CS 520 Knowledge Graphs: **Querying Property Graphs with [open]Cypher**

Dr. Petra Selmer

Team Lead, Query Languages Standards & Research Group @ Neo4j

petra.selmer@neo4j.com

April 8, 2021

Talk outline

- Recap of the Property Graph Data Model
- The Cypher query language
- Evolving Cypher through the openCypher project
- Introducing Graph Query Language (GQL)
- Proposed Extensions

The property graph data model

Property graph

Node

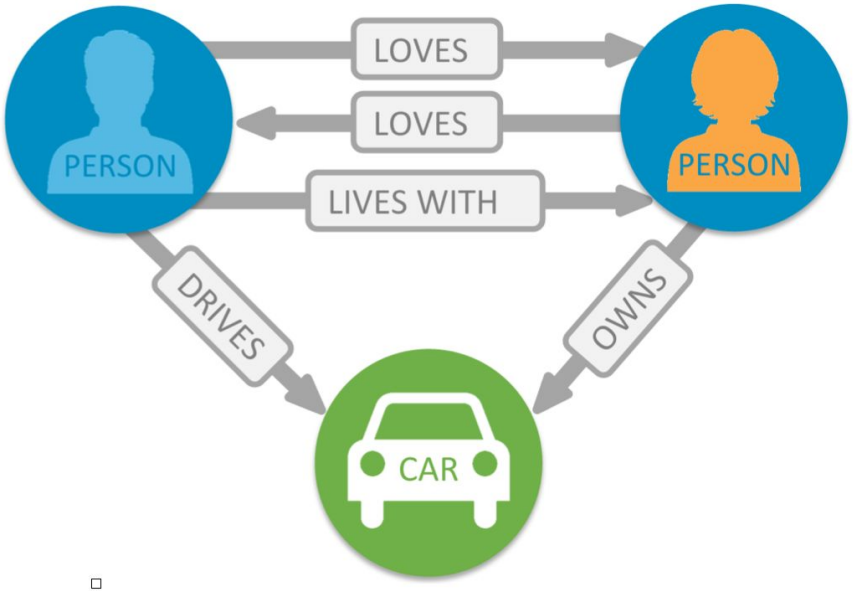
- Represents an entity within the graph
- Has zero or more labels
- Has zero or more properties
  - These may differ across nodes with the same label(s)
- Synonym: vertex

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

Property graph

Relationship

- Adds structure to the graph
  - Provides semantic context for nodes
- Synonym: edge
- Has one type
- Has zero or more properties
  - These may differ across relationships with the same type
- Relates nodes by type and direction
- Must have a start and an end node

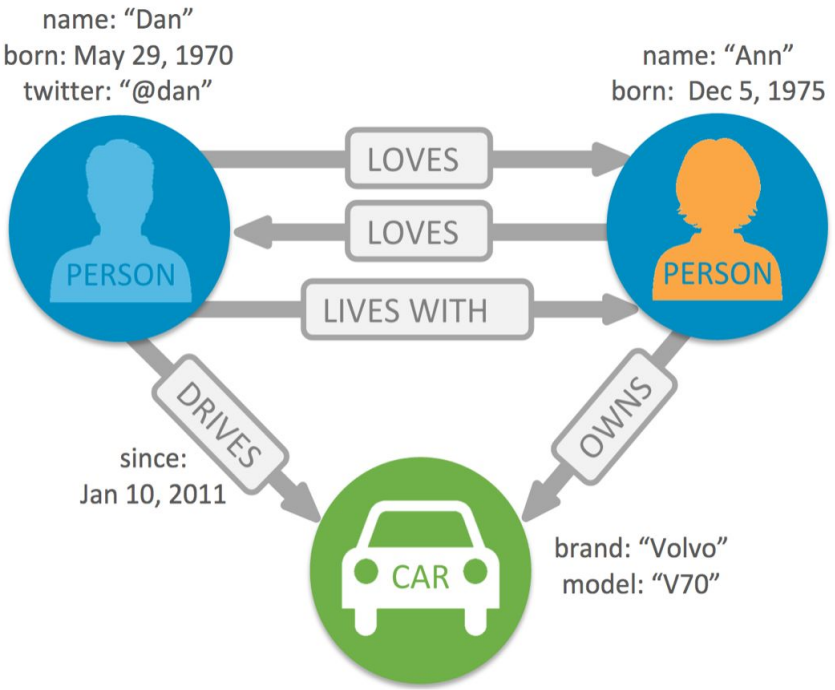


https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc

Property graph

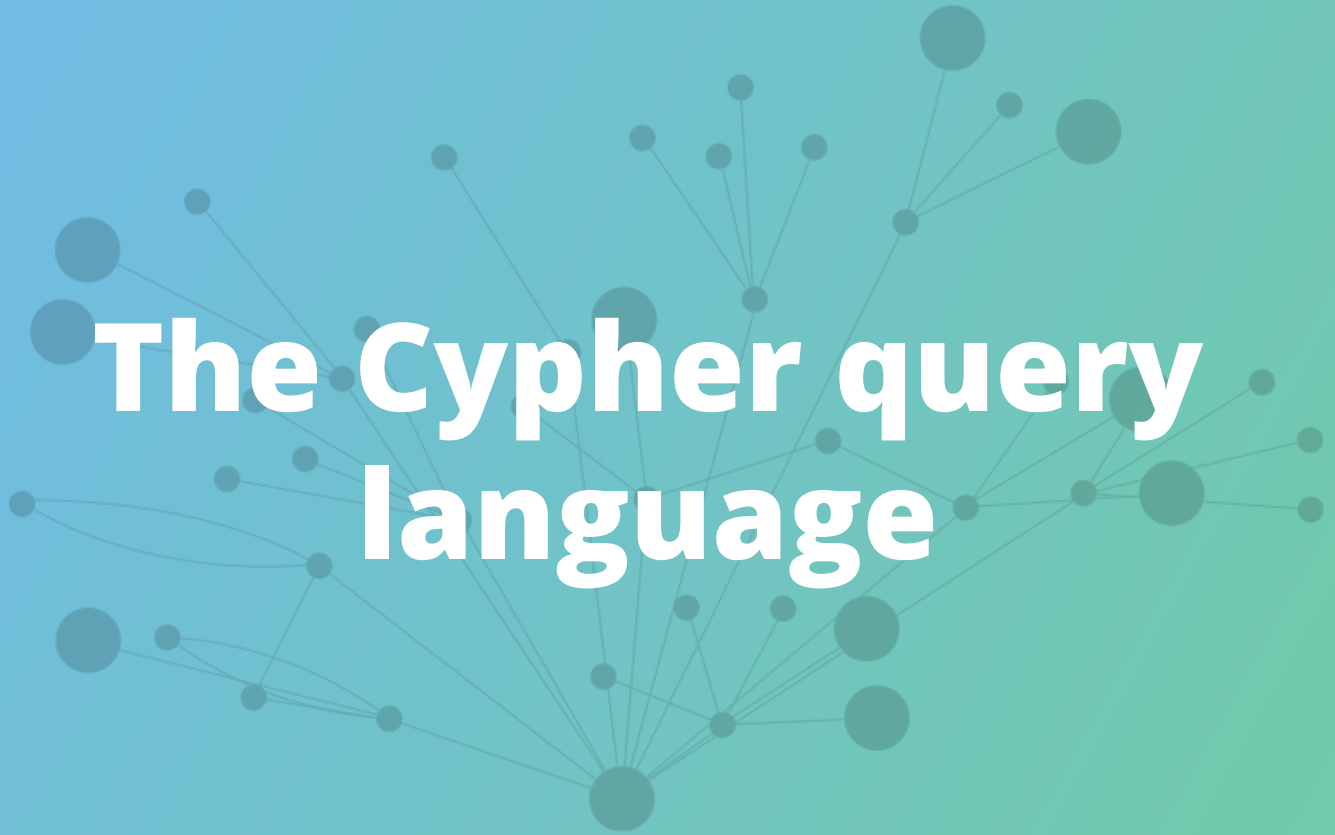
Property

- Name-value pair (map) that can go on nodes and edges
- Represents the data: e.g. name, age, weight etc
- String key; typed value (string, number, bool, list)



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

The Cypher query language



Introducing Cypher

- Created by Neo4j (2010)
- openCypher (2015)

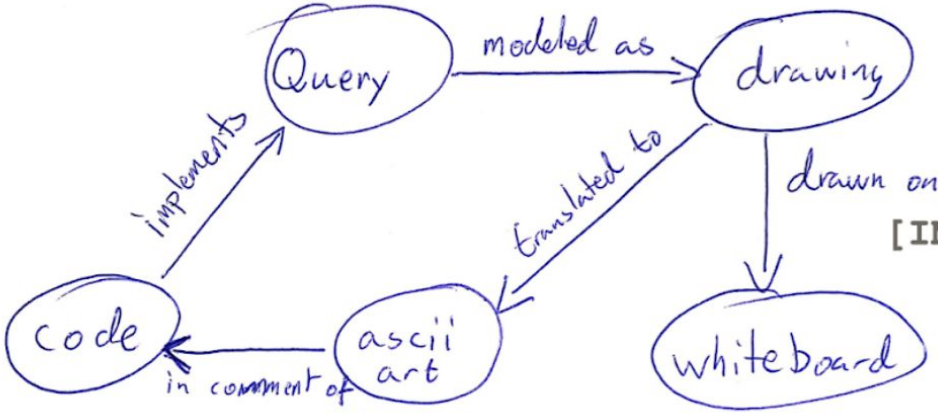
Declarative **graph pattern matching** language that adheres to modern paradigms and is intuitive

- Graph patterns are very easily expressed
- Recursive queries
- Variable-length relationship chains
- Returning paths

SQL-like syntax and features

- DQL for reads (focus of this talk)
- DML for updates
- DDL for creating constraints and indexes

Patterns are everywhere



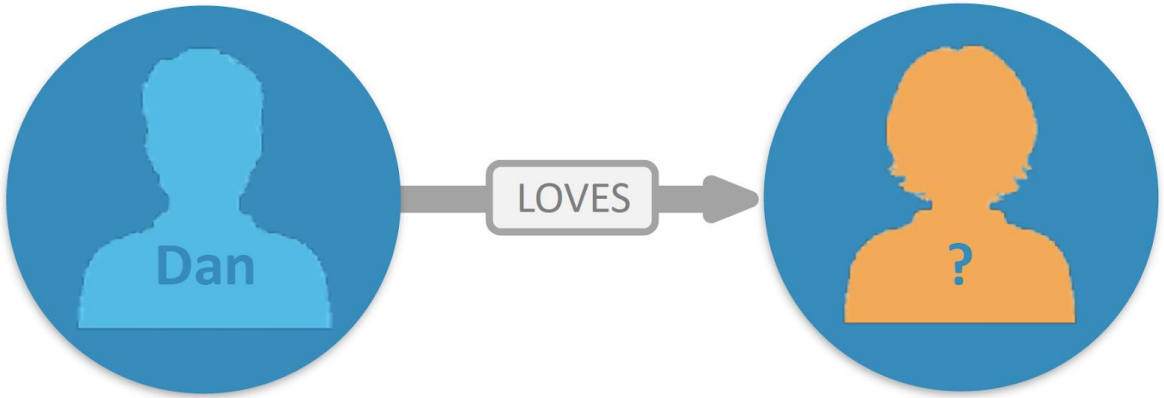
Expressed using “ASCII Art”

```
(query)--[MODELED_AS]--->(drawing)
   ^                        /    |
   |                 [DRAWN_ON]  |
[IMPLEMENTS]             /   [TRANSLATED_TO]
   |                   v          |
   |           (whiteboard)     v
(code)<-[IN_COMMENT_OF]-(ascii art)
```

```
MATCH (query)-[:MODELED_AS]->(drawing),
      (code)-[:IMPLEMENTS]->(query),
      (drawing)-[:TRANSLATED_TO]->(ascii_art),
      (ascii_art)-[:IN_COMMENT_OF]->(code),
      (drawing)-[:DRAWN_ON]->(whiteboard)
WHERE query.id = $query_id
RETURN code.source
```

Patterns are in
- Matching
- Updates
- DDL

Searching for (matching) graph patterns



NODE — Relationship — NODE

```
MATCH (:Person { name:"Dan"} ) -[:LOVES]-> ( whom ) RETURN whom
```

LABEL — PROPERTY — VARIABLE

Node and relationship patterns

`()` 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

`-->` 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

DQL: reading data

```
// Pattern description (ASCII art)
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
// Order results
ORDER BY name, title DESC
```

Multiple patterns can be defined in a single match clause (i.e. *conjunctive* patterns):

```
MATCH (a)-(b)-(c), (b)-(f)
```

Input: a property graph
Output: a table

More complex Cypher patterns

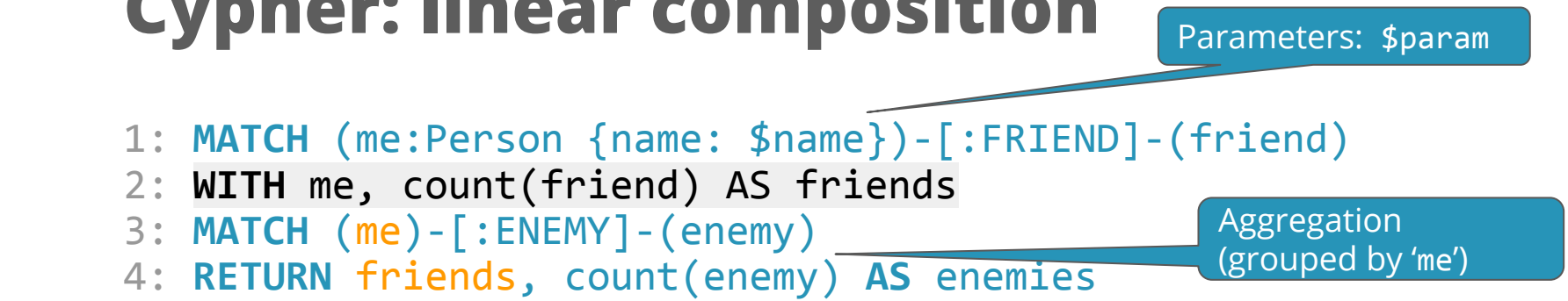
Variable-length relationship patterns

```
// Traverse 1 or more FRIEND relationships
MATCH (me)-[:FRIEND*]-(foaf)

// Traverse 2 to 4 FRIEND relationships
MATCH (me)-[:FRIEND*2..4]-(foaf)

// Traverse union of LIKES and KNOWS 1 or more times
MATCH (me)-[:LIKES|KNOWS*]-(foaf)
```

Returning **paths**

```
// Path binding returns all paths (p)
MATCH p = (a)-[:ONE]-()-[:TWO*]-()-[:THREE]-()
// Each path is a list containing the constituent nodes and relationships, in order
RETURN p

// Variation: return all constituent nodes/relationships of the path
RETURN nodes(p) / relationships(p)
```

Cypher: linear composition



`WITH` provides a *horizon*, allowing a query to be subdivided:

- Further matching can be done after a set of updates
- Expressions can be evaluated, along with aggregations
- Essentially acts like the pipe operator in Unix

Reading and writing statements may be composed linearly in a single query

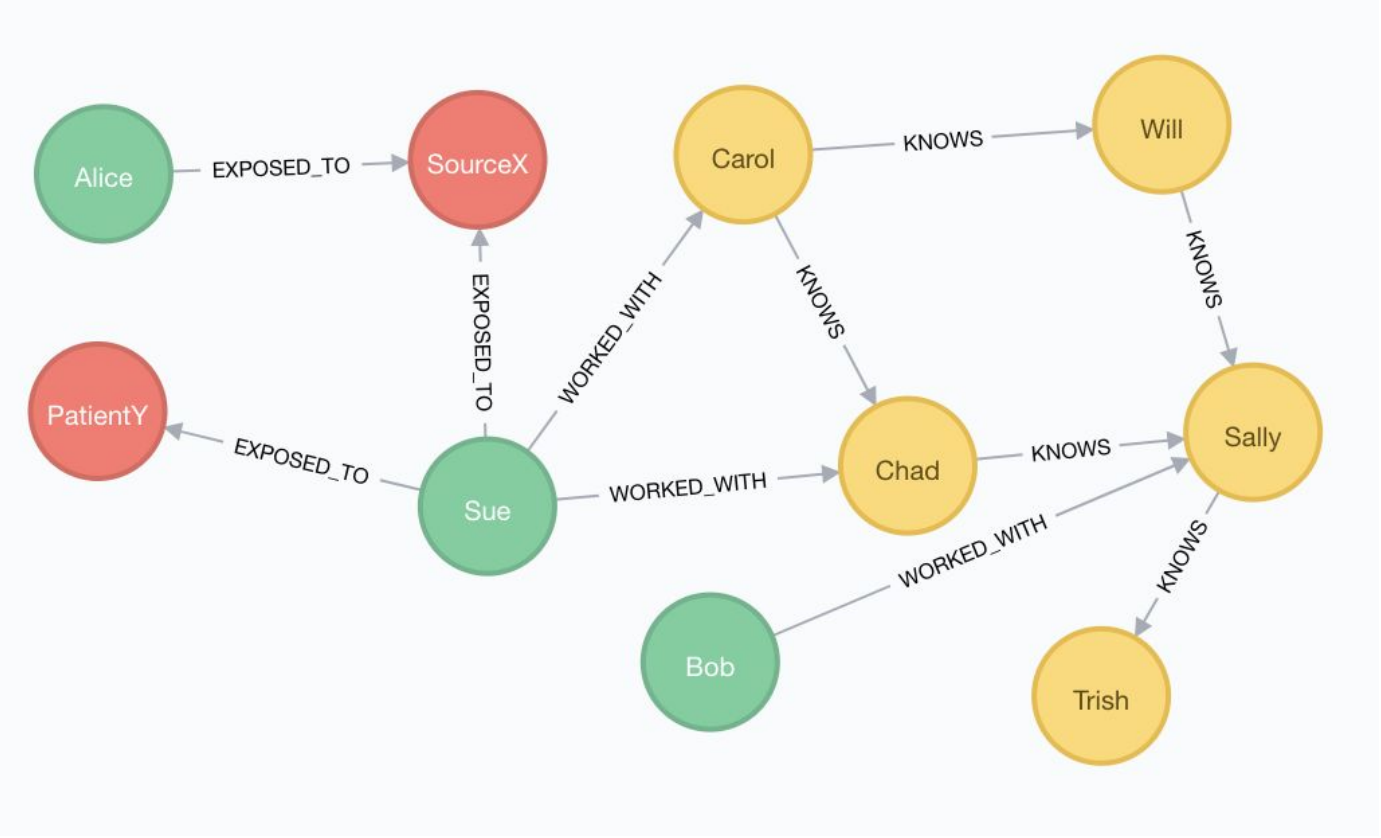
Linear composition

- Query processing begins at the top and progresses linearly to the end (top-down ordering)
- Each clause is a function taking in a table **T** (*line 1*) and returning a table **T'**
- **T'** then acts as a driving table to the next clause (*line 3*)

neo4j

Example query: epidemic

Example taken from EDBT/ICDT 2018



Assume a graph G containing doctors who have potentially been infected with a virus....

Example query

The following Cypher query returns the name of each doctor in G who has perhaps been exposed to some source of a viral infection, the number of exposures, and the number of people known (both directly and indirectly) to their colleagues

```
1: MATCH (d:Doctor)
2: OPTIONAL MATCH (d)-[:EXPOSED_TO]->(v:ViralInfection)
3: WITH d, count(v) AS exposures
4: MATCH (d)-[:WORKED_WITH]->(colleague:Person)
5: OPTIONAL MATCH (colleague)<-[:KNOWS*]-(p:Person)
6: RETURN d.name, exposures, count(DISTINCT p) AS thirdPartyCount
```

```
+-----------------------------------------+
| d.name | exposures | thirdPartyCount    |
+-----------------------------------------+
| Bob    |  0        | 3 (Will, Chad, Carol)|
| Sue    |  2        | 1 (Carol)          |
+-----------------------------------------+
```

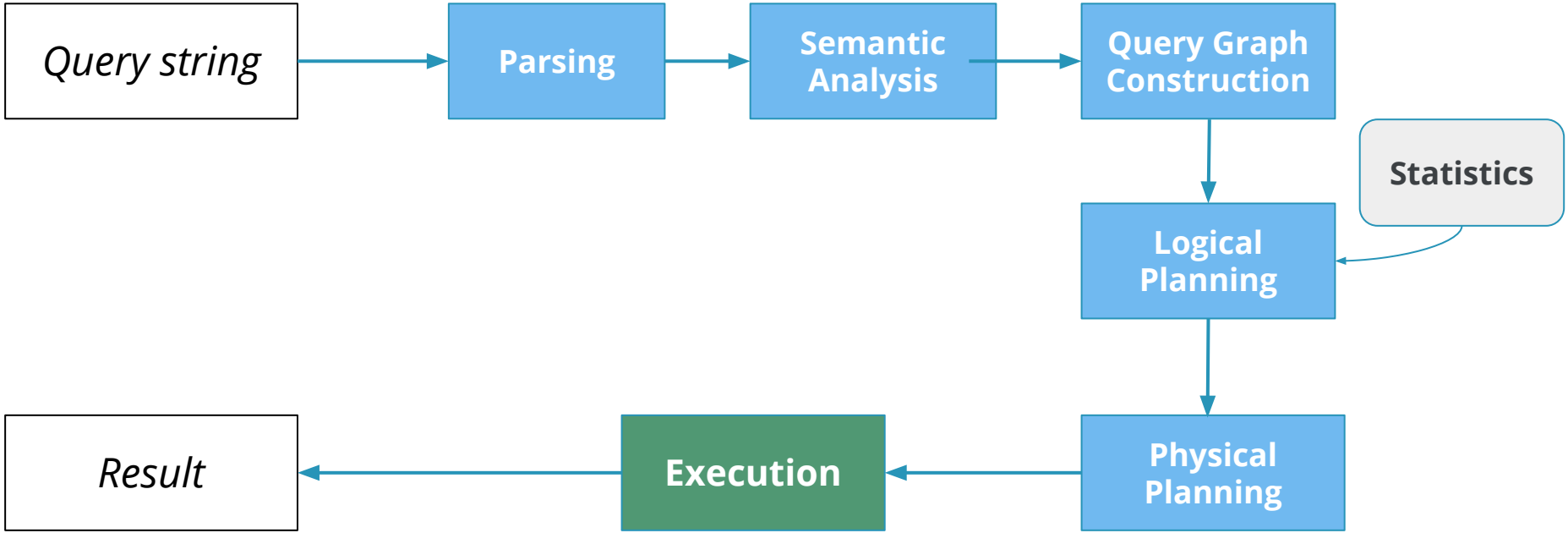
Contrasting Languages: SQL vs. Cypher

```
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.pid AS directReportees, 0 AS count
  FROM person_reportee manager
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
  SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
SELECT reportee.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
  FROM(
  SELECT reportee.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages)
AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT L2Reportees.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
)
```

```
MATCH (boss)-[:MANAGES*0..3]->(sub),
      (sub)-[:MANAGES*1..3]->(report)
WHERE boss.name = "John Doe"
RETURN sub.name AS Subordinate,
  count(report) AS Total
```

Neo4j Cypher Query Engine

Some stages skipped if in query cache



Evolving Cypher through the openCypher project

openCypher

opencypher.org

Established in 2015

openCypher Implementers Group (oCIG)

Evolve Cypher through an open process

Comprises vendors, researchers, implementers, interested parties

Language Artifacts

Cypher 9 reference

ANTLR and EBNF Grammars

Formal Semantics

Technology Compatibility Kit (TCK) - Cucumber test suite

Style Guide

Implementations & Code

openCypher for Apache Spark

openCypher for Gremlin

TCK (Technology Compliance Kit)

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:

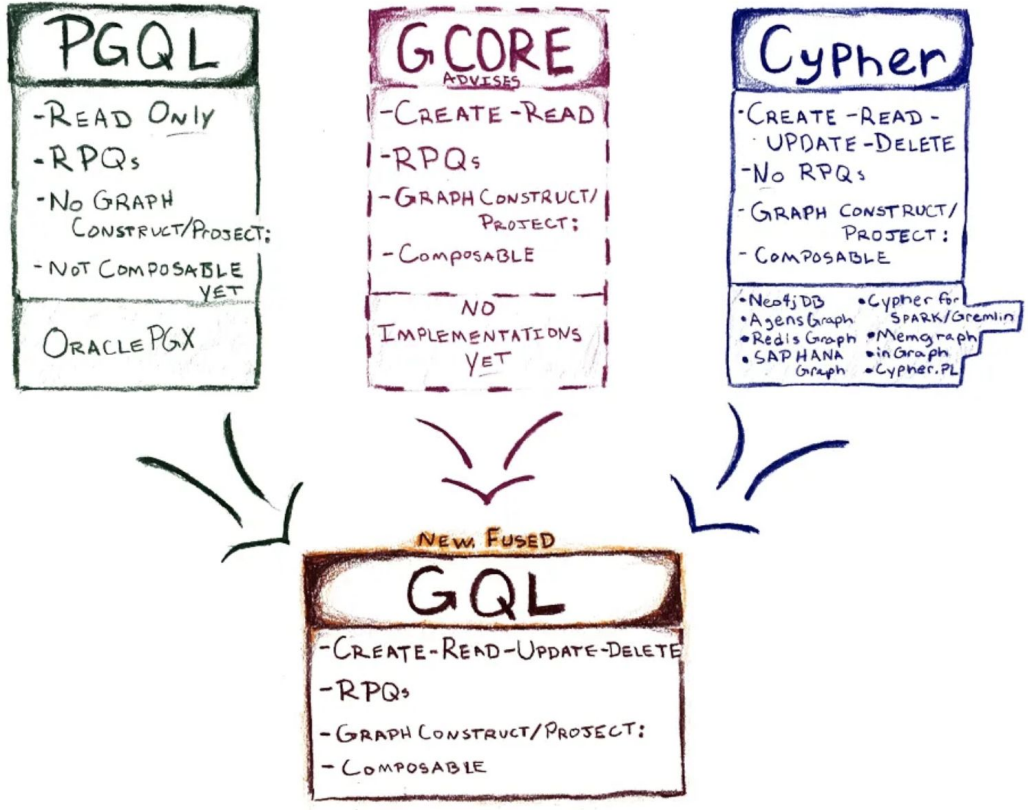
```
  | x                | p                                    |
  | ({name: 'B'})    | <({name: 'A'})-[:X]->({name: 'B'})> |
  | ({name: 'C'})    | null                                 |
```

And no side effects

Over 2K scenarios

Introducing Graph Query Language (GQL)

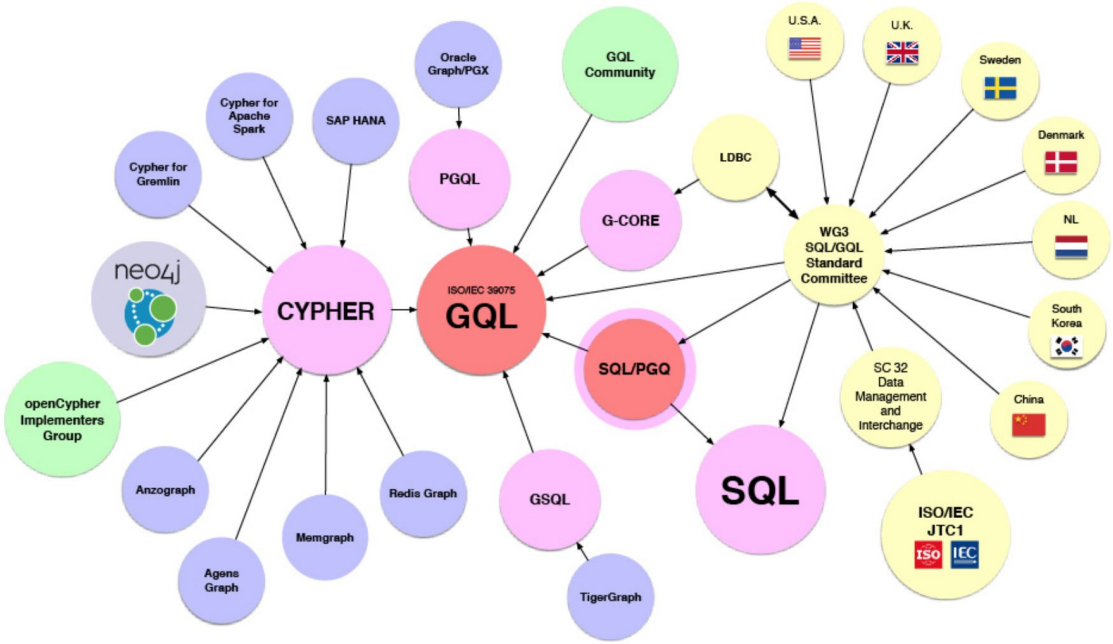
GQL Manifesto



gql.today

ISO GQL: A new standard Graph Query Language (GQL) for Property Graphs

- First International Standard Database Languages project since SQL in **1987**
- Successful ballot: Sept 2019
- 7 countries volunteer experts to define the language
- Cypher query language (openCypher) a major input



Support for Property Graph Queries in SQL

To appear in the next version of the **SQL Standard (ISO/IEC 9075-16)**

Represent a virtual graph, underpinned by a set of tables

Query this graph using pattern matching (syntax and semantics shared with GQL)

Optimistic release date (for Intl. Standard): 2022

Declarative Property Graph Language

GQL Standard (ISO/IEC 39075)

Undertaken in parallel with SQL/PGQ

Querying graphs (shared with SQL/PGQ) as well as DML

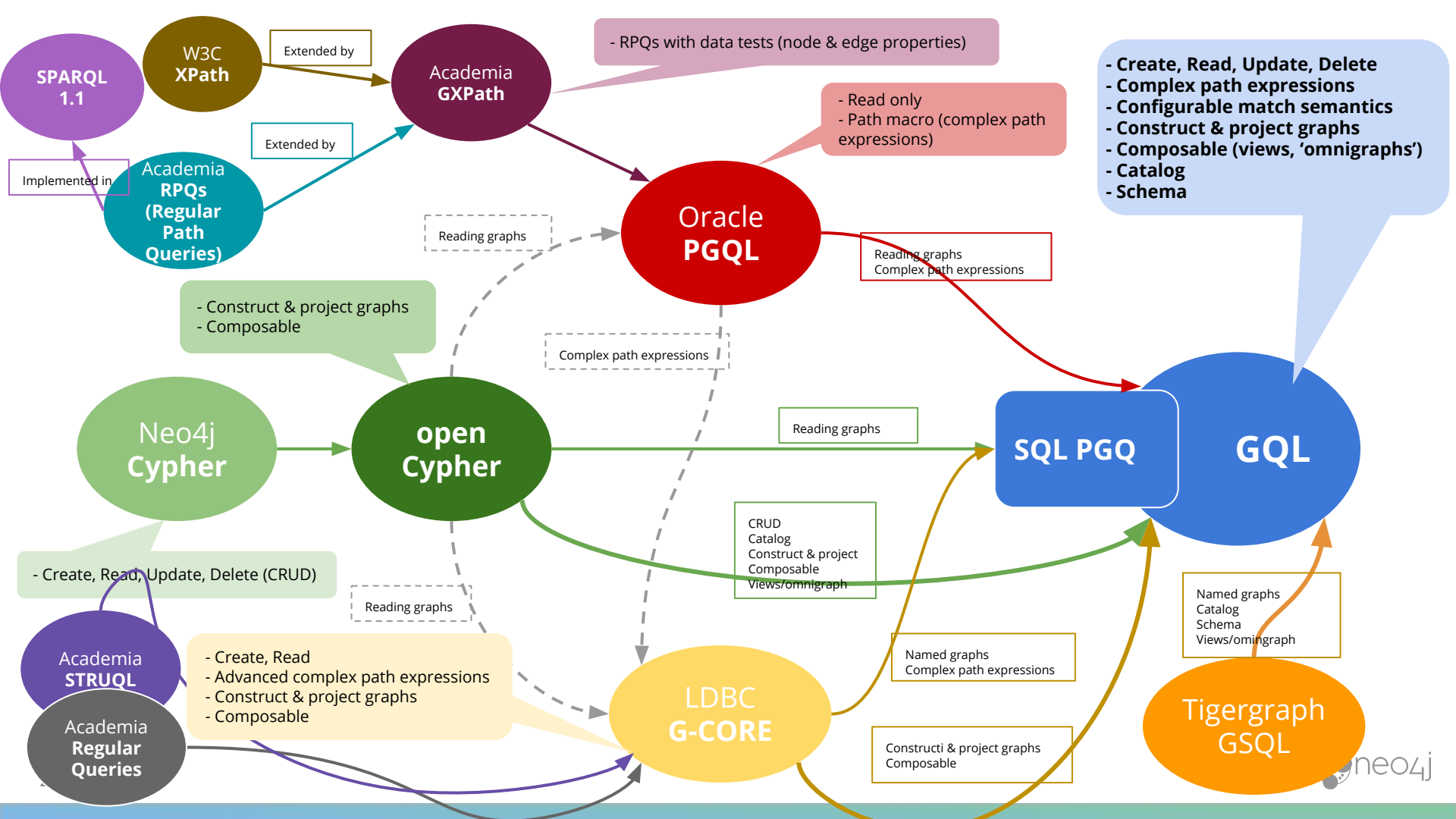
Allow for multiple graphs and composable querying in general - views, graph (& table) projections & transformations

Graph schema

Complex data types

Optimistic release date for **first version** (for Intl. Std): 2022

Future versions: streaming graphs, temporal support etc

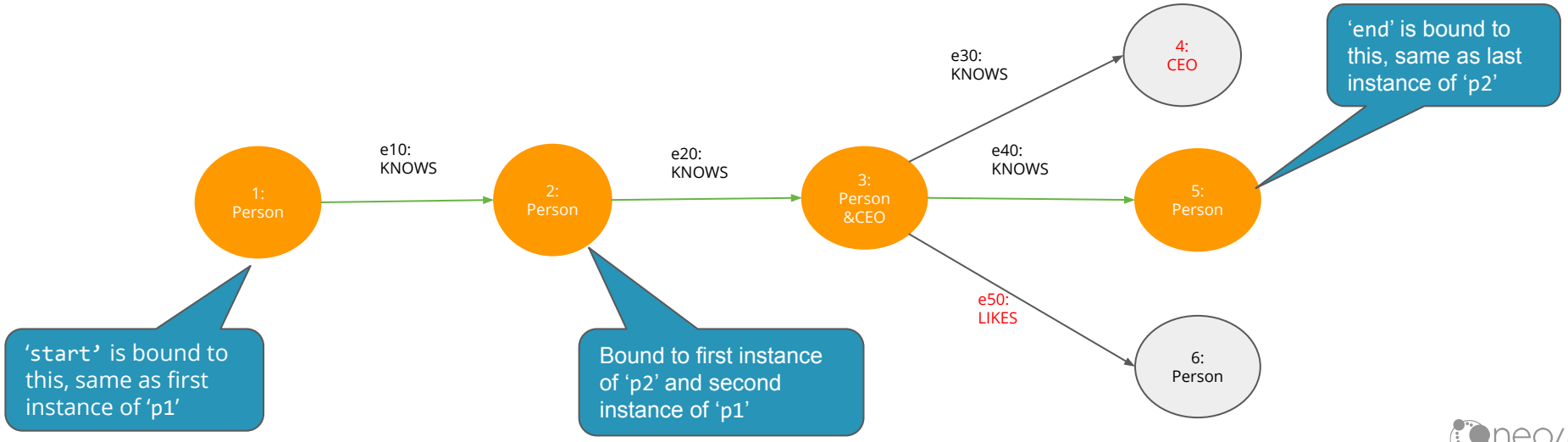

SPARQL 1.1
W3C XPath
Extended by
Academia GXPath
- RPQs with data tests (node & edge properties)
- Read only
- Path macro (complex path expressions)
- Create, Read, Update, Delete
- Complex path expressions
- Configurable match semantics
- Construct & project graphs
- Composable (views, 'omnigraphs')
- Catalog
- Schema
Implemented in
Extended by
Academia RPQs (Regular Path Queries)
Oracle PGQL
Reading graphs
Reading graphs
Complex path expressions
- Construct & project graphs
- Composable
Complex path expressions
Neo4j Cypher
open Cypher
Reading graphs
SQL PGQ
GQL
CRUD
Catalog
Construct & project
Composable
Views/omnigraph
- Create, Read, Update, Delete (CRUD)
Reading graphs
Named graphs
Catalog
Schema
Views/omingraph
Academia STRUQL
- Create, Read
- Advanced complex path expressions
- Construct & project graphs
- Composable
LDBC G-CORE
Named graphs
Complex path expressions
Academia Regular Queries
Constructi & project graphs
Composable
Tigergraph GSQL
neo4j

Proposed Extensions*

*Worked on under the auspices of the GQL standardization process, and will make it into a future version of openCypher as well as GQL

Repetition of Path Patterns
(based on Conjunctive Regular Path Queries)

```
MATCH (start) [(p1:Person)-[:KNOWS]-(p2:Person)]+ (end)
```



Path Patterns: some extensions

Node and edge label expressions:

```
MATCH (n:A&B)-[:!(C|D)]->(m:(E|F)&G)
```

Conjunction: `A&B`
Disjunction: `A|B`
Negation: `!A`
Grouping/nesting: `(A&B)|C`

Predicates on properties along a path:

```
MATCH (start) [ (p1:Person)-[r:KNOWS]-(p2:Person)
      WHERE p1.age < p2.age AND r.since < date("2001-09-11"]* (end)
```

Bounded repetition:

```
MATCH (me) [ (:Person)-[:KNOWS]->(:Person) ]{2,5} (you)
MATCH (me) [ (:Person)-[:KNOWS]->(:Person) ]{5} (you)
MATCH (me) [ (:Person)-[:KNOWS]->(:Person) ]{2,} (you)    //default upper bound = “infinity”
MATCH (me) [ (:Person)-[:KNOWS]->(:Person) ]{,5} (you)    //default lower bound = 0
```

Concatenation
a.b - a is followed by b
Alternation
a|b - either a or b
Transitive closure
***** - 0 or more
+ - 1 or more
{m, n} - at least m, at most n
Optionality:
? - 0 or 1
Grouping/nesting
() - allows nesting/defines scope

Configurable pattern-matching semantics

Node isomorphism

- No node occurs in a path more than once
- Most restrictive

Edge isomorphism

- No edge (relationship) occurs in a path more than once
- Proven in practice

Homomorphism

- A path can contain the same nodes and edges more than once
- Most efficient for some RPQs
- Least restrictive

Allow all three types of matching

All forms may be valid in different scenarios

Can be configured at a query level, or even at a pattern level

Path pattern output modifiers

Illustrative syntax only!

Controlling the path pattern-matching output semantics

`ALL`	- returns all paths
`[ALL] SHORTEST`	- for shortest path patterns of equal length (computed by number of edges).
`ANY SHORTEST`	- any of the shortest possible paths.

Variations also include getting the *k* shortest paths or groups of paths

Some of these operations may be non-deterministic

Data types

Scalar data types

- Numeric, string, boolean, temporal etc

Sharing some data types with SQL's type system

Collection data types

- Maps with arbitrary keys as well as maps with a fixed set of typed fields (anonymous structs): `{name: "GQL", type: "language", age: 0 }`
- Ordered and unordered sequences with and without duplicates: `[1, 2, 3]`

Graph-related data types

- Nodes and edges (with intrinsic identity)
- Paths
- Graphs

Support for
- Comparison and equality
- Sorting and equivalence

Schema

"Classic" property graphs: historically schema-free/optional

- This is very useful in practice - retain the ability to be schema free
- Also provide the ability to have a schema in cases where this is needed

Moving towards a more comprehensive graph schema

- Element types:
    - Permitted set of labels and properties {name, data type} on a node or edge
    - Future extension: permitted endpoint node types for an edge type
- Extended with unique key and cardinality constraints

Multiple graphs and graph projection

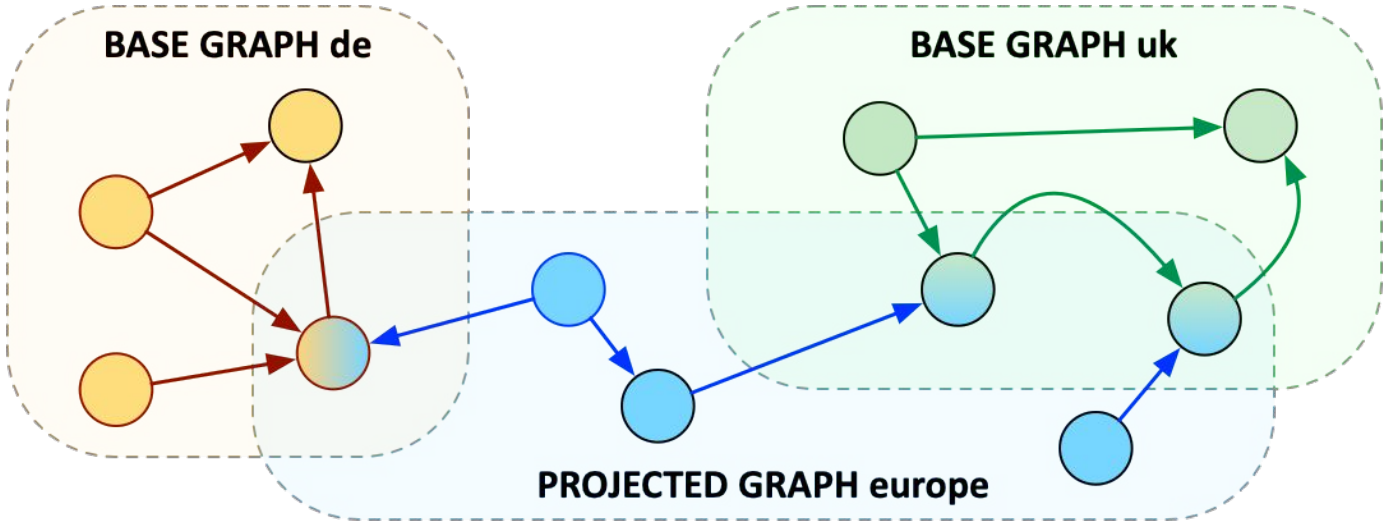


Image courtesy of Stefan Plantikow

Allowing for *multiple named* graphs

Allowing for graph projection:

- Sharing elements in the projected graph
- Deriving new elements in the projected graph
- Shared edges always point to the same (shared) endpoints in the projected graph

Queries are composable procedures

- Use the output of one query as input to another to enable abstraction and views
- Applies to queries with tabular output and graph output
- Support for nested subqueries
- Extract parts of a query to a view for re-use
- Replace parts of a query without affecting other parts
- Build complex workflows programmatically
- Enables: application views; access control; derived graphs / reasoning; data integration; graph operations

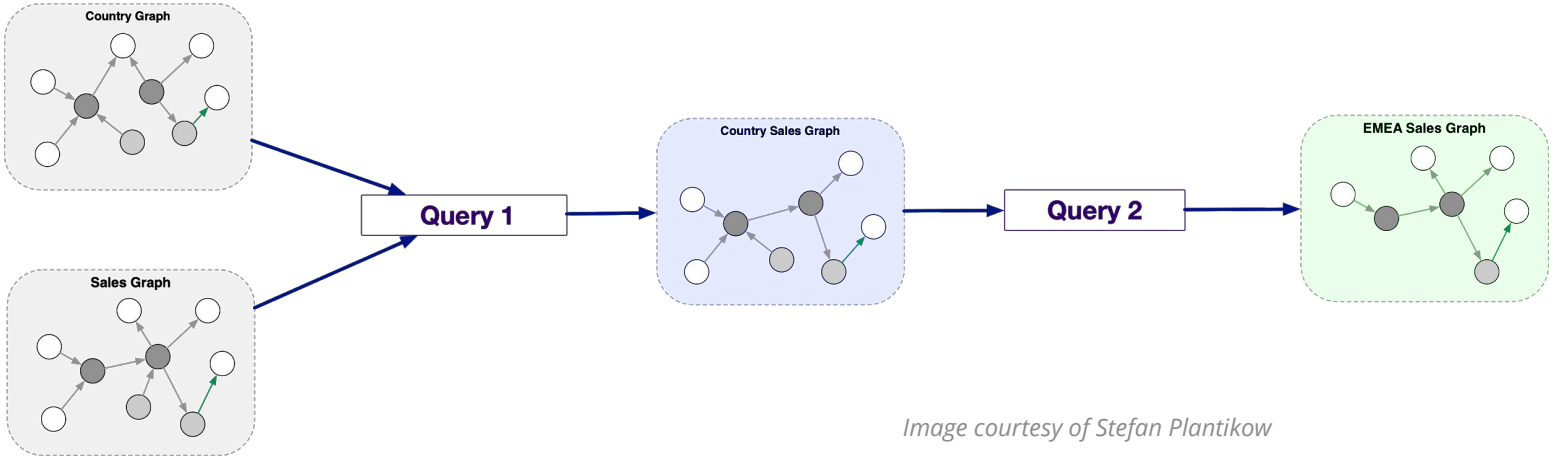


Image courtesy of Stefan Plantikow

(Some) key papers in the story so far…

A graphical query language supporting recursion.

I. F. Cruz, A. O. Mendelzon, and P. T. Wood. 1987.

Declarative specification of web sites with STRUDEL.

M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. 2000.

Querying Graphs with Data.

L. Libkin, W. Martens, and D. Vrgoč. 2016.

PGQL: A Property Graph Query Language.

O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. 2016.

Regular Queries on Graph Databases.

J. L. Reutter, M. Romero, and M. Y. Vardi. 2017.

Cypher: An Evolving Query Language for Property Graphs.

N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. 2018.

G-CORE: A Core for Future Graph Query Languages.

R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. 2018.

Updating Graph Databases with Cypher

A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Schuster, P. Selmer, and H. Voigt. 2019.

This is a very small subset. Research in this area spans decades.

Thank you!

Links:

- Neo4j Documentation: https://neo4j.com/docs/
- Use cases: https://neo4j.com/use-cases/
- Graph Databases (book available online at www.graphdatabases.com)
- Getting started: http://neo4j.com/developer/get-started/
- Online training: http://neo4j.com/graphacademy/
- openCypher: http://www.opencypher.org/

petra.selmer@neo4j.com