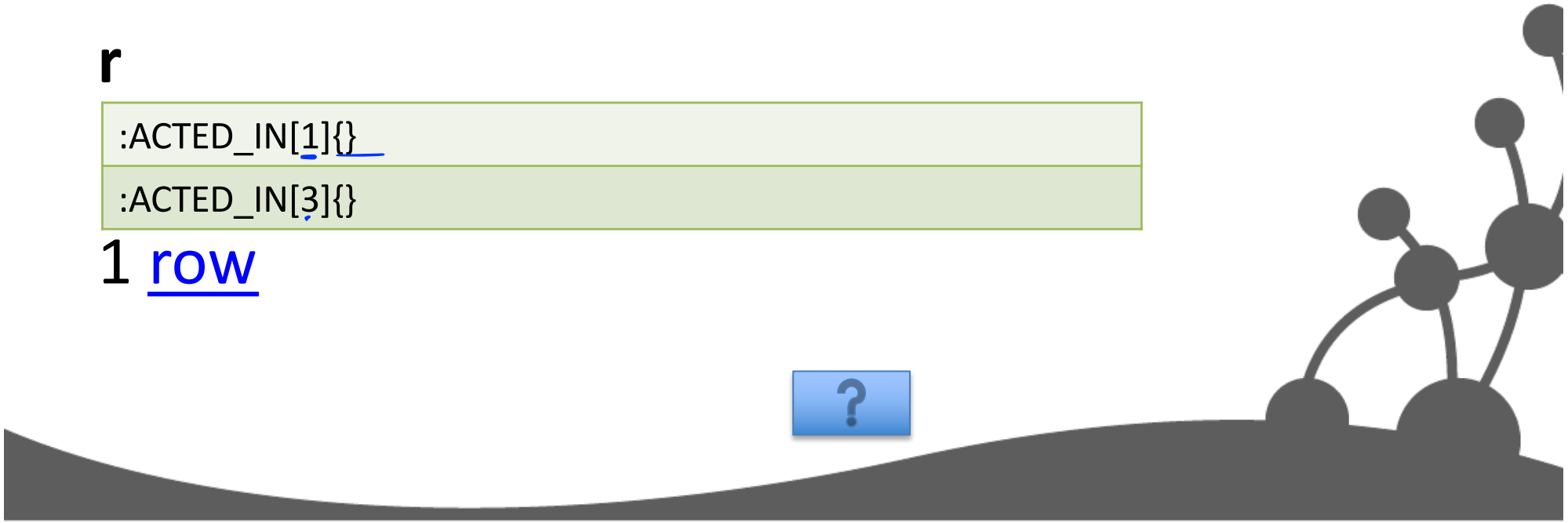
- Returns all outgoing relationships from Martin.

Results

r

:ACTED_IN[1]{}
:ACTED_IN[3]{}

1 row



Match by multiple relationship types

```
MATCH (wallstreet { title:'Wall Street' })<-[:ACTED_IN|:DIRECTED]-(person)
RETURN person
```

- Returns nodes with a ACTED_IN or DIRECTED relationship to Wall Street.
- Results

Person

Node[0]{name:"Oliver Stone"}
Node[1]{name:"Charlie Sheen"}
Node[2]{name:"Martin Sheen"}
Node[6]{name:"Michael Douglas"}

4 rows

Shortest path

```
MATCH (martin:Person { name:"Martin Sheen" }),(oliver:Person { name:"Oliver Stone" }),
p = shortestPath((martin)-[*..15]-(oliver))
 RETURN p
```

- This means: find a single shortest path between two nodes, as long as the path is max 15 relationships long.
- Results

p

[Node[2]{name:"Martin Sheen"},:ACTED_IN[1]{},Node[4]{name:"WallStreet",title:"Wall Street"},:DIRECTED[5]{},Node[0]{name:"Oliver Stone"}]

1 row

Variable length relationships

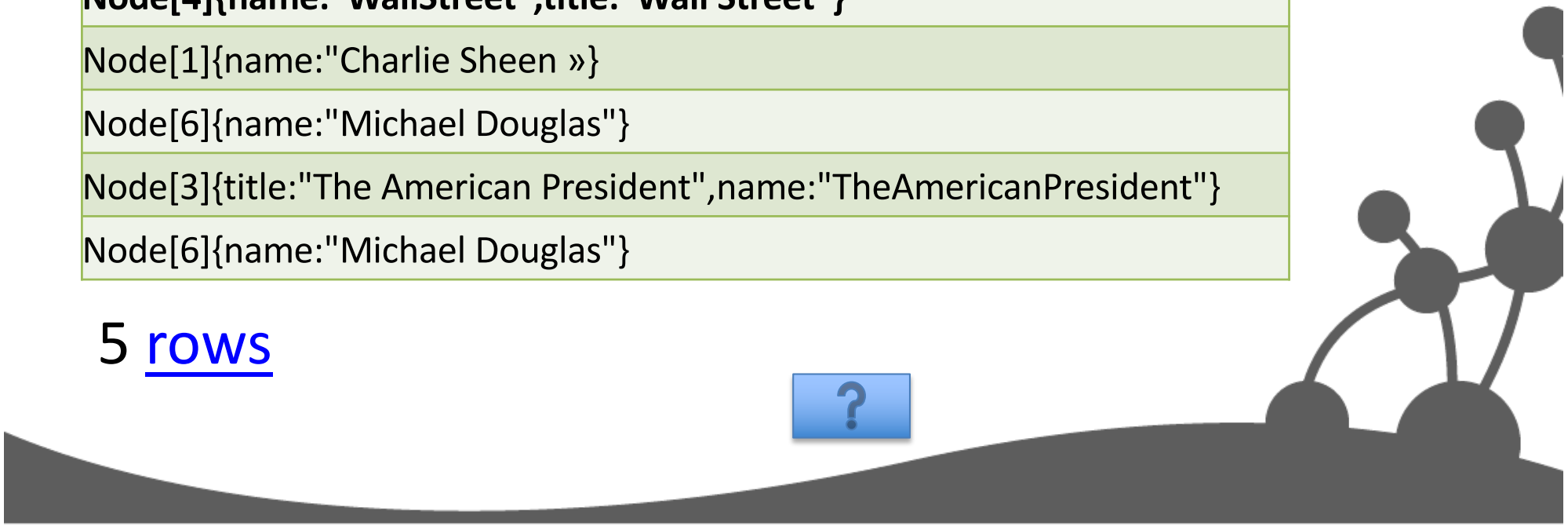
```
MATCH (martin { name:"Martin Sheen" })-[:ACTED_IN*1..2]-(x)
 RETURN x
```

- Returns nodes that are 1 or 2 relationships away from Martin.Results

x

Node[4]{name:"WallStreet",title:"Wall Street"}
Node[1]{name:"Charlie Sheen »}
Node[6]{name:"Michael Douglas"}
Node[3]{title:"The American President",name:"TheAmericanPresident"}
Node[6]{name:"Michael Douglas"}

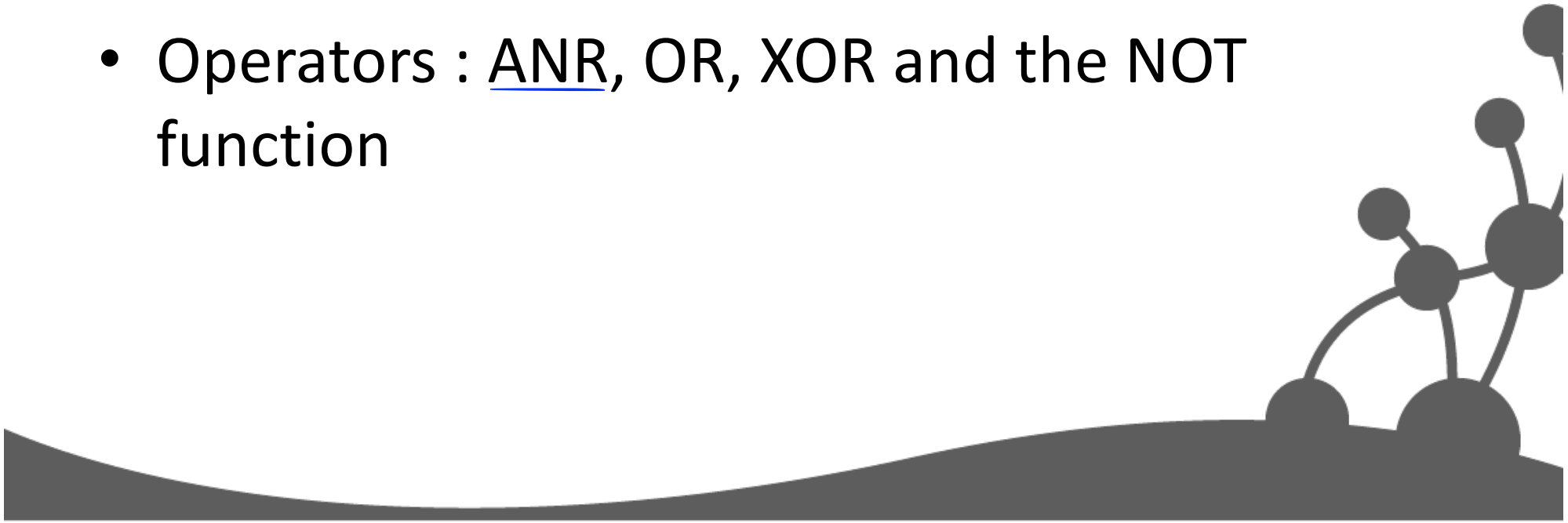
5 rows



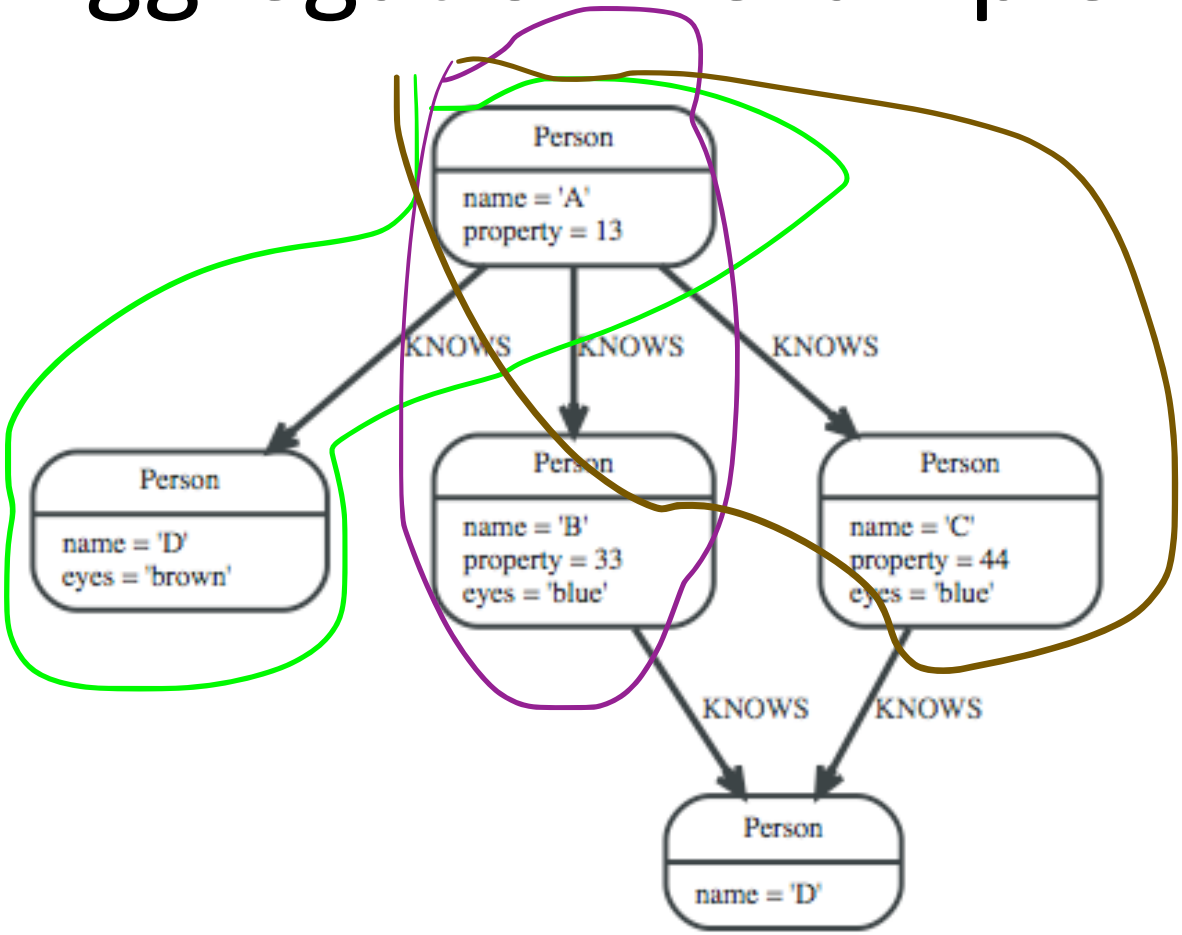
Where clause

```
MATCH (n)
WHERE n:Swedish AND n.age < 30 AND n.name  =~ 'Tob.*’ AND
(n)-[:KNOWS]-({ name:’Peter' })
RETURN n
```

- WHERE adds constraints to the patterns described in Match.
- Operators : ANR, OR, XOR and the NOT function



Aggregation - example



Aggregation

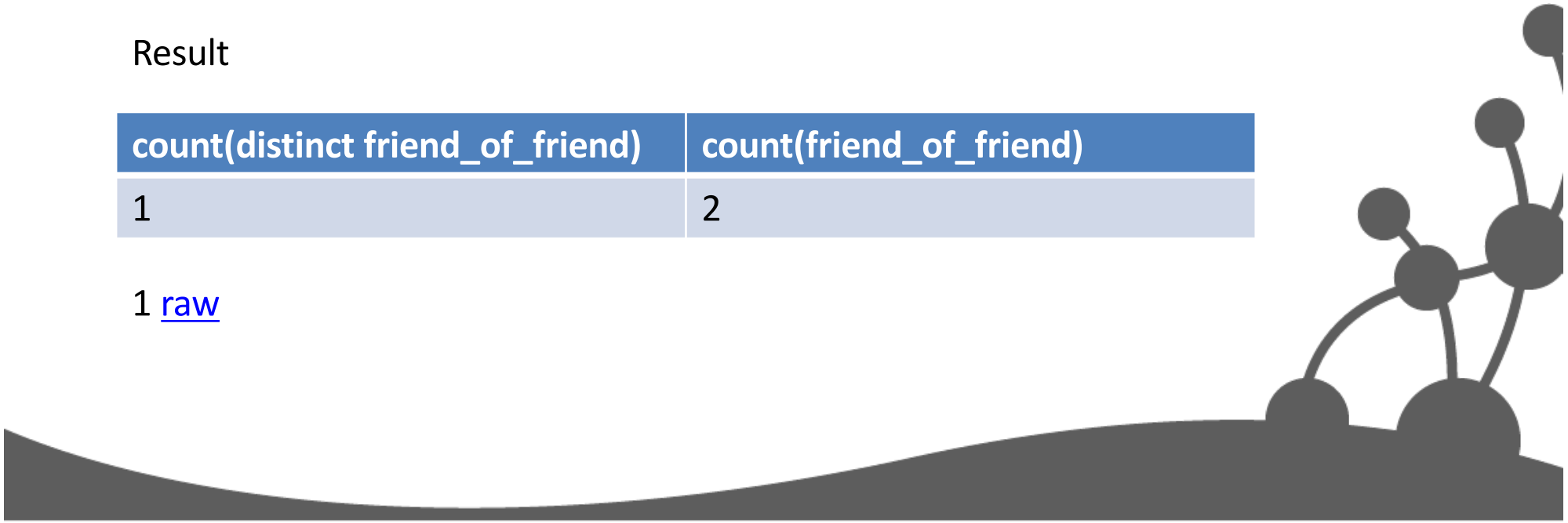
- **Similar to** SQL's GROUP BY.
- **Count(),** `AVG, MAX, MIN, COLLECT,..`

```
MATCH (me:Person)-->(friend:Person)-->(friend_of_friend:Person)
WHERE me.name = 'A'
RETURN count(DISTINCT friend_of_friend), count(friend_of_friend)
```

Result

count(distinct friend_of_friend)	count(friend_of_friend)
1	2

1 raw



Count

- count(*) counts the number of matching rows

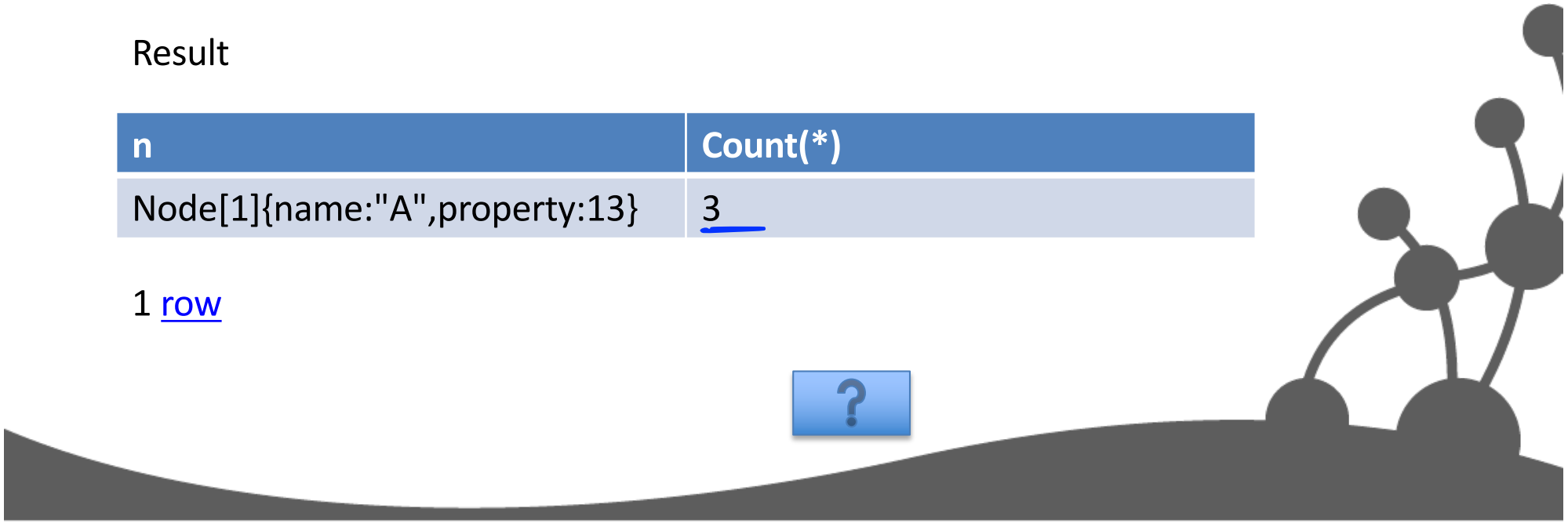
```
MATCH (n { name: 'A' })-->(x)
RETURN n, count(*)
```

This returns the start node and the count of related nodes.

Result

n	Count(*)
Node[1]{name:"A",property:13}	3

1 row



Count

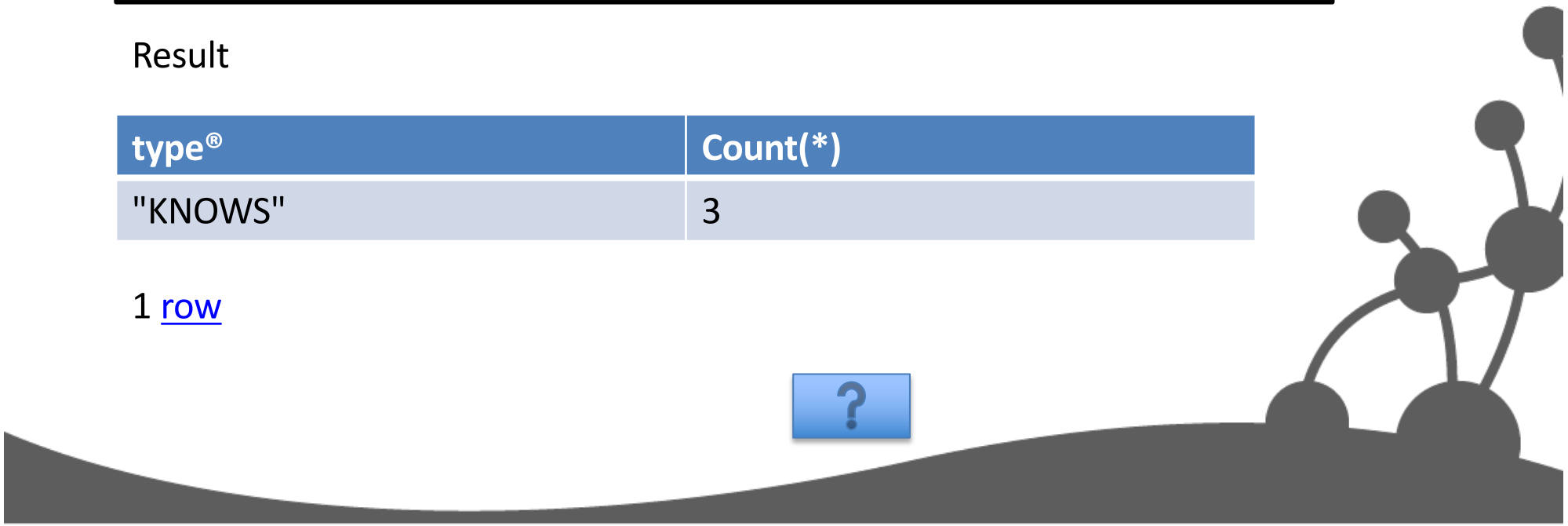
- to count the groups of relationship types, return the types and count them with count(*).

```
MATCH (n { name: 'A' })-[r]->()
RETURN type(r), count(*)
```

Result

type®	Count(*)
"KNOWS"	3

1 row



Count

- COUNT(<identifier>), which counts the number of non-NULL values in <identifier>.

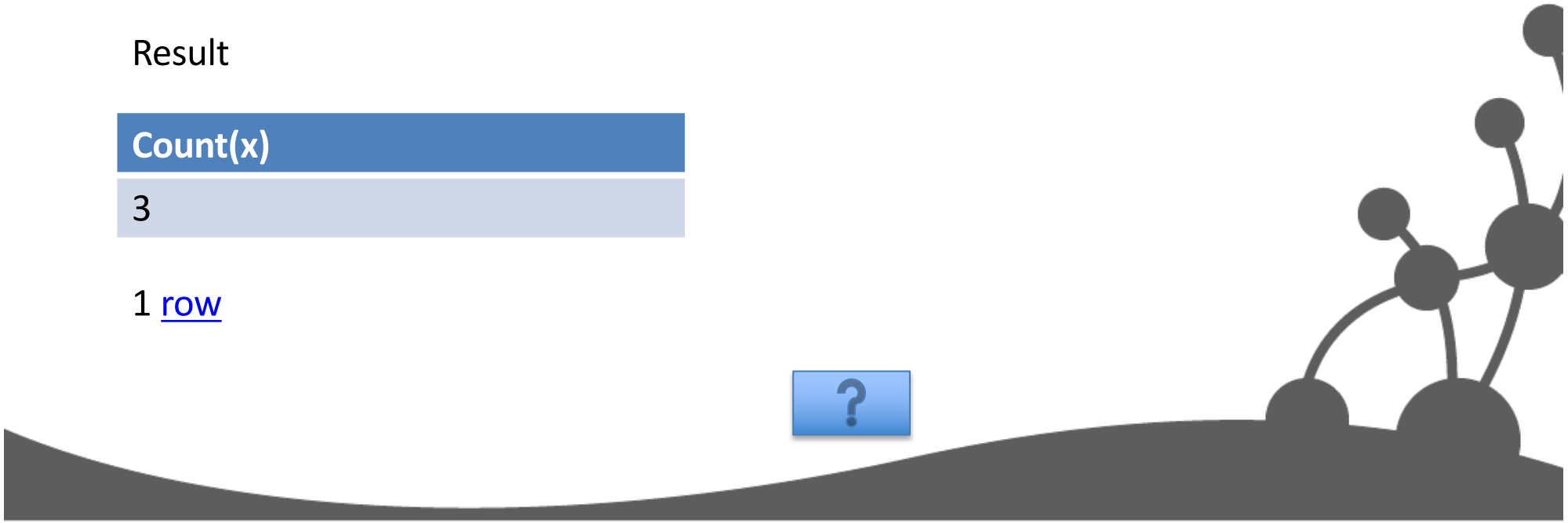
```
MATCH (n { name: 'A' })-->(x)
RETURN count(x)
```

returns the number of connected nodes from the start node.

Result

Count(x)
3

1 row



Statistics

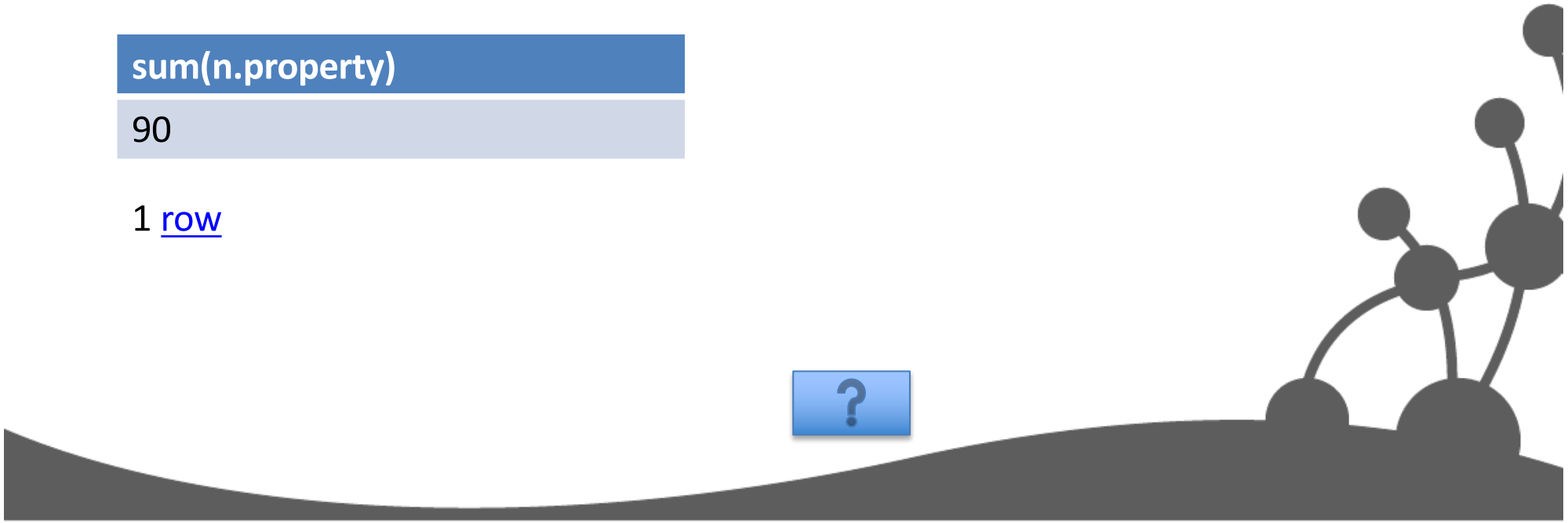
```
MATCH (n:Person)
RETURN sum(n.property)
```

This returns the sum of all the values in the property property.

Result

sum(n.property)
90

1 row



Writing clauses

- Create a node with a label

```
CREATE (n:Person { name : 'Andres', title : 'Developer' })
```

- To return the node, add **Return n**

Writing clauses

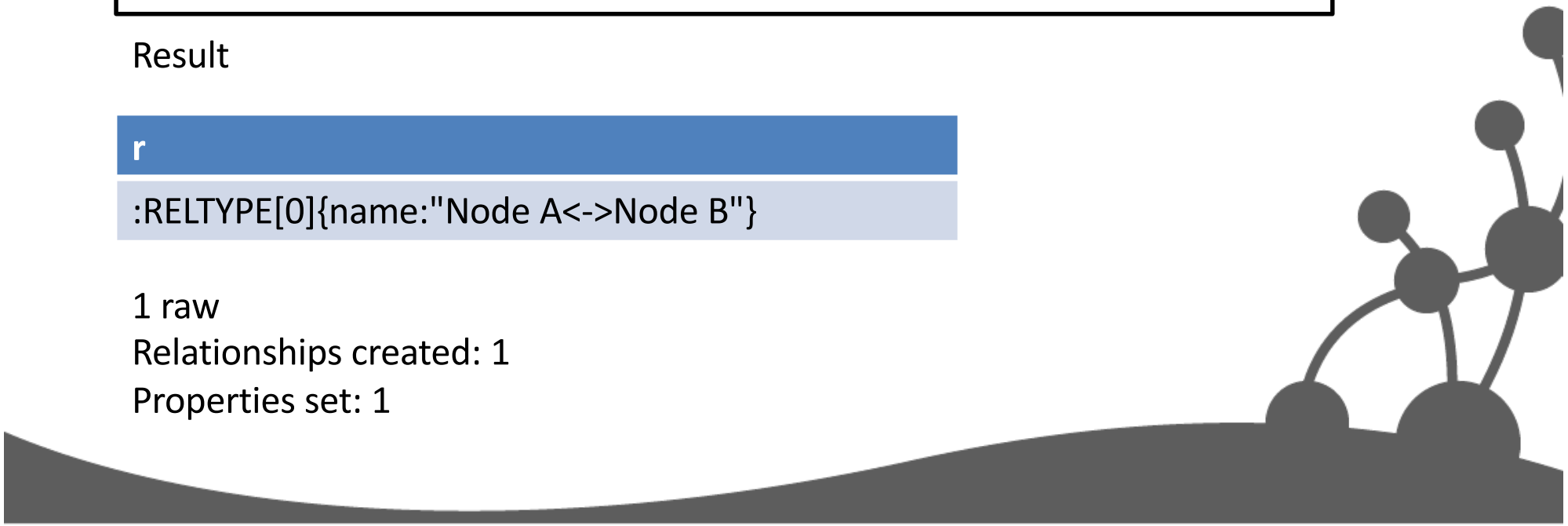
- **Create a relationship and set properties**

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'Node A' AND b.name = 'Node B'
CREATE (a)-[r:RELTYPE { name : a.name + '<->' + b.name }]->(b)
RETURN r
```

Result

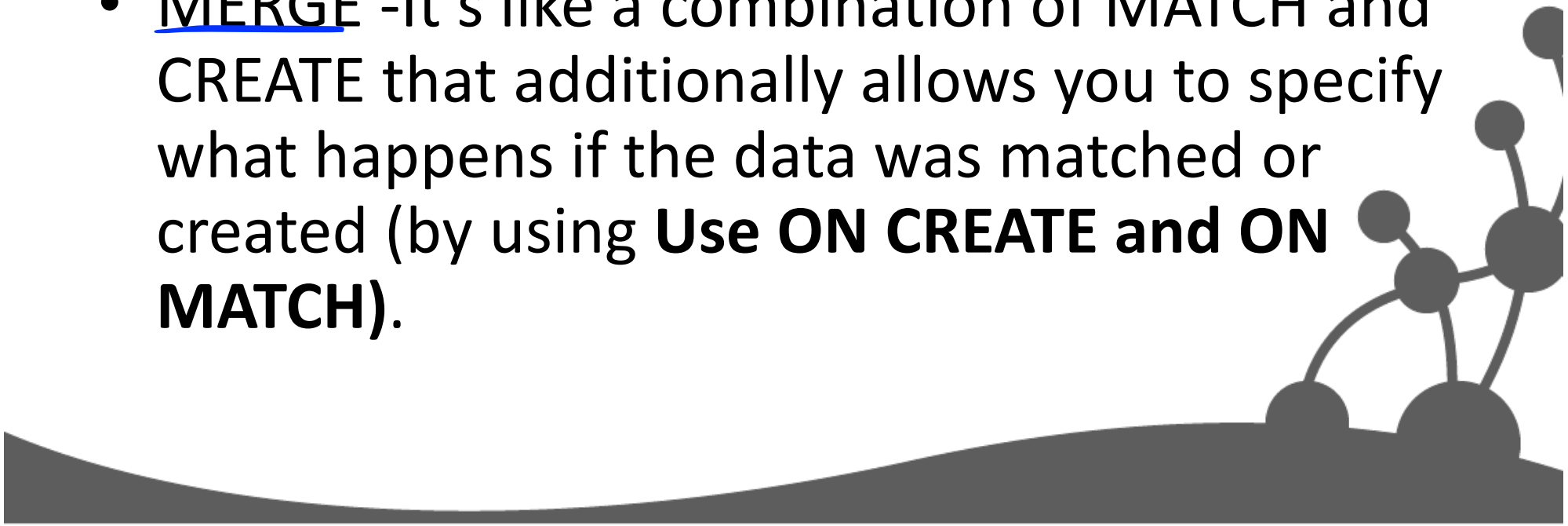
r
:RELTYPE[0]{name:"Node A<->Node B"}

1 raw
Relationships created: 1
Properties set: 1



Writing clauses

- DELETE –deleting graph elements — nodes and relationships
- REMOVE - Removing properties and labels from graph elements.
- MERGE -It's like a combination of MATCH and CREATE that additionally allows you to specify what happens if the data was matched or created (by using **Use ON CREATE and ON MATCH)**.



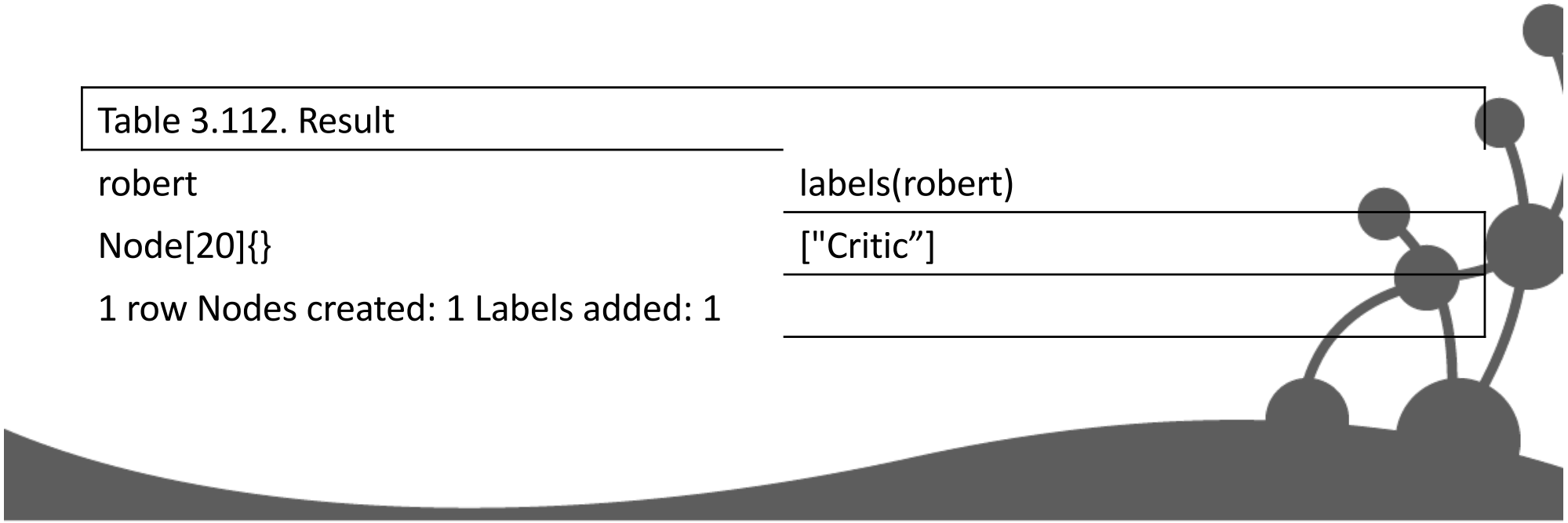
Merge single node with a label

```
MERGE (robert:Critic)
RETURN robert, labels(robert)
```

A new node is created because there are no nodes labeled Critic in the database.

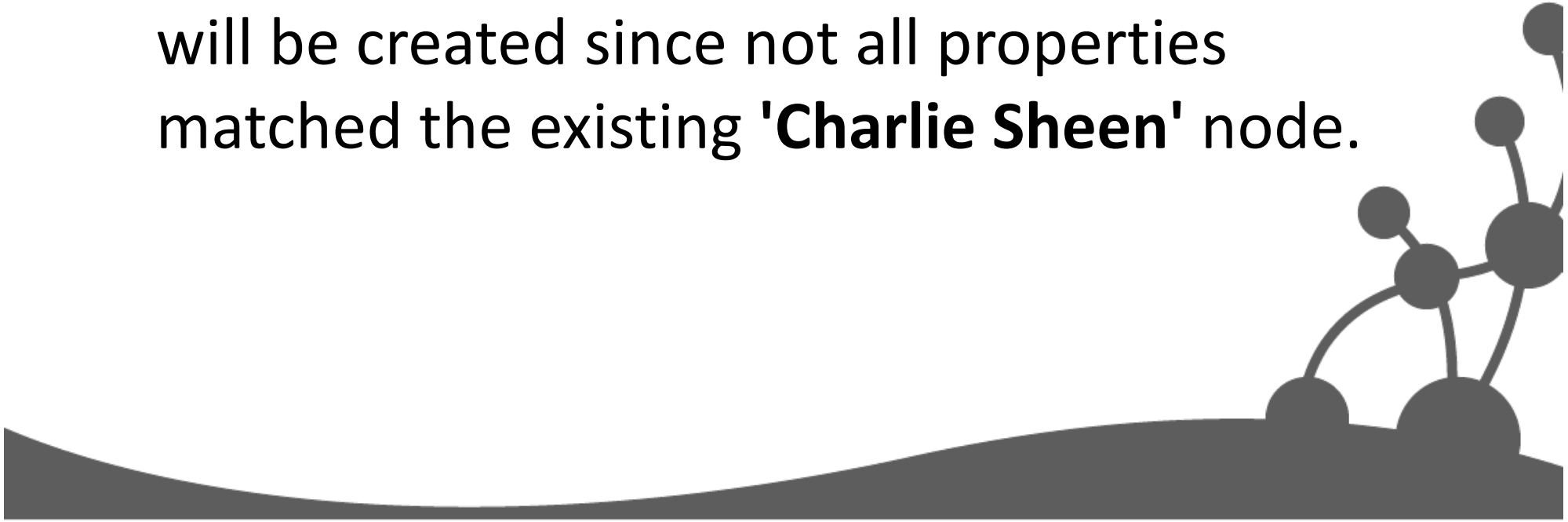
Table 3.112. Result

robert	labels(robert)
Node[20]{}	["Critic”]
1 row Nodes created: 1 Labels added: 1	



Merge single node with properties

- MERGE (charlie { name: 'Charlie Sheen', age: 10 })
- RETURN charlie
- A new node with the name **'Charlie Sheen'** will be created since not all properties matched the existing **'Charlie Sheen'** node.

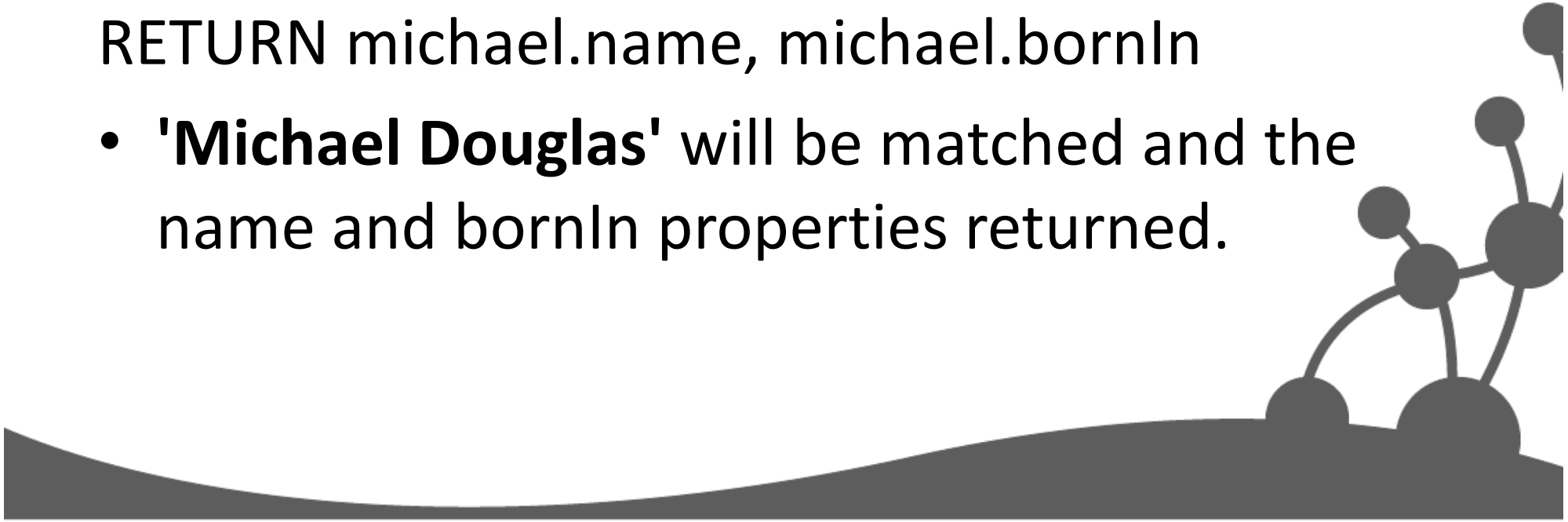


Merge single node specifying both label and property

- Merging a single node with both label and property matching an existing node.

```
MERGE (michael:Person { name: 'Michael Douglas' })
RETURN michael.name, michael.bornIn
```

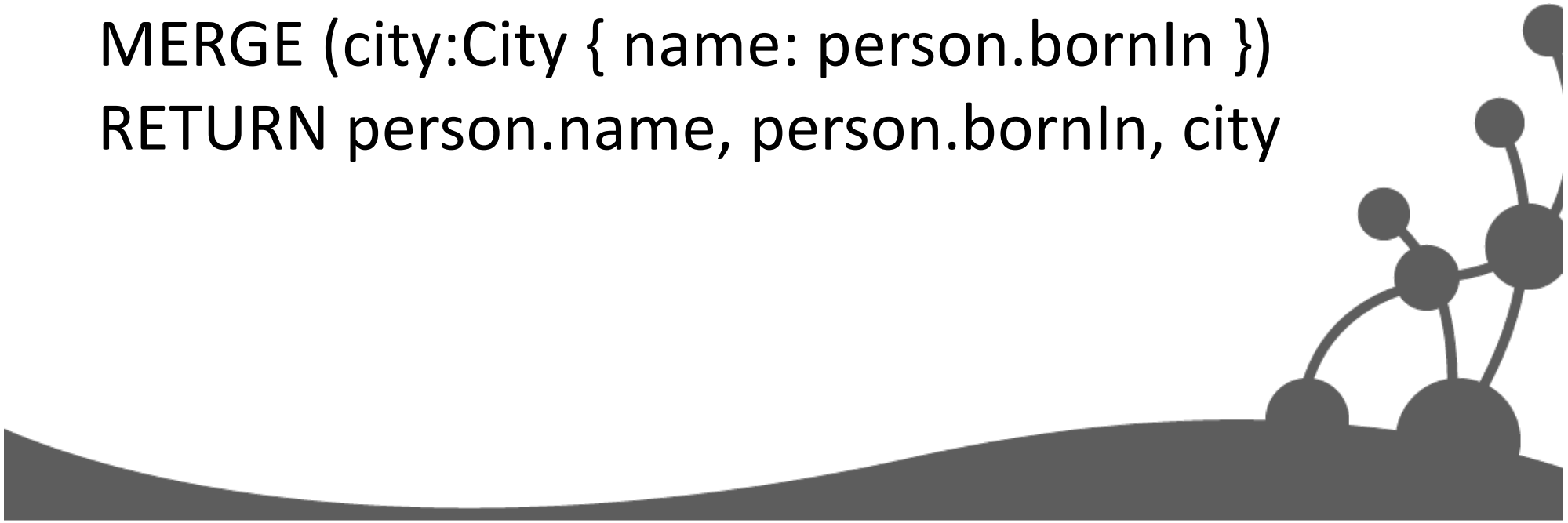
- **'Michael Douglas'** will be matched and the name and bornIn properties returned.



Merge single node derived from an existing node property

- For some property 'p' in each bound node in a set of nodes, a single new node is created for each unique value for 'p'.

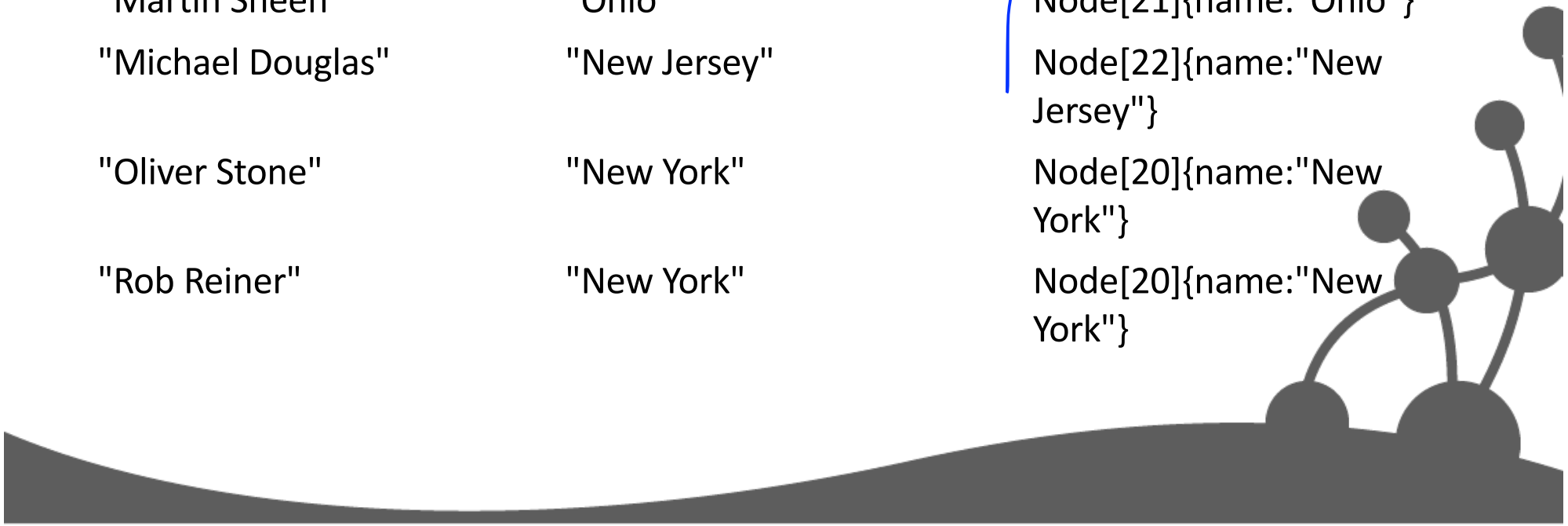
```
MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
RETURN person.name, person.bornIn, city
```



Query Result

Table 3.115. Result

person.name	person.bornIn	city
5 rows Nodes created: 3 Properties set: 3 Labels added: 3		
"Charlie Sheen"	"New York"	Node[20]{name:"New York"}
"Martin Sheen"	"Ohio"	Node[21]{name:"Ohio"}
"Michael Douglas"	"New Jersey"	Node[22]{name:"New Jersey"}
"Oliver Stone"	"New York"	Node[20]{name:"New York"}
"Rob Reiner"	"New York"	Node[20]{name:"New York"}

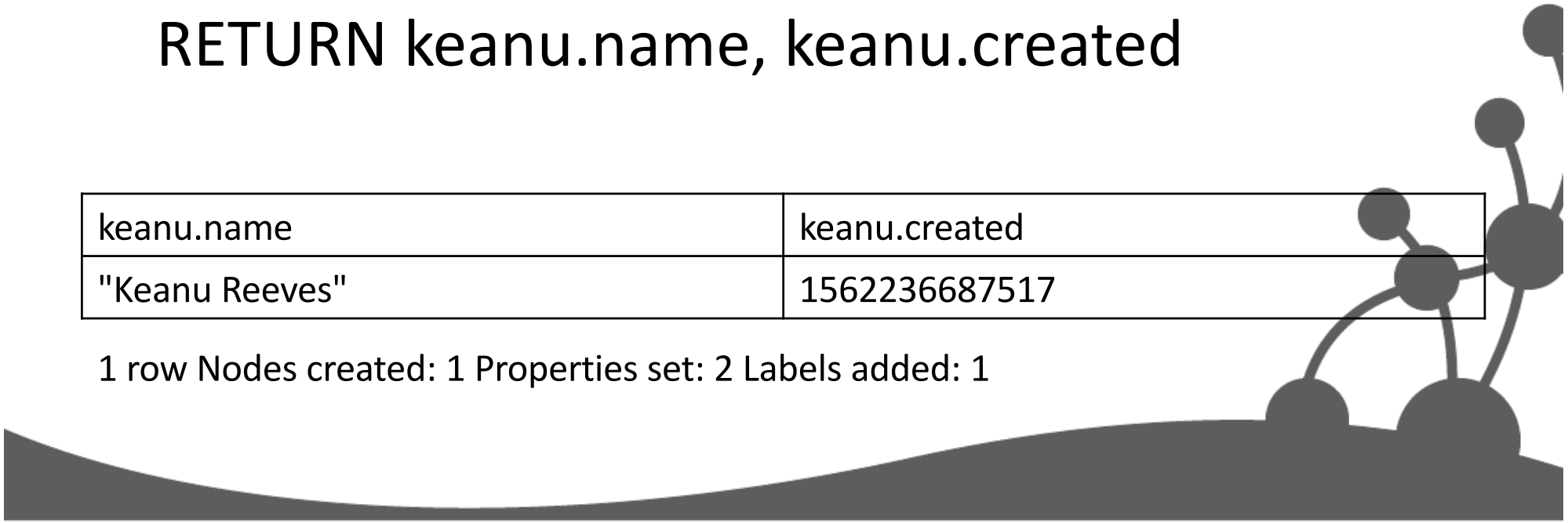


Merge with On CREATE

- Merge a node and set properties if the node needs to be created.
- MERGE (keanu:Person { name: 'Keanu Reeves' })
- ON CREATE SET keanu.created = timestamp()
  RETURN keanu.name, keanu.created

keanu.name	keanu.created
"Keanu Reeves"	1562236687517

1 row Nodes created: 1 Properties set: 2 Labels added: 1

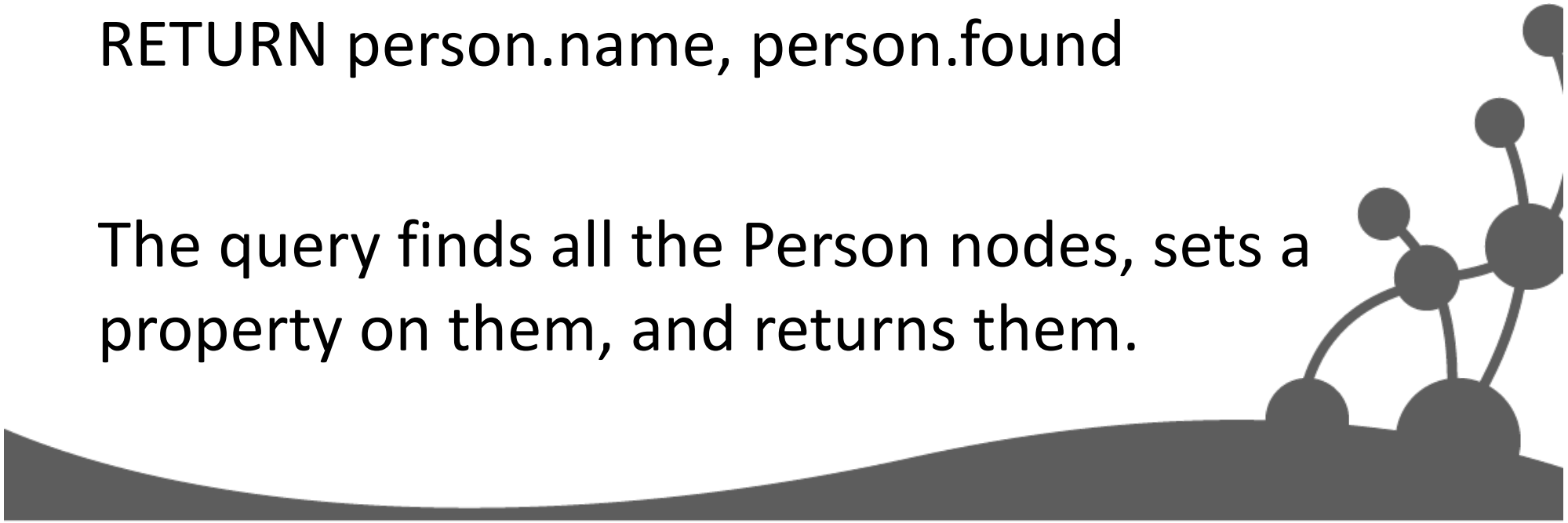


Merge with ON MATCH

- Merging nodes and setting properties on found nodes.

```
MERGE (person:Person) ON MATCH SET
person.found = TRUE
RETURN person.name, person.found
```

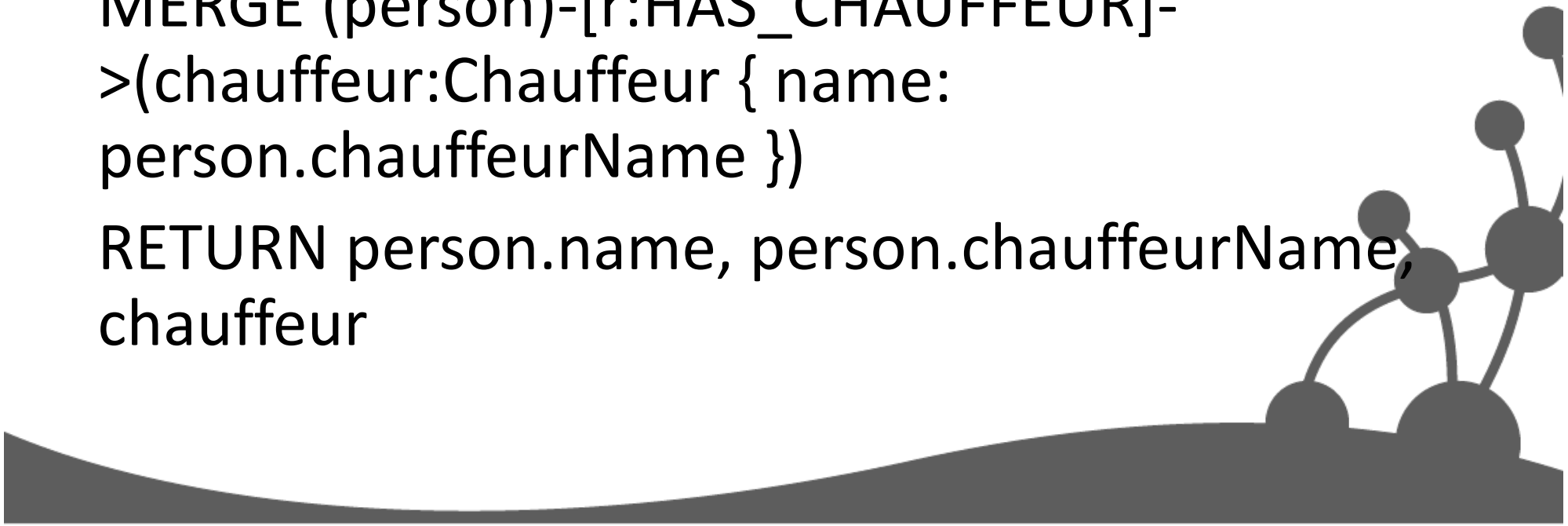
The query finds all the Person nodes, sets a property on them, and returns them.



Merge on a relationship between an existing node and a merged node derived from a node property

- MERGE can be used to simultaneously create both a new node 'n' and a relationship between a bound node 'm' and 'n'.

```
MATCH (person:Person)
MERGE (person)-[r:HAS_CHAUFFEUR]->(chauffeur:Chauffeur { name: person.chauffeurName })
RETURN person.name, person.chauffeurName, chauffeur
```



Schema - indexes

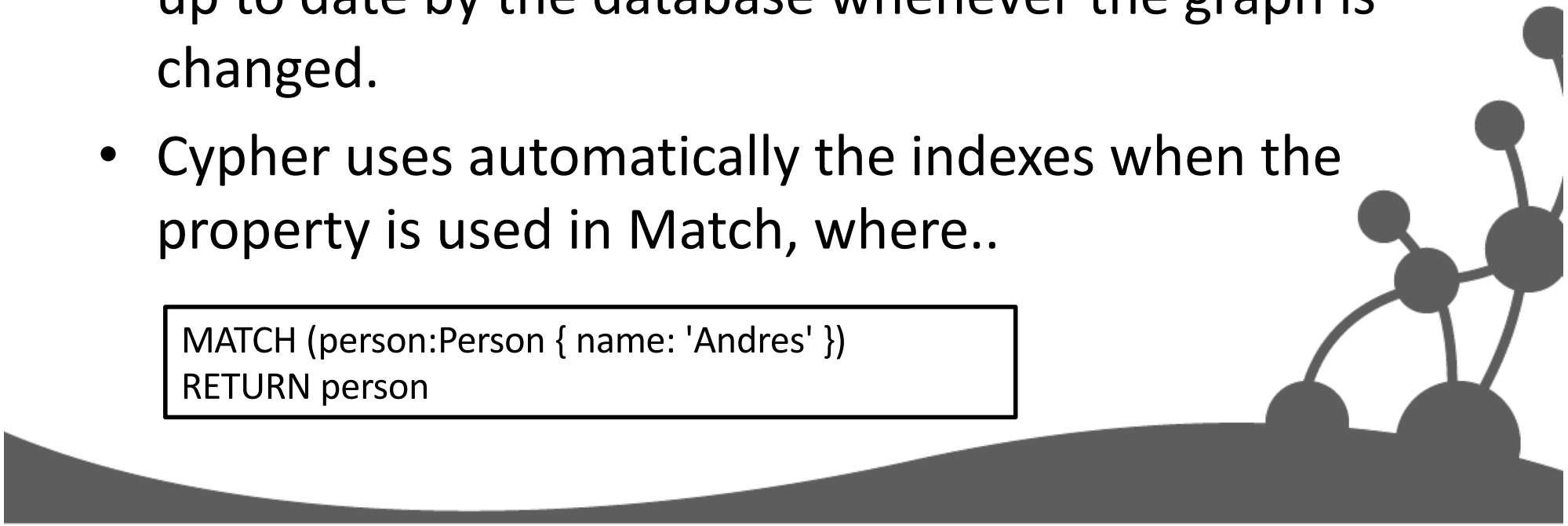
- Cypher allows the creation of indexes over a property for all nodes that have a given label.

```
CREATE INDEX ON :Person(name)
```

```
Drop INDEX ON :Person(name)
```

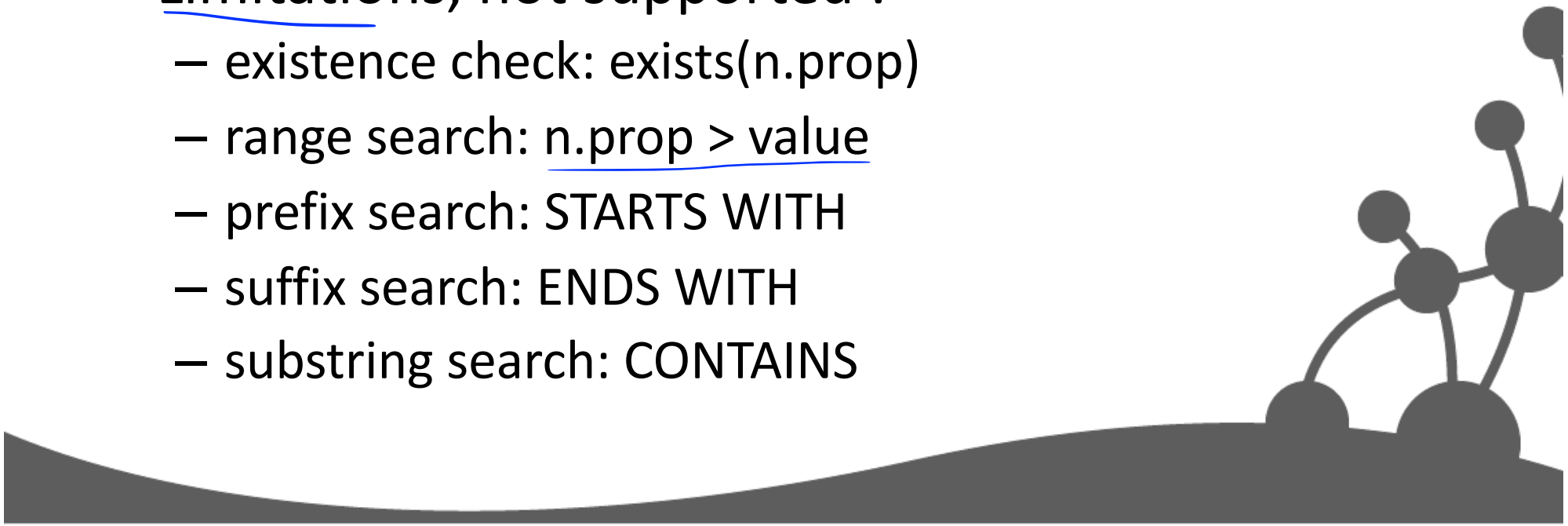
- These indexes are automatically managed and kept up to date by the database whenever the graph is changed.
- Cypher uses automatically the indexes when the property is used in Match, where..

```
MATCH (person:Person { name: 'Andres' })
RETURN person
```



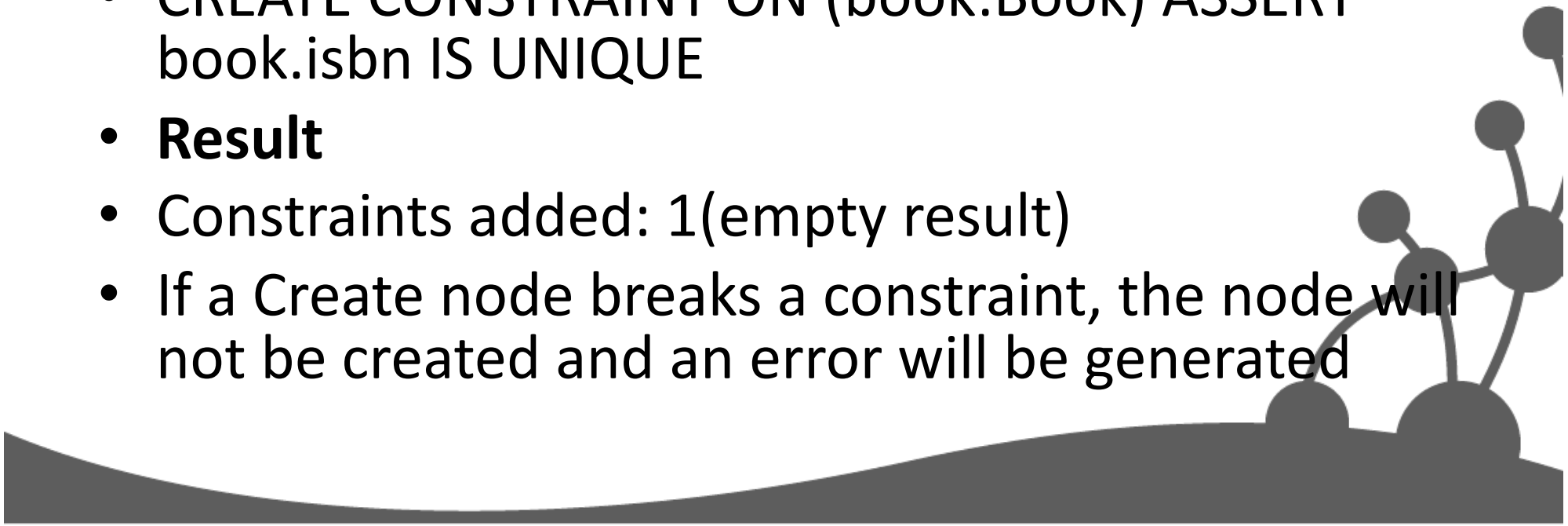
Composite index

- CREATE INDEX ON :Label(prop1, …, propN)
- Only nodes labeled with the specified label and which contain **all the properties** in the index definition will be added to the index.
- Limitations, not supported :
  - existence check: exists(n.prop)
  - range search: n.prop > value
  - prefix search: STARTS WITH
  - suffix search: ENDS WITH
  - substring search: CONTAINS



Schema - constraints

- **Create uniqueness constraint -** IS UNIQUE
  - to make sure that your database will never contain more than one node with a specific label and one property value
- **Query**
- CREATE CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
- **Result**
- Constraints added: 1(empty result)
- If a Create node breaks a constraint, the node will not be created and an error will be generated

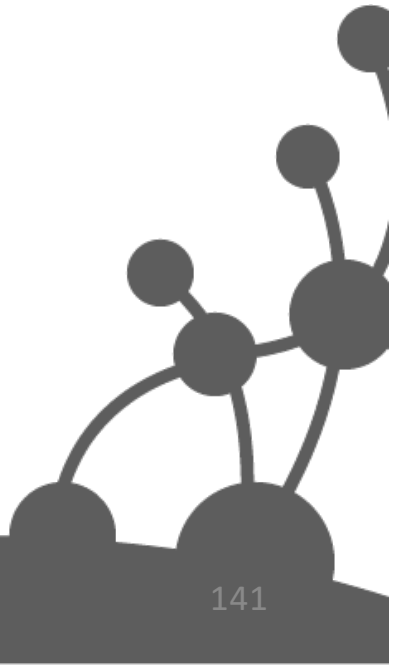


Other Cypher Clauses

- WHERE
  - Provides criteria for filtering pattern matching results.
- Ordering:

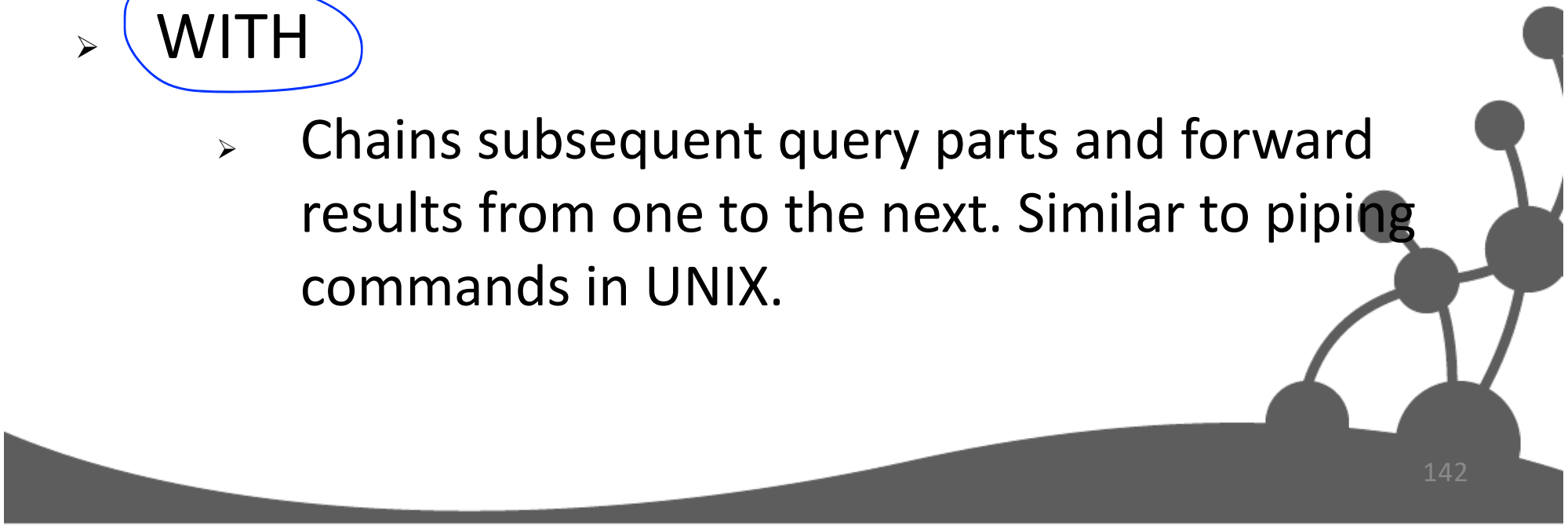
```
order by <property>
order by <property> desc
```

- CREATE
  - Create nodes and relationships
- DELETE
  - Removes nodes, relationships and properties
- SET
  - Sets property values

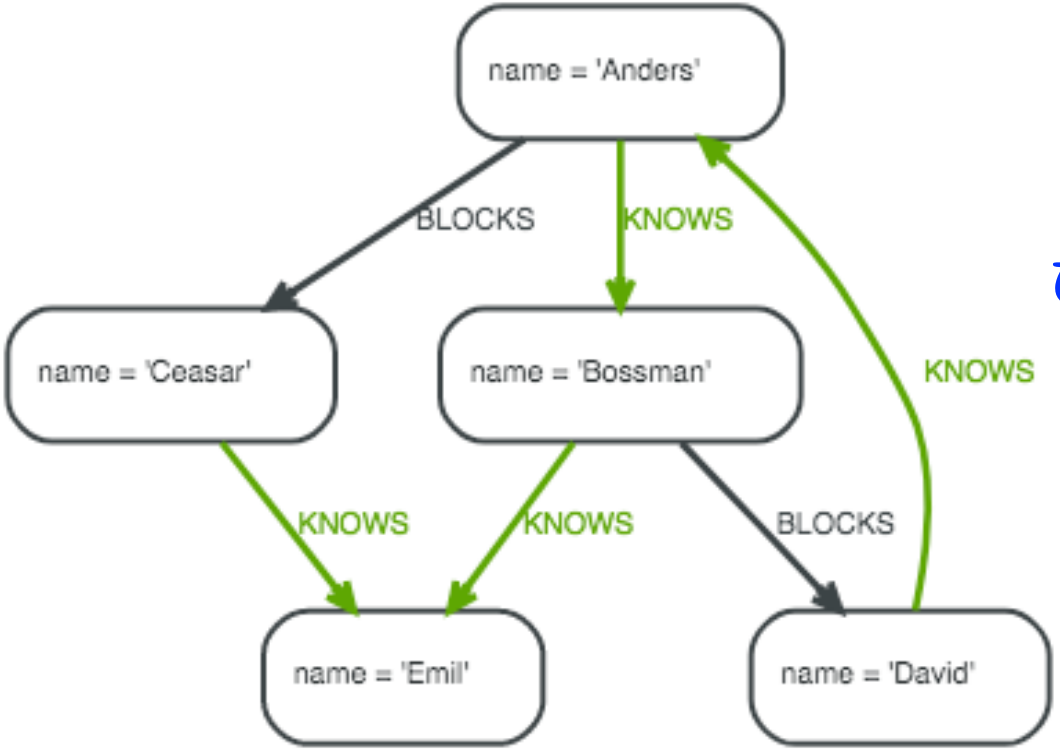


Other Cypher Clauses

- FOREACH
  - Performs an updating action for graph element in a list (a path, ..).
- UNION
  - Merge results from two or more queries.
- WITH
  - Chains subsequent query parts and forward results from one to the next. Similar to piping commands in UNIX.



WITH example

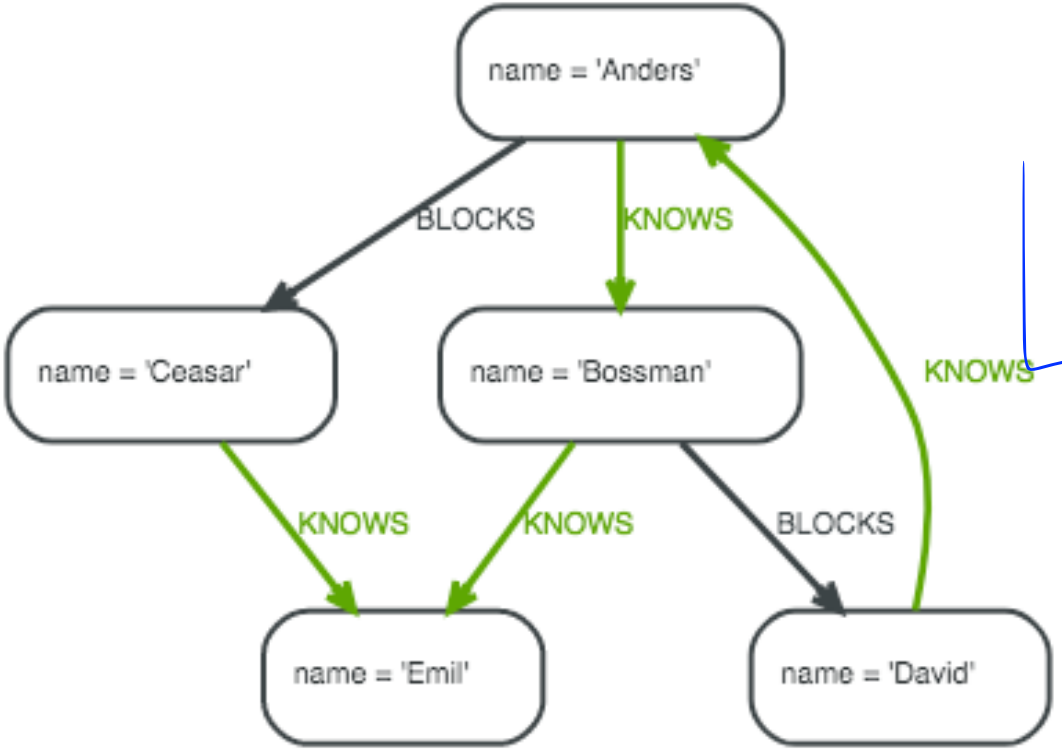


Filter on aggregate function results

```
MATCH (david { name: 'David' })-->(otherPerson)-->()
WITH otherPerson, count(*) AS foaf
WHERE foaf > 1
RETURN otherPerson.name
```

Result
otherPerson.name
"Anders »
1 row

WITH example



limit the number of entries for 2nd match

```
MATCH (n { name: 'Anders' })--(m)
WITH m
ORDER BY m.name DESC LIMIT 1
MATCH (m)--(o)
RETURN o.name
```

Result
o.name
"Bossman"
"Anders"
2 rows

Example Query

- The top 5 most frequently appearing companions:

```
match (doctor:characters{name = 'Doctor}),
    (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	

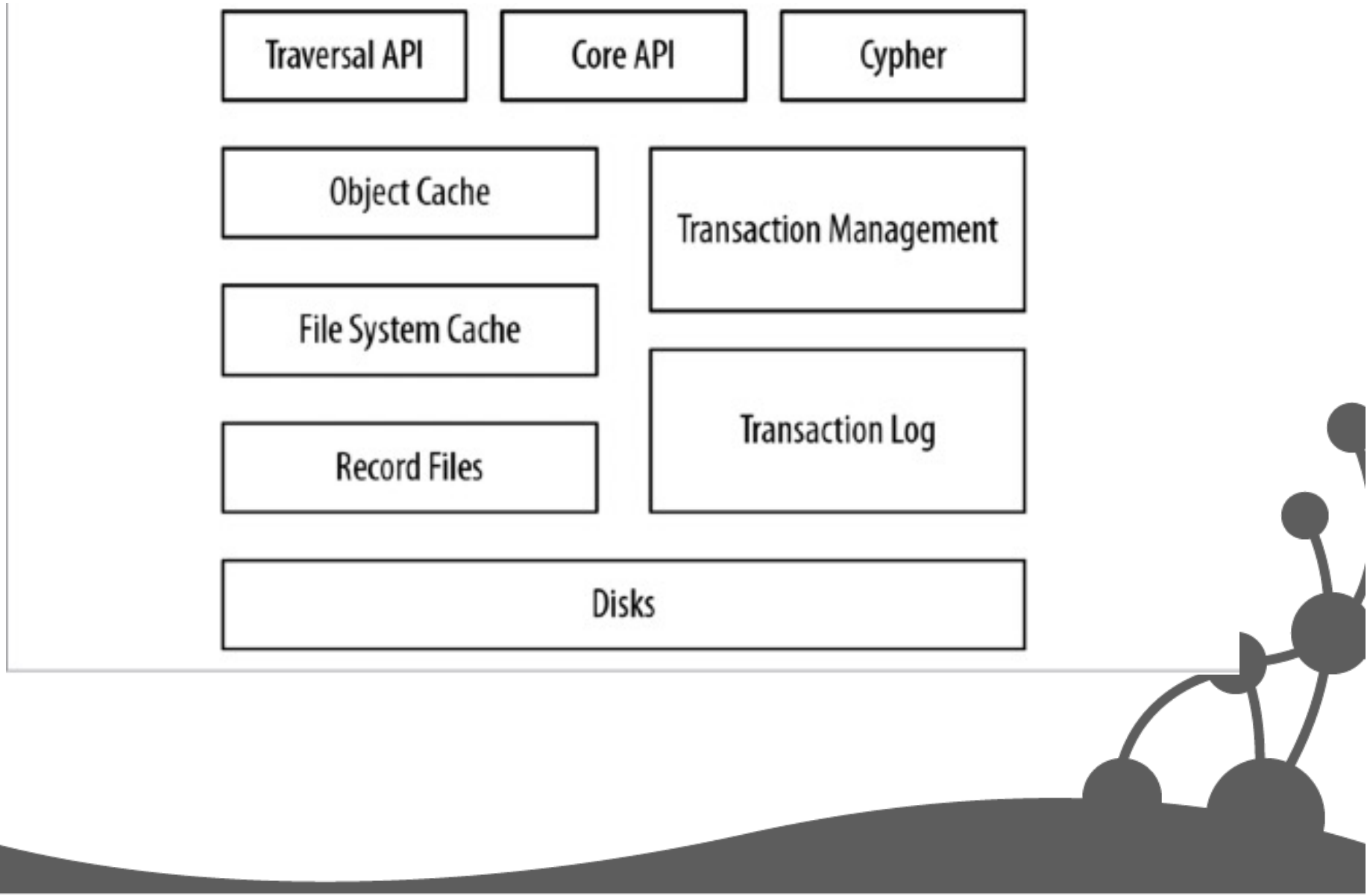
Execute Cypher queries from Java

GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase(databaseDirectory);

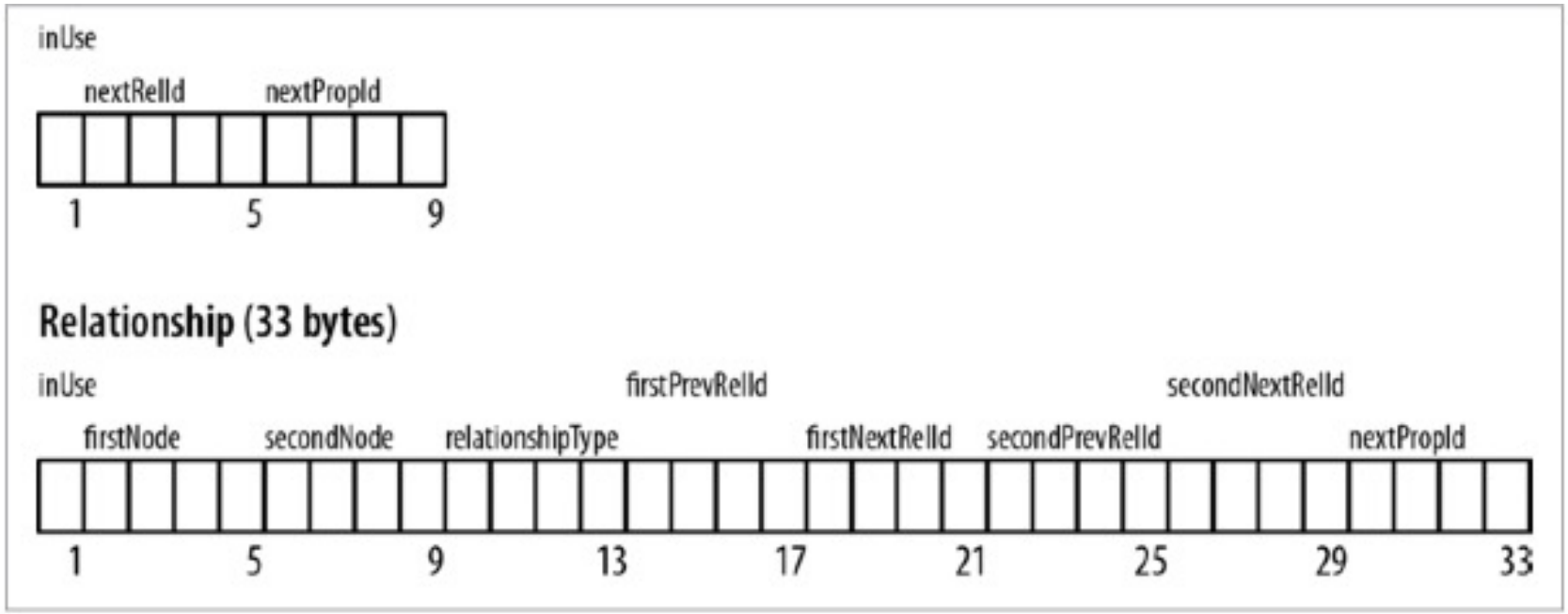
```
String cql = "match (doctor:characters{name = 'Doctor})"
           + " match (doctor)<-[:COMPANION_OF]-(companion)"
           + "-[:APPEARED_IN]->(episode)"
           + " return companion.name, count(episode)"
           + " order by count(episode) desc limit 5";

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

Graph Database internals

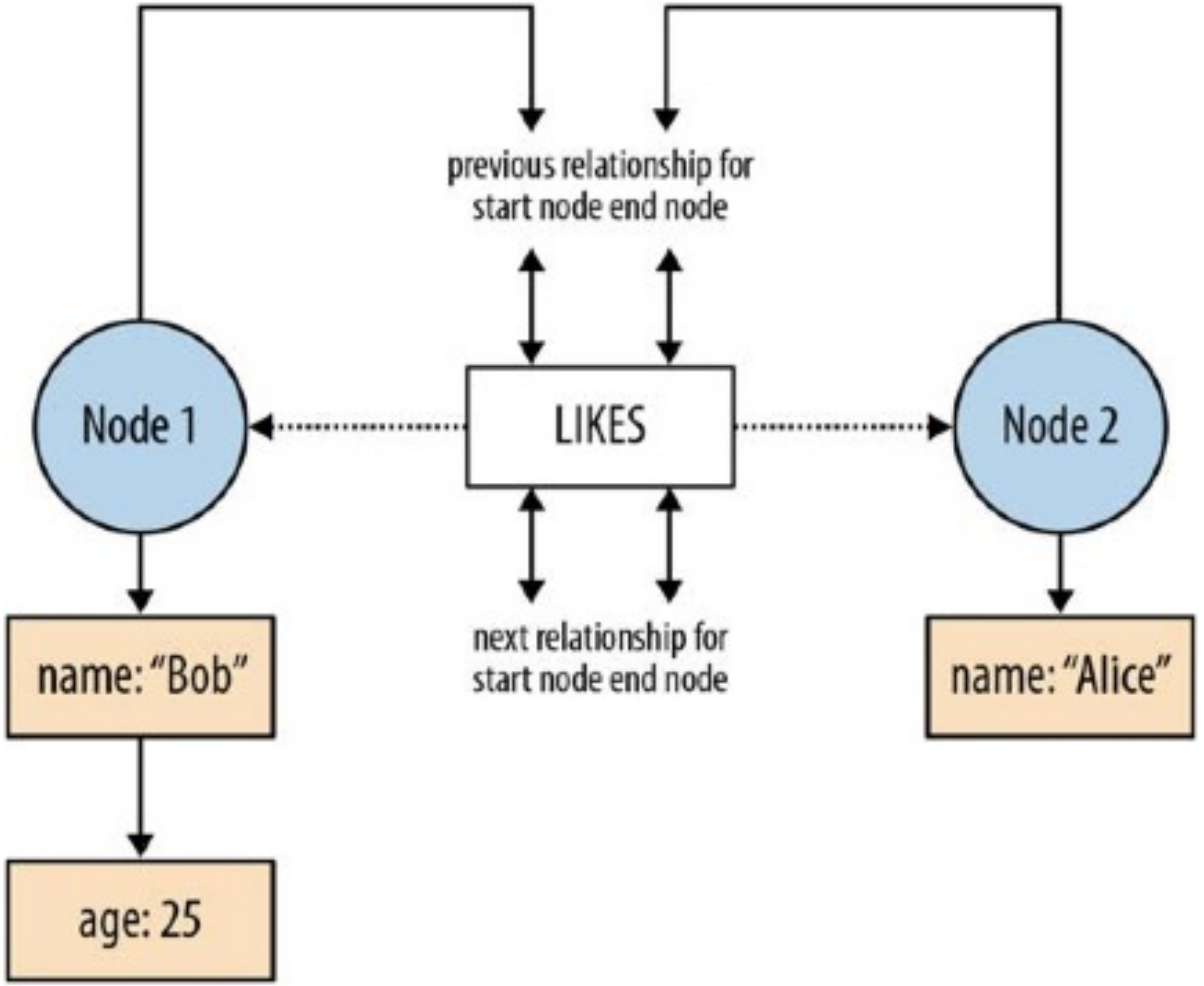
Neo4j Architecture



Neo4j node and relationship store file record structure

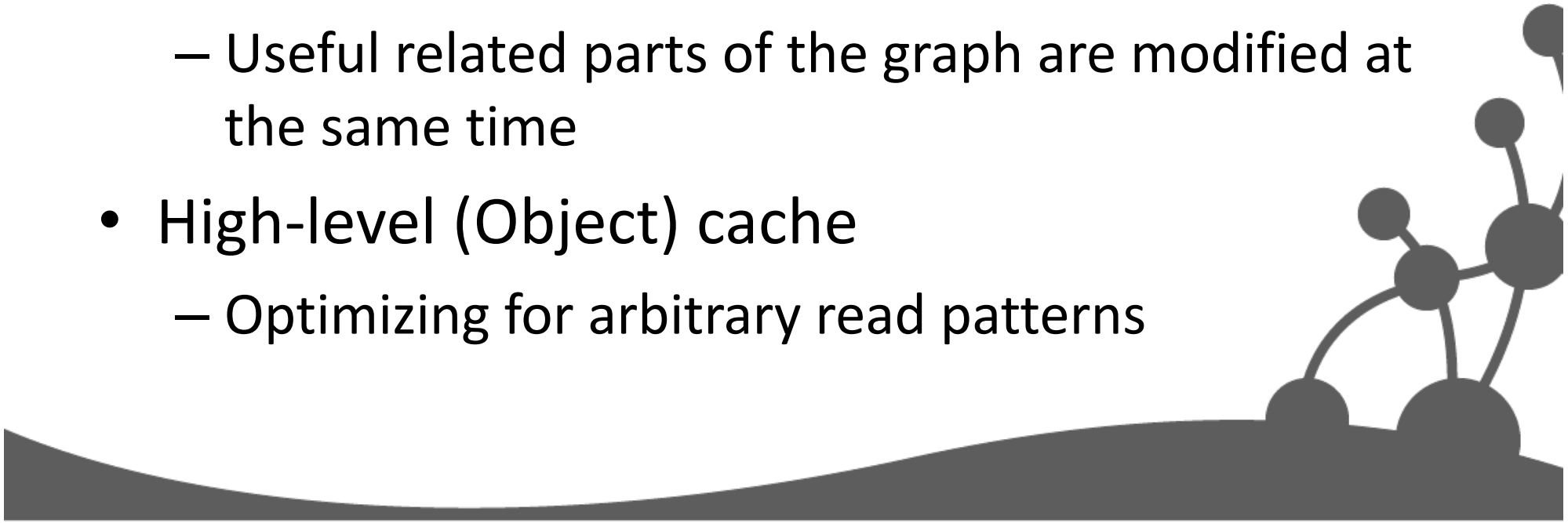


How a graph is physically stored

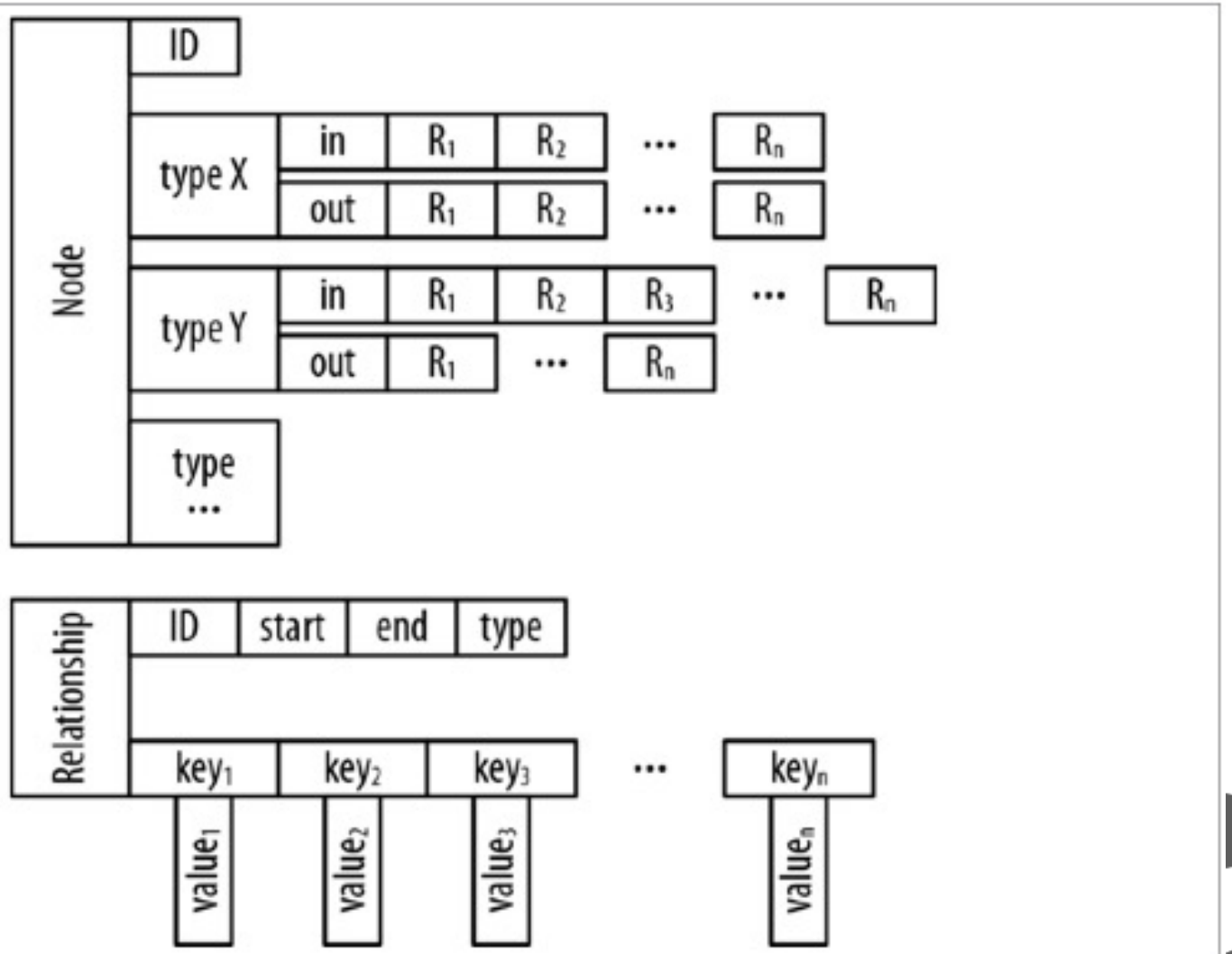


Caches

- Efficient storage layout
- Two tiers caching architecture to provide probabilistic low-latency access to the graph
- Filesystem cache
  - Useful related parts of the graph are modified at the same time
- High-level (Object) cache
  - Optimizing for arbitrary read patterns



Object cache



Non Functional Characteristics

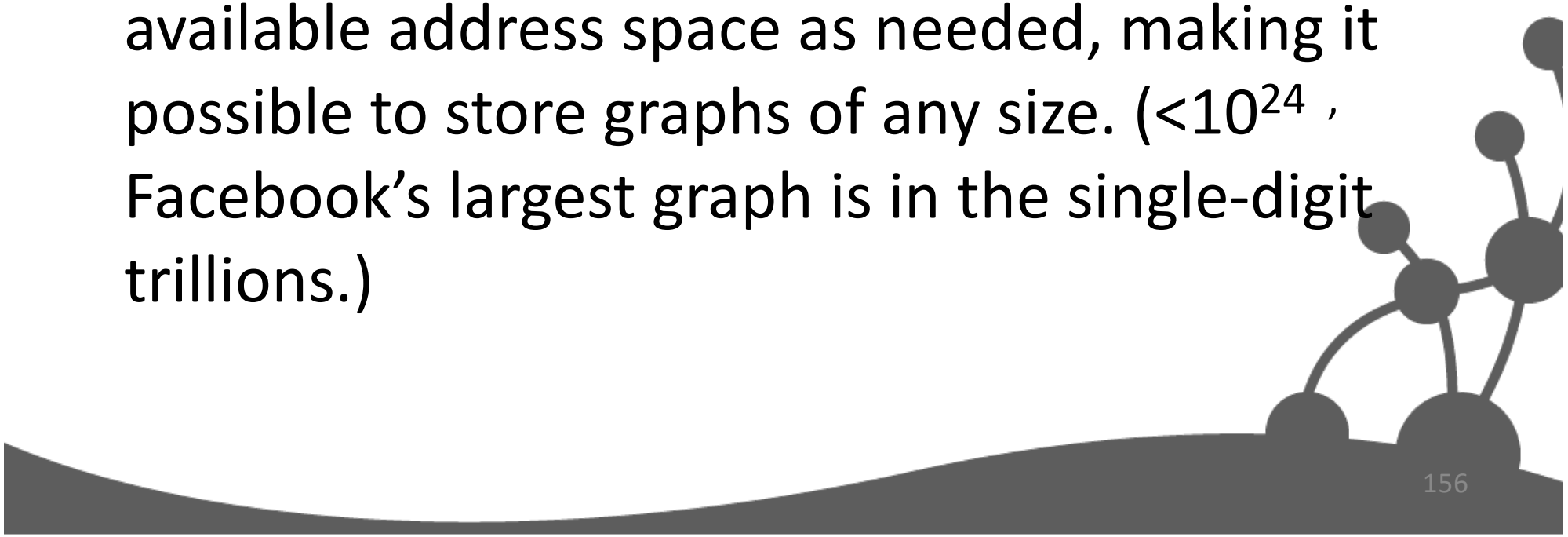
- Transactions
  - Fully ACID
- Recoverability
- Availability
- Scalability

Scalability

- Capacity (Graph Size)
- Latency (Response Time)
- Read and Write Throughput

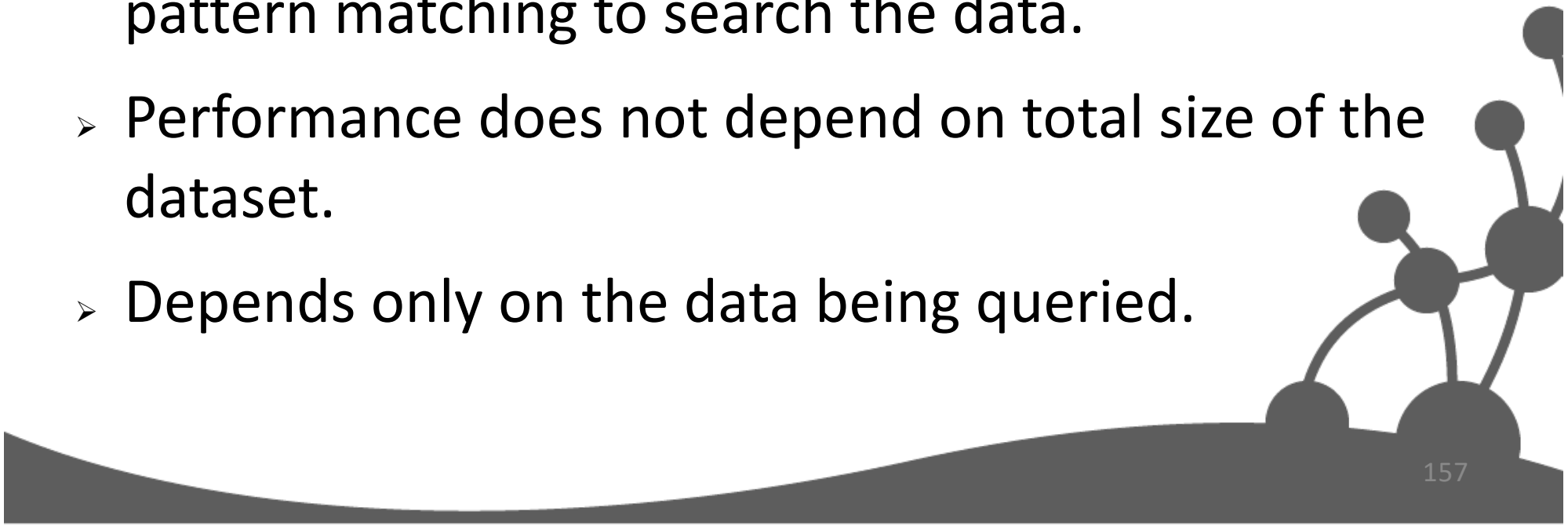
Capacity

- 1.9 Release of Neo4j can support single graphs having **10s of billions of nodes, relationships and properties**.
- Starting with 3.0 :
- Dynamic pointer compression expands Neo4j's available address space as needed, making it possible to store graphs of any size. ($<10^{24}$, Facebook's largest graph is in the single-digit trillions.)



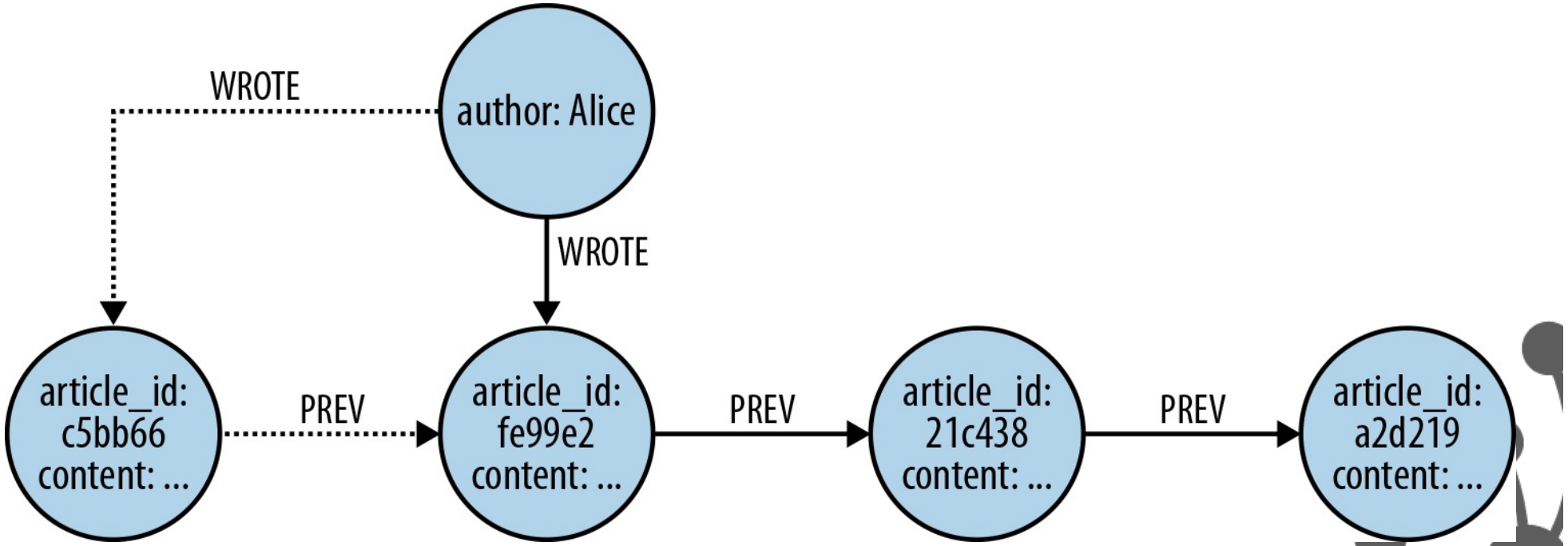
Latency

- RDBMS – more data in tables/indexes result in longer join operations.
- Graph DB doesn't suffer the same latency problem.
- Index is used to find starting node.
- Traversal uses a combination of pointer chasing and pattern matching to search the data.
- Performance does not depend on total size of the dataset.
- Depends only on the data being queried.



Throughput

- Constant performance irrespective of graph size.



Operations and Big Data

Neo4j in Production, in the Large

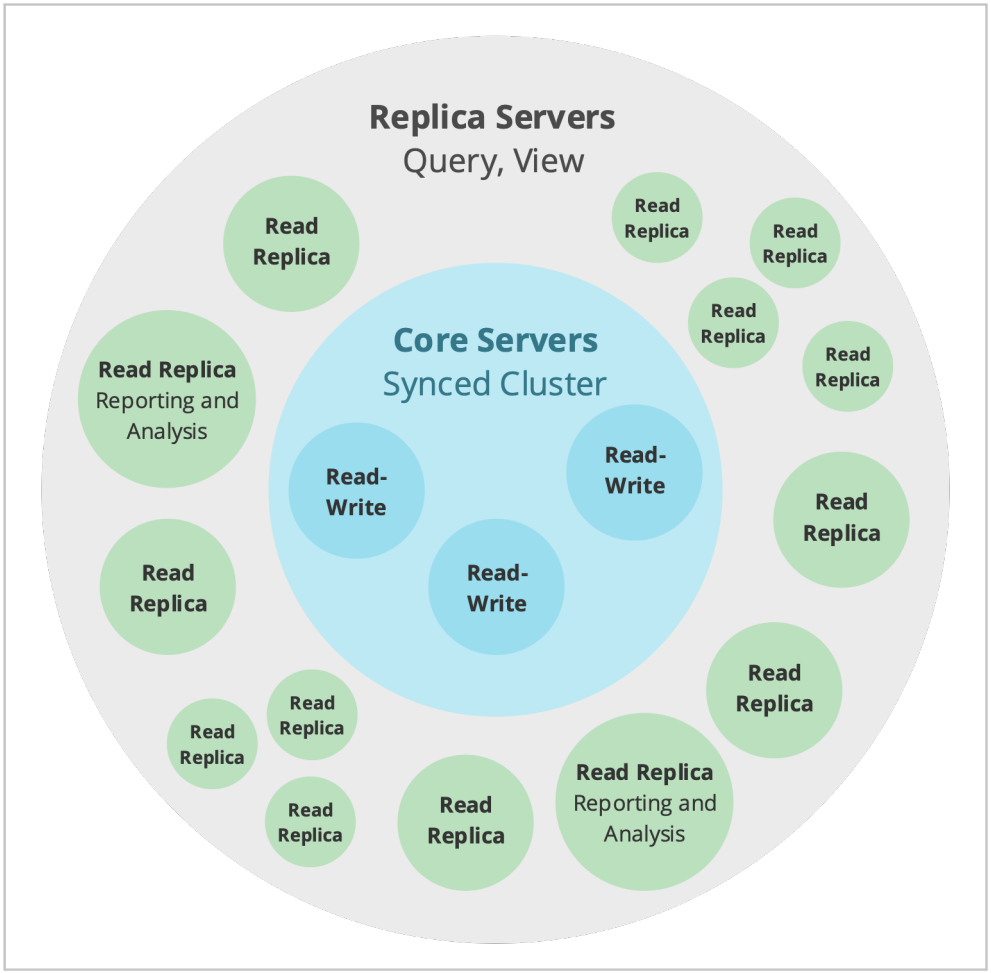
Two Challenges

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

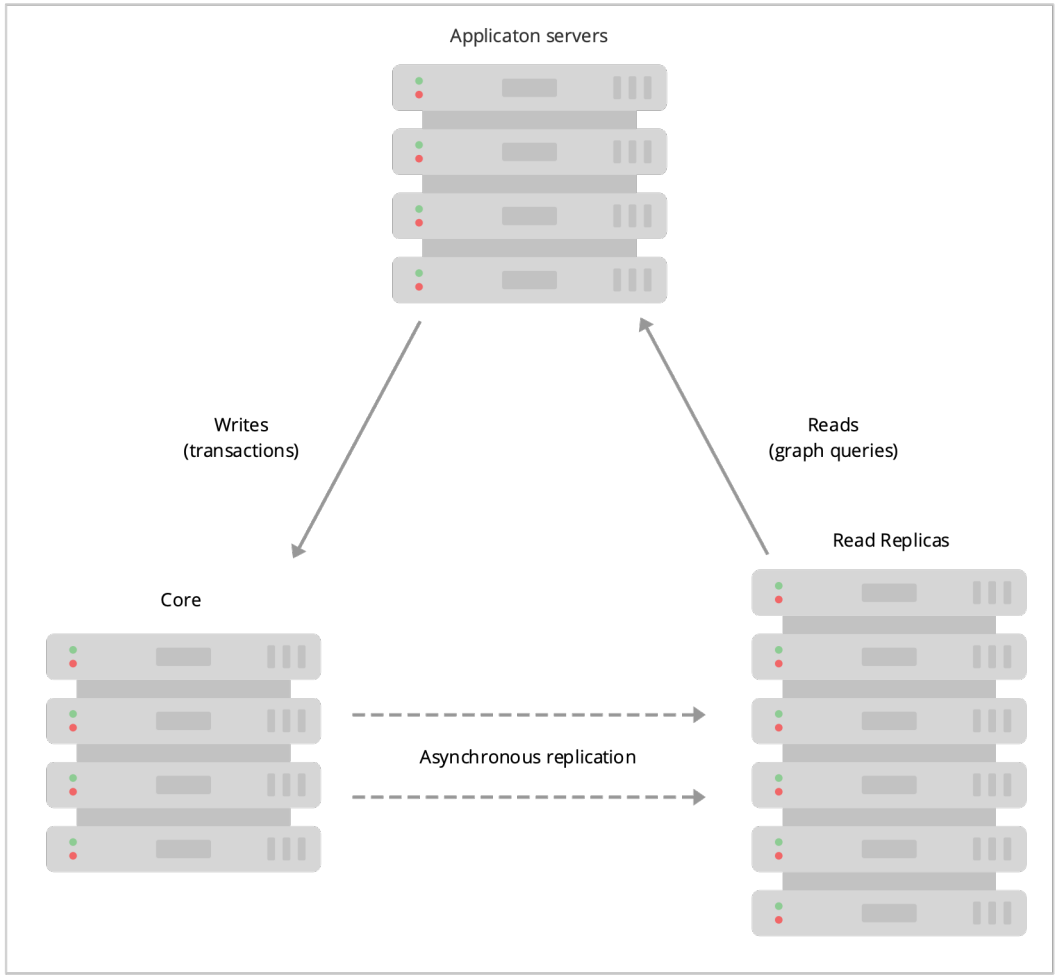
Neo4j High Availability

- Enterprise Edition only
- The HA component supports replication
  - For load balancing
  - For DR across sites

Causal cluster architecture

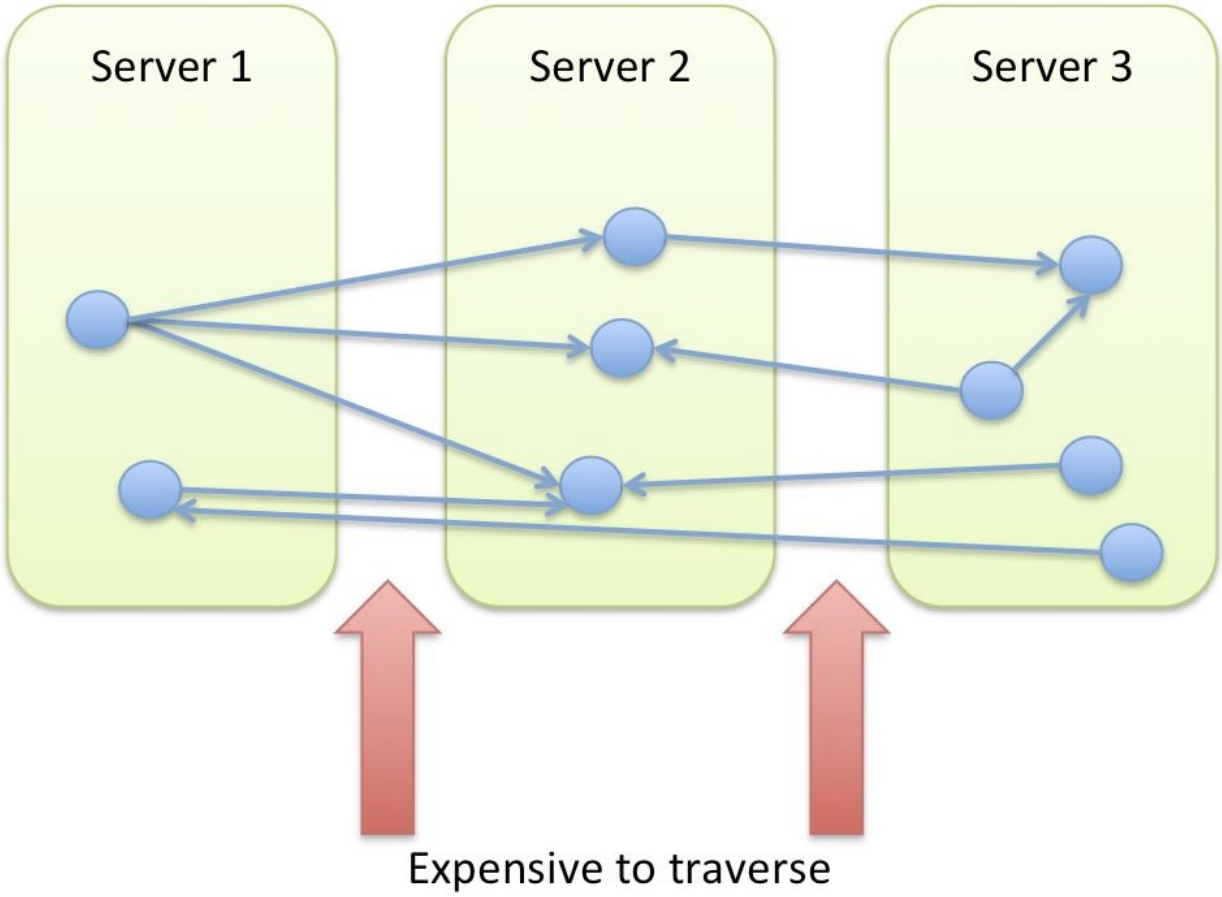


Causal cluster setup

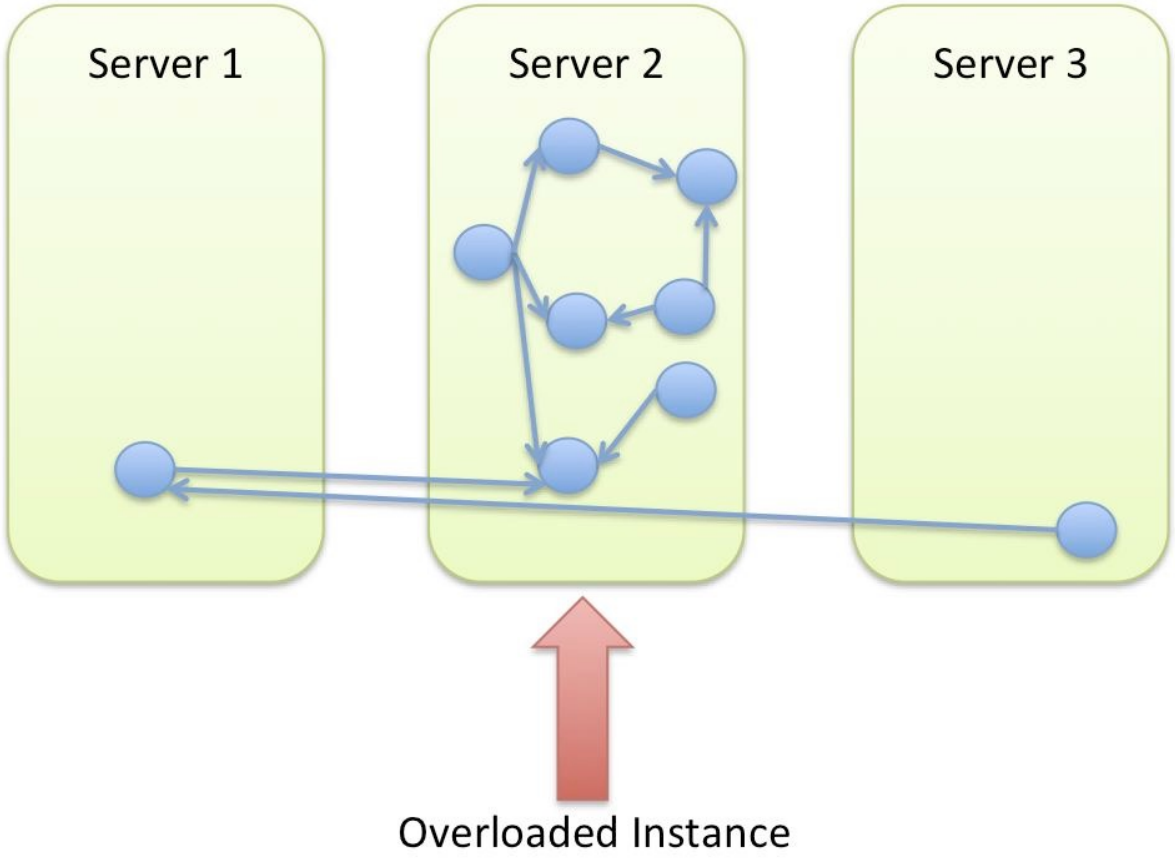


Scaling graphs is **hard**

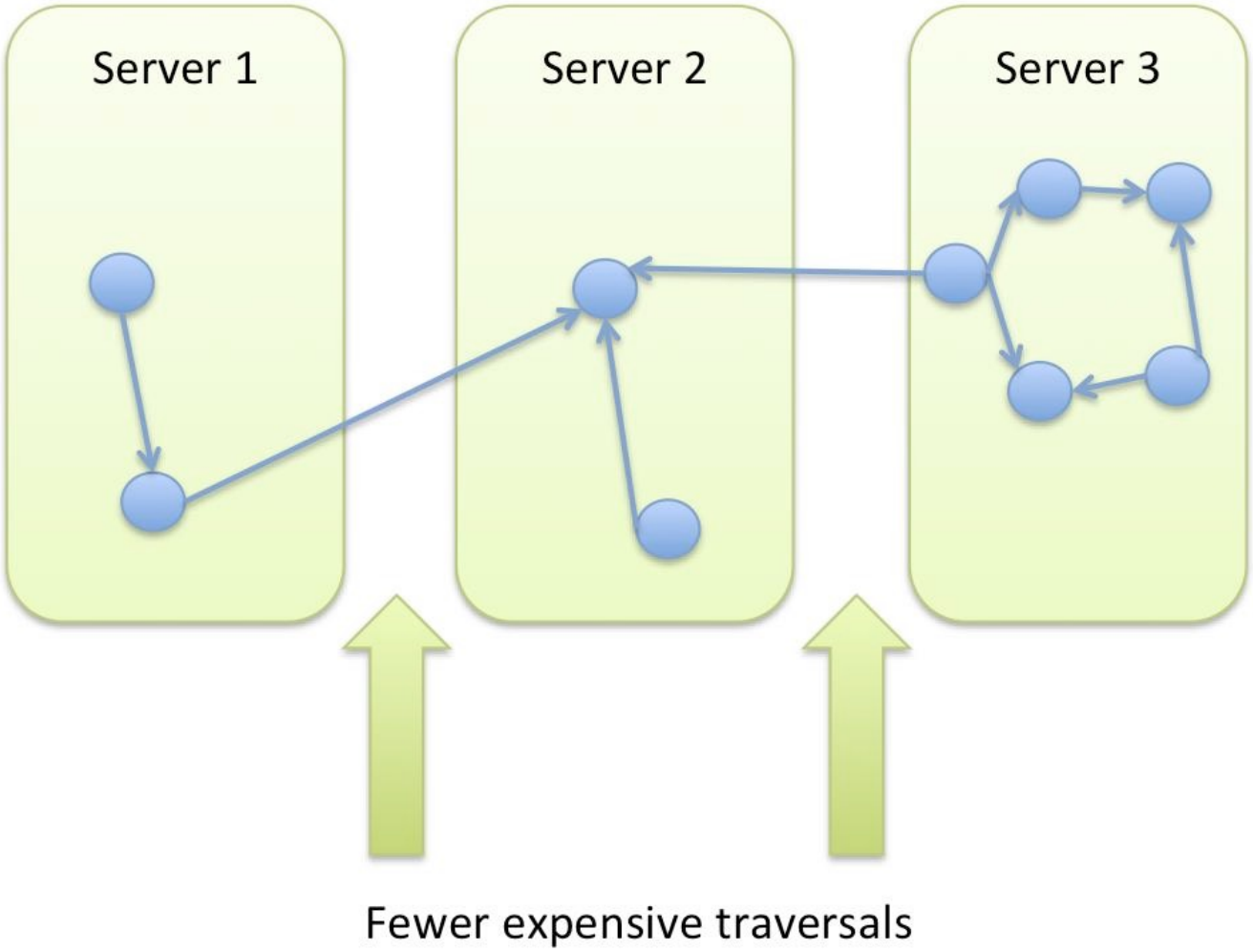
Chatty Network



“Black Hole” server



Minimal Point Cut



So how do we scale Neo4j?

Domain-specific sharding

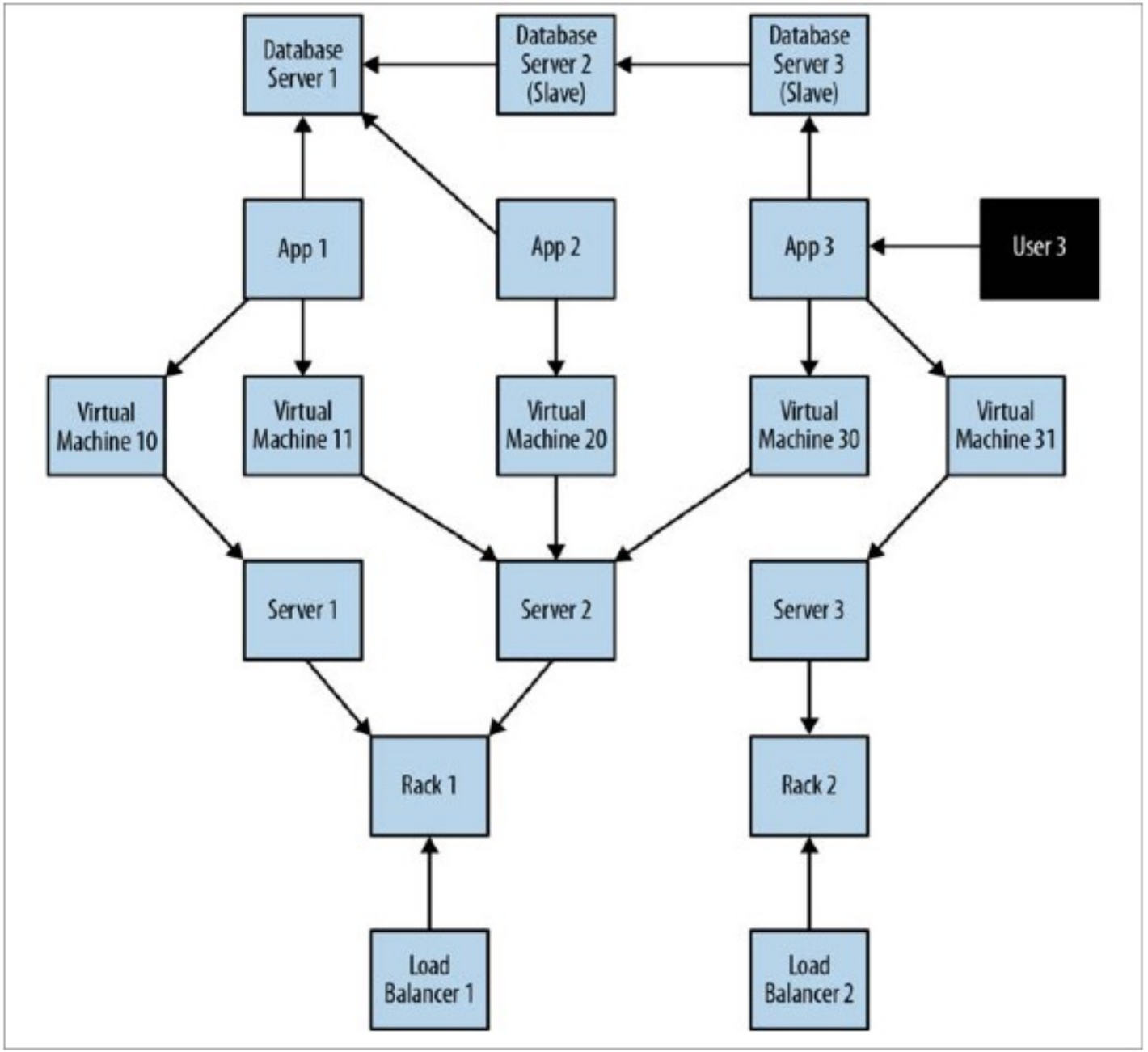
- Eventually (Petabyte) level data cannot be replicated practically
- Need to shard data across machines
- **Remember: no perfect algorithm exists**

- But we humans sometimes have *domain insight*

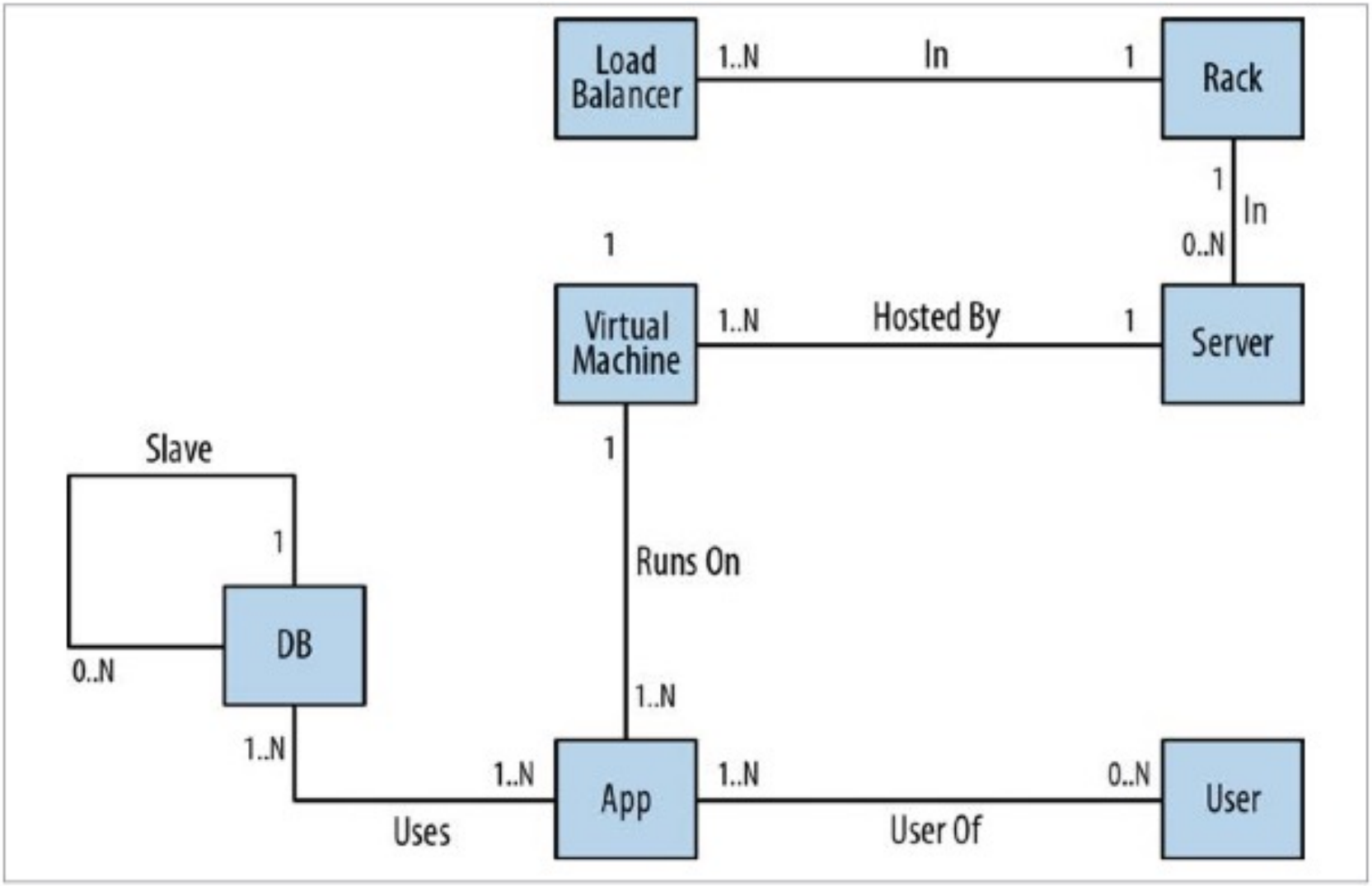
Data modeling with graphs

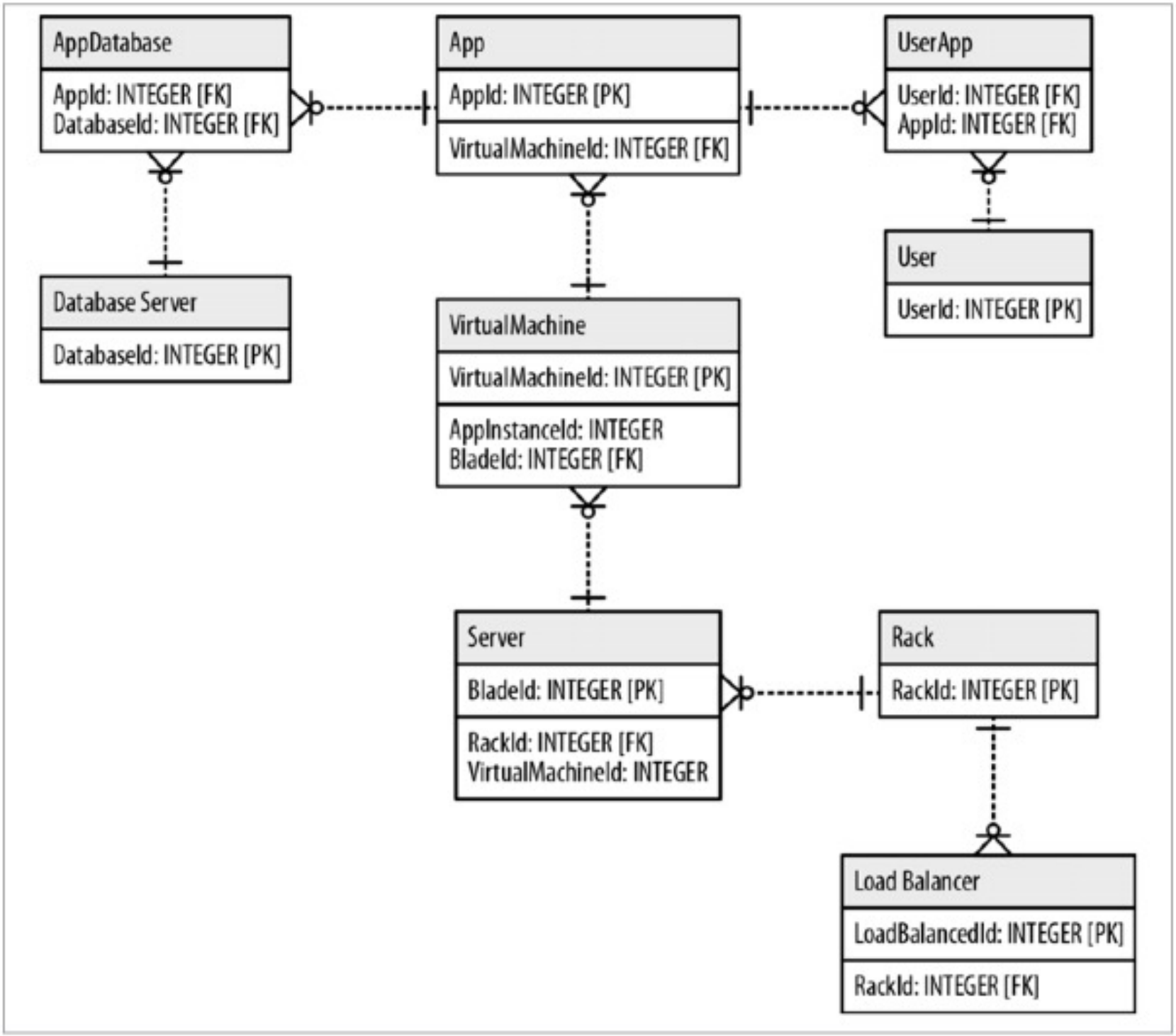
A comparison of relational and graph modelling

Simplified snapshot of application deployment within a data center

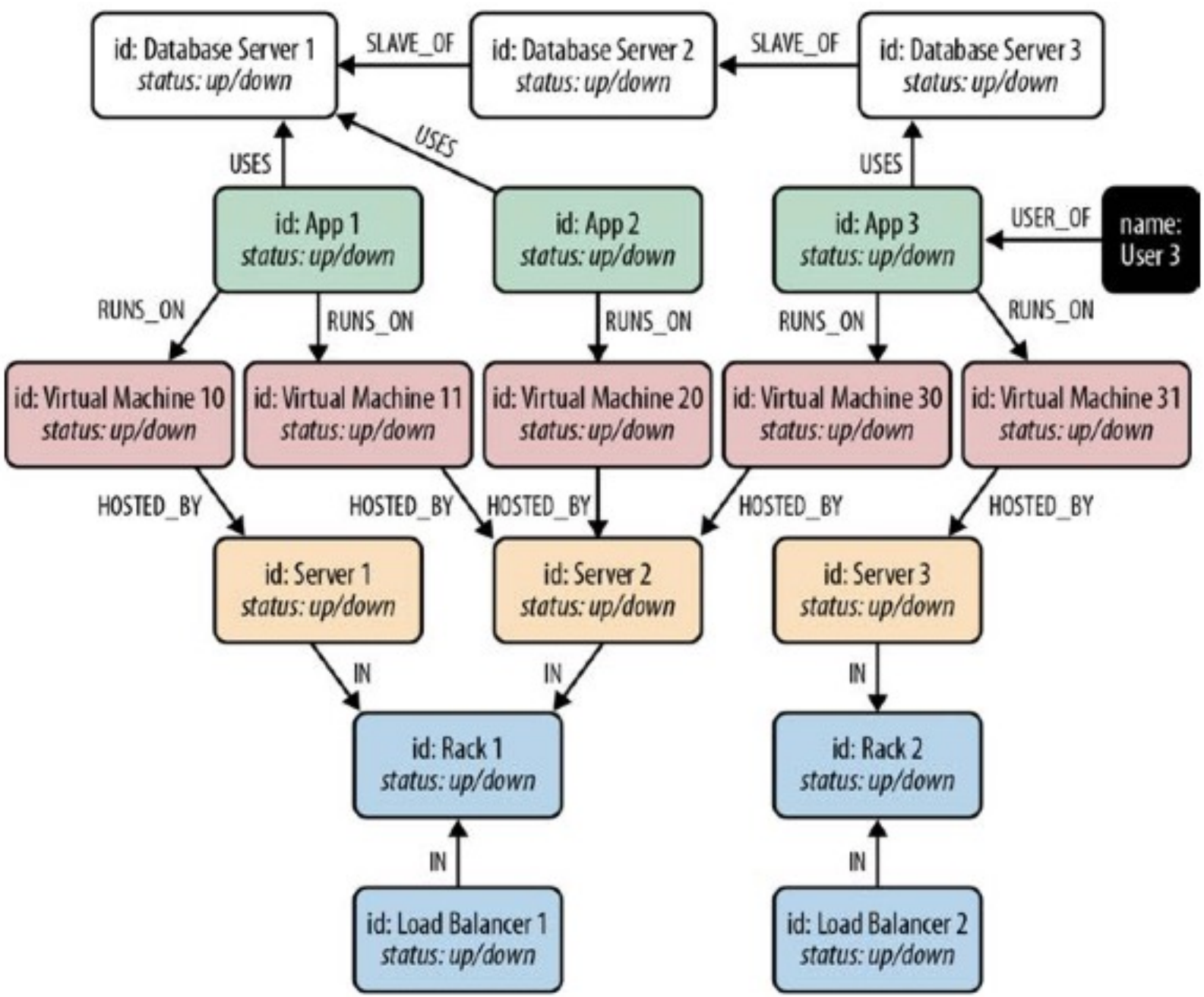


An entity-relationship diagram for the data center domain

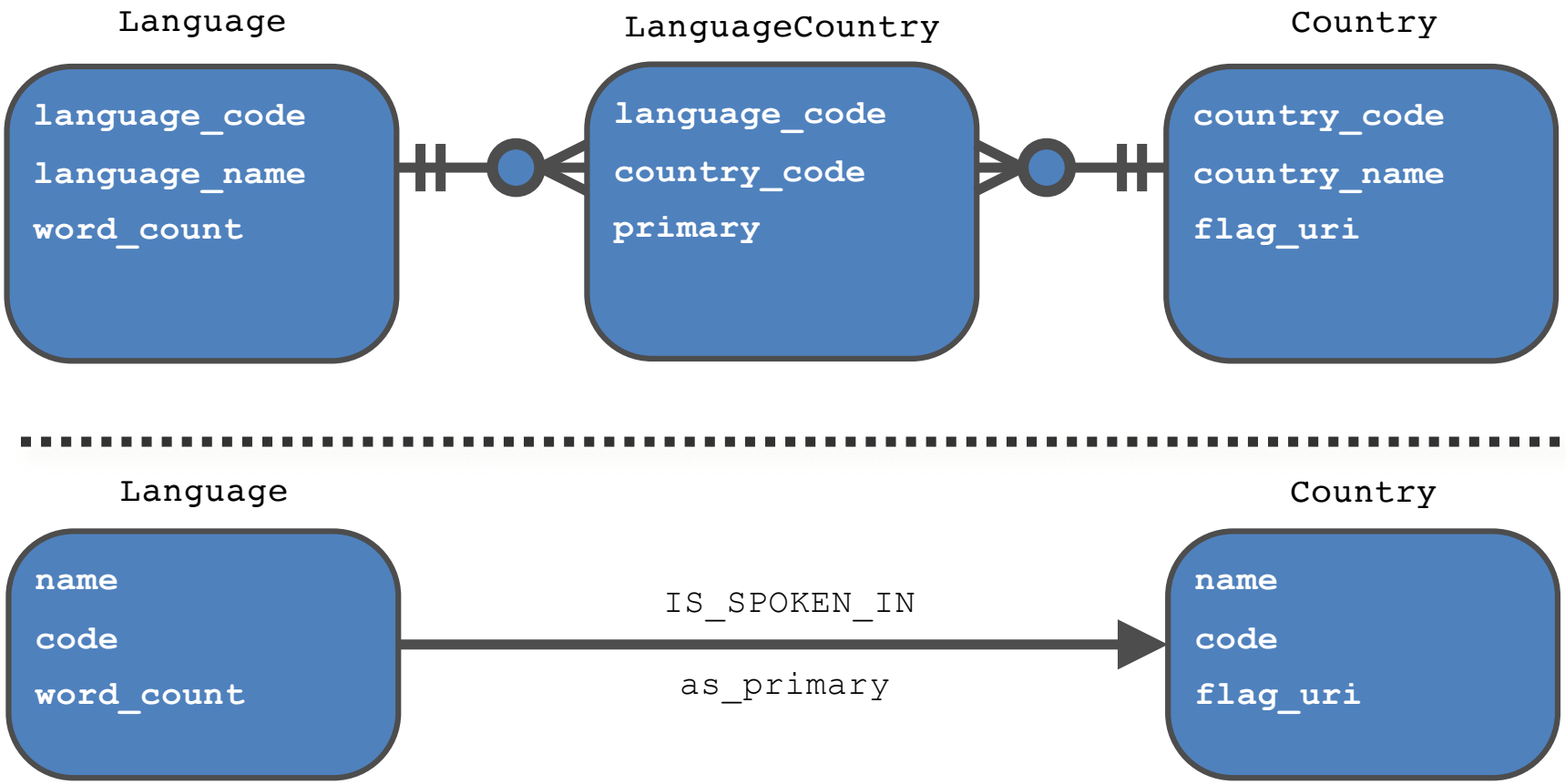


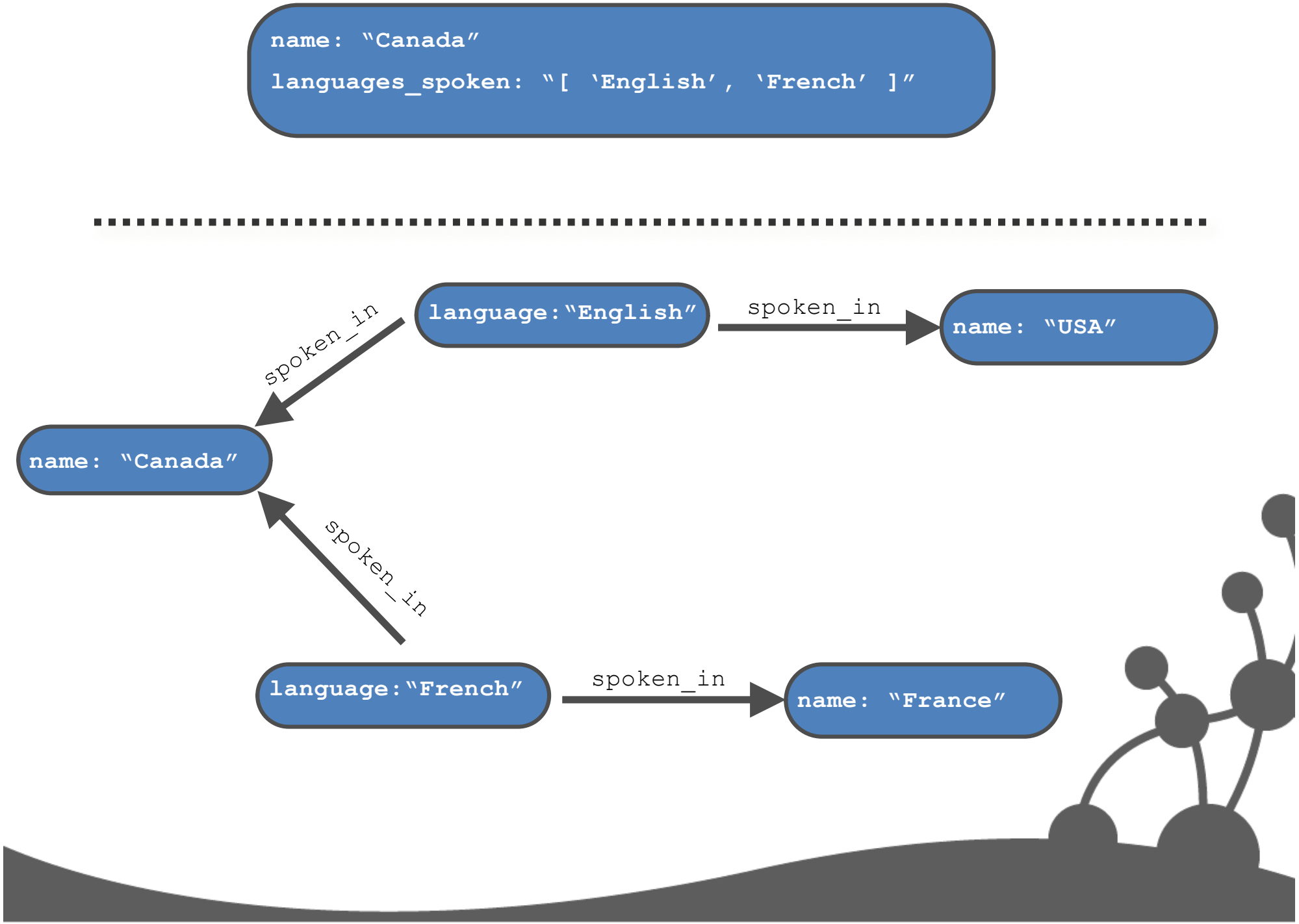

AppDatabase
AppId: INTEGER [FK]
DatabaseId: INTEGER [FK]
App
AppId: INTEGER [PK]
VirtualMachineId: INTEGER [FK]
UserApp
UserId: INTEGER [FK]
AppId: INTEGER [FK]
Database Server
DatabaseId: INTEGER [PK]
VirtualMachine
VirtualMachineId: INTEGER [PK]
AppInstanceId: INTEGER
BladeId: INTEGER [FK]
User
UserId: INTEGER [PK]
Server
BladeId: INTEGER [PK]
RackId: INTEGER [FK]
VirtualMachineId: INTEGER
Rack
RackId: INTEGER [PK]
Load Balancer
LoadBalancedId: INTEGER [PK]
RackId: INTEGER [FK]

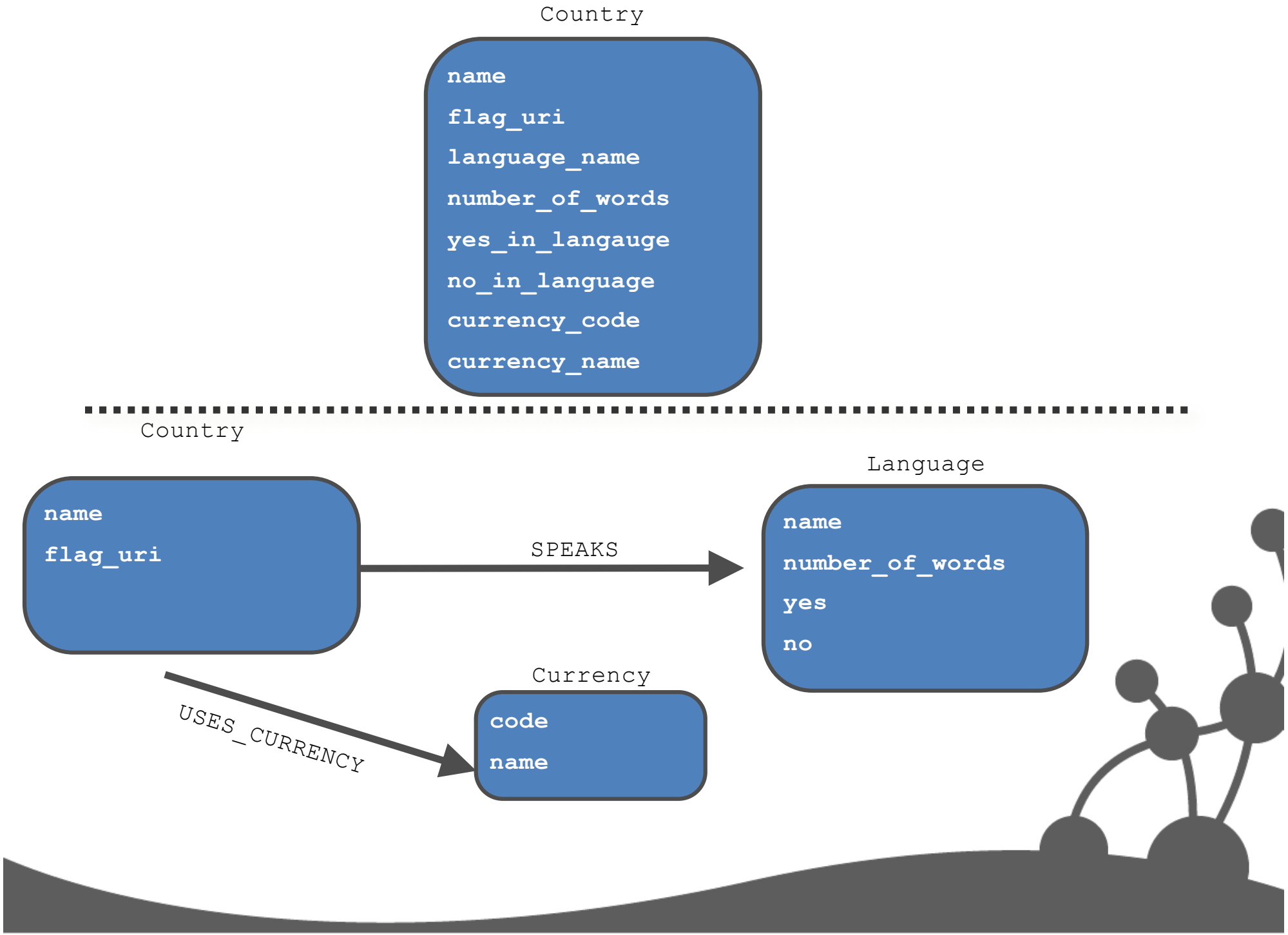
Graph for the data center application



Other example

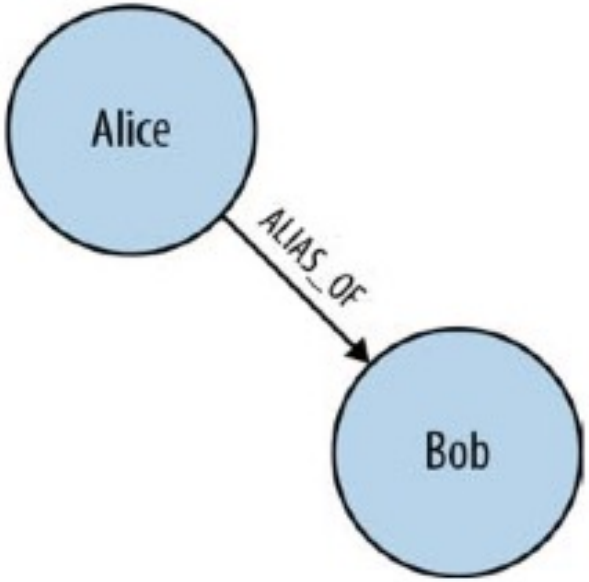



name: "Canada"
languages_spoken: "['English', 'French']"
language:"English"
spoken_in
name: "USA"
spoken_in
name: "Canada"
spoken_in
language:"French"
spoken_in
name: "France"


Country
name
flag_uri
language_name
number_of_words
yes_in_langauge
no_in_language
currency_code
currency_name
Country
name
flag_uri
SPEAKS
Language
name
number_of_words
yes
no
USES_CURRENCY
Currency
code
name

Common modeling pitfalls

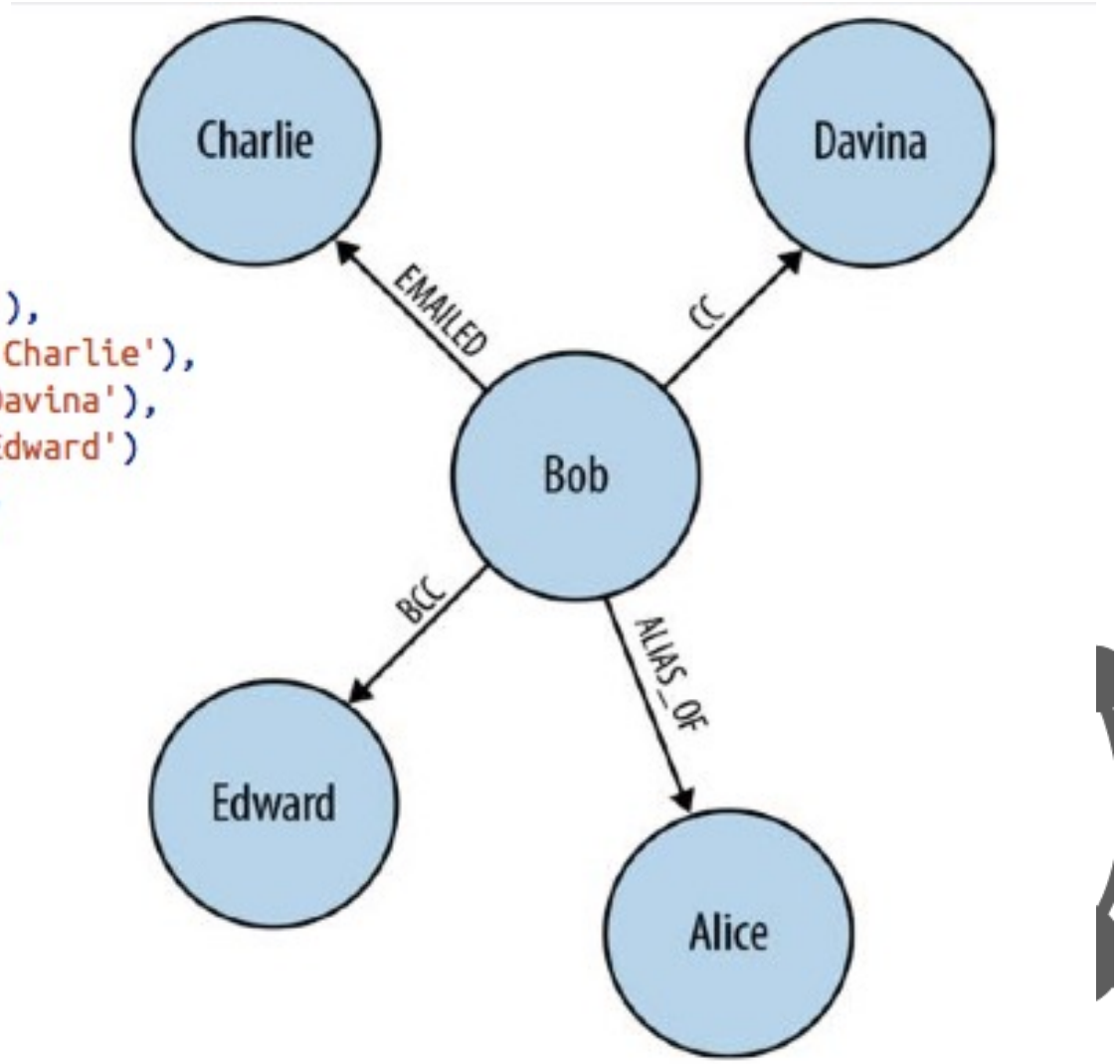
- Example: email provenance problem domain
  - *Analysis of email communications to detect suspicious patterns of email communication*
  - *Users and aliases :*



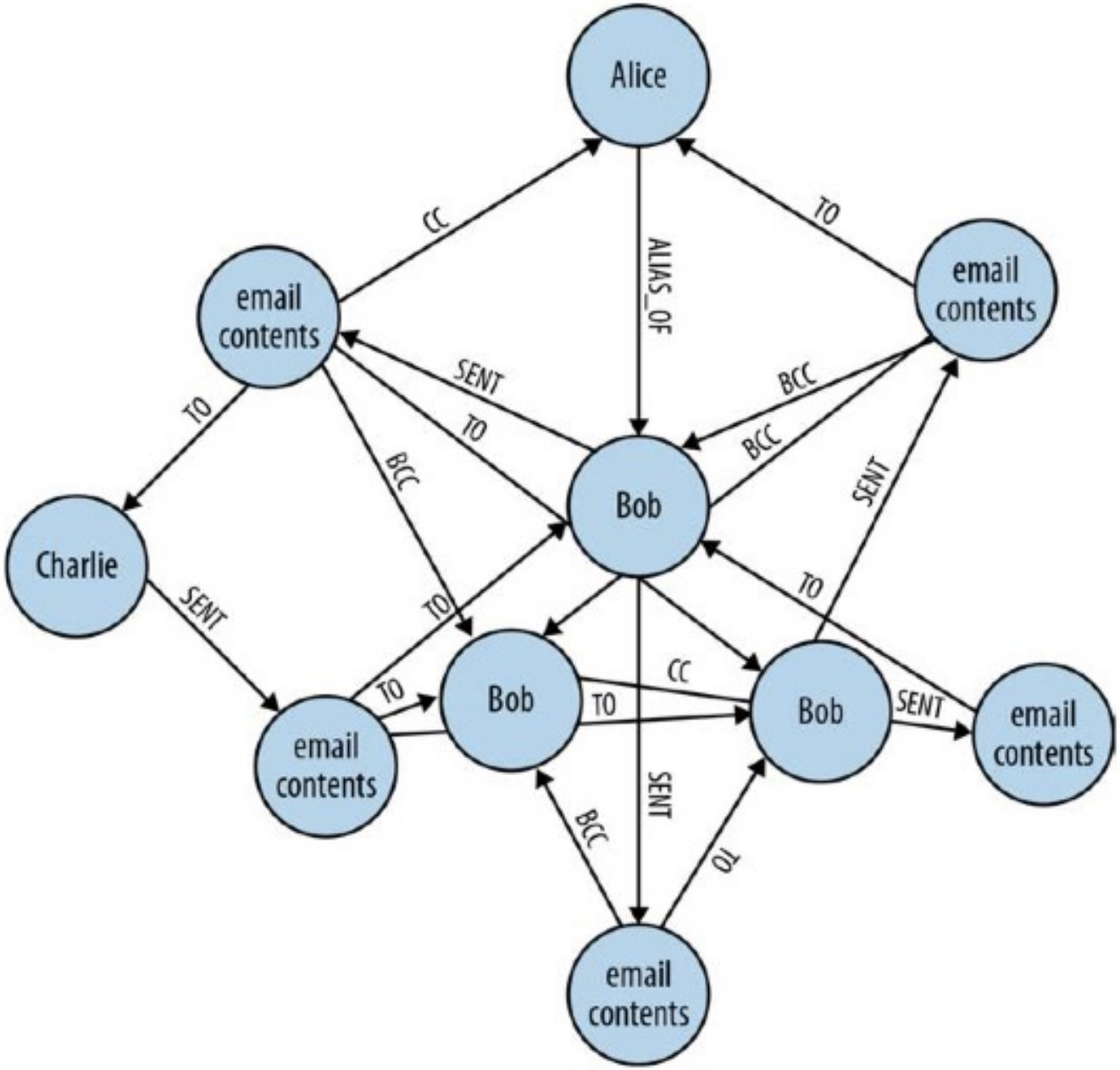
Common modeling pitfalls

- Exchanged emails

```
START  bob=node:user(username='Bob'),
       charlie=node:user(username='Charlie'),
       davina=node:user(username='Davina'),
       edward=node:user(username='Edward')
CREATE (bob)-[:EMAILED]->(charlie),
       (bob)-[:CC]->(davina),
       (bob)-[:BCC]->(edward)
```

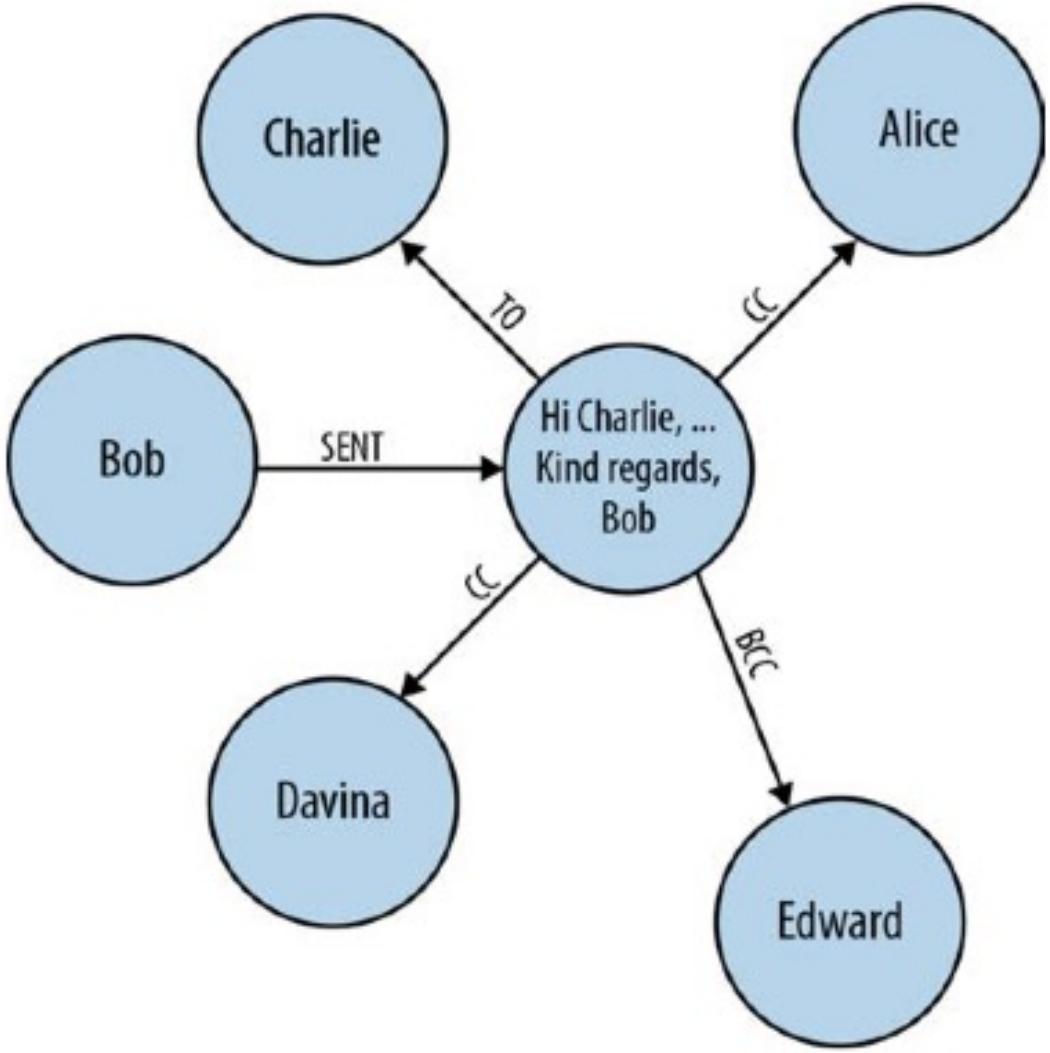


A graph of email interactions

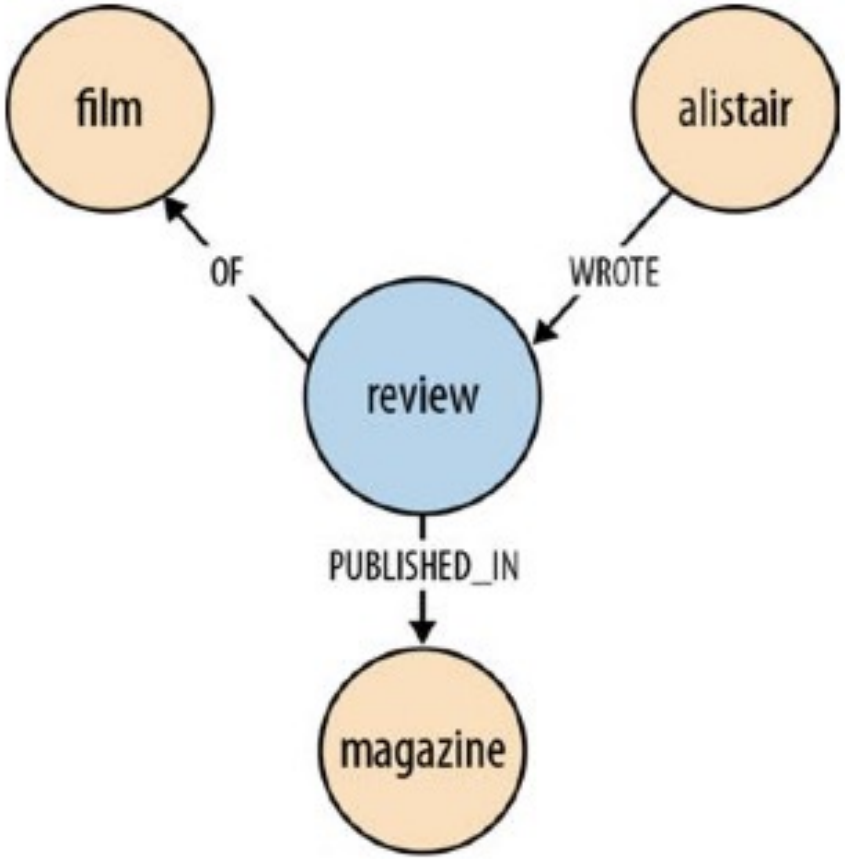


Avoiding loss of information

- Introducing an email node



Representing facts as nodes



Graph Databases popularity ranking

27 systems in ranking, November 2017

Rank Nov 2017	Rank Oct 2017	Rank Nov 2016	DBMS	Database Model	Score Nov 2017	Score Oct 2017	Score Nov 2016
1.	1.	1.	Neo4j	Graph DBMS	38.45	+0.50	+1.70
2.	2.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	13.03	+0.40	+9.78
3.	3.	↓ 2.	OrientDB	Multi-model	6.10	-0.03	+0.03
4.	4.	↓ 3.	Titan	Graph DBMS	5.68	+0.21	+0.22
5.	5.	↑ 6.	ArangoDB	Multi-model	3.15	-0.01	+0.87
6.	6.	↓ 5.	Virtuoso	Multi-model	1.88	+0.03	-0.67
7.	7.	7.	Giraph	Graph DBMS	1.02	-0.01	+0.05
8.	8.	↑ 12.	GraphDB	Multi-model	0.74	+0.10	+0.56
9.	9.	9.	AllegroGraph	Multi-model	0.60	-0.02	+0.12
10.	10.	↓ 8.	Stardog	Multi-model	0.54	+0.01	+0.01

Source :http://db-engines.com/en/ranking/graph+dbms

Source : InfiniteGraph blog

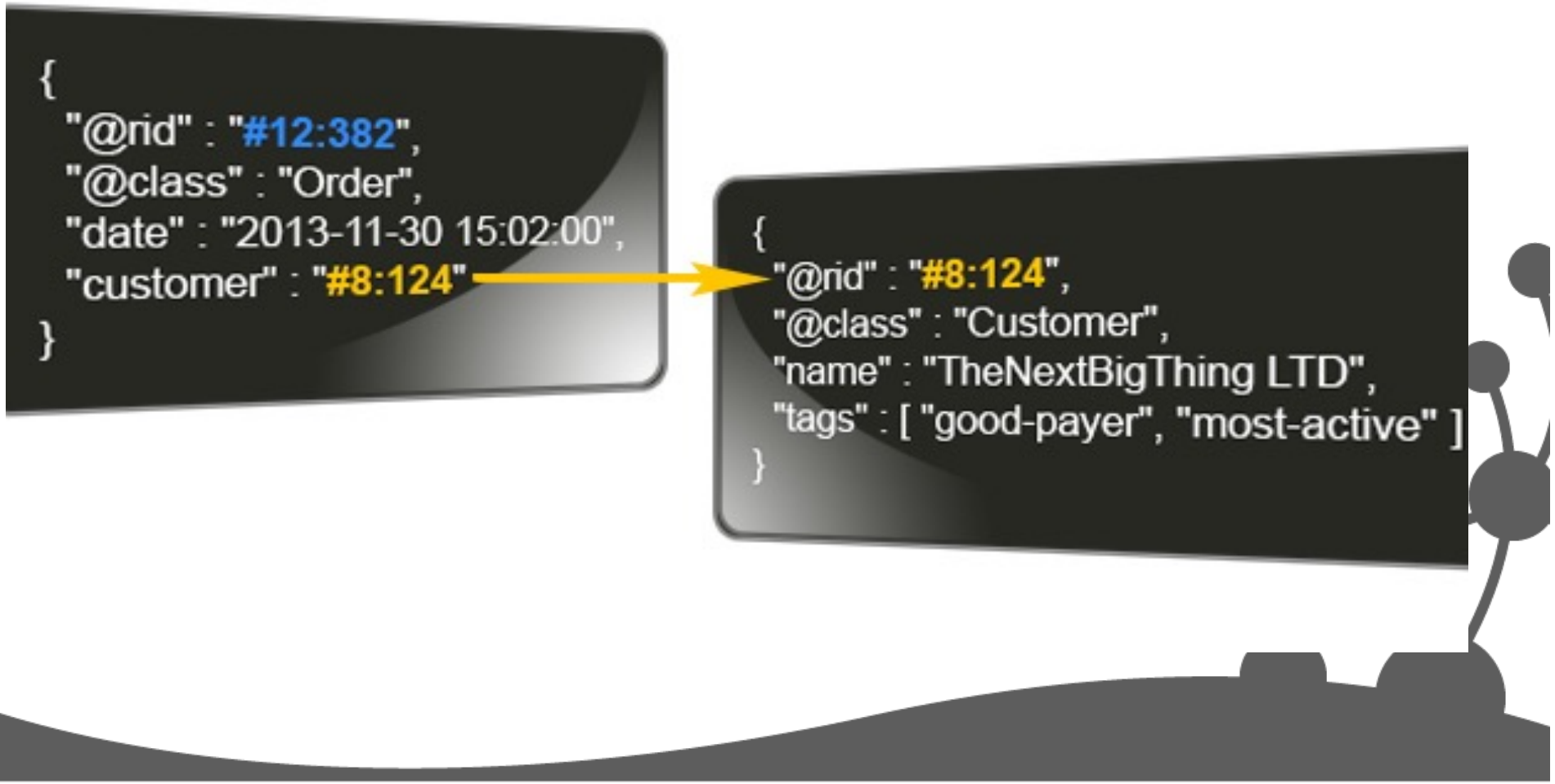
Features / Functionality	InfiniteGraph	Neo4J	AllegroGraph	Titan	FlockDB	Dex	OrientDB
Licensing	Commercial	Dual-licensed: GPLv3 and AGPLv3 / commercial	Commercial	Apache License, Version 2.0	Apache License, Version 2.0	personal evaluation use / commercial use	Apache License, Version 2.0
Source	Proprietary with open source extensions, integrated components and 3rd party connectors	Open Source	Proprietary	Open Source	Open Source	Open source	Open source
Scalability	Massively scalable distributed architecture			Dependent on 3rd party storage back end			
Graph Model	Labeled directed multi-property-graph	Directed Graph	RDF, XML	Directed graph	adjacency lists of a directed graph	Labelled, directed and attributed multi-graph	Directed graph
Schema Model	Hybrid	Property model		Property model	SQL	Property model	Document - graph
API	Java, gremlin, Blueprints	Java, gremlin, Blueprints	Java, Python, Ruby, C#, Perl, Clojure, Lisp	Gremlin, Blueprints	Java/Scala	Java, C++, .NET	Java
Query Method	Graph navigation API, predicate language qualification, Gremlin, Blueprints	Cypher (traversal), index lookup, Gremlin, Blueprints	SPARQL, Prolog, RDFS++	Gremlin, Blueprints	SQL	API	Extended SQL, Gremlin
Platforms	Windows, Linux, Mac OS X	Windows, Linux, Mac OS X	Linux x86 64bit, others as VMs or Cloud (Windows, Linux, Mac OS X, Free BSD, Solaris)		Various	Windows, Linux, Mac OS X, Free BSD, Solaris	Windows, Linux + any Java supported platforms
Consistency	Flexible (ACID to Relaxed)	ACID possible	ACID	ACID, vertex consistent, eventually consistent (dependent on 3rd party storage back end)	ACID	aCID (Partial support)	ACID
Concurrency (Distributed processing)	Concurrent non-blocking ingest & distributed queries	Single server & embedded versions. Multiple access through REST interface	All clients access server by REST protocol	Dependent on 3rd party back-end storage	Multiple application servers "flapps".	Multiple application servers	Single server & embedded versions
	Lock server and 64-bit object IDs			Faunus, Hadoop based distributing			

Platforms	Windows, Linux, Mac OS X	Windows, Linux, Mac OS X	Linux x86 64bit, others as VMs or Cloud (Windows, Linux, Mac OS X, Free BSD, Solaris)		Various	Windows, Linux, Mac OS X, Free BSD, Solaris	Windows, Linux + any Java supported platforms
Consistency	Flexible (ACID to Relaxed)	ACID possible	ACID	ACID, vertex consistent, eventually consistent (dependent on 3rd party storage back end)	ACID	aCID (Partial support)	ACID
Concurrency (Distributed processing)	Concurrent non-blocking ingest & distributed queries	Single server & embedded versions. Multiple access through REST interface	All clients access server by REST protocol	Dependent on 3rd party back-end storage	Multiple application servers "flapps".	Multiple application servers	Single server & embedded versions
Partitioning (Distributed data)	Lock server and 64-bit object IDs support dynamic addressing space (with each federation capable of managing up to 65,356 individual DBS)			Faunus, Hadoop based distributing computing framework, required to handle iterations over distributed storage backends.	Gizzard library. Horizontally scalable.	Multiple node HA enabled installation	Multiple server configuration using Hazelcast
Extensibility (Plugin Framework)	Navigators, Formatters, Visualizer	Server functionality can be extended by adding plugins.		Rexster			3rd party plugin framework
Visualizer Tool	JSON, GraphML, IG Visualizer	Web administration tool		Rexster/Frames			Visualization through 3rd party plugins, e.g. Gephi
Storage Back End/Persistency	Embedded, Disk, or Persistent SSD, Industry Standard file System Storage	Embedded, Disk Based, Proprietary file system	Industry Standard File System Storage	Apache Cassandra, Apache Hbase, Oracle Berkeley DB	MySQL	Embedded, Disk Based, Proprietary file system	Embedded, Disk Based, Proprietary file system
Language	Java, (core C++), Groovy, Scala	Jruby, Python, Django, JavaScript, Java, Groovy, Scala, Clojure	Java, Python, Ruby, Perl, C#, Clojure, Common Lisp, Sparql	Java, Ruby, Scala	Scala, Java, Ruby	C++ & Java	Java
Backup/Restore	Incremental backup, full restore	Neo4J Enterprise Edition Only	Online full backups and restores	Dependent on 3rd Party storage back- end			

Source : InfiniteGraph blog

OrientDB

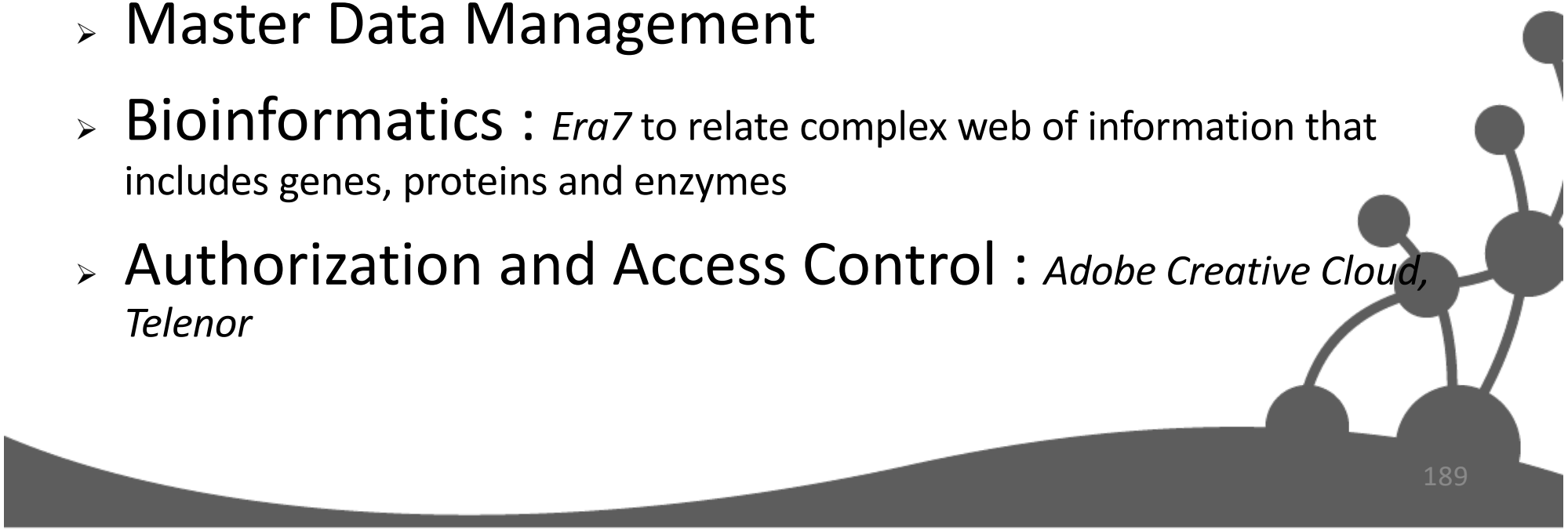
Document database that supports relationships between documents
(using persistent pointers between records)



Graphs in the Real World

Common Use Cases

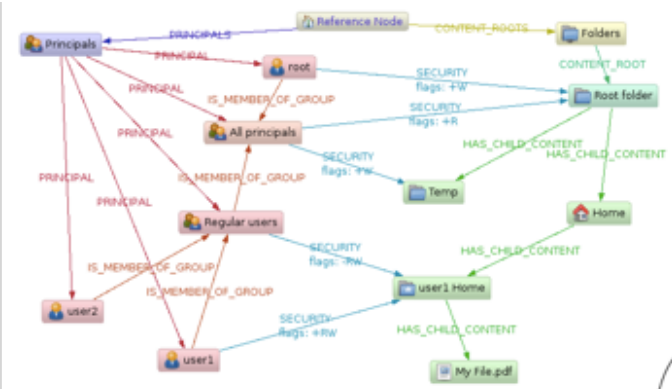
- Social
- Recommendations
- Geo
- Logistics Networks : for package routing, finding shortest Path
- Financial Transaction Graphs : for fraud detection
- Master Data Management
- Bioinformatics : *Era7* to relate complex web of information that includes genes, proteins and enzymes
- Authorization and Access Control : *Adobe Creative Cloud, Telenor*



Emergent Graph in Other Industries

(Actual Neo4j Graphs)

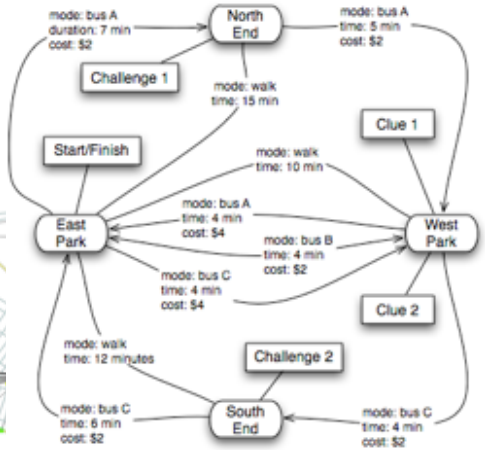
Content Management & Access Control



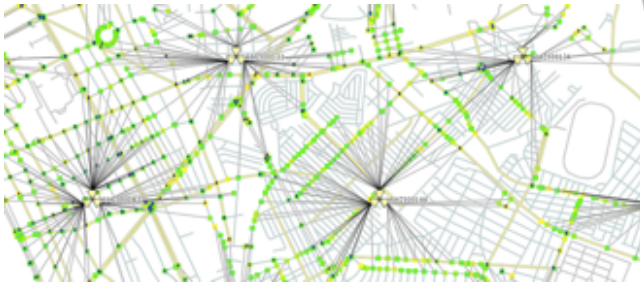
Insurance Risk Analysis

Geo Routing

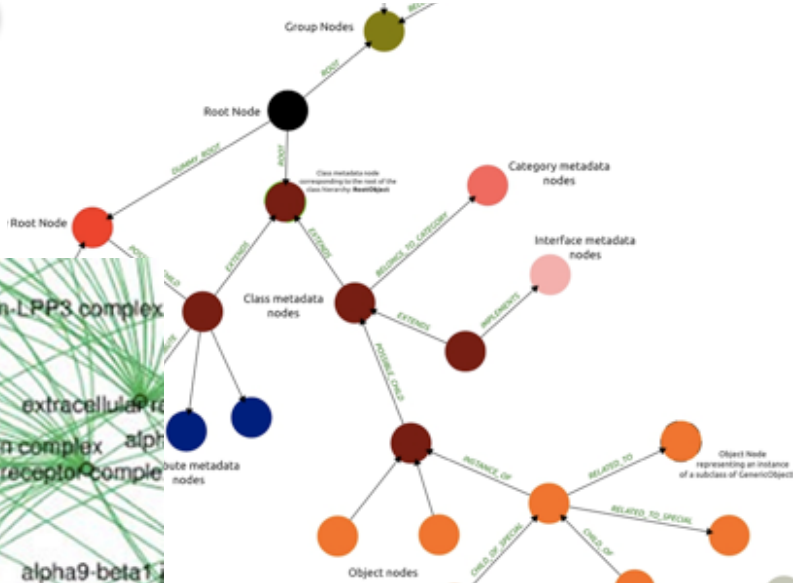
(Public Transport)



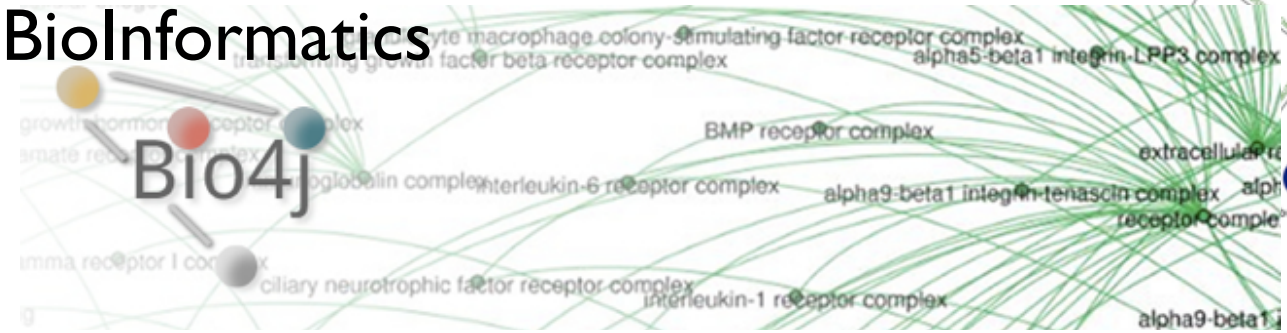
Network Cell Analysis



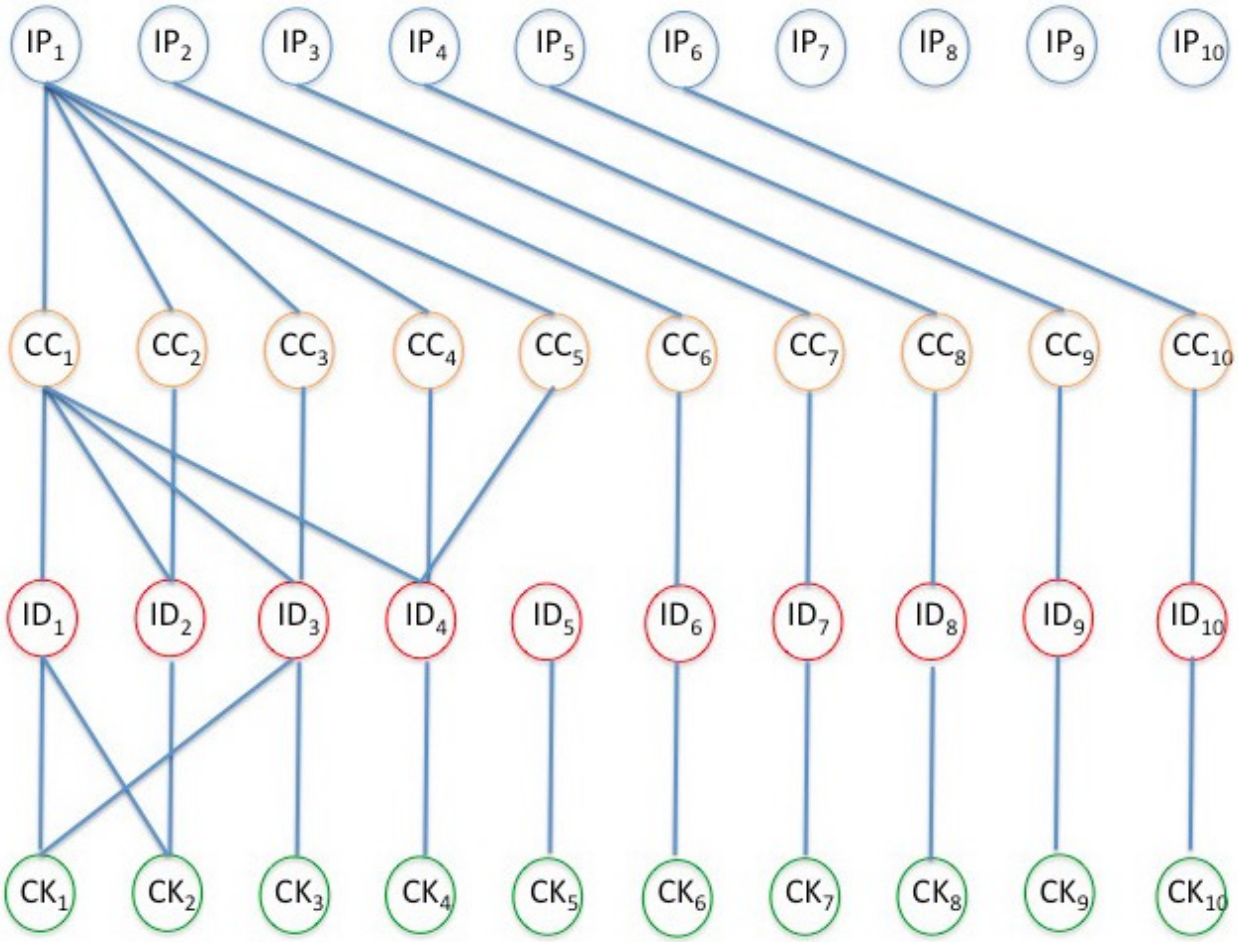
Network Asset Management



BioInformatics

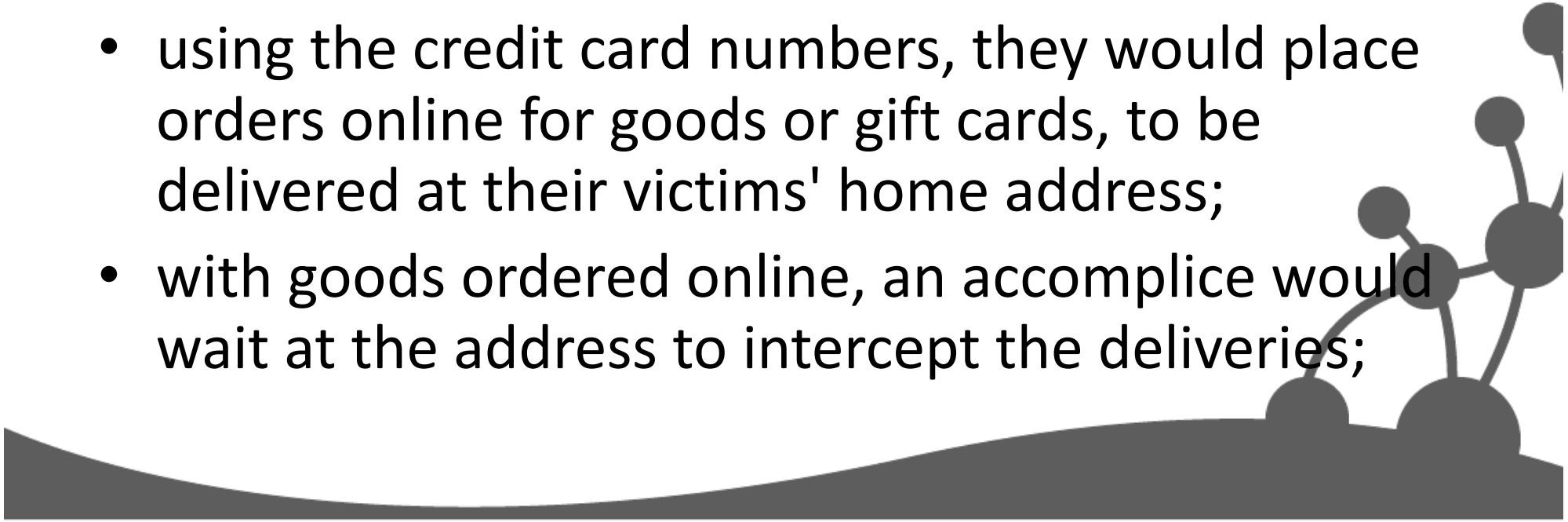


E-commerce fraud detection

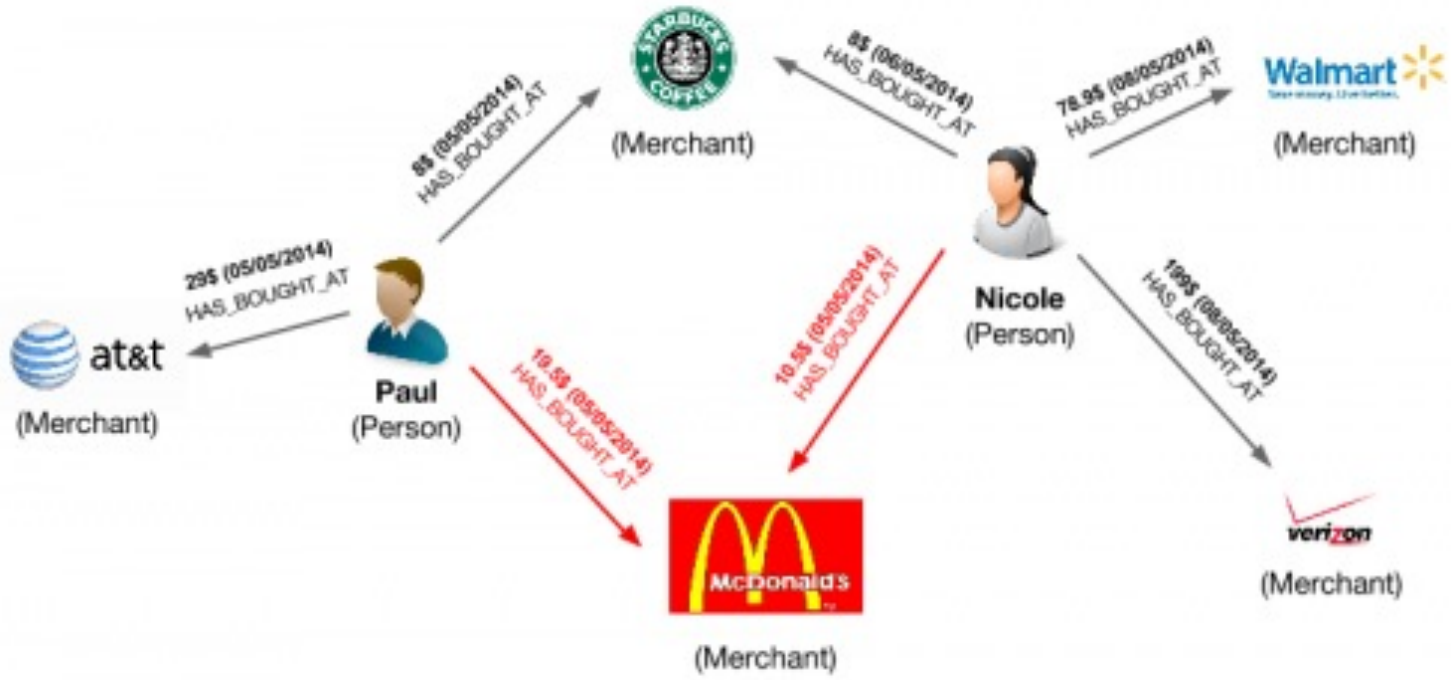


Credit Card Fraud Detection

- the Post Office clerks copied the credit card information of some of their customers while processing transactions;
- they then located these customers' home addresses;
- using the credit card numbers, they would place orders online for goods or gift cards, to be delivered at their victims' home address;
- with goods ordered online, an accomplice would wait at the address to intercept the deliveries;

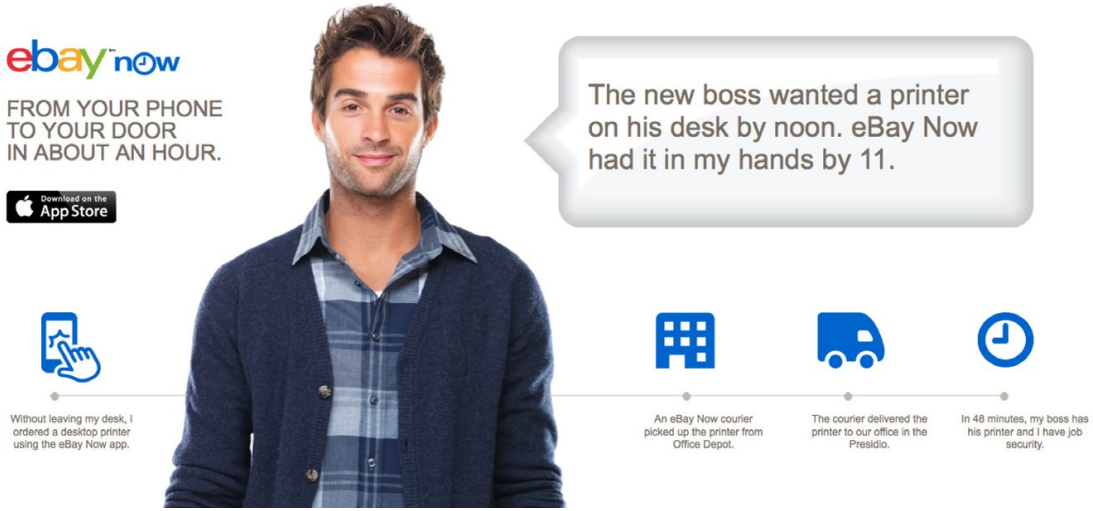


Credit Card Fraud Detection



eBay case study

- “Our Neo4j solution is literally thousands of times faster than the prior MySQL solution, with queries that require 10-100 times less code. At the same time, Neo4j allowed us to add functionality that was previously not possible.”



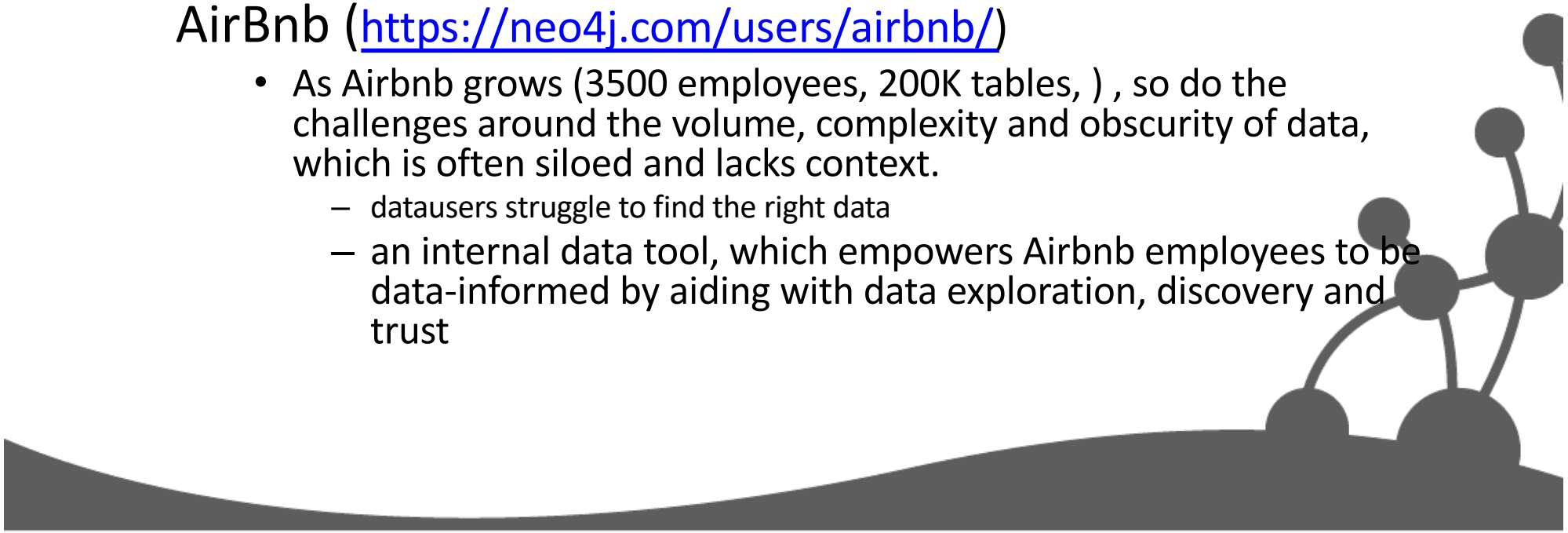
Source :https://neo4j.com/case-studies/ebay/

Use Cases

- NASA (https://neo4j.com/blog/david-meza-chief-knowledge-architect-nasa/)
  - « recording comments from astronauts as they come off the space station. This allows us to connect their experiences with those of astronauts over the last 10 or 15 years, which we can then tie to subsequent actions and projects. «

AirBnb (https://neo4j.com/users/airbnb/)

- As Airbnb grows (3500 employees, 200K tables,) , so do the challenges around the volume, complexity and obscurity of data, which is often siloed and lacks context.
  - datausers struggle to find the right data
  - an internal data tool, which empowers Airbnb employees to be data-informed by aiding with data exploration, discovery and trust



Open Cypher project

- https://www.opencypher.org/
- open query language for property graph databases.
- one of the graph languages at the basis of the graph query language (GQL) standard being developed by ISO
- Members of the openCypher Implementers Group include **Spark**, RedisGraph, AnzoGraph, and Neo4j, ..
- used by over 10 products