NOSQL, graph databases & Cypher

Advances in Data Management, 2018

Dr. Petra Selmer
Engineer at Neo4j and member of the openCypher Language Group

About me

Member of the Cypher Language Group

- Design new features for Cypher
- Manage the openCypher project

Engineer at Neo4j

- Work on the Cypher Features Team
- Maintainer of the Cypher chapter in the Neo4j Developer Manual

PhD in flexible querying of graph-structured data (Birkbeck, University of London)

Agenda

The wider landscape

NOSQL in brief

Introduction to property graph databases (in particular Neo4j)

The Cypher query language

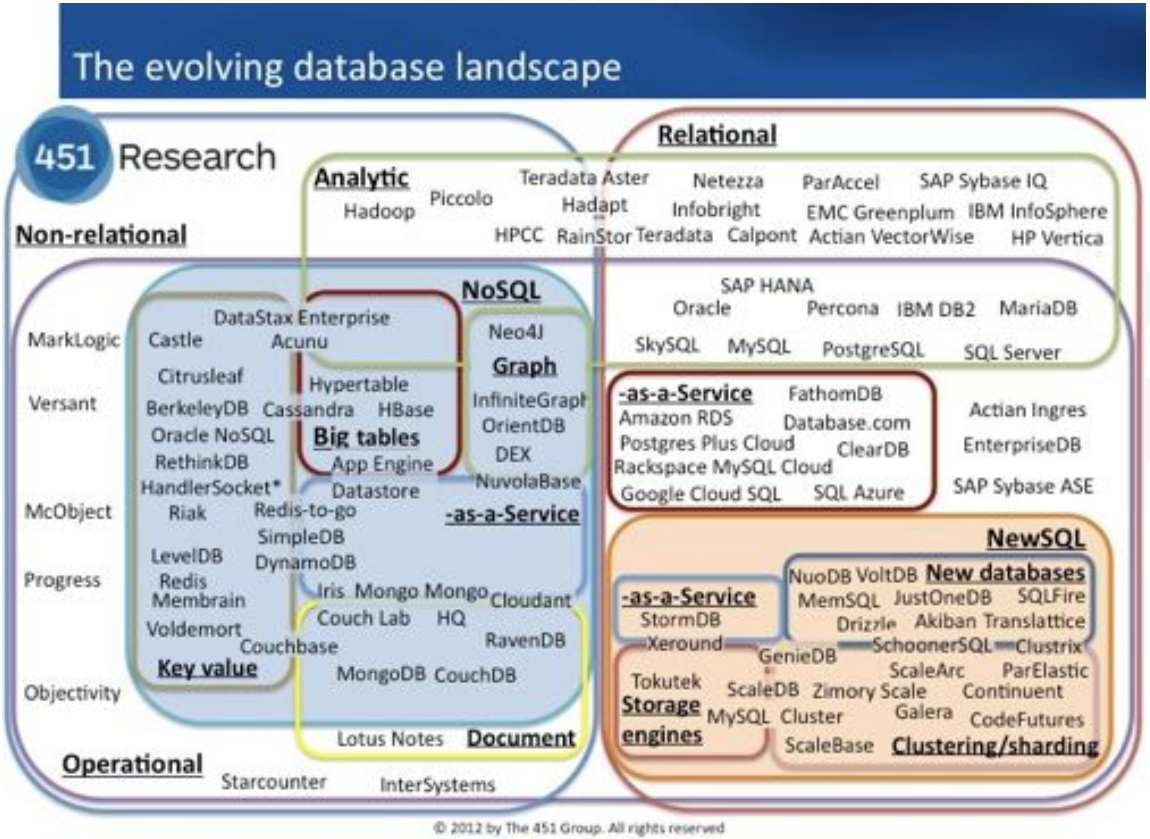
Evolving Cypher

Preamble

The area is HUGE

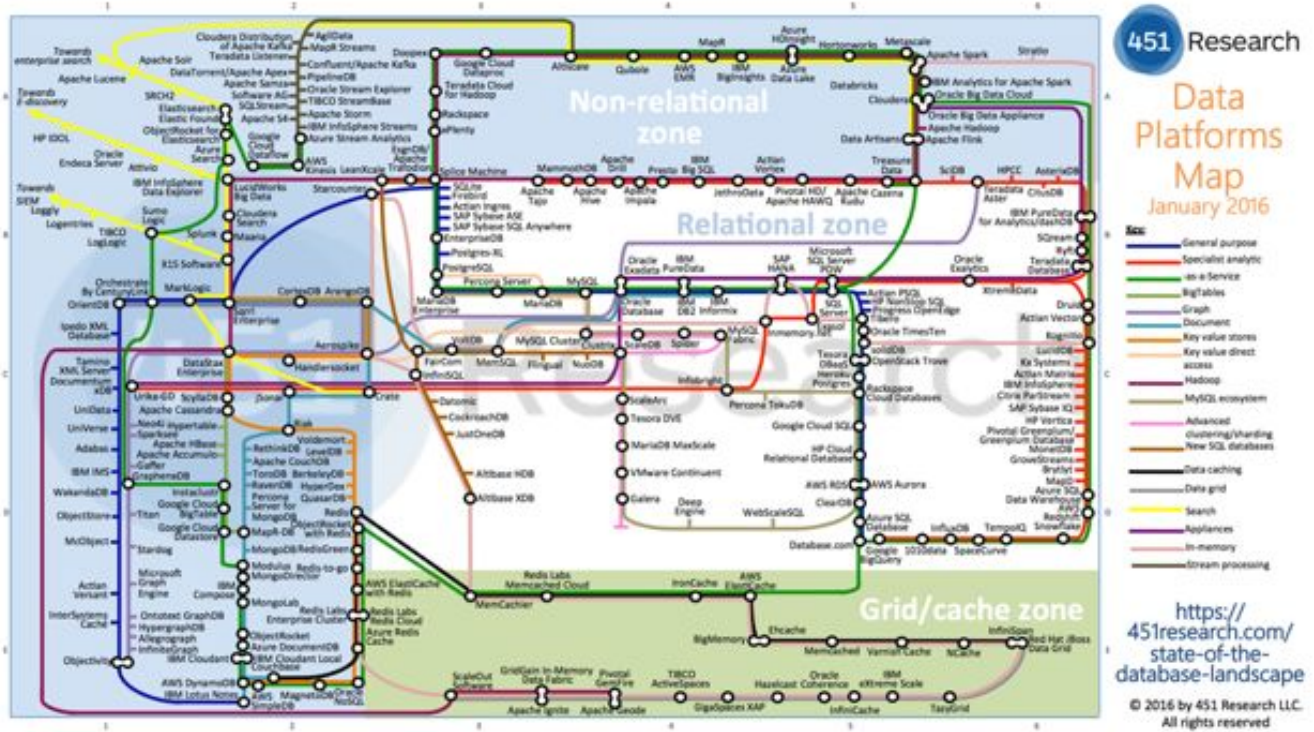
The area is ever-changing!

The wider landscape: 2012



Matthew Aslett, The 451 Group

The wider landscape: 2016



Matthew Aslett, The 451 Group

The wider landscape

Several dimensions in one picture:

Relational vs. Non-relational

Analytic (batch, offline) vs. Operational (transactional, real-time)

Increasingly difficult to categorise these data stores:

Everyone is now trying fiercely to integrate features from databases found in other spaces.

The emergence of “multi-model” data stores:

One may start with one data model and add other models as new requirements emerge.

A brief tour of NOSQL

NOSQL: non-relational

NOSQL: “Not Only SQL”, not “No SQL”

Basically means “not relational” – however this also doesn't quite apply, because graph data stores are very relational; they just track different forms of relationships than a traditional RDBMS.

A more precise definition would be the union of different data management systems differing from Codd’s classic relational model

NOSQL: non-relational

The name is not a really good one, because some of these support SQL and SQL is really orthogonal to the capabilities of these systems. However, tricky to find a suitable name.

A good way to think of these is as “the rest of the databases that solve the rest of our problems”

Scalability:

Horizontal (scale out): the addition of more nodes (commodity servers) to a system (cluster) - simple NOSQL stores

Vertical (scale up): the addition of more resources – CPU, memory – to a single machine

Non-relational vs. relational

What's wrong with relational DBs? They're great!

ACID

Enforcement of referential integrity and constraints

SQL

Excellent support by many languages and technology stacks

Excellent tooling

Well-understood operational processes (DBAs): backups, recovery, tuning etc

Good security management (user access, groups etc)

Problems with relational

Scaling with large and high-velocity data

- ‘Big Data’
- Expensive / difficult / impossible to scale reads and writes vertically and horizontally

Complexity of data

- Impedance mismatch
- Performance issues (joins)
- Difficult to develop and maintain

Problems with relational

Schema flexibility and evolution

- Not trivial
- Application downtime

Non-relational

Not intended as a replacement for RDBMS

One size doesn’t fit all

Use the right tool for the job

Non-relational

Today's data problems are getting complicated: the scalability, performance (low latency), and volume needs are greater.

In order to solve these problems, we're going to have to use an alternative data store or use more than one database technology.

Relational vs. Aggregate Data Model

Relational

Data are divided into rows (tuples) with pre-defined columns (attributes)

There is no nesting of tuples

There is no list of values

Aggregate

Think of this as a collection of related objects, which should be treated as a unit

Relational vs. Aggregate Data Model

Relational Instance

CUSTOMER

ID	NAME
1	Guido

PRODUCT

ID	NAME
1000	IPod Touch
1020	Monster Beat

BILLING_ADDRESS

ID	CUSTOMER_ID	ADDRESS_ID
1	1	55

ADDRESS

ID	STREET	CITY	POST_CODE
55	Chaumontweg	Spiegel	3095

ORDER

ID	CUSTOMER_ID	SHIPPING_ADDRESS_ID
90	1	55

ORDER_ITEM

ID	ORDER_ID	PRODUCT_ID	PRICE
1	90	1000	250.55
1	90	1020	199.55

Aggregate Instance

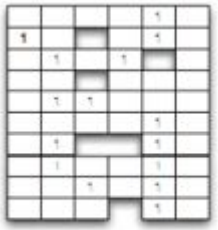
```
{
„id":1,
„name":"Guido",
„billingAddress":[{„street":"Chaumontweg","city":"Spiegel","postCode":"3095"}]
}

{
„id":90,
„customerId":1,
„orderItems":[
 {
 „productId":1000,"price": 250.55, „produtName": „iPod Touch"
 },
 {
 „productId":1020,"price": 199.55, „produtName": „Monster Beat"
 }],
„sippingAddress":[{„street":"Chaumontweg","city":"Spiegel","postCode":"3095"}]
}
```

Non-relational families

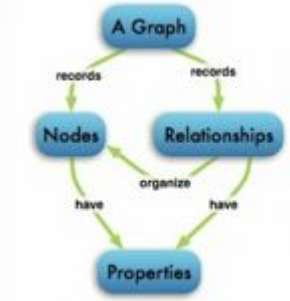
Key-Value

Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.



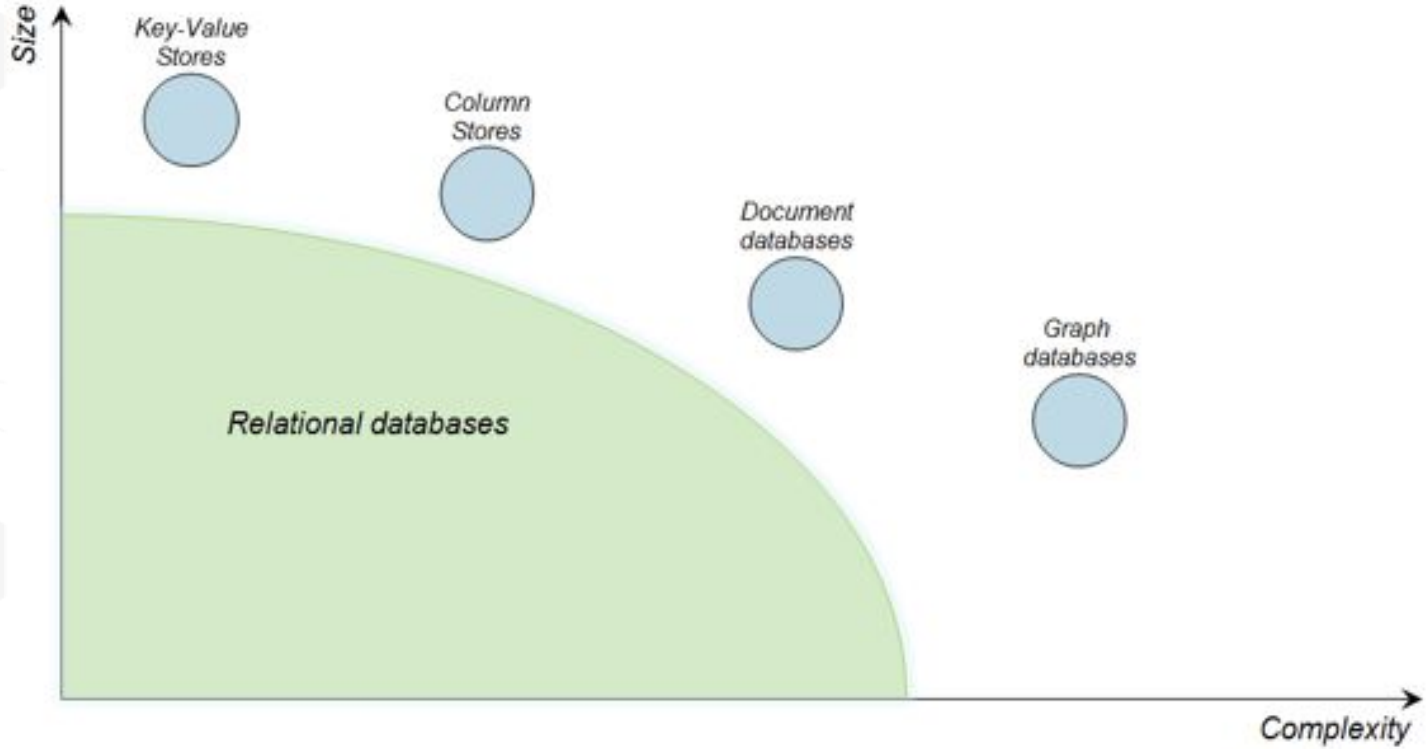
Column Store

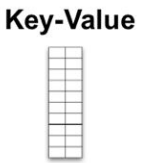


Non-relational families

Store	Key/Value	Column	Document	Graph
Design	Key/Value pairs; indexed by key	Columns and Column Families. Directly accesses the column values	Multiple Key/Value pairs form a document. Values may be nested documents or lists as well as scalar values	Focus on the connections between data and fast navigation through these connections
Scalability	+++	+++	++	++
Aggregate-oriented	Yes	Yes	Yes	No
Complexity	+	++	++	+++
Inspiration/ Relation	Berkley DB, Memcached, Distributed Hashmaps	SAP Sybase IQ, Google BigTable	Lotus Notes	Graph theory
Products	Voldemort, Redis, Riak(?)	HBase, Cassandra, Hypertable	MongoDB, Couchbase	Neo4j, DataStax Enterprise Graph

Non-relational families





Key/Value stores

A key-value store is a simple hash table

Generally used when all access to the data is via a primary key

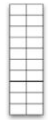
Simplest non-relational data store

Value is a BLOB data store does not care or necessarily know what is ‘inside’

Use cases

- Storing Session Information
- User Profiles, Preferences
- Shopping Cart Data
- Sensor data, log data, serving ads

Key-Value



Key/Value stores

Strengths

Simple data model

Great at scaling out horizontally for reads and writes

Scalable

Available

No database maintenance required when adding / removing columns

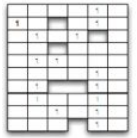
Weaknesses

Simplistic data model – moves a lot of the complexity of the application into the application layer itself

Poor for complex data

Querying is simply by a given key: more complex querying not supported

Column stores



Rows are split across multiple nodes through sharding on the primary key

A big table, with column families. Column families are groups of related data, often accessed together

Example (see diagram):

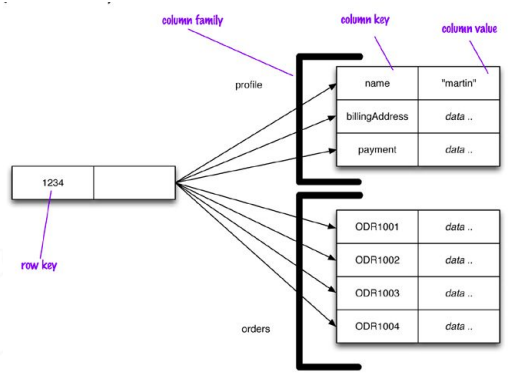
One row for Customer 1234

Customer table partitioned into 2 column families: profile and orders

Each column family has columns (e.g. name and payment) and supercolumns (have a name and an arbitrary number of associated columns)

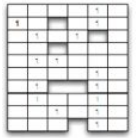
Each column family may be treated as a separate table in terms of sharding:

Profile for Customer 1234 may be on Node 1, orders for Customer 1234 may be on Node 2



Source: NOSQL Distilled

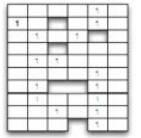
Column stores



Use cases

- Logging and customer analytics
- Event Logging
- Counters
- Smart meters and monitoring
- Sensor data

Column stores



Strengths

- Data model supports (sparse) semi-structured data
- Naturally indexed (columns)
- Good at scaling out horizontally
- Can see results of queries in real time

Weaknesses

- Unsuited for interconnected data
- Unsuited for complex data reads and querying
- Require maintenance – when adding / removing columns and grouping them
- Queries need to be pre-written; no ad-hoc queries defined “on the fly”

Document stores

```
{
  person: {
    first_name: "Peter",
    last_name: "Peterson",
    addresses: [
      {street: "123 Peter St"},
      {street: "504 Not Peter St"}
    ],
  }
}
```

Collections of documents

A document is a key-value collection

Stores and retrieves documents, which can be XML, JSON, BSON..

Documents are self-describing, hierarchical tree data structures which can consist of maps, collections and scalar values, as well as nested documents

Documents stored are similar to each other but do not have to be exactly the same

Document stores

Use cases

- High Volume Data Feeds
- Tick Data capture
- Risk Analytics & Reporting
- Product Catalogs & Trade Capture
- Portfolio and Position Reporting
- Reference Data Management
- Portfolio Management
- Quantitative Analysis
- Automated Trading

Document stores

Strengths

Simple but powerful data model – able to express nested structures

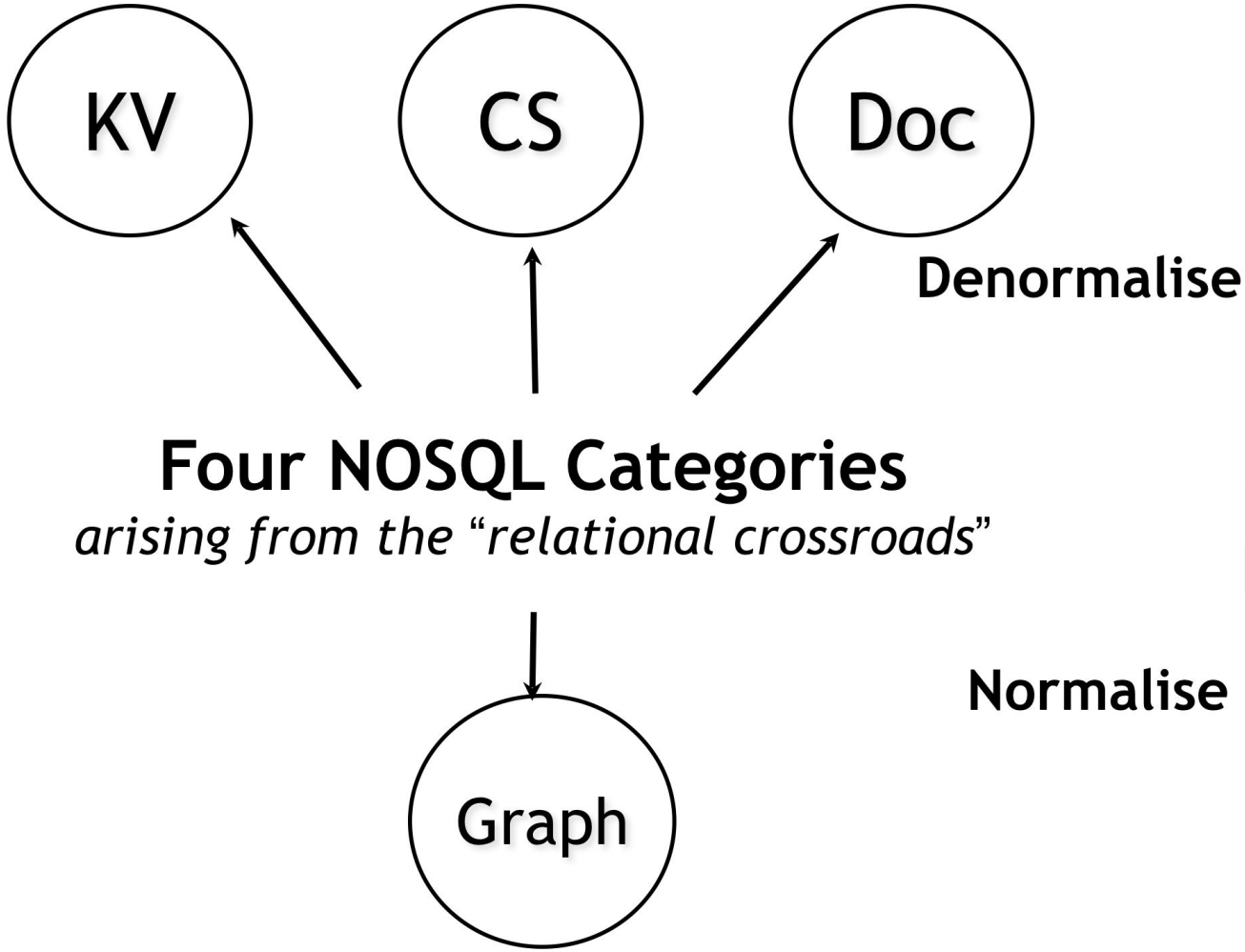
Good scaling (especially if sharding supported)

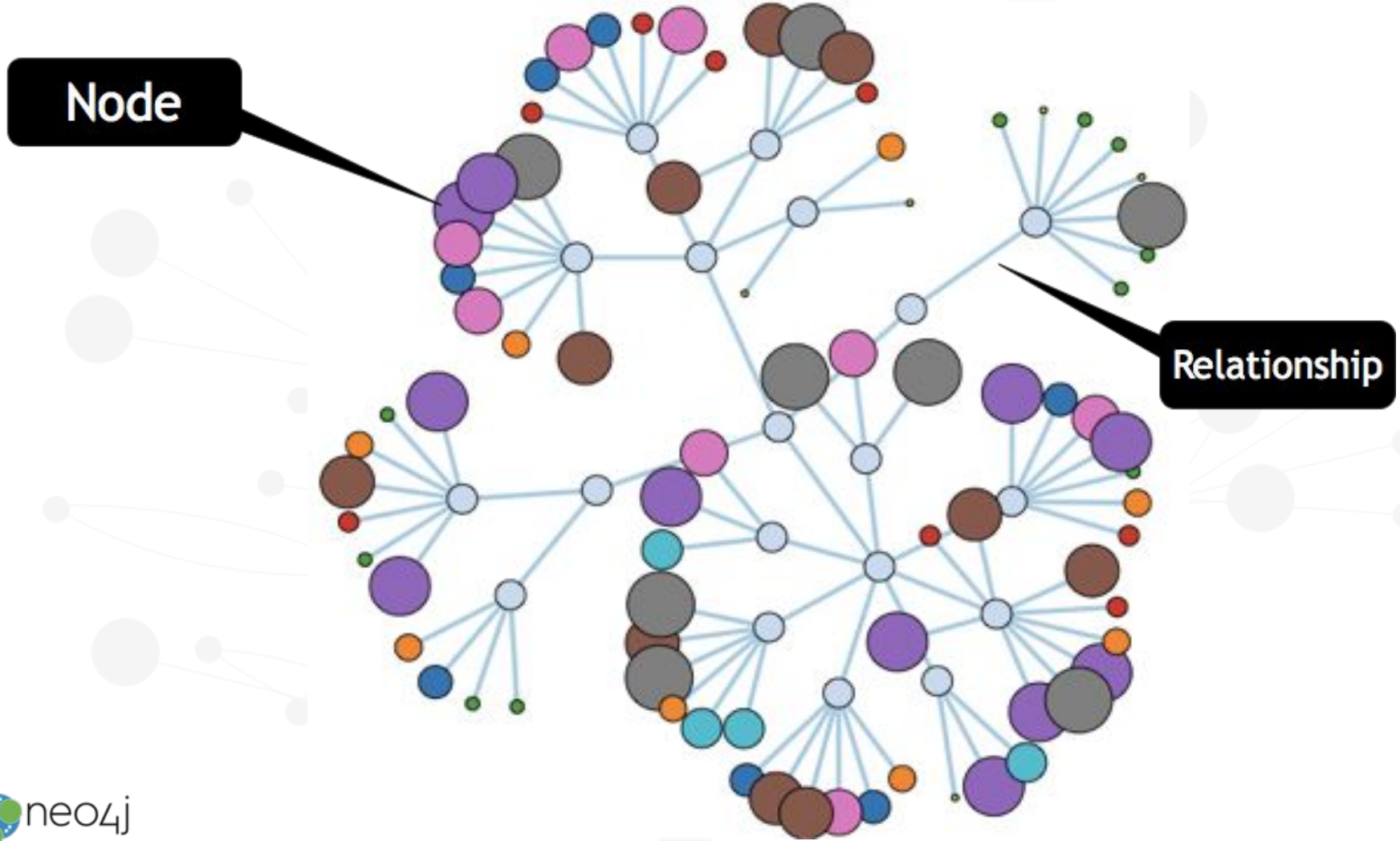
No database maintenance required to add / remove ‘columns’

Powerful query expressivity (especially with nested structures) – able to pose fairly sophisticated queries

Weaknesses

Unsuited for interconnected data


KV
CS
Doc
Denormalise
Four NOSQL Categories
arising from the "relational crossroads"
Normalise
Graph



neo4j

Graph stores

"Odd man out" in the non-relational group: not aggregate-oriented

Designed for COMPLEX data – richer data, a lot of expressive power

Data model – nodes and edges:

- Nodes
- Edges are named relationships between nodes

A query on the graph is also known as traversing the graph: traversing the relationships is very fast

Graph theory:

- People talk about Codd's relational model being mature because it was proposed in 1969: 49 years old.
- Euler's graph theory was proposed in 1736: 282 years old!

Semantic Web technologies: RDF, ontologies, triple stores and SPARQL

Graph stores

Strengths

*complexity = f(size, variable structure, **connectedness**)*

Powerful data model

Fast

For connected data, can be many orders of magnitude faster than RDBMS

Good, well-established querying models: Cypher, SPARQL and Gremlin

Schema-optional model

Weaknesses

If the data has no / few connections, there is not much benefit in using a graph database

Graph stores: use cases

Connected data

Hierarchical data

Recommendation engines, Business intelligence

Network impact analysis, Social computing, Geospatial

Systems management, web of things / Internet of things

Genealogy

Product catalogue, Access Control

Life Sciences and scientific computing (especially bioinformatics)

Routing, Dispatch, Logistics and Location-Based Services

Financial services – finance chain, dependencies, risk management, fraud detection etc. For example, if you want to find out how vulnerable a company is to a bit of "bad news" for another company, the directness of the relationship can be a critical calculation. Querying this in several SQL statements takes a lot of code and won't be fast, but a graph store excels at this task.

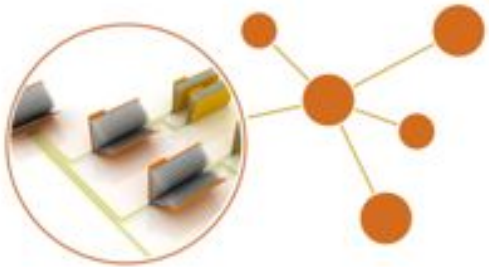
Neo4j: a property graph database

Verticals

Impact Analysis

Logistics and Routing

Recommendations



Access Control

Fraud Analysis

Social Network

Graph stores: Neo4j

Labelled property graph database

Four building blocks:

- Nodes
- Relationships
- Properties
- Labels

https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc

(Thanks to Stefan Plantikow, Tobias Lindaaker & Mark Needham for some of the following slides/images)

Graph stores: Neo4j

Nodes

Represent objects in the graph

Can be *labelled*

Graph stores: Neo4j

Nodes

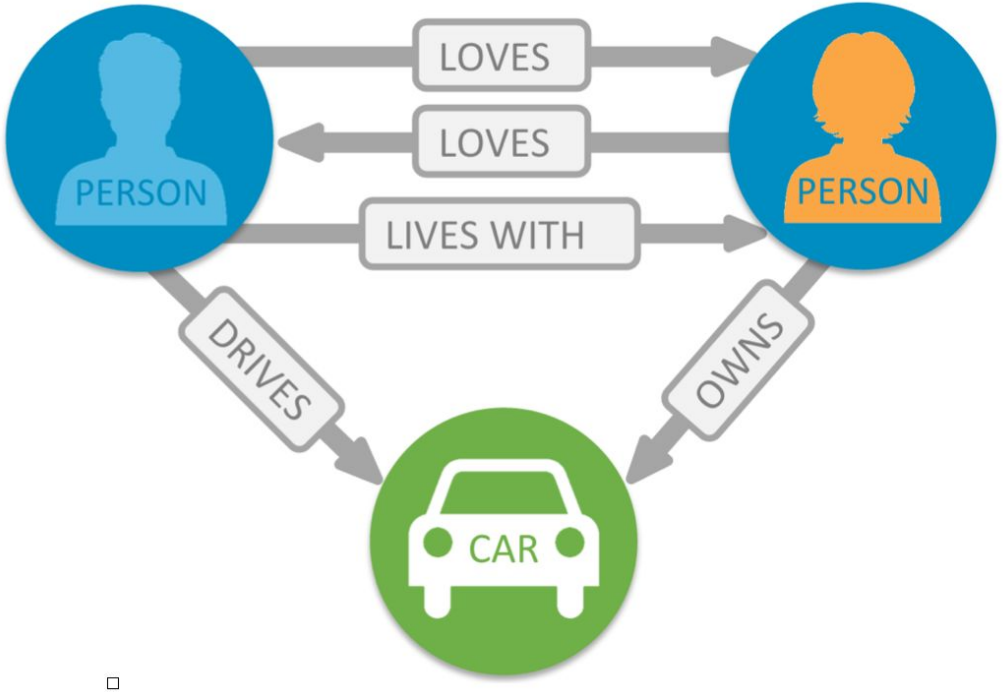
Represent objects in the graph

Can be *labelled*

Relationships

Relate nodes by *type* and

direction



Graph stores: Neo4j

Nodes

Represent objects in the graph

Can be *labelled*

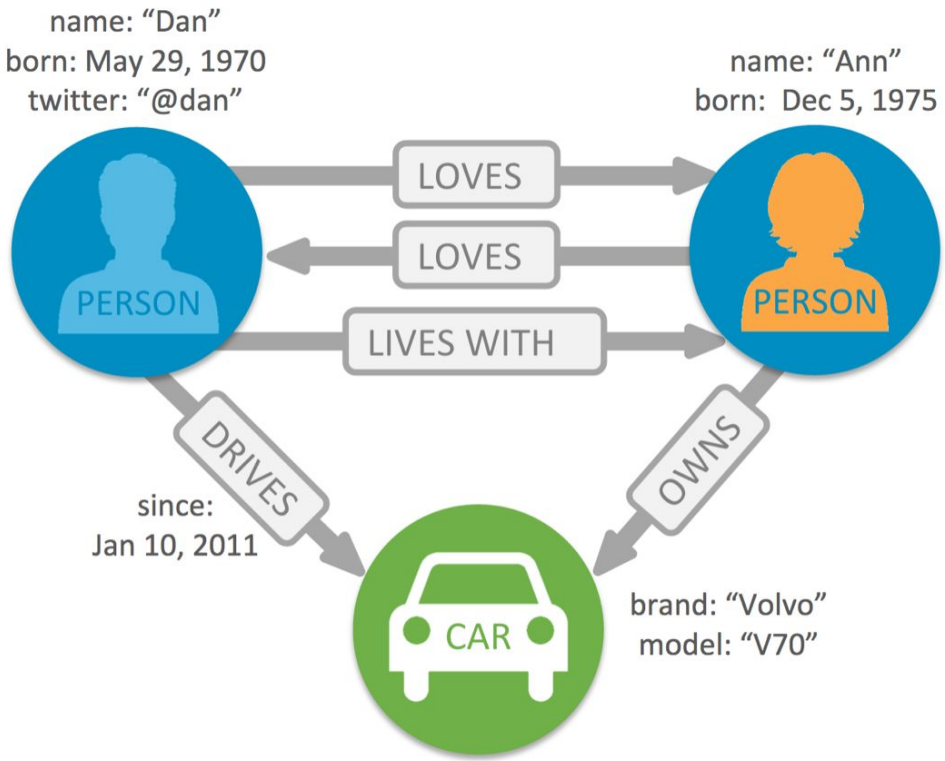
Relationships

Relate nodes by *type* and

Direction

Properties

Name-value pairs that can go on nodes and relationships



Nodes

Used to represent *entities* and *complex value types* in your domain

Can contain properties

Nodes of the same type can have different properties

Labels

Every node can have *zero* or *more* labels

Used to represent roles (e.g. user, product, company)

- Group nodes
- Allows us to associate *indexes* and *constraints* with groups of nodes

Relationships

Every relationship has a type and a direction

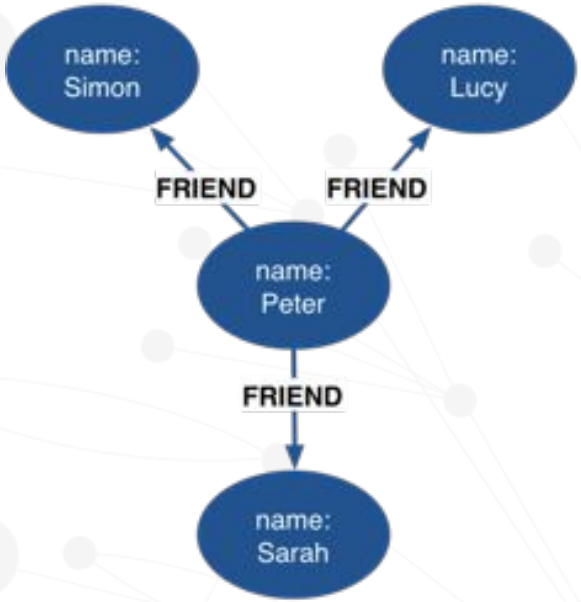
- Adds structure to the graph
- Provides semantic context for nodes

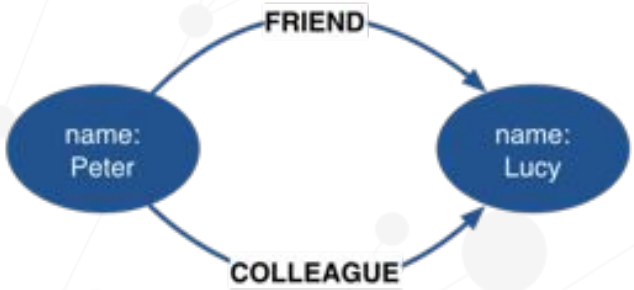
Can contain properties

Every relationship must have a start node and end node

- No dangling relationships

Relationships





Properties

Each node and relationship may have *zero* or *more* properties

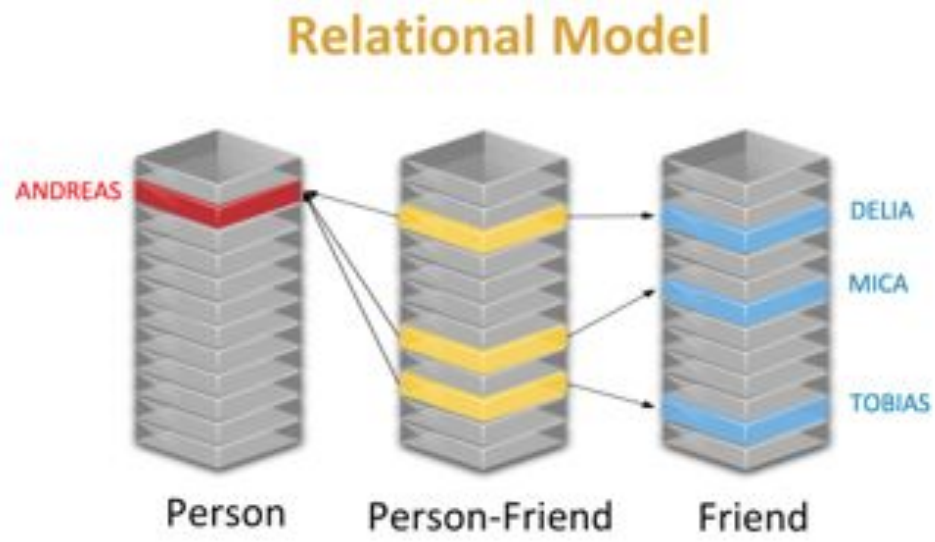
Represent the data: name, age, weight, createdAt etc…

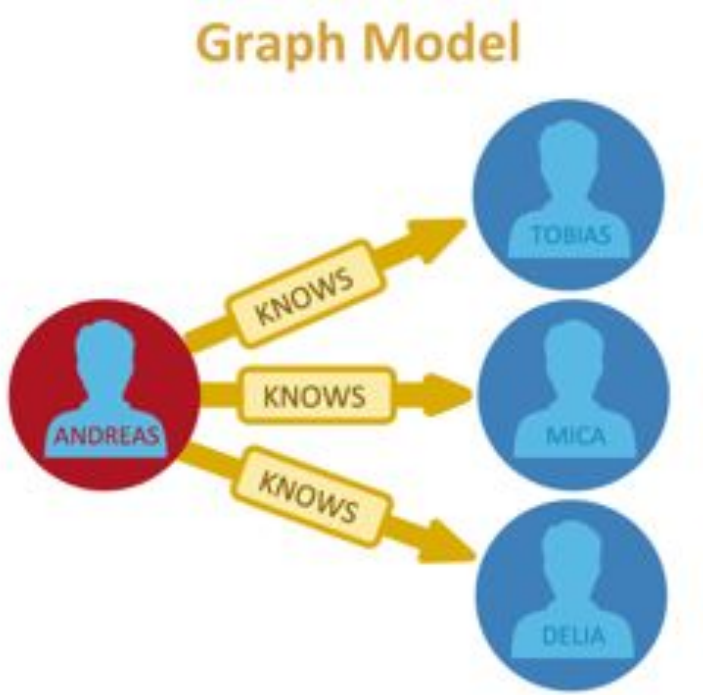
Key-value pairs (a map):

String **key**: “name”

Typed **value**: string, number, boolean, lists

Relational vs. graph models





Language drivers

Java

```
<dependency>
  <groupId>org.neo4j.driver</groupId>
  <artifactId>neo4j-java-driver</artifactId>
</dependency>
```

Python

```
pip install neo4j-driver
```

.NET

```
PM> Install-Package Neo4j.Driver
```

JavaScript

```
npm install neo4j-driver
```

Graph stores

**where a real time response is needed

Less about the volume of data or availability

More about how your data is related

Densely-connected, variably structured domains**

Lots of join tables? Connectedness**

Lots of sparse tables? Variable structure**

Path finding**

Deep joins**

Use in any case where the relationship between the data is just as important as the data itself.

Don’t use if your data is simple or tabular.

More use cases for graphs at http://neo4j.com/customers/

Neo4j: Resources

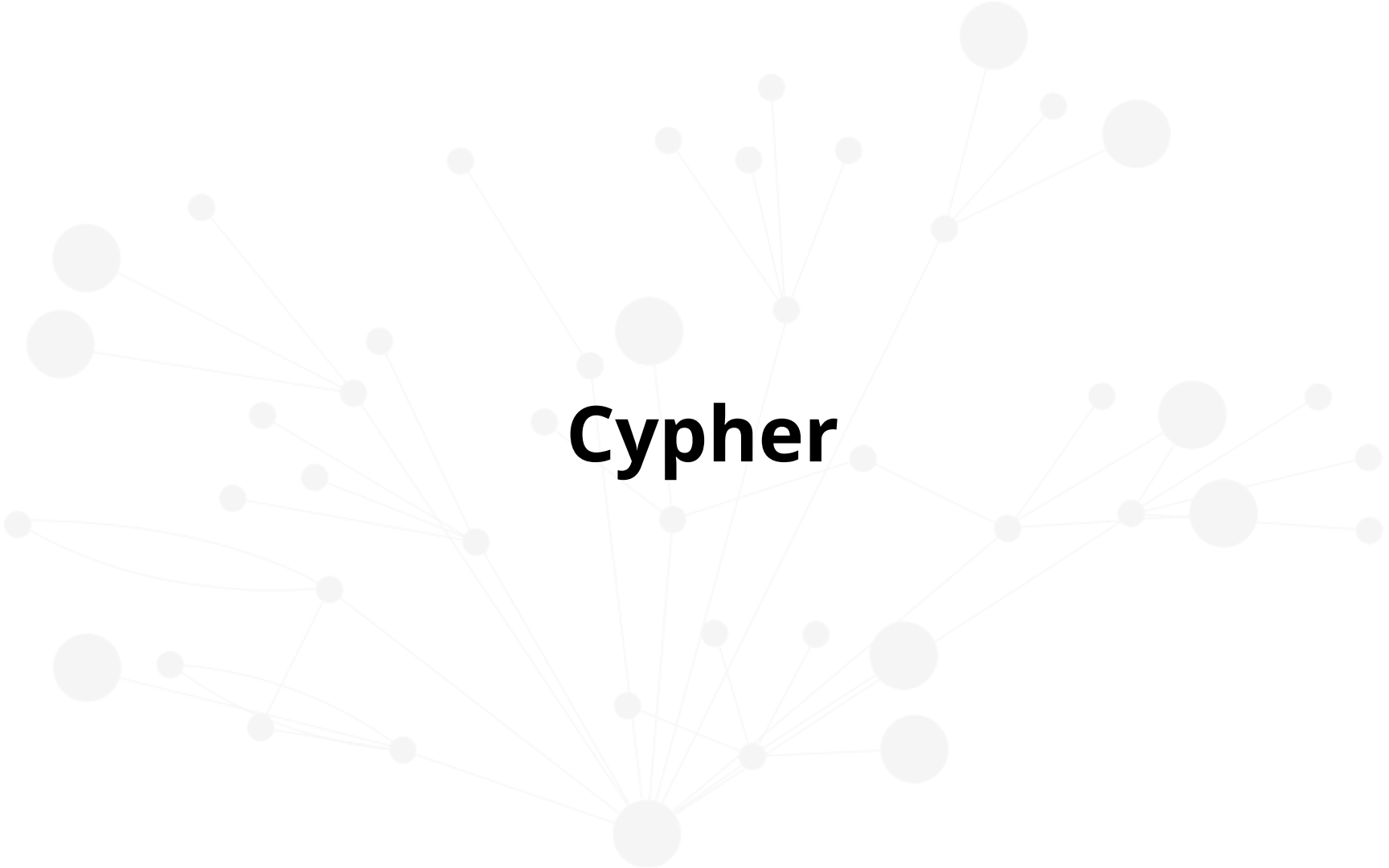
Neo4j Manual: https://neo4j.com/docs/developer-manual/current/

Graph Databases (book available online at www.graphdatabases.com)

Getting started: http://neo4j.com/developer/get-started/

Online training: http://neo4j.com/graphacademy/

Meetups (last Wed of the month) at http://www.meetup.com/graphdb-london (free talks and training sessions)



Cypher

Introducing Cypher

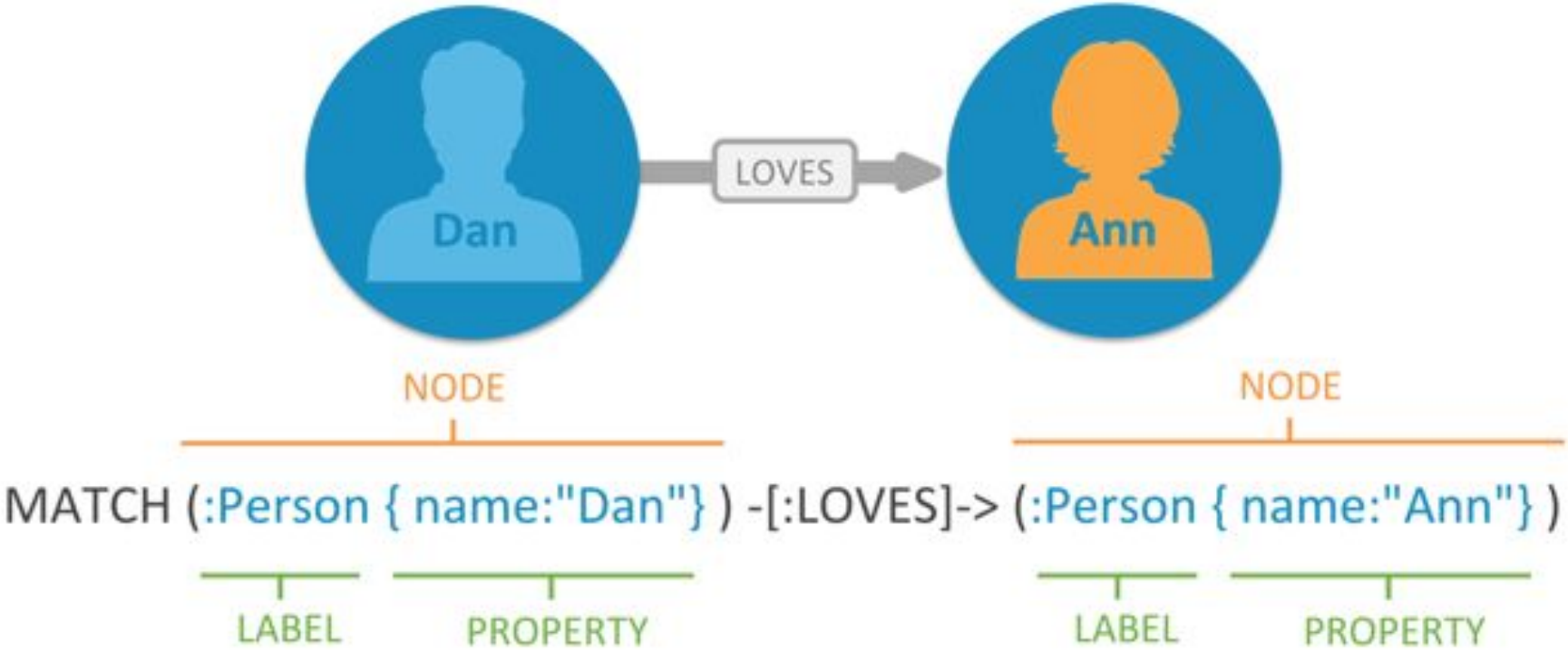
Declarative graph pattern matching language

SQL-like syntax

ASCII art based

Able to read and mutate the data, as well as perform various aggregate functions such as count and so on

Cypher: matching patterns



Cypher: nodes

`()` or `(n)`

Surround with parentheses

Use an alias `n` to refer to our node later in the query

`(n:Label)`

Specify a `Label`, starting with a colon `:`

Used to group nodes by roles or types (similar to tags)

`(n:Label {prop: 'value'})`

Nodes can have properties

Cypher: relationships

`-->` or `-[r:TYPE]->`

Wrapped in hyphens and square brackets

A relationship type starts with a colon `:`

`<>`

Specify the direction of the relationships

`-[:KNOWS {since: 2010}]->`

Relationships can have properties

Cypher: patterns

Used to query data

```
(n:Label {prop: ‘value’})-[:TYPE]->(m:Label)
```

Cypher: patterns

Find Alice who knows Bob

In other words:

find `Person` with the `name 'Alice'`

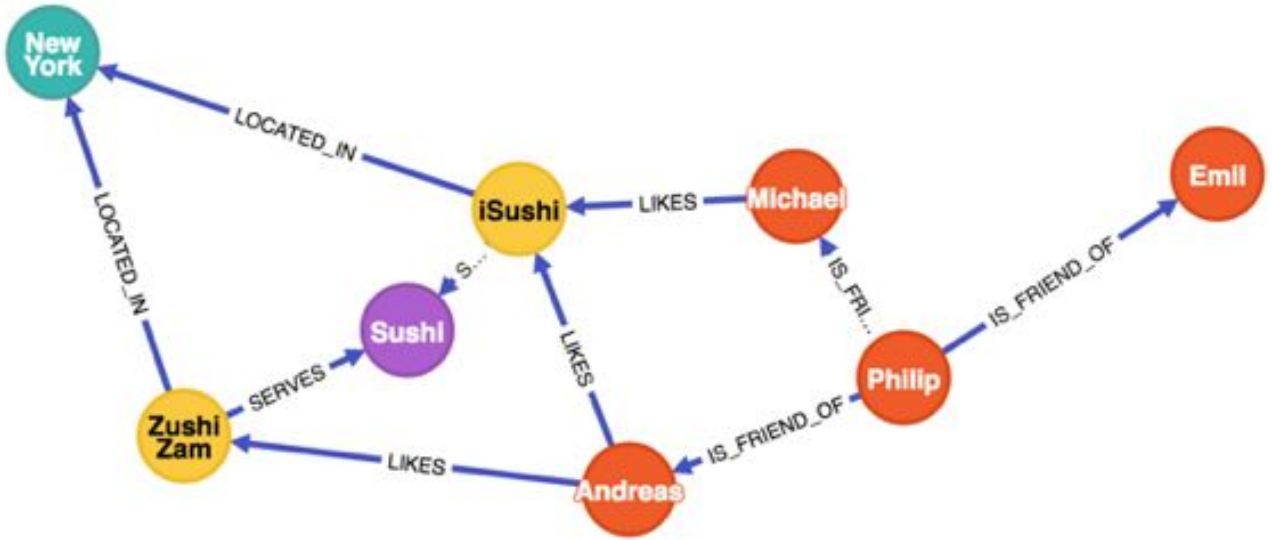
who `KNOWS`

a `Person` with the `name 'Bob'`

```
(p1:Person {name: 'Alice'})-[:KNOWS]->(p2:Person {name: 'Bob'})
```

Cypher: restaurant recommendations

Friends, restaurants in cities, their cuisines, and restaurants liked by people



Cypher: restaurant recommendations

Find Sushi restaurants in New York liked by Philip's friends

Four connected facts:

1. `People` who are `friends` of `Philip`
2. `Restaurants` `located` in `New York`
3. `Restaurants` `serving` `Sushi`
4. `Restaurants` `liked` by `Philip's Friends`

Cypher: restaurant recommendations

```
MATCH (philip:Person {name: 'Philip'}),
(philip)-[:IS_FRIEND_OF]-(friend),
(restaurant:Restaurant)-[:LOCATED_IN]->(:City {name: 'New York'}),
(restaurant)-[:SERVES]->(:Cuisine {name: 'Sushi'}),
(friend)-[:LIKES]->(restaurant)
RETURN restaurant.name, collect(friend.name) AS likers, count(*) AS occurrence
ORDER BY occurrence DESC
```

restaurant.name	likers	occurrence
iSushi	[Michael, Andreas]	2
Zushi Zam	[Andreas]	1

Cypher in a nutshell

```
// Pattern matching
MATCH (me:Person)-[:FRIEND]->(friend)
// Filtering with predicates
WHERE me.name = "Frank Black"
AND   friend.age > me.age
// Projection of expressions
RETURN toUpper(friend.name) AS name, friend.title AS title

// Data creation and manipulation
CREATE (you:Person)
SET you.name = "Aaron Fletcher"
CREATE (you)-[:FRIEND]->(me)

// Sequential query composition and aggregation
MATCH (me:Person {name: $name})-[:FRIEND]-(friend)
WITH me, count(friend) AS friends
MATCH (me)-[:ENEMY]-(enemy)
RETURN friends, count(enemy) AS enemies
```

based on slide by T. Lindaaker

Cypher patterns in a nutshell

```
// Node patterns
MATCH (), (node), (node:Node), (:Node), (node {type:"NODE"})

// Rigid relationship patterns
MATCH ()-->(), ()-[edge]->(),
      ()-[edge:RELATES]->(),
      ()-[:RELATES]->(),
      ()-[edge {score:5}]->(),
      (a)-[edge]->(b)
      (a)<-[edge]-(b), (a)-[edge]-(b)

// Variable length relationship patterns
MATCH (me)-[:FRIEND*]-(foaf)
MATCH (me)-[:FRIEND*1..3]-(foaf)

// Path binding
MATCH p=(a)-[:ONE]-()-[:TWO]-()-[:THREE]-()
```

based on slide by T. Lindaaker

Evolving Cypher

Designing a query language: what is involved?

Syntax

Semantics

Academic research

Compare and contrast with SQL, SPARQL, ...

Designing a query language: considerations

```
(node1)-[:RELATIONSHIP]->(node2)
```

Keywords

- Suitability e.g. `CREATE` or `ADD`
- Symmetry e.g. `ADD` and `DROP`

Delimiters

- Do not reuse “`(`”, “`[`”...

Consistent behaviour with existing implementation

Complexity

Ensure the constructs are future-proof

openCypher...

...is a community effort to evolve Cypher, and make it the de-facto language for querying property graphs

openCypher implementations

SAP, Redis, Agens Graph, Cypher.PL, Neo4j, CAPS, CoG, ...

openCypher

opencypher.org

openCypher Implementers Group (oCIG)

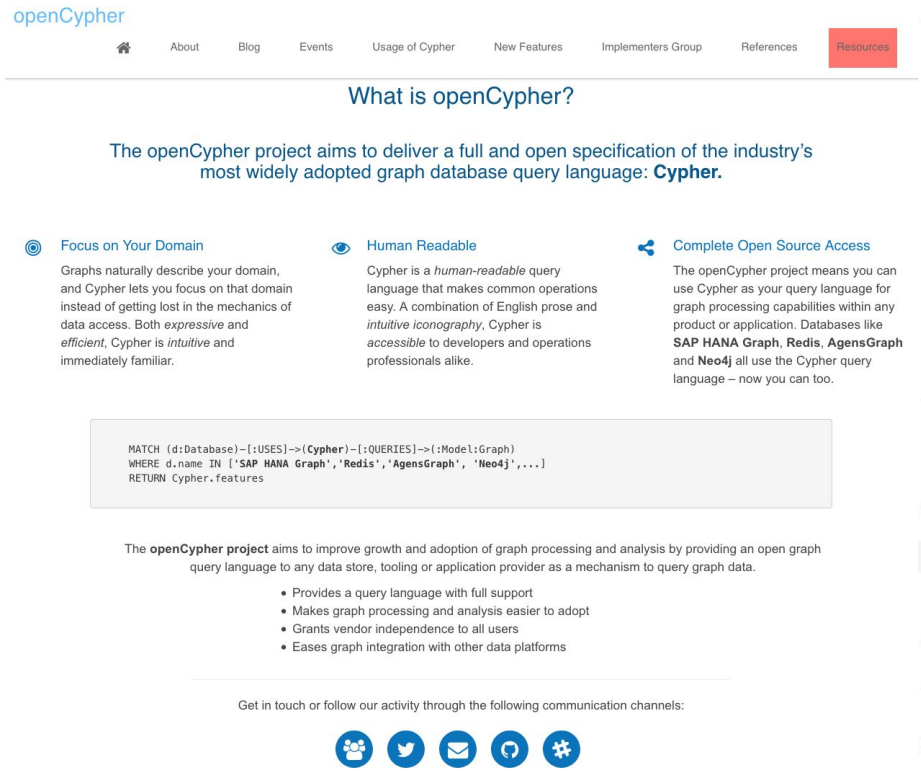
Evolve Cypher through an open process

Comprises vendors, researchers, implementers, interested parties

Regular meetings to discuss and agree upon new features

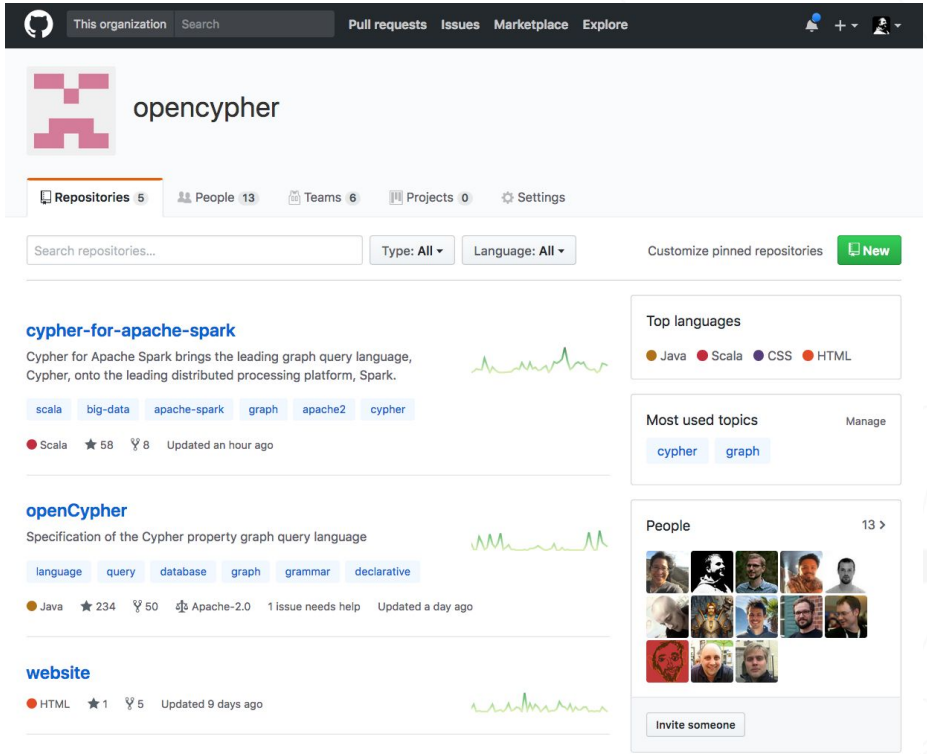
Consensus-based system

openCypher website



- Blog
- New Features
- Upcoming Meetings
- Recordings and Slides
- References (Links, Papers)
- Artifacts

github.com/openCypher



Language Artifacts

Cypher 9 reference

ANTLR and EBNF Grammars

Formal Semantics (SIGMOD, to be published here)

Technology Compatibility Kit (TCK) - Cucumber test suite)

Style Guide

Implementations & Code

openCypher for Apache Spark

openCypher for Gremlin

open source frontend (part of Neo4j, to be published here)

Formal Semantics

SIGMOD 2018

http://homepages.inf.ed.ac.uk/pguaglia/papers/sigmod18.pdf

Cypher: An Evolving Query Language for Property Graphs

Nadime Francis*
Université Paris-Est

Alastair Green
Neo4j

Paolo Guagliardo
University of Edinburgh

Leonid Libkin
University of Edinburgh

Tobias Lindaaker
Neo4j

Victor Marsault
University of Edinburgh

Stefan Plantikow
Neo4j

Mats Rydberg
Neo4j

Petra Selmer
Neo4j

Andrés Taylor
Neo4j

ABSTRACT

The Cypher property graph query language is an evolving language, originally designed and implemented as part of the Neo4j graph database, and it is currently used by several commercial database products and researchers. We describe Cypher 9, which is the first version of the language governed by the openCypher Implementers Group. We first introduce the language by example, and describe its uses in industry. We then provide a formal semantic definition of the core read-query features of Cypher, including its variant of the property graph data model, and its 'ASCII Art' graph pattern matching mechanism for expressing subgraphs of interest to an application. We compare the features of Cypher to other property graph query languages, and describe extensions, at an advanced stage of development, which will form part of Cypher 10, turning the language into a compositional language which supports graph projections and multiple named graphs.

ACM Reference Format:
Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2017. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of (SIGMOD'18).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the last decade, property graph databases [33] such as Neo4j, JanusGraph and Sparksee have become more widespread in industry and academia. They have been used in multiple domains, such as master data and knowledge management, recommendation engines, fraud detection, IT operations and network management, authorization and access control [51], bioinformatics [38], social networks [17], software system analysis [24], and in investigative journalism [11]. Using graph databases to manage graph-structured data confers many benefits such as explicit support for modeling graph data, native indexing and storage for fast graph traversal operations, built-in support for graph algorithms (e.g., Page Rank, subgraph matching and so on), and the provision of graph languages, allowing users to express complex pattern-matching operations.

In this paper we describe Cypher, a well-established language for querying and updating property graph databases, which began life in the Neo4j product, but has now been implemented commercially in other products such as SAP HANA Graph, Redis Graph, Agens Graph (over PostgreSQL) and Memgraph. Neo4j [51] is one of the most popular property graph databases[1] that stores graphs natively on disk and provides a framework for traversing graphs and executing graph operations. The language therefore is used in hundreds of production applications across many industry vertical domains.

Since 2015 the openCypher project[2] has sought to enable the use of Cypher as a standardized language capable of multiple independent implementations, and to provide a framework for cross-implementer collaborative evolution of new language features. The goal is that Cypher will mature into an industry standard language for property graph querying, playing a complementary role to that of the SQL standard for relational data querying. Here we present Cypher 9 [46], the first version of the language governed by openCypher. We give an introduction to the language, describe its uses in industry, provide a formal definition of its data model and the semantics of its queries and clauses, and then describe current work that will lead to Cypher 10, a compositional language supporting graph projections and multiple named graphs.

The data model of Neo4j that is used by Cypher is that of *property graphs*. It is the most popular graph data model in industry, and is becoming increasingly prevalent in academia [37]. The model comprises *nodes*, representing entities (such as people, bank accounts, departments and so on), and *relationships* (synonymous with *edges*), representing the connections or relationships between the entities. In the graph model, the relationships are as important as the entities themselves. Moreover, any number of attributes (henceforth termed *properties*), in the form of key-value pairs, may be associated with the nodes and relationships. This allows for the modeling and querying of complex data.

The language comes with a fully formalized semantics of its core constructs. The need for it stems from the fact that Cypher,

*Affiliated with the School of Informatics at the University of Edinburgh during the time of contributing to this work.

SIGMOD'18, submission, industry track

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://db-engines.com/en/ranking/graph+dbms
[2] https://www.opencypher.org

TCK

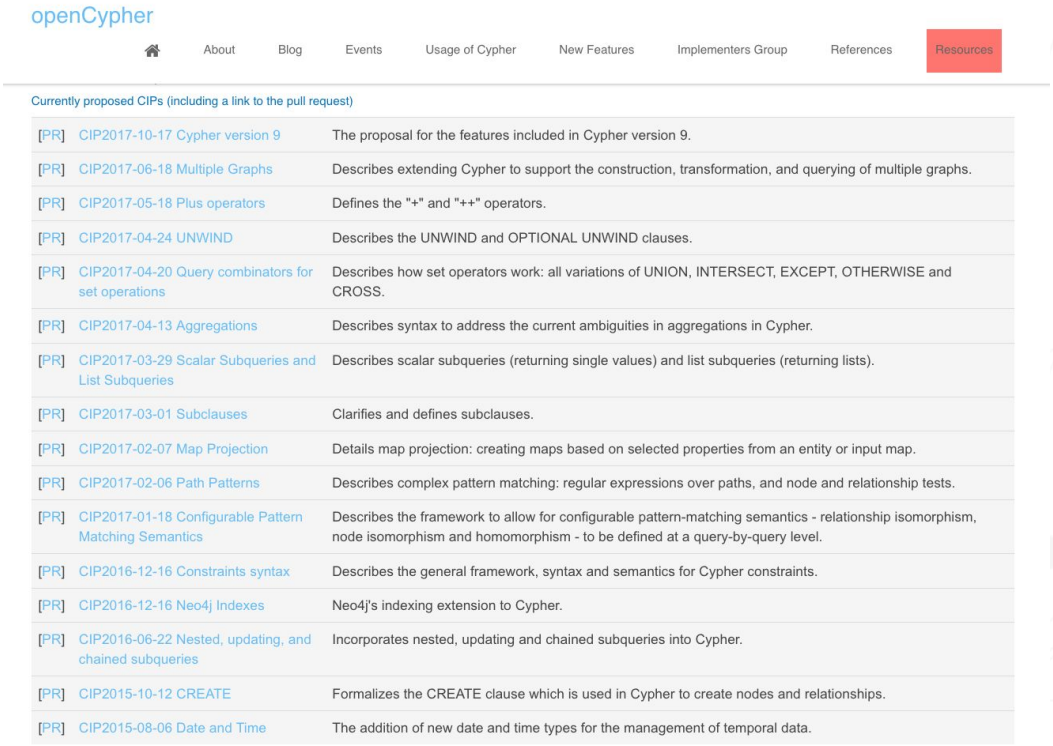
Scenario: Optionally matching named paths
  Given an empty graph
  And having executed:
    """
    CREATE (a {name: 'A'}), (b {name: 'B'}), (c {name: 'C'})
    CREATE (a)-[:X]->(b)
    """
  When executing query:
    """
    MATCH (a {name: 'A'}), (x)
    WHERE x.name IN ['B', 'C']
    OPTIONAL MATCH p = (a)-->(x)
    RETURN x, p
    """
  Then the result should be:
    | x | p |
    | ({name: 'B'}) | <({name: 'A'})-[:X]->({name: 'B'})> |
    | ({name: 'C'}) | null |
  And no side effects

Background:
  Given any graph

Scenario: Creating a node
  When executing query:
    """
    CREATE ()
    """
  Then the result should be empty
  And the side effects should be:
    | +nodes | 1 |

based on slide by M. Rydberg

Language specification and improvements



openCypher

About | Blog | Events | Usage of Cypher | New Features | Implementers Group | References | Resources

Currently proposed CIPs (including a link to the pull request)

[PR]	CIP2017-10-17 Cypher version 9	The proposal for the features included in Cypher version 9.
[PR]	CIP2017-06-18 Multiple Graphs	Describes extending Cypher to support the construction, transformation, and querying of multiple graphs.
[PR]	CIP2017-05-18 Plus operators	Defines the "+" and "++" operators.
[PR]	CIP2017-04-24 UNWIND	Describes the UNWIND and OPTIONAL UNWIND clauses.
[PR]	CIP2017-04-20 Query combinators for set operations	Describes how set operators work: all variations of UNION, INTERSECT, EXCEPT, OTHERWISE and CROSS.
[PR]	CIP2017-04-13 Aggregations	Describes syntax to address the current ambiguities in aggregations in Cypher.
[PR]	CIP2017-03-29 Scalar Subqueries and List Subqueries	Describes scalar subqueries (returning single values) and list subqueries (returning lists).
[PR]	CIP2017-03-01 Subclauses	Clarifies and defines subclauses.
[PR]	CIP2017-02-07 Map Projection	Details map projection: creating maps based on selected properties from an entity or input map.
[PR]	CIP2017-02-06 Path Patterns	Describes complex pattern matching: regular expressions over paths, and node and relationship tests.
[PR]	CIP2017-01-18 Configurable Pattern Matching Semantics	Describes the framework to allow for configurable pattern-matching semantics - relationship isomorphism, node isomorphism and homomorphism - to be defined at a query-by-query level.
[PR]	CIP2016-12-16 Constraints syntax	Describes the general framework, syntax and semantics for Cypher constraints.
[PR]	CIP2016-12-16 Neo4j Indexes	Neo4j's indexing extension to Cypher.
[PR]	CIP2016-06-22 Nested, updating, and chained subqueries	Incorporates nested, updating and chained subqueries into Cypher.
[PR]	CIP2015-10-12 CREATE	Formalizes the CREATE clause which is used in Cypher to create nodes and relationships.
[PR]	CIP2015-08-06 Date and Time	The addition of new date and time types for the management of temporal data.

Cypher 9 reference

Cypher Improvement Request (CIR)

Cypher Improvement Proposal (CIP)

Next version: Cypher 10

Upcoming Cypher features

Query composition

"Meaning of the whole is determined by the meanings of its constituents and the rules used to combine them"

Organize a query into multiple parts

Extract parts of a query to a view for re-use

Replace parts of a query without affecting other parts

Build complex workflows programmatically

based on slide by S. Plantikow

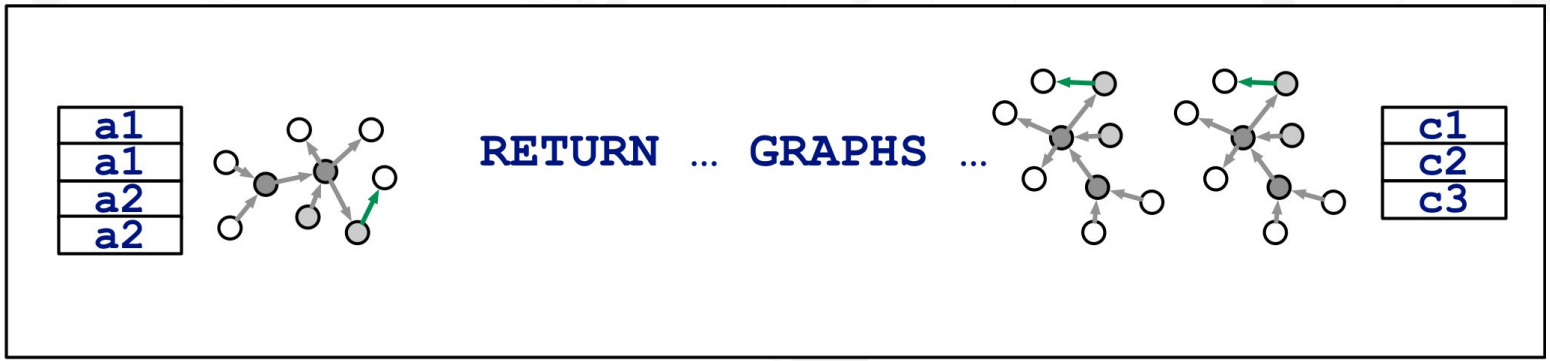
Implications for Cypher

Pass both multiple graphs and tabular data into a query

Return both multiple graphs and tabular data from a query

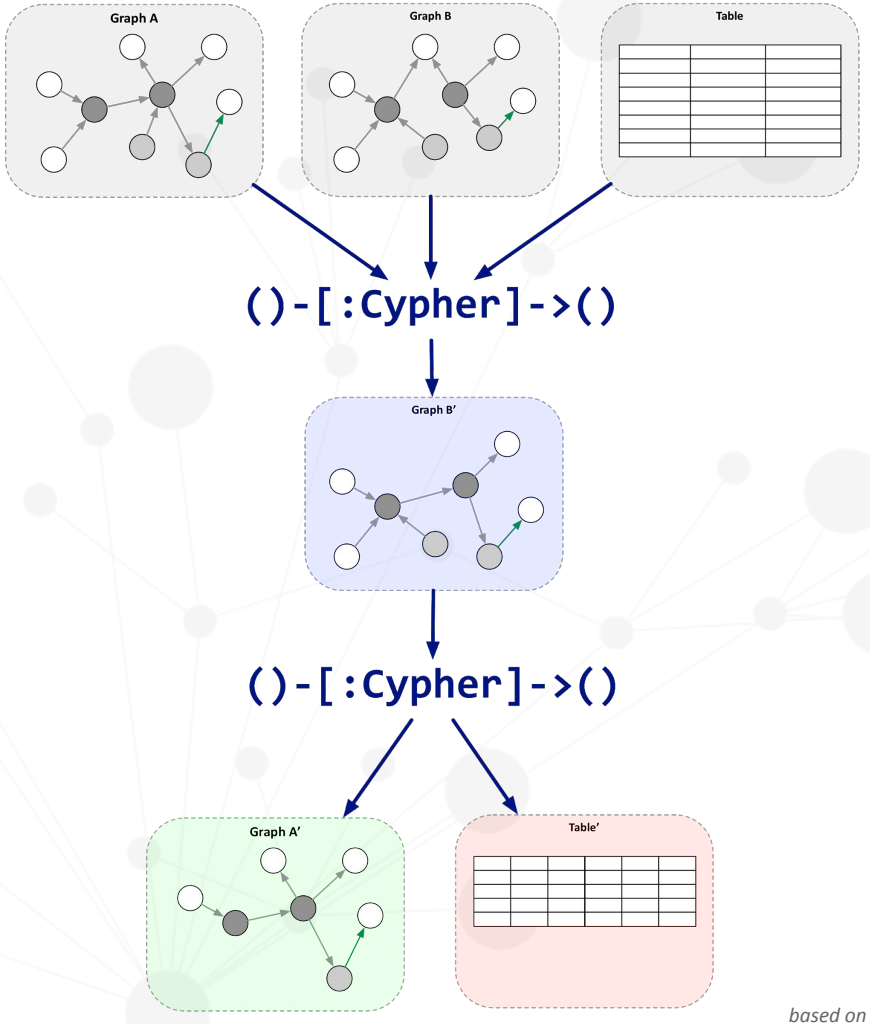
Select which graph to query

Construct new graphs from existing graphs



based on slide by S. Plantikow

Cypher query pipeline composition



based on slide by S. Plantikow

Complex path patterns

Regular path queries

```
X, (likes.hates)*(eats|drinks)+, Y
```

Inclusion of node and relationship tests

Path patterns

```
PATH PATTERN
  older_friends = (a)-[:FRIEND]-(b) WHERE b.age > a.age

MATCH p=(me)-/~older_friends+/-(you)
WHERE me.name = $myName AND you.name = $yourName
RETURN p AS friendship
```

based on slide by T. Lindaaker

Getting involved

Please follow news at **opencypher.org** and **@opencypher** on twitter

There's a great slack channel for implementers

Next openCypher Implementer Group call on Wednesday, 14 March

Language change request issues (CIRs) and full proposals (CIPs)

Own ideas? Talk to us! Or create a Pull Request at
https://github.com/opencypher/openCypher

Thank you!