Formal semantics of Cypher:
towards a standard language for querying property graphs

N. Francis [1]
V. Marsault [1]
A. Green [2]
T. Lindaaker [2]
S. Plantikow [2]
M. Rydberg [2]
P. Selmer [2]
A. Taylor [2]
P. Guagliardo [3]
L. Libkin [3]
M. Schuster [3]

1. Univ. Paris-Est Marne-la-Vallée, ENPC, ESIEE, CNRS
2. Neo Technology
3. University of Edinburgh

GT Alga
Inria Nord Europe, Lille
2018-10-15

Outline

1 Introduction

2 Property graphs

3 Regular Path Queries

4 Cypher by example

5 Principles of the semantics

6 Towards a standard language for querying property graphs

Representing data as graphs

Most databases use the relational model

- Relational algebra in theory
- The language SQL in practice

Representing data as graphs

Most databases use the relational model

- Relational algebra in theory
- The language SQL in practice

Some data have intrinsically the structure of graphs:

- Semantic web
- Social Networks
- Bioinformatic networks

Native representation of data as graphs allows:

- Efficient algorithms on graphs
- Pattern matching
- Optimisations

More and more ways to manipulate data as graphs

Data model

Property graphs, RDF

Query languages

Cypher, Gremlin, PGQL, SPARQL, G-Core

Engines

JanusGraph, Jena, Neo4j, Virtuoso

Domain

Fraud detection, Investigative journalism

More and more ways to manipulate data as graphs

Data model

Property graphs, RDF

Query languages

Cypher, Gremlin, PGQL, SPARQL, G-Core

Engines

JanusGraph, Jena, Neo4j, Virtuoso

Domain

Fraud detection, Investigative journalism

Cypher

- Language for querying and updating *property graphs*

- Invented by Neo Technology
- Originally, part of engine Neo4j $\longrightarrow$ commercial success
- Now, in multiple datagraph engines (e.g., SAP HANA Graph, Redis Graph, Agens Graph)

Cypher

- Language for querying and updating *property graphs*

- Invented by Neo Technology
- Originally, part of engine Neo4j $\longrightarrow$ commercial success
- Now, in multiple datagraph engines (e.g., SAP HANA Graph, Redis Graph, Agens Graph)

The openCypher project

- Since 2015
- Seeks to standardise Cypher (SQL for property graphs?)
  - Community-led evolution
  - Complete specification

Short term goal

Full denotational semantics for the language Cypher.

- Industrial partnership Neo4j/University of Edinburgh
- Reverse engineering and formalisation from Neo4j
- Done semantics of the “core fragment’ [Francis et al’18]
- Soon: semantics of the “update clauses

Short term goal

Full denotational semantics for the language Cypher.

- Industrial partnership Neo4j/University of Edinburgh
- Reverse engineering and formalisation from Neo4j
- Done semantics of the “core fragment’ [Francis et al’18]
- Soon: semantics of the “update clauses

Long term goal

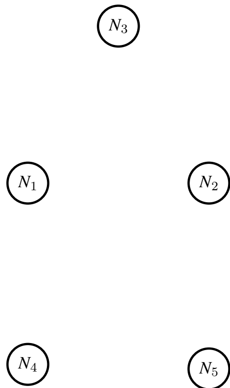
Design a standard language for querying property graphs: GQL.

- Merging Cypher, PGQL, G-Core.
- Involvement of Neo Technology, Oracle and LDBC.

Outline

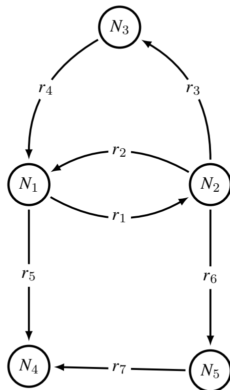
1 Introduction
2 Property graphs
3 Regular Path Queries
4 Cypher by example
5 Principles of the semantics
6 Towards a standard language for querying property graphs

Property graph



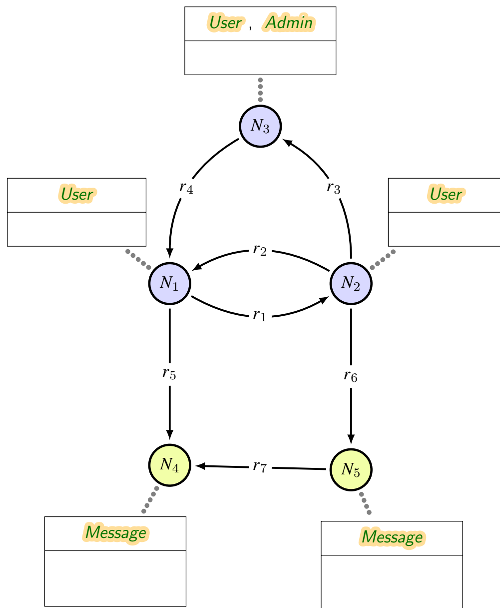
- Nodes : $N_1, N_2, \cdots, N_5$

Property graph



- Nodes : $N_1, N_2, \cdots, N_5$
- Relationships : $r_1, r_2, \cdots, r_7$

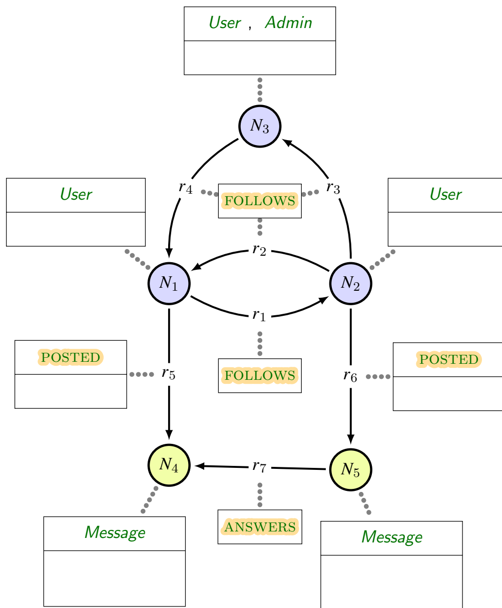
Property graph



- Nodes : $N_1, N_2, \cdots, N_5$
- Relationships : $r_1, r_2, \cdots, r_7$

- Labels (de nœuds) :
  - *User*
  - *Message*

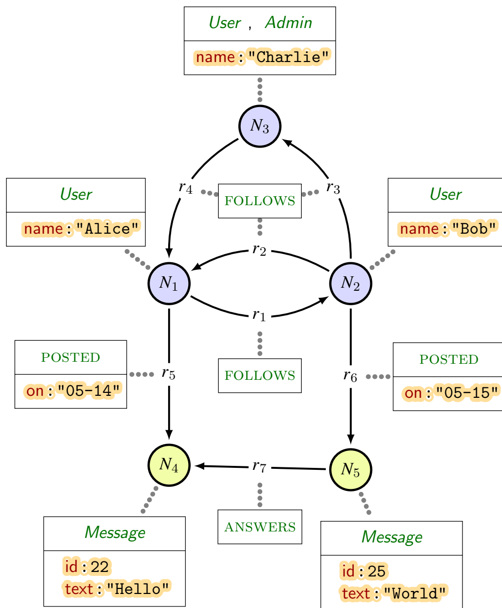
Property graph



- Nodes : $N_1, N_2, \cdots, N_5$
- Relationships : $r_1, r_2, \cdots, r_7$

- Labels (de nœuds) :
  - *User*
  - *Message*
- Types (of relationships) :
  - FOLLOWS
  - POSTED
  - ANSWERS

Property graph



- Nodes : $N_1, N_2, \cdots, N_5$
- Relationships : $r_1, r_2, \cdots, r_7$

- Labels (de nœuds) :
  - *User*
  - *Message*
- Types (of relationships) :
  - FOLLOWS
  - POSTED
  - ANSWERS
- Properties (*i.e.* Key/Value pairs) :
  - name : "Alice"
  - id : 22
  - text : "Hello"

Outline

1 Introduction

2 Property graphs

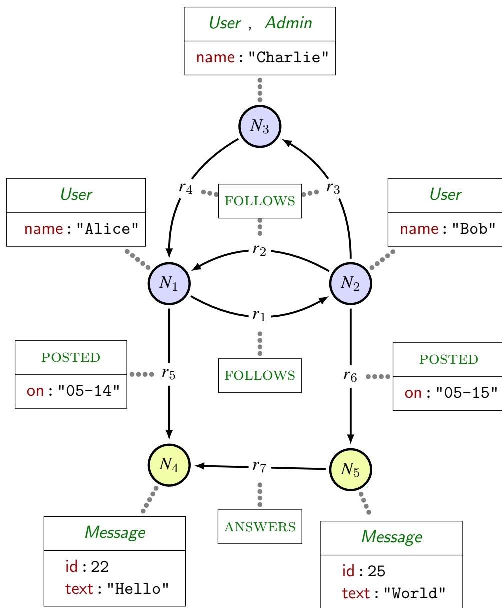
3 Regular Path Queries

4 Cypher by example

5 Principles of the semantics

6 Towards a standard language for querying property graphs

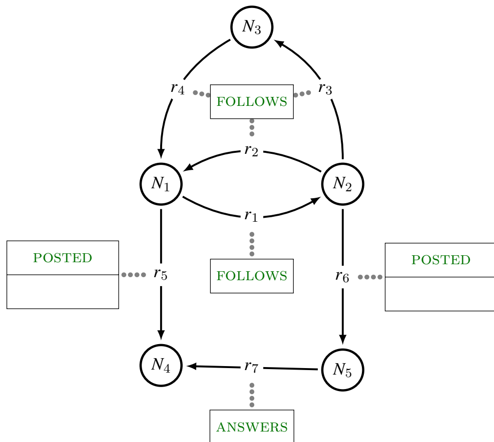
Data model for theoretical query languages



Graph database

- Relation bear types
- Node do not bear types (could be simulated)
- **Neither bears property**

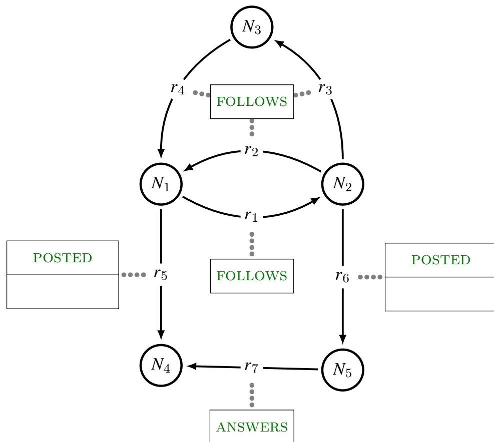
Data model for theoretical query languages



Graph database

- Relation bear types
- Node do not bear types (could be simulated)
- **Neither bears property**

Data model for theoretical query languages



Graph database

- Relation bear types
- Node do not bear types (could be simulated)
- **Neither bears property**

Finite number of relation types (and node types)
→ "Dataless graph"

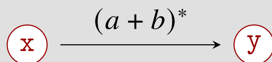
A: finite alphabet (of relation types)
N: nodes in the graph

Definition (RPQ R):

$$R = (x, E, y)$$

$$\begin{cases} E : & \text{regular expr. over } A \\ x, y : & \text{two variables} \end{cases}$$

Example:

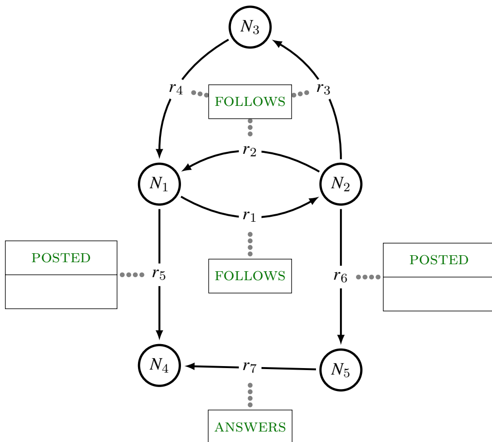


Definition (Answer to R):

Set of the functions

$$F : \{x, y\} \longrightarrow N \ , \quad \exists u \in E,\ F(x) \xrightarrow{u} F(y)$$

Regular Path Query (RPQ) – Example 1

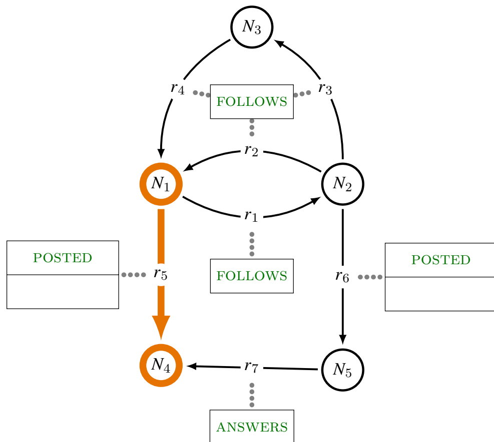


Query:

Answers:

$$F_1 : \quad x \mapsto N_1 \quad y \mapsto N_4$$
$$F_2 : \quad x \mapsto N_2 \quad y \mapsto N_5$$
$$F_3 : \quad x \mapsto N_2 \quad y \mapsto N_4$$

Regular Path Query (RPQ) – Example 1



Query:

$x \xrightarrow{\text{POSTED.(ANSWERS)}^*} y$

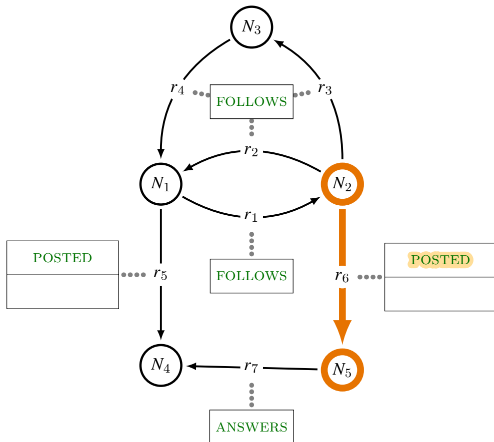
Answers:

$F_1: \quad x \mapsto N_1 \quad y \mapsto N_4$

$F_2: \quad x \mapsto N_2 \quad y \mapsto N_5$

$F_3: \quad x \mapsto N_2 \quad y \mapsto N_4$

Regular Path Query (RPQ) – Example 1



Query:

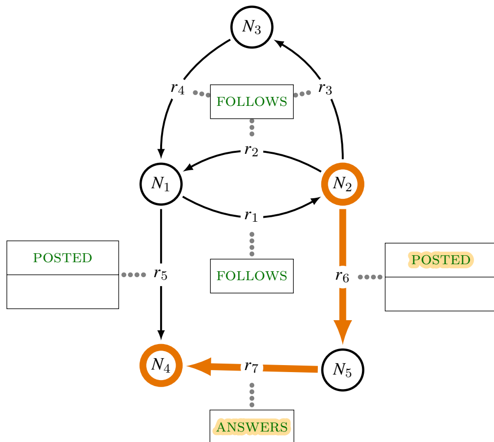
Answers:

$F_1 : \quad x \mapsto N_1 \quad y \mapsto N_4$

$F_2 : \quad x \mapsto N_2 \quad y \mapsto N_5$

$F_3 : \quad x \mapsto N_2 \quad y \mapsto N_4$

Regular Path Query (RPQ) – Example 1



Query:

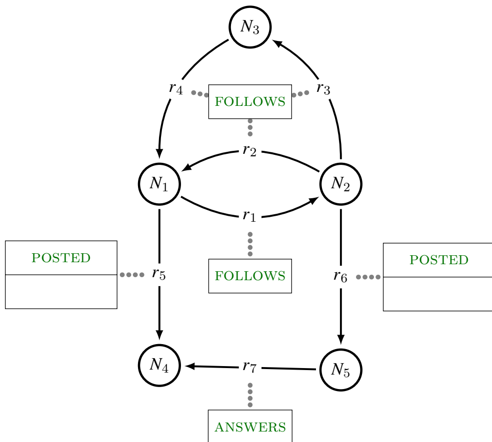
Answers:

$F_1: \quad x \mapsto N_1 \quad y \mapsto N_4$

$F_2: \quad x \mapsto N_2 \quad y \mapsto N_5$

$F_3: \quad x \mapsto N_2 \quad y \mapsto N_4$

Regular Path Query (RPQ) – Example 1

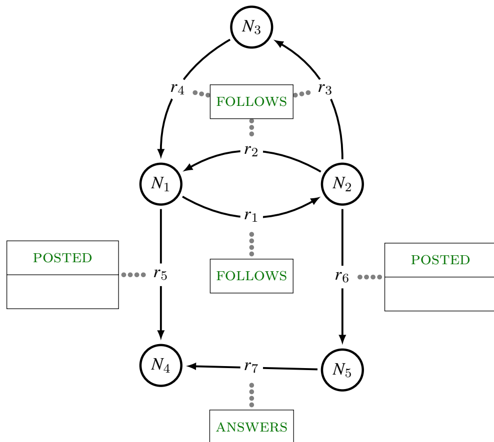


Query:

Answers:

$$F_1 : \quad x \mapsto N_1 \quad y \mapsto N_4$$
$$F_2 : \quad x \mapsto N_2 \quad y \mapsto N_5$$
$$F_3 : \quad x \mapsto N_2 \quad y \mapsto N_4$$

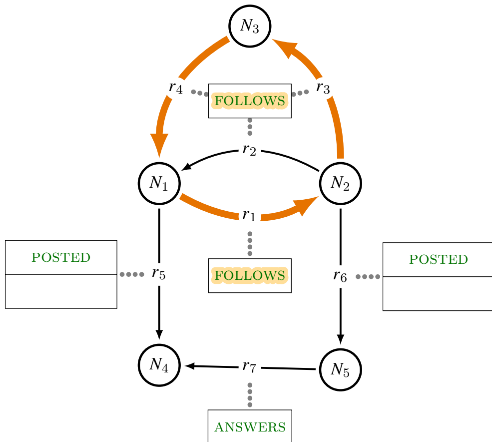
Regular Path Query (RPQ) – Example 2



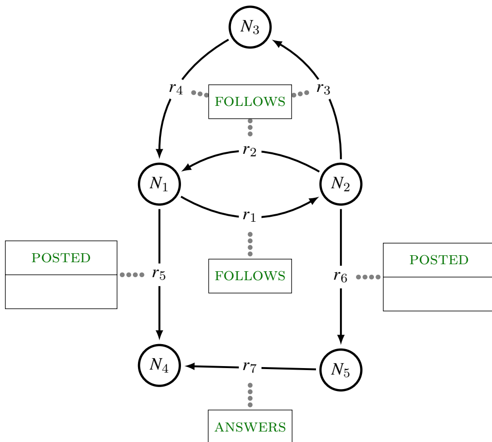
Query:

Regular Path Query (RPQ) – Example 2

Query:



Regular Path Query (RPQ) – Example 2

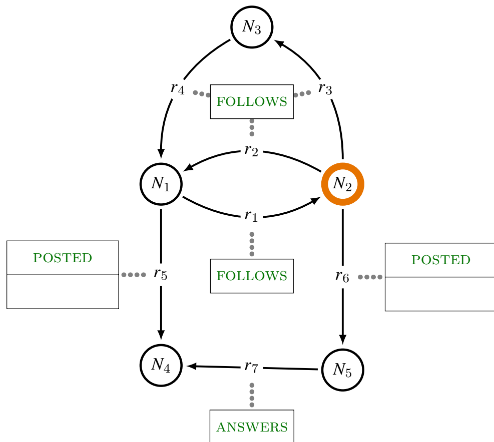


Query:

Answers:

$$
\begin{array}{llll}
F_1: & x \mapsto N_1 & y \mapsto N_1 \\
F_2: & x \mapsto N_1 & y \mapsto N_2 \\
F_3: & x \mapsto N_1 & y \mapsto N_3 \\
\vdots & \vdots & \vdots \\
F_9: & x \mapsto N_3 & y \mapsto N_3
\end{array}
$$

Regular Path Query (RPQ) – Example 2

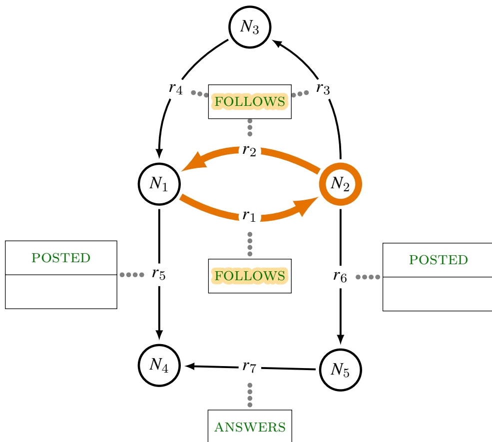


Query:

Answers:

F_1 :	$x \mapsto N_1$	$y \mapsto N_1$
F_2 :	$x \mapsto N_1$	$y \mapsto N_2$
F_3 :	$x \mapsto N_1$	$y \mapsto N_3$
$\vdots$	$\vdots$	$\vdots$
F_9 :	$x \mapsto N_3$	$y \mapsto N_3$

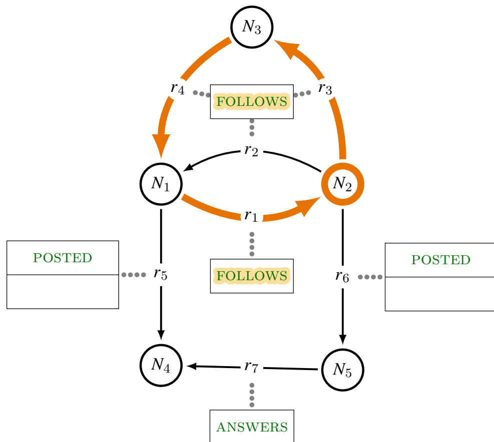
Regular Path Query (RPQ) – Example 2



Query:

Answers:

$$F_1 : \quad x \mapsto N_1 \quad y \mapsto N_1$$
$$F_2 : \quad x \mapsto N_1 \quad y \mapsto N_2$$
$$F_3 : \quad x \mapsto N_1 \quad y \mapsto N_3$$
$$\vdots \quad \vdots \quad \vdots$$
$$F_9 : \quad x \mapsto N_3 \quad y \mapsto N_3$$

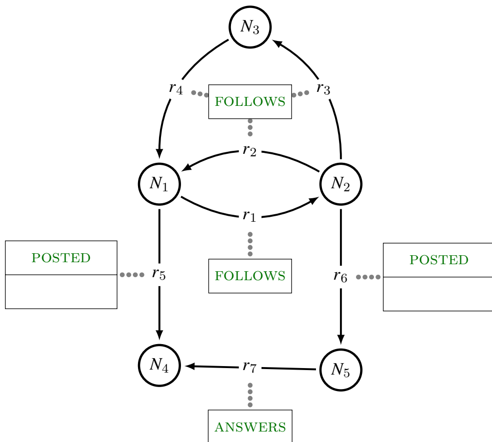


Query:

Answers:

F_1 :	$x \mapsto N_1$	$y \mapsto N_1$
F_2 :	$x \mapsto N_1$	$y \mapsto N_2$
F_3 :	$x \mapsto N_1$	$y \mapsto N_3$
$\vdots$	$\vdots$	$\vdots$
F_9 :	$x \mapsto N_3$	$y \mapsto N_3$

Regular Path Query (RPQ) – Example 2



Query:

Answers:

F_1 :	$x \mapsto N_1$	$y \mapsto N_1$
F_2 :	$x \mapsto N_1$	$y \mapsto N_2$
F_3 :	$x \mapsto N_1$	$y \mapsto N_3$
$\vdots$	$\vdots$	$\vdots$
F_9 :	$x \mapsto N_3$	$y \mapsto N_3$

$\longrightarrow$ ∞-many path realise F_1

$\longrightarrow$ RPQ follows set-semantics

Conjunction of RPQs (CRPQ) – Definition

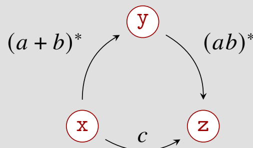
A: alphabet (of relation types)
N: nodes in the graph

Definition (a CRPQ C):

$$C = (R_1 \wedge R_2 \wedge \cdots \wedge R_n)$$

where $R_1, , \ldots, R_n$ are RPQs

Example:

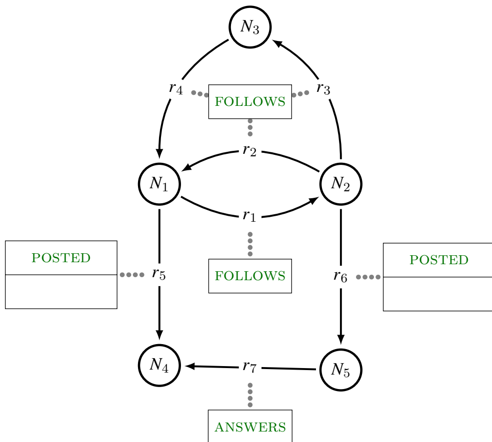


Answer to C:

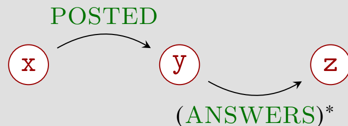
Set of the functions $F : \mathsf{var}(C) \longrightarrow N$

such that $\forall i,\ F_{|\mathsf{var}(R_i)}$ is an answer to R_i

Conjunction of RPQs (CRPQ) – Example



Query:



Answers:

	x	y	z
	↧	↧	↧
F_1 :	N_1	N_4	N_4
F_2 :	N_2	N_5	N_4
F_3 :	N_2	N_5	N_5

A: alphabet (of relation types)
N: nodes in the graph

Definition (UCRPQ Q)

$$Q = (C_1 \cup C_2 \cup \cdots \cup C_n)$$

where C_1, C_2, $\ldots$, C_n are CRPQs

Answer to Q:

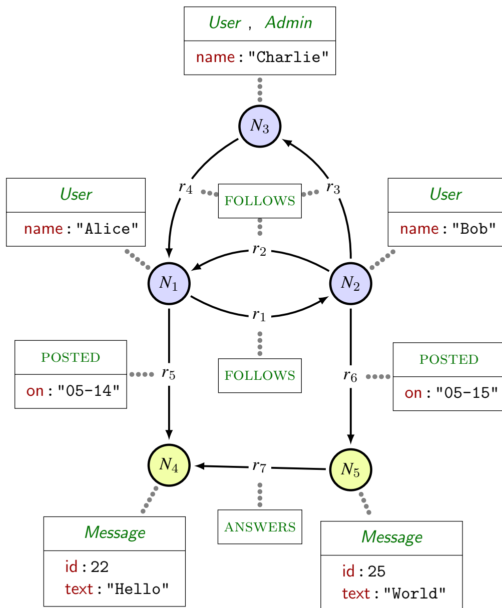
Set of partial functions: $\big(\mathrm{var}(C_1) \cup \cdots \cup \mathrm{var}(C_n)\big) \to N$

$$\bigcup_{i=1}^{n} F_i \ ,$$

where,$\forall i$, F_i is the answer to C_i.

1. Introduction
2. Property graphs
3. Regular Path Queries
4. Cypher by example
5. Principles of the semantics
6. Towards a standard language for querying property graphs

Evaluating a Cypher query



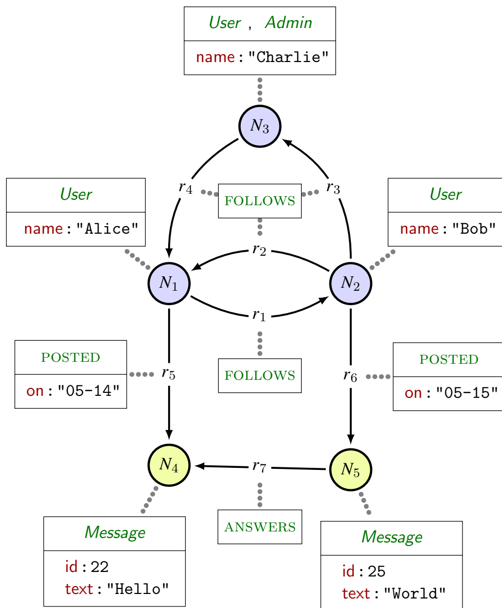
Example of Cypher query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
WHERE t1 = "Hello"
```

A Cypher statement

- is a sequence of *clauses*

Evaluating a Cypher query



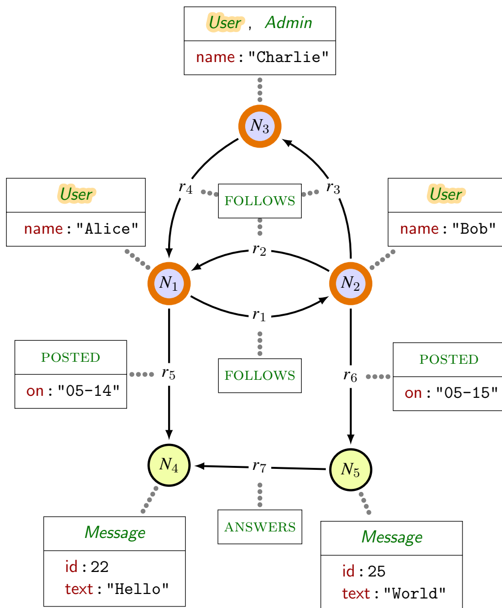
Example of Cypher query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
WHERE t1 = "Hello"
```

A Cypher statement

- is a sequence of *clauses*
- queries a graph
- returns a table

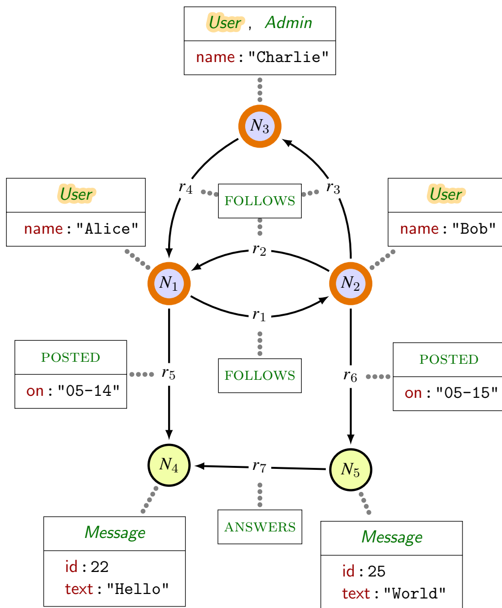
Matching nodes (1)



Query:

```
MATCH (u1:User)
```

Matching nodes (1)



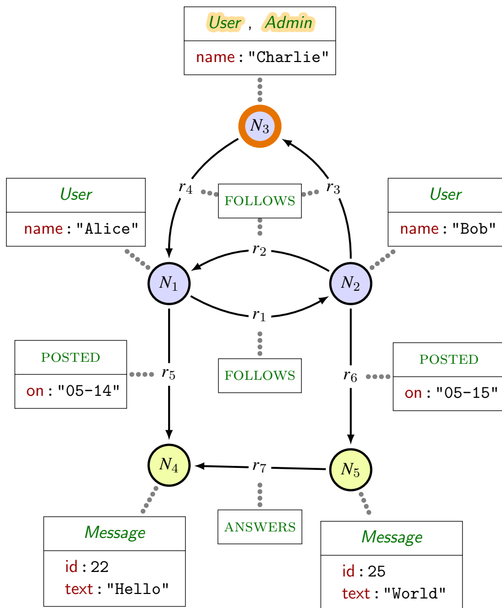
Query:

```
MATCH (u1:User)
```

Result:

u1
N_1
N_2
N_3

Matching nodes (2)



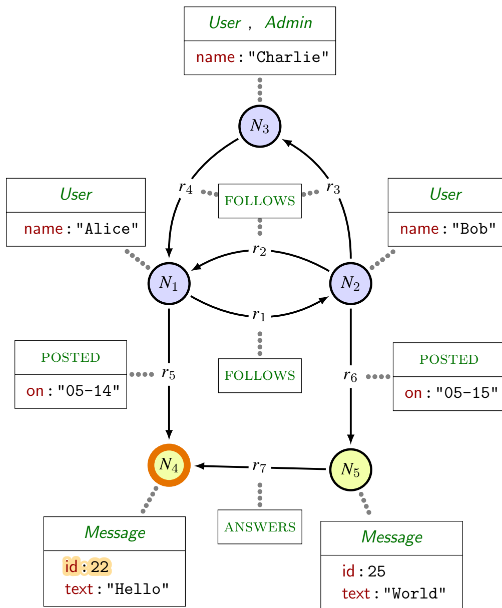
Query:

```
MATCH (u1:User:Admin)
```

Result:

u1
N_3

Matching nodes (3)



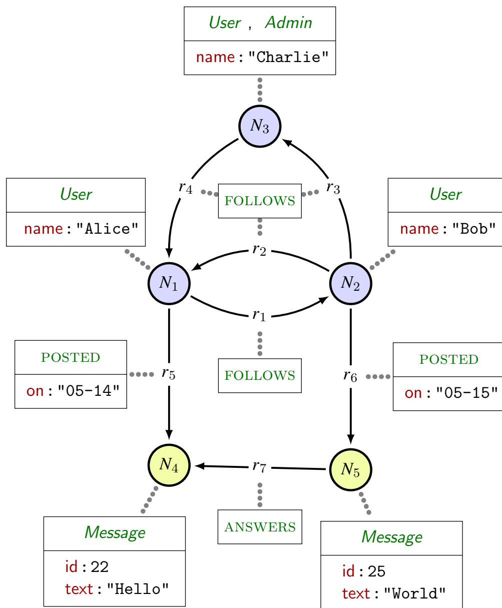
Query:

```
MATCH (u1{id:22})
```

Result:

u1
N_4

Matching relations (1)



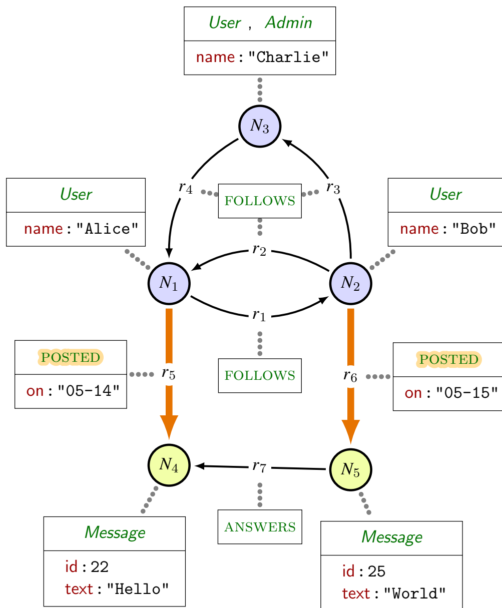
Query:

```
MATCH ()-[p1]->()
```

Result:

p1
r_1
r_2
r_3
r_4
r_5
r_6
r_7

Matching relation (2)



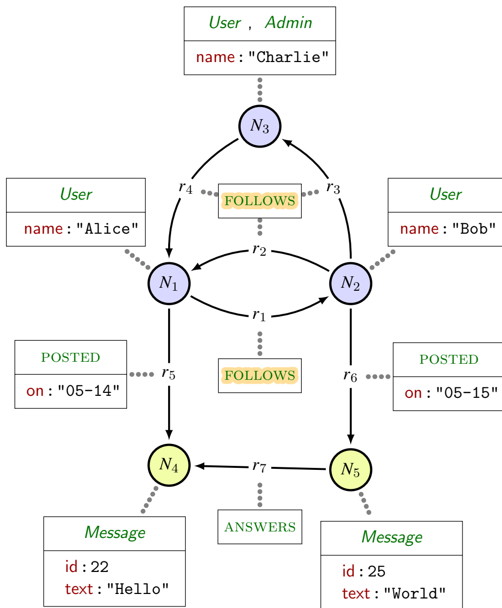
Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
```

Result:

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Matching relationships (3)



Query:

```
MATCH (u1)-[:FOLLOWS]->()
```

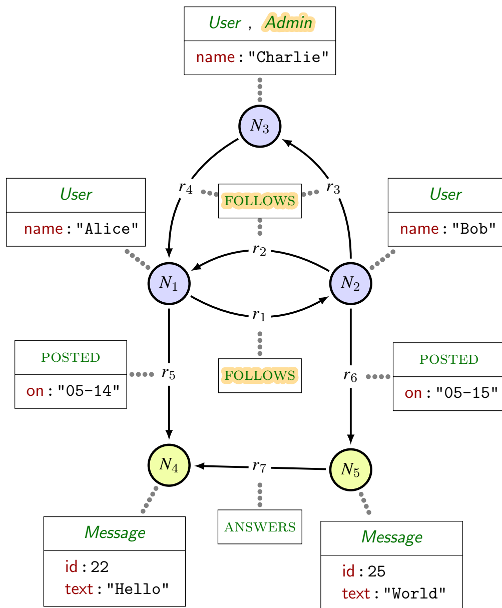
Result:

u1
N_1
N_2
N_2
N_3

Cypher has bag semantics:

N_2 has two outgoing FOLLOWS relations $\Rightarrow$ two lines N_2

Matching paths (1)



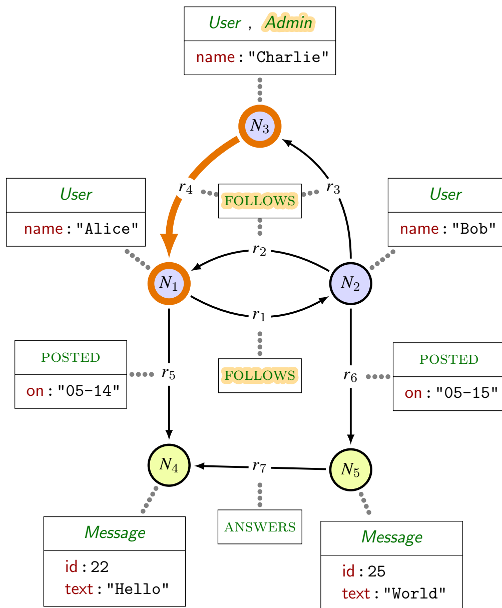
Query:

```
MATCH (u1:Admin)
          -[l1:FOLLOWS*]->(m1)
```

Result:

u1	l1	m1
N_3	$[r_4]$	N_1
N_3	$[r_4, r_1]$	N_2
N_3	$[r_4, r_1, r_2]$	N_1
N_3	$[r_4, r_1, r_3]$	N_3

Matching paths (1)



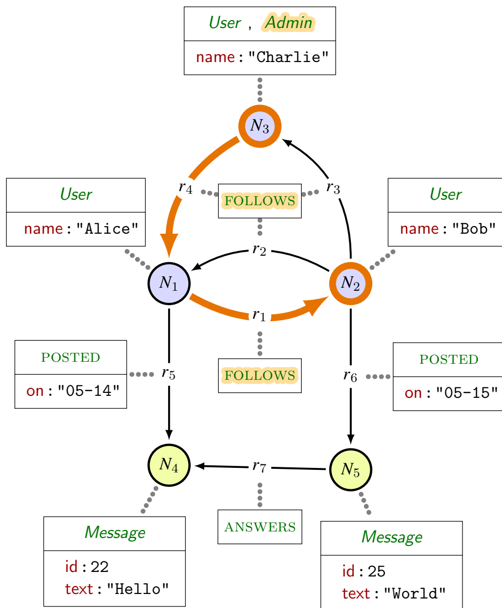
Query:

```
MATCH (u1:Admin)
        -[l1:FOLLOWS*]->(m1)
```

Result:

u1	l1	m1
N_3	$[r_4]$	N_1
N_3	$[r_4, r_1]$	N_2
N_3	$[r_4, r_1, r_2]$	N_1
N_3	$[r_4, r_1, r_3]$	N_3

Matching paths (1)



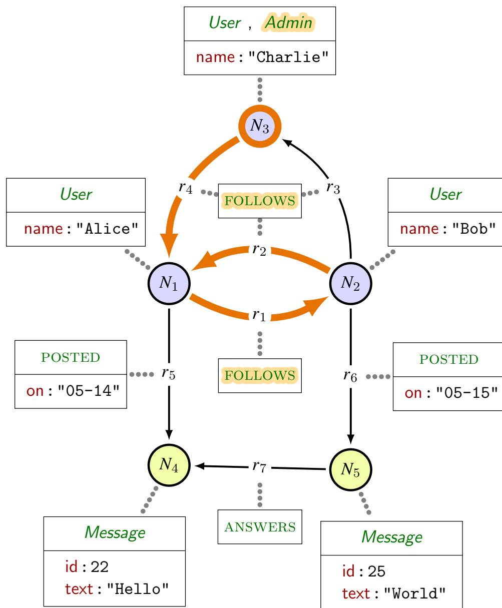
Query:

```
MATCH (u1:Admin)
        -[l1:FOLLOWS*]->(m1)
```

Result:

u1	l1	m1
N_3	$[r_4]$	N_1
N_3	$[r_4, r_1]$	N_2
N_3	$[r_4, r_1, r_2]$	N_1
N_3	$[r_4, r_1, r_3]$	N_3

Matching paths (1)



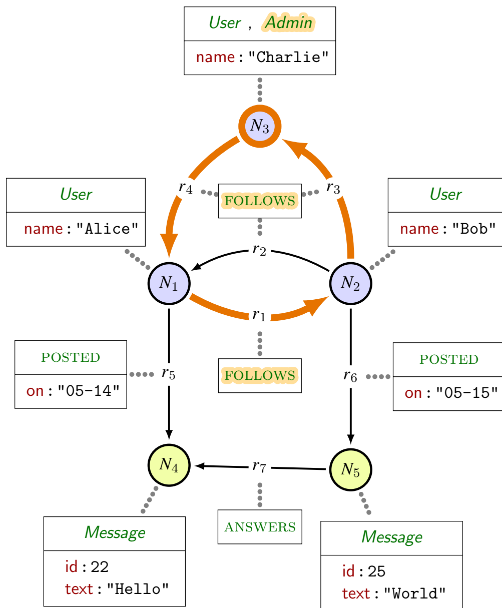
Query:

```
MATCH (u1:Admin)
          -[l1:FOLLOWS*]->(m1)
```

Result:

u1	l1	m1
N_3	$[r_4]$	N_1
N_3	$[r_4, r_1]$	N_2
N_3	$[r_4, r_1, r_2]$	N_1
N_3	$[r_4, r_1, r_3]$	N_3

Matching paths (1)

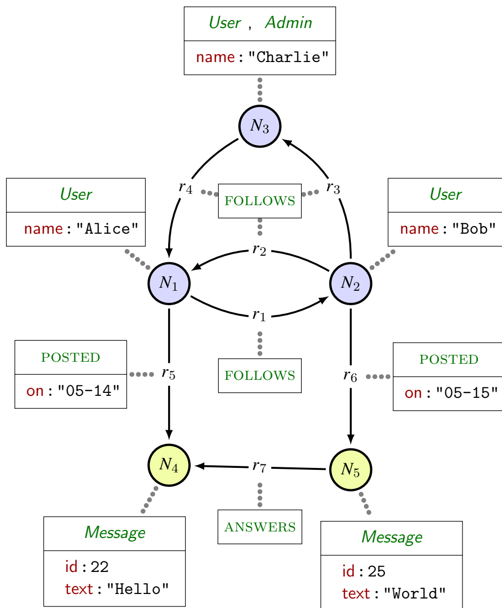


Query:

```
MATCH (u1:Admin)
        -[l1:FOLLOWS*]->(m1)
```

Result:

u1	l1	m1
N_3	$[r_4]$	N_1
N_3	$[r_4, r_1]$	N_2
N_3	$[r_4, r_1, r_2]$	N_1
N_3	$[r_4, r_1, r_3]$	N_3



Query:

```
MATCH (u1:Admin)
        -[l1:FOLLOWS*]->(m1)
```

Result:

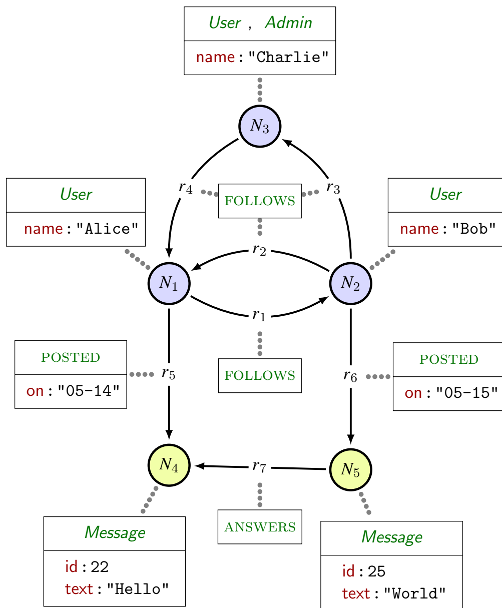
u1	l1	m1
N_3	$[r_4]$	N_1
N_3	$[r_4, r_1]$	N_2
N_3	$[r_4, r_1, r_2]$	N_1
N_3	$[r_4, r_1, r_3]$	N_3

Cypher-Morphism

- Each rel. matched ≤ 1 time

$\Rightarrow$ Finitely many results

Matching paths (2)



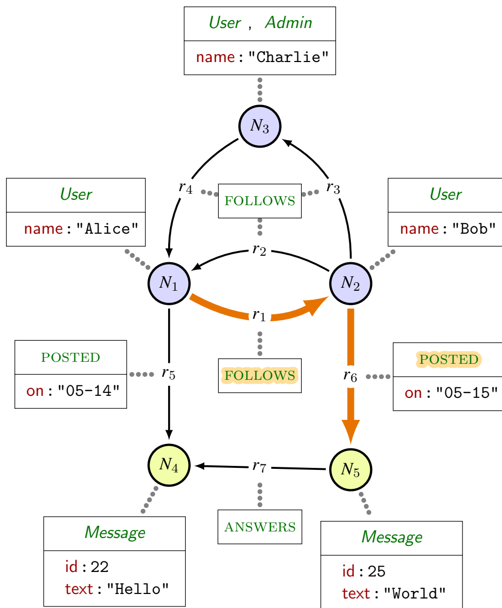
Query:

```
MATCH (u1)-[:FOLLOWS]->()
            -[:POSTED]->(m1)
```

Result:

u1	m1
N_1	N_5
N_2	N_4
N_3	N_5

Matching paths (2)



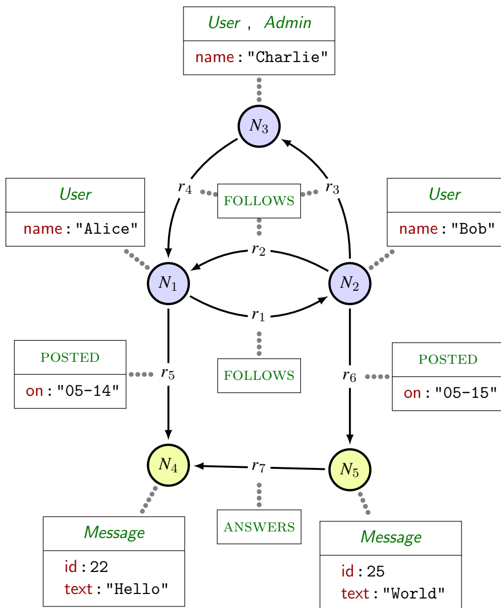
Query:

```
MATCH (u1)-[:FOLLOWS]->()
              -[:POSTED]->(m1)
```

Result:

u1	m1
N_1	N_5
N_2	N_4
N_3	N_5

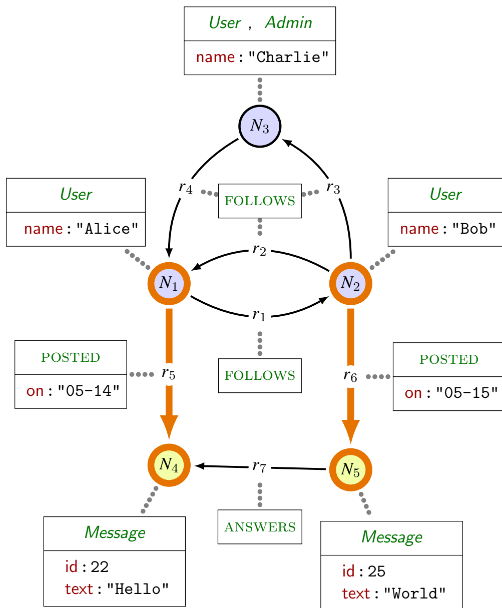
Chaining clauses



Query:

```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
               -[:FOLLOWS]->(u3)
```

Chaining clauses



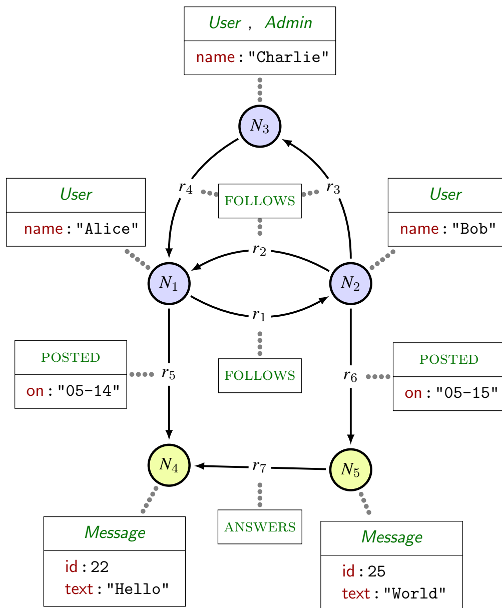
Query:

```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
                -[:FOLLOWS]->(u3)
```

Table after first MATCH:

u1	m1
N_1	N_4
N_2	N_5

Chaining clauses



Query:

```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
                  -[:FOLLOWS]->(u3)
```

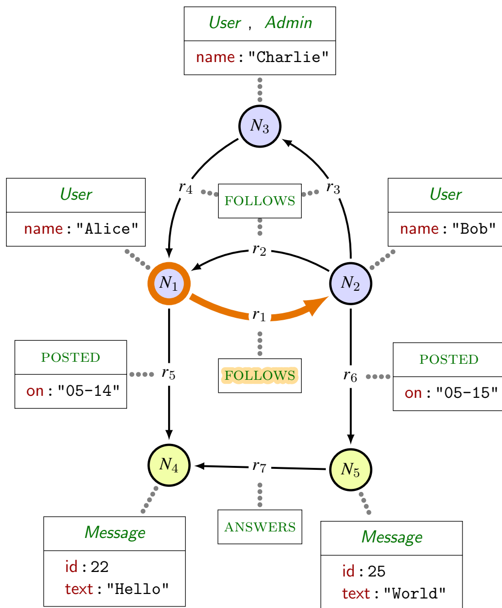
Table after first MATCH:

u1	m1
N_1	N_4
N_2	N_5

Table after second MATCH:

u1	m1	u2	u3
N_1	N_4	·	·
N_2	N_5	·	·

Chaining clauses



Query:

```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
                      -[:FOLLOWS]->(u3)
```

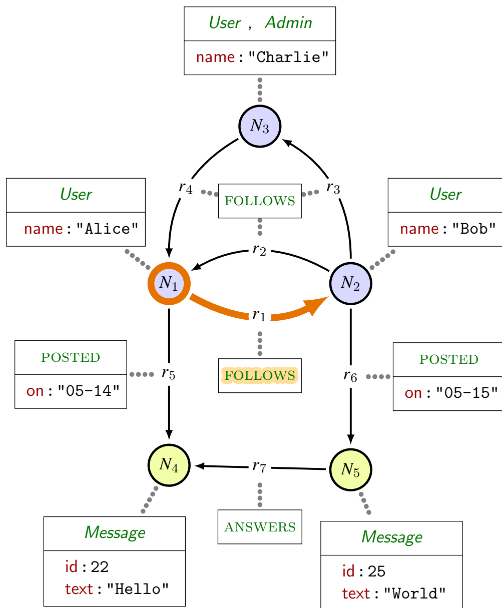
Table after first MATCH:

u1	m1
N_1	N_4
N_2	N_5

Table after second MATCH:

u1	m1	u2	u3
N_1	N_4	·	·
N_2	N_5	·	·

Chaining clauses



Query:

```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
                -[:FOLLOWS]->(u3)
```

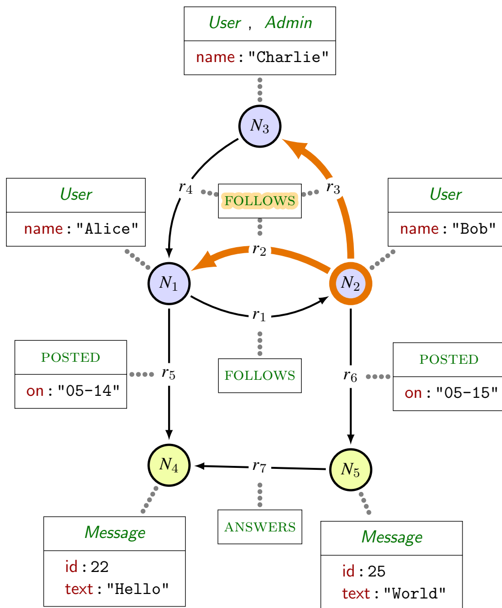
Table after first MATCH:

u1	m1
N_1	N_4
N_2	N_5

Table after second MATCH:

u1	m1	u2	u3
~~N_1~~	~~N_4~~		
N_2	N_5	·	·

Chaining clauses



Query:

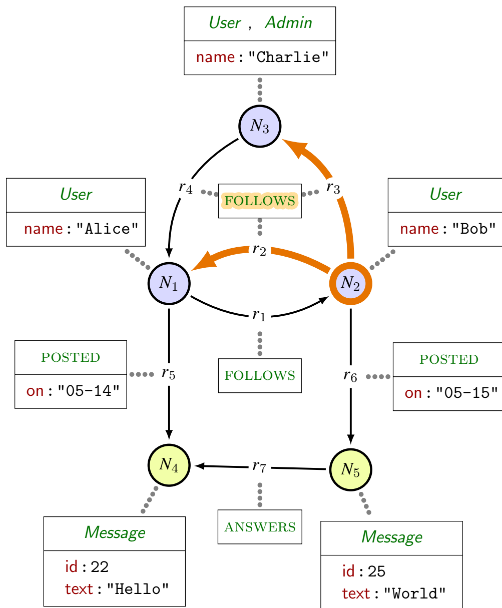
```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
                -[:FOLLOWS]->(u3)
```

Table after first MATCH:

u1	m1
N_1	N_4
N_2	N_5

Table after second MATCH:

u1	m1	u2	u3
~~N_1~~	~~N_4~~		
N_2	N_5	·	·



Query:

```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
              -[:FOLLOWS]->(u3)
```

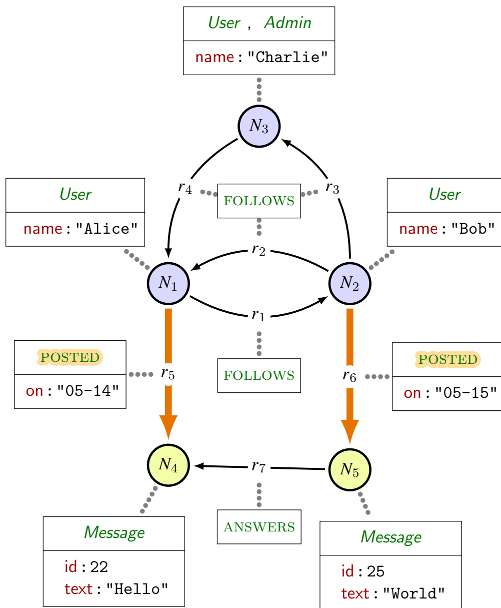
Table after first MATCH:

u1	m1
N_1	N_4
N_2	N_5

Table after second MATCH:

u1	m1	u2	u3
N_2	N_5	N_1	N_3
N_2	N_5	N_3	N_1

Column manipulation (WITH clause)



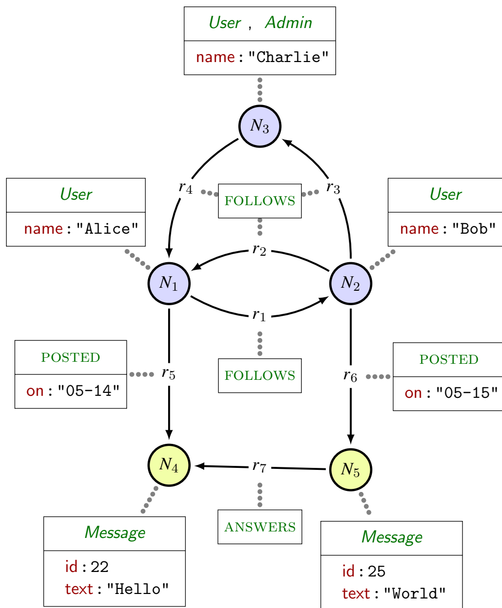
Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

After the MATCH clause

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Column manipulation (WITH clause)



Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

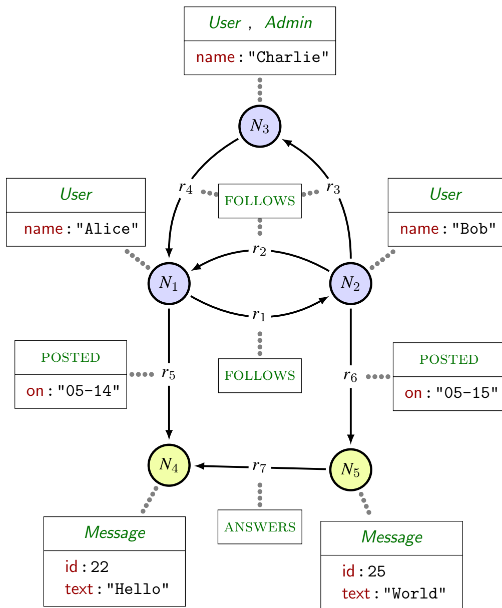
After the MATCH clause

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Execution of the WITH clause

u1	p1	t1
N_1	r_5	
N_2	r_6	

Column manipulation (WITH clause)



Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

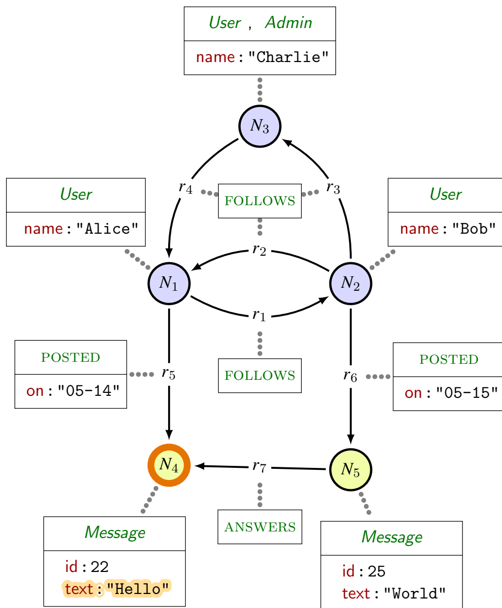
After the MATCH clause

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Execution of the WITH clause

u1	p1	t1
N_1	r_5	
N_2	r_6	

Column manipulation (WITH clause)



Query:

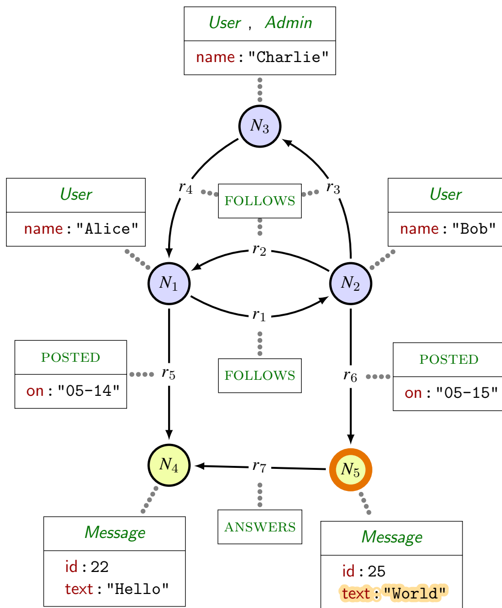
```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

After the MATCH clause

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Execution of the WITH clause

u1	p1	t1
N_1	r_5	"Hello"
N_2	r_6	



Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

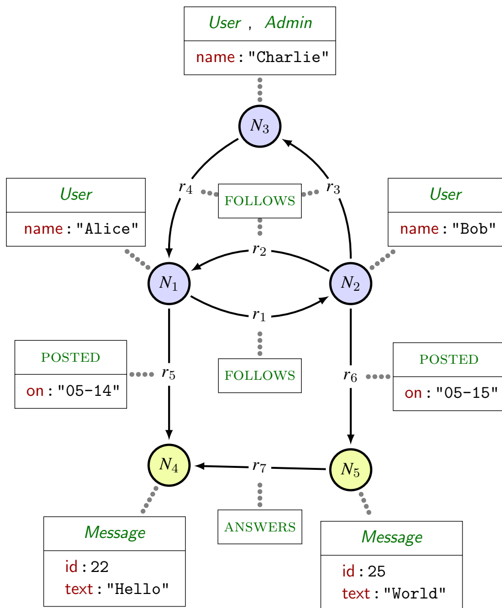
After the MATCH clause

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Execution of the WITH clause

u1	p1	t1
N_1	r_5	"Hello"
N_2	r_6	"World"

Column manipulation (WITH clause)



Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

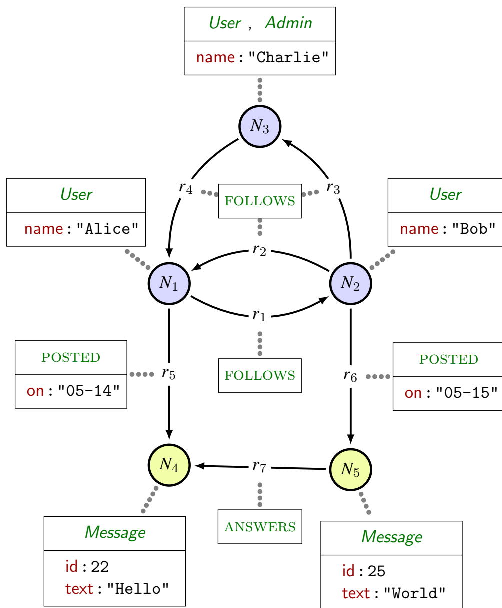
After the MATCH clause

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Final result

u1	p1	t1
N_1	r_5	"Hello"
N_2	r_6	"World"

Line filtering (WHERE clause)



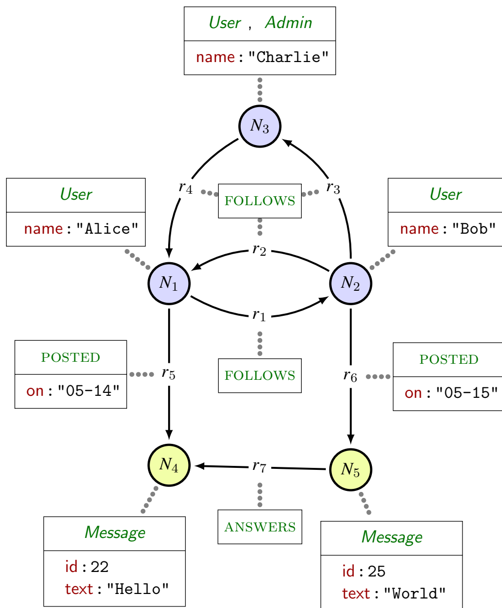
Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
WHERE t1 = "Hello"
```

After the WITH clause

u1	p1	t1
N_1	r_5	"Hello"
N_2	r_6	"World"

Line filtering (WHERE clause)



Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
WHERE t1 = "Hello"
```

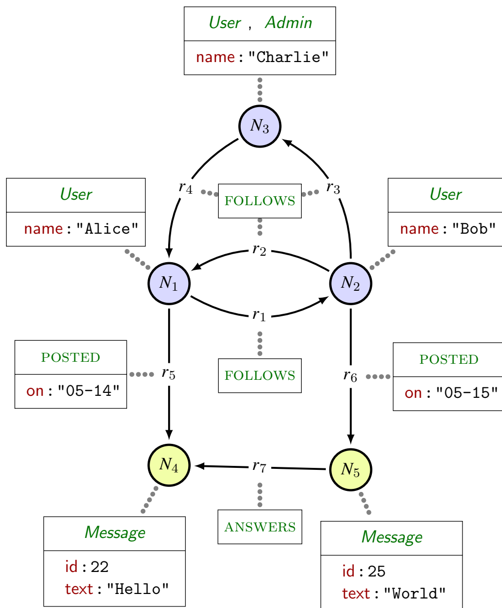
After the WITH clause

u1	p1	t1
N_1	r_5	"Hello"
N_2	r_6	"World"

Execution of the WHERE clause

u1	p1	t1
N_1	r_5	"Hello"
~~N_2~~	~~r_6~~	~~"World"~~

Line filtering (WHERE clause)



Query:

```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
WHERE t1 = "Hello"
```

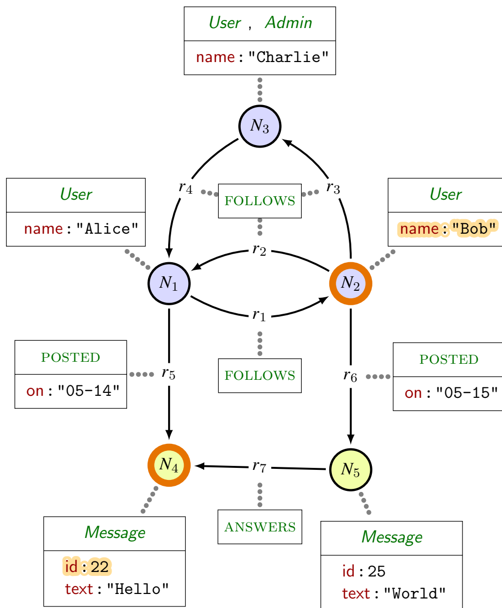
After the WITH clause

u1	p1	t1
N_1	r_5	"Hello"
N_2	r_6	"World"

Final result

u1	p1	t1
N_1	r_5	"Hello"

A last read-only example



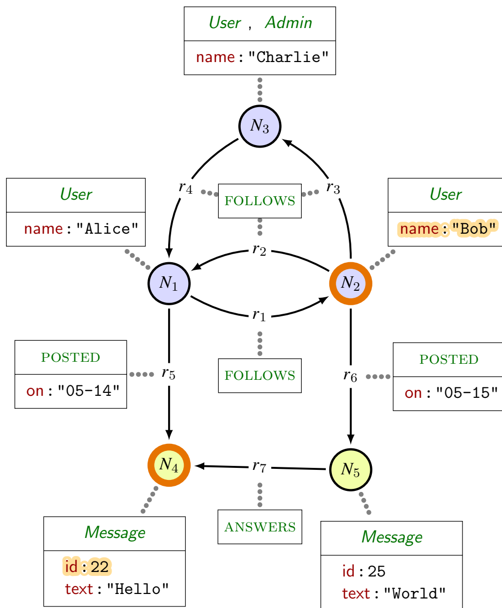
Query:

```
MATCH (a{name:"Bob"})
        -[*]->(b{id:22})
              <-[*]-(a)
```

Question

What does this computes ?

A last read-only example



Query:

```
MATCH (a{name:"Bob"})
        -[*]->(b{id:22})
              <-[*]-(a)
```

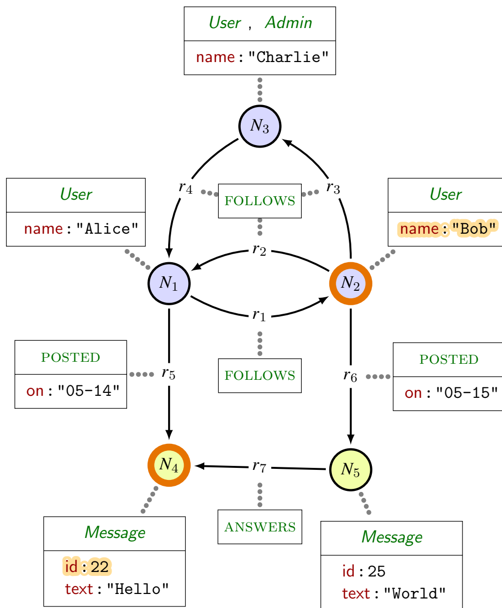
Question

What does this computes ?

Answer

The # of pairs of disjoint paths from N_2 to N_4.

A last read-only example



Query:

```
MATCH (a{name:"Bob"})
        -[*]->(b{id:22})
              <-[*]-(a)
```

Question

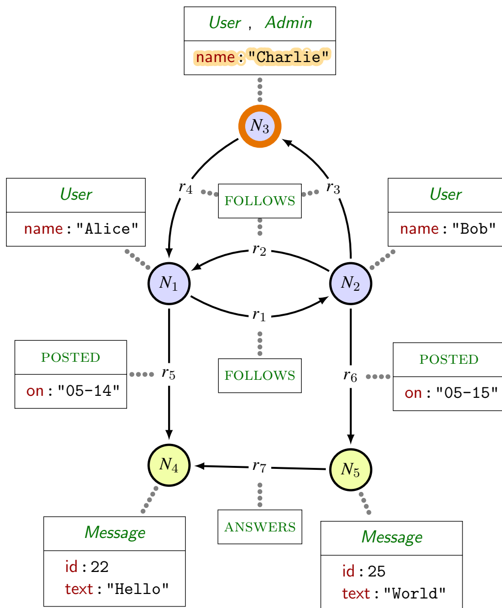
What does this computes ?

Answer

The # of pairs of disjoint paths from N_2 to N_4.

$\Rightarrow$ Evaluation of one constant `MATCH` is NP-HARD

Node and relation creation (CREATE)

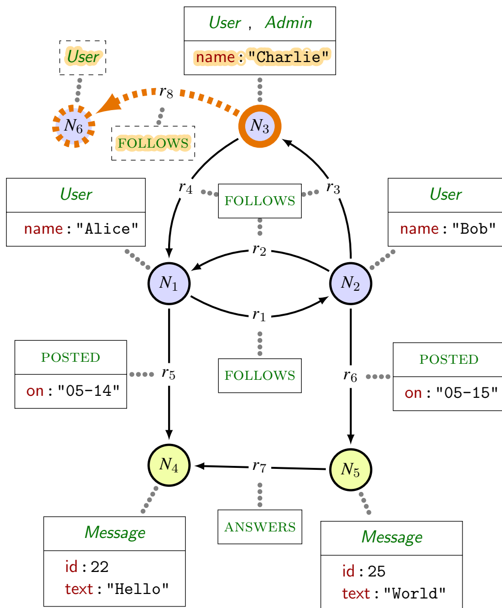


Query:

```
MATCH (a{name:"Charlie"})
CREATE (a)-[:FOLLOWS]->
                  (b:User)
```

Table after MATCH clause:

a
N_3



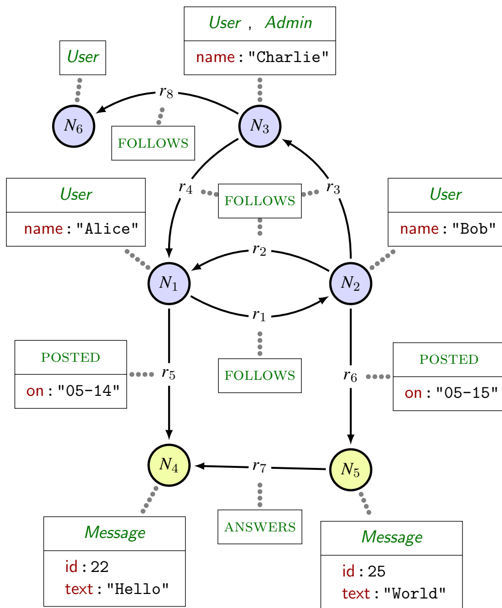
Query:

```
MATCH (a{name:"Charlie"})
CREATE (a)-[:FOLLOWS]->
                    (b:User)
```

Table after MATCH clause:

a
N_3

Node and relation creation (CREATE)



Query:

```
MATCH (a{name:"Charlie"})
CREATE (a)-[:FOLLOWS]->
                        (b:User)
```

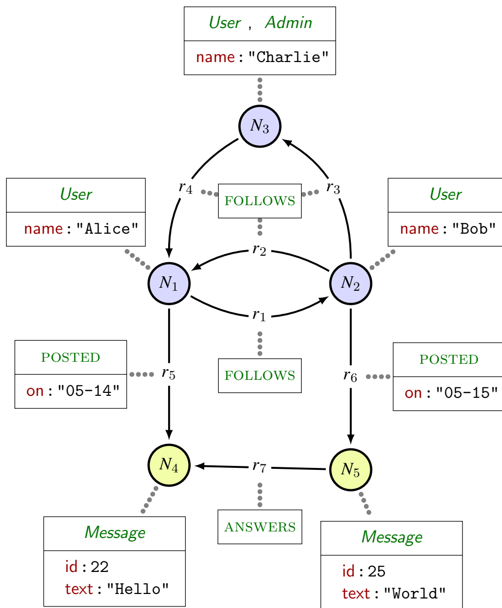
Table after MATCH clause:

a
N_3

Table after CREATE clause:

a	b
N_3	N_6

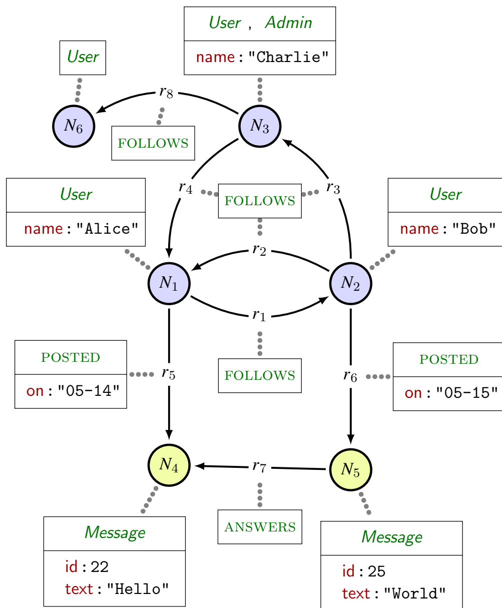
The example graph stored as CREATE clauses



Query:

```
CREATE
  (n1:User{name:"Alice"}),
  (n2:User{name:"Bob"}),
  (n3:User:Admin
            {name:"Charlie"}),
  (n4:Message {id:22,
               text:"Hello"}),
  (n5:Message {id:25,
               text:"World"})
CREATE
  (n1)-[:FOLLOWS]->(n2),
  (n1)-[:POSTED
          {on:"05-04"}]->(n4),
  (n2)-[:FOLLOWS]->(n1),
  (n2)-[:FOLLOWS]->(n3),
  (n2)-[:POSTED
          {on:"05-04"}]->(n5),
  (n3)-[:FOLLOWS]->(n1),
  (n5)-[:ANSWERS]->(n4),
```

Node/Relation modification (SET clause)



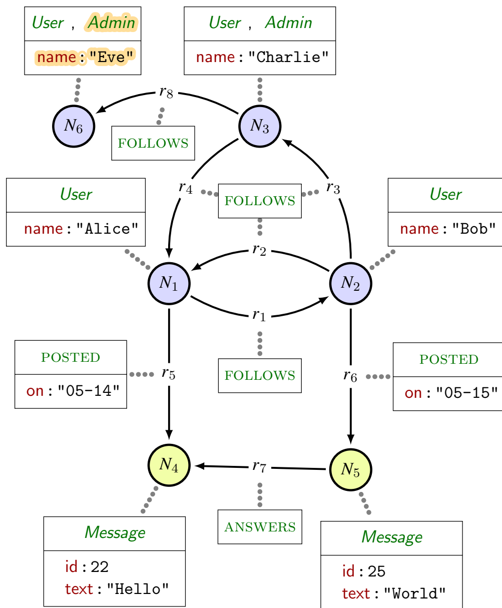
Query:

```
MATCH (a{name:"Charlie"})
CREATE (a)-[:FOLLOWS]->
                    (b:User)
SET b:Admin, b.name="Eve"
```

Table after CREATE clause:

a	b
N_3	N_6

Node/Relation modification (SET clause)



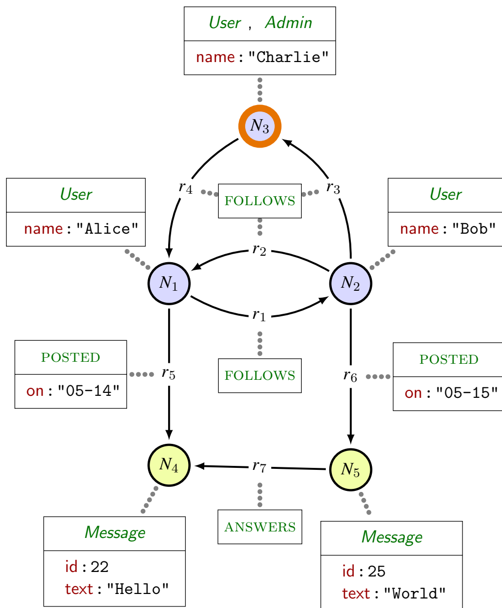
Query:

```
MATCH (a{name:"Charlie"})
CREATE (a)-[:FOLLOWS]->
                    (b:User)
SET b:Admin, b.name="Eve"
```

Table after CREATE clause:

a	b
N_3	N_6

The MERGE clause : MATCH else CREATE



Input table:

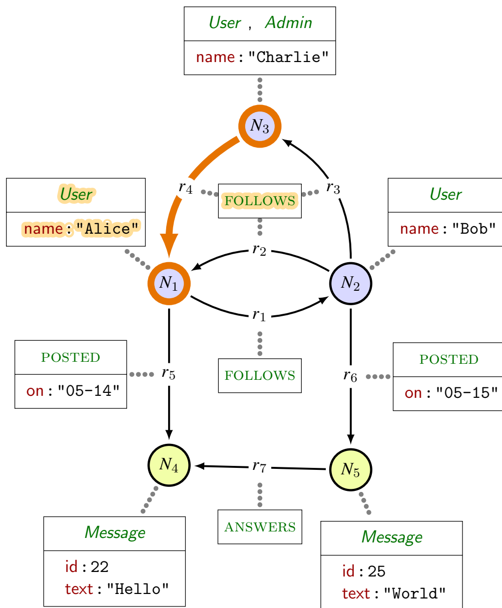
a	n
N_3	"Alice"
N_3	"Eve"

Query:

```
MERGE (a)-[:FOLLOWS]->
          (b:User {name:n})
```

Output table:

a	n	c
N_3	"Alice"	
N_3	"Eve"	



Input table:

a	n
N_3	"Alice"
N_3	"Eve"

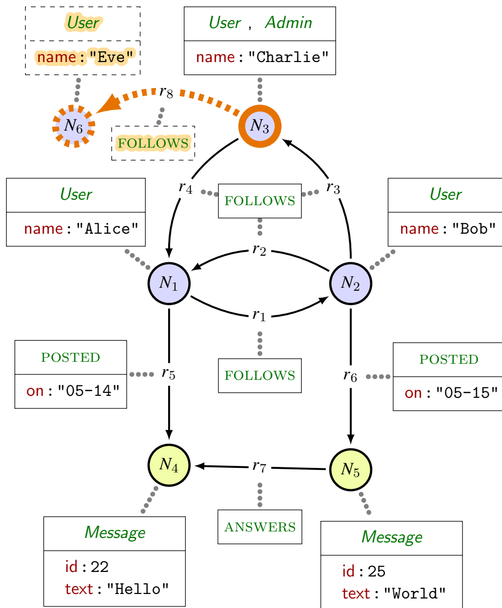
Query:

```
MERGE (a)-[:FOLLOWS]->
            (b:User {name:n})
```

Output table:

a	n	c
N_3	"Alice"	N_1
N_3	"Eve"	

The MERGE clause : MATCH else CREATE



Input table:

a	n
N_3	"Alice"
N_3	"Eve"

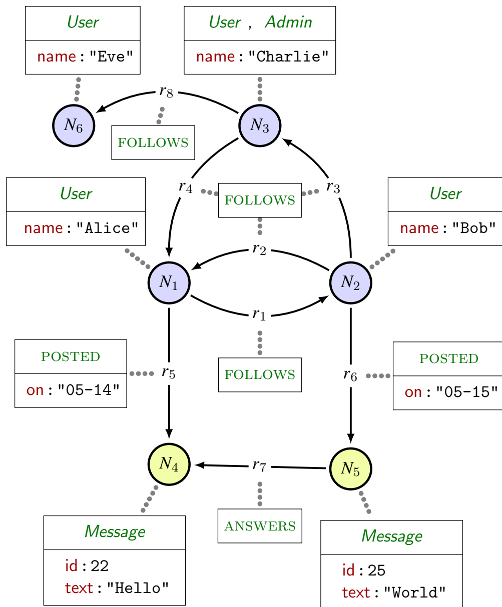
Query:

```
MERGE (a)-[:FOLLOWS]->
          (b:User {name:n})
```

Output table:

a	n	c
N_3	"Alice"	N_1
N_3	"Eve"	N_6

The MERGE clause : MATCH else CREATE



Input table:

a	n
N_3	"Alice"
N_3	"Eve"

Query:

```
MERGE (a)-[:FOLLOWS]->
          (b:User {name:n})
```

Output table:

a	n	c
N_3	"Alice"	N_1
N_3	"Eve"	N_6

Other constructs

- DELETE deletes node and relations.

Ex: `MATCH (a{name:"Eve"}) DELETE a`

- REMOVE removes labels or properties.

Ex: `MATCH (a{name:"Charlie"}) REMOVE a:Admin,a.name`

- WITH allows to perform aggregations.

Ex: `MATCH (a)-[:FOLLOWS]->(b) WITH a, count(b) as c`

- ORDER BY limits size of table.

Ex: `MATCH (a:User) ORDER BY a.name LIMIT 1`

Outline

1 Introduction
2 Property graphs
3 Regular Path Queries
4 Cypher by example
5 Principles of the semantics
6 Towards a standard language for querying property graphs

Record (table row)

A *record* is a partial function from variables to values.

Example: $(x \mapsto \texttt{"Bob"}\ ;\ y \mapsto \texttt{1})$

Record (table row)

A *record* is a partial function from variables to values.

Example: $(x \mapsto$ `"Bob"` ; $y \mapsto$ `1`$)$

Table

A *table* is a multi-set (or bag) of records with the same domain.

Example:

x	y
"Bob"	1
"Alice"	999
"Bob"	1

Modelling Tables

Record (table row)

A *record* is a partial function from variables to values.

Example: $(x \mapsto$ `"Bob"` ; $y \mapsto$ `1`$)$

Table

A *table* is a multi-set (or bag) of records with the same domain.

Example:

x	y
"Bob"	1
"Alice"	999
"Bob"	1

=

y	x
999	"Alice"
1	"Bob"
1	"Bob"

Semantics brackets – Read-Only

G: a graph

Semantics of expressions

$[\![\cdot]\!]_{u,G}$: expression $\longmapsto$ value (where u is a record)

Semantics of clauses

$[\![\cdot]\!]_{G}$: clause $\longmapsto$ (function: Tables $\to$ Tables)

Semantics of queries

$[\![\cdot]\!]_{G}$: query $\longmapsto$ (function: Tables $\to$ Tables)

output : (Graphs $\times$ Queries) $\longmapsto$ Tables

Principle of Read-Only Semantics

G: a graph
Q: a query

To compute the output of Q

- Q is a sequence of clauses $Q = C_1 \, C_2 \, \cdots \, C_n$

Principle of Read-Only Semantics

G: a graph
Q: a query

To compute the output of Q

- Q is a sequence of clauses $Q = C_1\, C_2\, \cdots\, C_n$
- Compute $\left[\!\left[C_1 \right]\!\right]_G, \ \left[\!\left[C_2 \right]\!\right]_G, \ \ldots, \ \left[\!\left[C_n \right]\!\right]_G$

Principle of Read-Only Semantics

G: a graph
Q: a query

To compute the output of Q

- Q is a sequence of clauses $Q = C_1\, C_2\, \cdots\, C_n$
- Compute $[\![C_1]\!]_G$, $[\![C_2]\!]_G$, $\ldots$, $[\![C_n]\!]_G$
- Let $[\![Q]\!]_G = [\![C_n]\!]_G \circ \cdots \circ [\![C_2]\!]_G \circ [\![C_1]\!]_G$

Principle of Read-Only Semantics

G: a graph
Q: a query

To compute the output of Q

- Q is a sequence of clauses $Q = C_1\, C_2\, \cdots\, C_n$
- Compute $[\![C_1]\!]_G, \; [\![C_2]\!]_G, \; \ldots, \; [\![C_n]\!]_G$
- Let $[\![Q]\!]_G = [\![C_n]\!]_G \circ \cdots \circ [\![C_2]\!]_G \circ [\![C_1]\!]_G$
- $\text{output}(G, Q) = [\![Q]\!]_G \big(T_{\text{unit}}\big)$

where T_{unit} is the 1-line 0-column table.

Excerpt of Read-Only semantics (2)

- $\left[\!\left[\texttt{WHERE}\ e\right]\!\right]_G(T) = \left\{ u \in T \;\middle|\; \left[\!\left[e\right]\!\right]_{G,u} = \texttt{true} \right\}$

- $\left[\!\left[\texttt{MATCH}\ \bar{\pi}\right]\!\right]_G(T) = \biguplus_{u \in T} \{u \cdot u' \mid u' \in \mathsf{match}(\bar{\pi}, G, u)\}$
- $\left[\!\left[\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e\right]\!\right]_G(T) = \left[\!\left[\texttt{WHERE}\ e\right]\!\right]\left(\left[\!\left[\texttt{MATCH}\ \bar{\pi}\right]\!\right]_G(T)\right)$
- $\left[\!\left[\texttt{OPTIONAL MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e\right]\!\right]_G(T)$
  $$= \biguplus_{u \in T} \begin{cases} \left[\!\left[\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e\right]\!\right]_G(\{u\}) & \text{if } \left[\!\left[\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e\right]\!\right]_G(\{u\}) \neq \emptyset \\ (u, (\mathsf{free}(u, \bar{\pi}) : \texttt{null})) & \text{otherwise} \end{cases}$$
- $\left[\!\left[\texttt{OPTIONAL MATCH}\ \bar{\pi}\right]\!\right]_G(T) = \left[\!\left[\texttt{OPTIONAL MATCH}\ \bar{\pi}\ \texttt{WHERE true}\right]\!\right]_G(T)$

Excerpt of Read-Only semantics (2)

- $\llbracket \texttt{WITH } * \rrbracket_G(T) = T$ if T has at least one column
- $\llbracket \texttt{WITH } *, e_1 \ [\texttt{AS } a_1], \ldots, e_m \ [\texttt{AS } a_m] \rrbracket_G(T) =$
  $$\llbracket \texttt{WITH } b_1 \texttt{ AS } b_1, \ldots, b_q \texttt{ AS } b_q, e_1 \ [\texttt{AS } a_1], \ldots, e_m \ [\texttt{AS } a_m] \rrbracket_G(T)$$
- $\llbracket \texttt{WITH } e_1 \ [\texttt{AS } a_1], \ldots, e_m \ [\texttt{AS } a_m] \rrbracket_G(T) =$
  $$\biguplus_{u \in T} \left\{ (a'_1 : \llbracket e_1 \rrbracket_{G,u}, \ldots, a'_m : \llbracket e_m \rrbracket_{G,u}) \right\}$$

- $\llbracket \texttt{UNWIND } e \texttt{ AS } a \rrbracket_G(T) = \biguplus_{u \in T} \biguplus_{v \in E_u} \{(u, a : v)\}$,
  $$\text{with} \quad E_u = \begin{cases} \biguplus_{0 \le i < m} \{v_i\} & \text{if } \llbracket e \rrbracket_{G,u} = \text{list}(v_0, \ldots, v_{m-1}) \\ \{\} & \text{if } \llbracket e \rrbracket_{G,u} = \text{list}() \\ \left\{ \llbracket e \rrbracket_{G,u} \right\} & \text{otherwise} \end{cases}$$

Semantics of expressions: (Nothing changes)

$[\![\cdot]\!]_{u,G}$: expression $\longmapsto$ value (where u is a record)

Semantics of clauses:

$[\![\cdot]\!]$: clause $\longmapsto$ (function: (Graphs $\times$ Tables) $\rightarrow$ (Graphs $\times$ Tables))

Semantics of queries:

$[\![\cdot]\!]$: query $\longmapsto$ (function: (Graphs $\times$ Tables) $\rightarrow$ (Graphs $\times$ Tables))

output : (Graphs $\times$ Queries) $\longmapsto$ (Graphs $\times$ Tables)

(Computed just like RO $\rightarrow$ composition of clause semantics)

Desirable properties of the semantics at Clause-level

Atomicity:

Each clause is executed as a single unit

Consistency:

Each clause should on valid Graph/Table pair

General scheme of the semantics if a clause is:

1. Ensure that the input Graph/Table is valid w.r.t. clause
2. Compute output Table and all changes to graph
3. Apply all changes to Graph
4. Ensure validity of output Graph/Table

If any fails, semantics is undefined.

Desirable properties of the semantics at Clause-level

Atomicity:

Each clause is executed as a single unit

Consistency:

Each clause should on valid Graph/Table pair

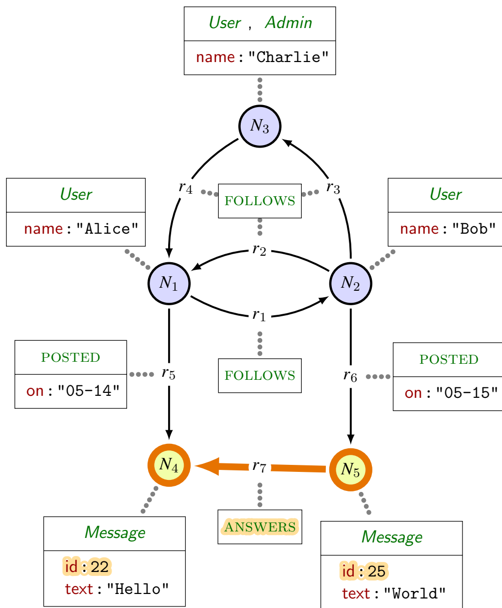
General scheme of the semantics if a clause is:

1. Ensure that the input Graph/Table is valid w.r.t. clause
2. Compute output Table and all changes to graph
3. Apply all changes to Graph
4. Ensure validity of output Graph/Table

If any fails, semantics is undefined.

In Neo4j, Atomicity and Consistency are verified at query level only.

In Neo4j, SET violate Atomicity



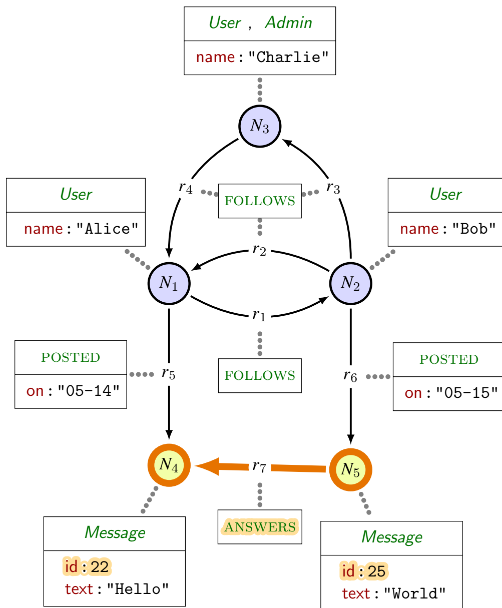
Query:

```
MATCH (m1)-[:ANSWERS]->(m2)
SET m1.id = m2.id,
    m2.id = m1.id
```

Table before SET clause:

m1	m2
N_5	N_4

In Neo4j, SET violate Atomicity



Query:

```
MATCH (m1)-[:ANSWERS]->(m2)
SET m1.id = m2.id,
    m2.id = m1.id
```

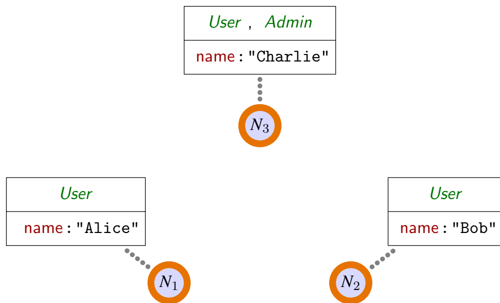
Table before SET clause:

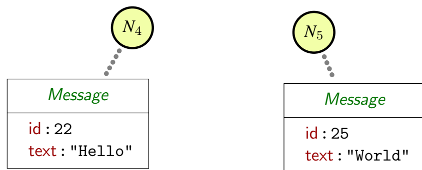
m1	m2
N_5	N_4

Neo4j sets both id to 25.

Semantics exchange id values

In Neo4j, MERGE is highly non-deterministic





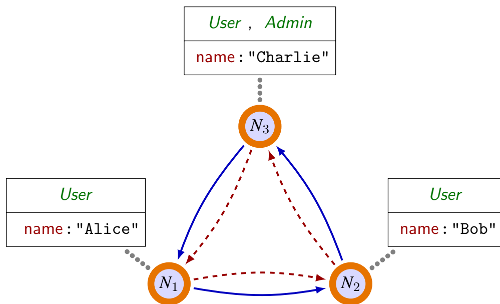
Input Table:

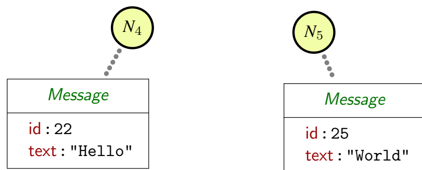
a	b	c
N_1	N_2	N_3
N_2	N_3	N_1
N_3	N_1	N_2

Query:

```
MERGE (a)-[:FOLLOWS]->(b)
         -[:FOLLOWS]->(c)
```

In Neo4j, MERGE is highly non-deterministic





Input Table:

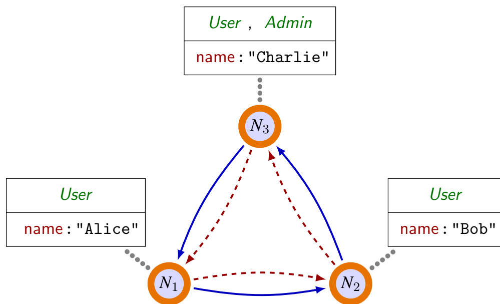
a	b	c
N_1	N_2	N_3
N_2	N_3	N_1
N_3	N_1	N_2

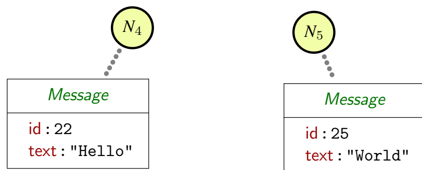
Query:

```
MERGE (a)-[:FOLLOWS]->(b)
         -[:FOLLOWS]->(c)
```

Neo4j creates 4 edges

In Neo4j, MERGE is highly non-deterministic





Input Table:

a	b	c
N_1	N_2	N_3
N_2	N_3	N_1
N_3	N_1	N_2

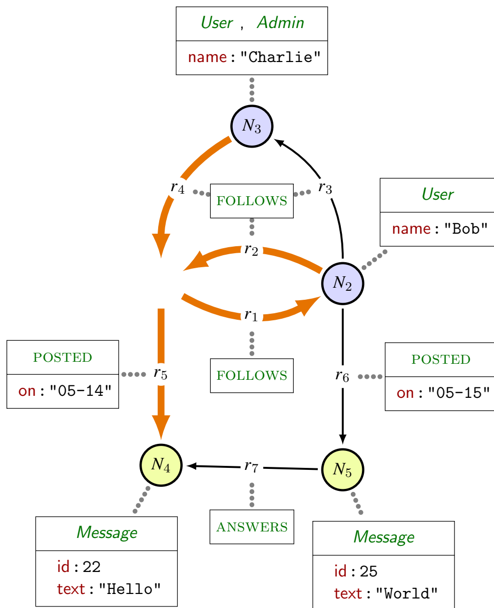
Query:

```
MERGE (a)-[:FOLLOWS]->(b)
         -[:FOLLOWS]->(c)
```

Neo4j creates 4 edges

Semantics propose different semantics creating either 3 or 6.

In Neo4j, DELETE violates clause-level consistency

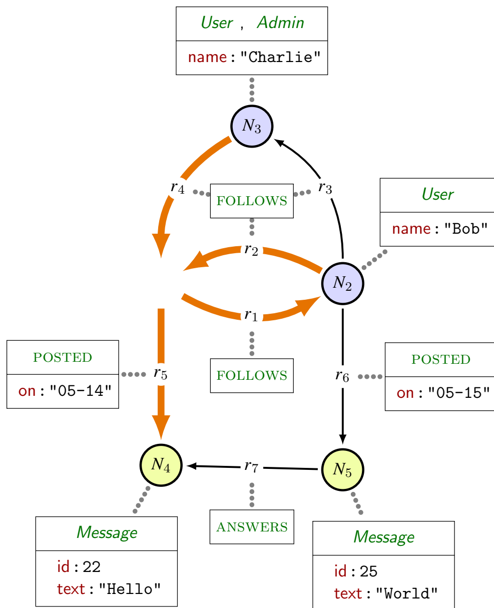


Query:

```
MATCH (a {name:"Alice"})
MATCH (a)-[r]-()
DELETE a
  [...]   // Arbitrary clauses
DELETE r
RETURN a.name AS n
```

- Execution of arbitrary code on invalid graph
- Access to deleted-entity

In Neo4j, DELETE violates clause-level consistency



Query:

```
MATCH (a {name:"Alice"})
MATCH (a)-[r]-()
DELETE a
  [...]    // Arbitrary clauses
DELETE r
RETURN a.name AS n
```

- Execution of arbitrary code on invalid graph
- Access to deleted-entity

Semantics

- is undefined if it should produce invalid graphs
- replaces deleted entities by `null` in the table

Outline

1 Introduction

2 Property graphs

3 Regular Path Queries

4 Cypher by example

5 Principles of the semantics

6 Towards a standard language for querying property graphs

Cypher vs RPQ

Cypher queries vs UCRPQs

- Cypher has bag + cypher-morphism semantics
  (Set+ standard morphism semantics may be emulated...)
- Cypher has relationship/path variables
- (Data model is different)

RPQs not expressible in Cypher

- $(ab)^*$: no concatenation under star in Cypher
- $(a^*)^*$: no nested stars in Cypher
- $(a + b^{-1})^*$: some unions are not allowed under star in Cypher

- $(ab + cd)^3$: nested alternations of concatenations and unions require multi-exponential blow-up of the query
  $\rightarrow (ab)^3 + (ab)^2cd + ab(cd)^2 + (cd)^3$

PGQL (Property Graph Query Language)

- Designed by Oracle Inc.
- Support full UCRPQ
- ASCII-art representation of patterns (similar to Cypher)
- Syntax close to SQL

Example: (from http://pgql-lang.org/)

```
SELECT p1.name
FROM facebook_graph          /* In the Facebook graph,..   */
MATCH (p1:Person)            /* ..find persons such that.. */
WHERE NOT EXISTS (           /* ..there does not exist..   */
  SELECT p2
    FROM twitter_graph       /* ..in the Twitter graph..   */
    MATCH (p2:Person)        /* ..a person..               */
    WHERE p1.name = p2.name  /* ..with the same name.      */
)
```

- Designed by LDBC
- Experimental
- Support full UCRPQ
- ASCII-art representation of patterns (similar to Cypher)
- Support Multiple graphs, graph views, and paths cost
- Multiple semantics

Example:

```
CONSTRUCT (n)-[:To{distance:=c}]->(m)
  MATCH (n) -/SHORTEST p<:KNOWS*> COST c/->(m)
    ON facebook_graph
  MATCH (n),(m) ON upem_graph
```

PGQL

- Read Only
- RPQs
- No Graph Construct/Project;
- Not Composable yet

Oracle PGX

GCORE

Advises

- Create - Read
- RPQs
- Graph Construct/Project;
- Composable

No implementations yet

Cypher

- Create - Read - Update - Delete
- No RPQs
- Graph Construct/Project:
- Composable

- Neo4j DB
- AgensGraph
- Redis Graph
- SAP HANA Graph
- Cypher for Spark/Gremlin
- Memgraph
- inGraph
- Cypher.PL

New, Fused

GQL

- Create - Read - Update - Delete
- RPQs
- Graph Construct/Project;
- Composable

cf. GQL manifesto:
`http://gql.today`

cf. GQL manifesto:
`http://gql.today`

To be continued...

Detailed outline

1 Introduction
2 Property graphs
3 Regular Path Queries
- RPQ
- CRPQ
- UCRPQ

4 Cypher by example
- Query evaluation
- MATCH
- WITH
- WHERE
- CREATE
- SET
- MERGE
- Others

5 Principles of the semantics
- Modelling tables
- Read-Only
- Read-Write

6 Towards a standard language for querying property graphs
- Cypher vs RPQ
- PGQL
- G-Core

7 The MATCH clause
- Path patterns
- Rigid path patterns
- General case

8 Remainder of the core fragment
- Caveat on the MATCH clause
- OPTIONAL MATCH
- WHERE
- WITH
- UNWIND
- RETURN
- UNION of queries

9 Ongoing additions
- Aggregation
- Line order

Node patterns

⟨node_pattern⟩ ::= (⟨name⟩? ⟨labels⟩? ⟨properties⟩?)

Examples

`()`	Any node
`(a)`	Any node; verify/map id to column a
`(:User)`	Nodes bearing label *User*
`({name:"Alice"})`	Nodes with the property name set to `"Elin"`
`(a:User{name:"Alice"})`	Nodes of type User and with a property name set to `"Elin"`; verify/map id to column a.

Relationship patterns

⟨pattern⟩ ::= **[** ⟨name⟩? ⟨types⟩? ⟨iter⟩? ⟨properties⟩? **]**

Example

`[]`	Any relationship
`[r]`	Any relationship; verify/map relationship id to column r
`[r*]`	Any path; verify/map the list of relationship id to column r
`[*..3]`	Any path of length 1–3
`[:FOLLOWS*3..]`	Any FOLLOWS path of length at least 3
`[*{isNice:true}]`	Path such that every relationship has the property isNice set to true

Path patterns

⟨pattern⟩ ::=
⟨node_pattern⟩ [$<^?$ – ⟨rel_pattern⟩ – $>^?$ ⟨node_pattern⟩ $]^*$

Examples

```
()
()-[*2..3]-(b)
(a)<-[]->()-[{id:220}]->()
(a)<-[*]-(b)-[*]->(a)
({name:a.name})-[*]->(a)
({name:"Alice"})-[:POSTED]->()<-[:ANSWERS*]-()
```

Rigid path-patterns

Definition

A path pattern is *Rigid* if its length is fixed
(*i.e.* all ⟨iter⟩ are absent or derive to $*i..i$ for some $i \in \mathbb{N}$)

Examples

`(a)<-[]->()-[{id:220}]->()`	Rigid
`()-[]->()-[*42..42]->(:User)`	Rigid
`()-[*2..3]->(b)` and `(a)<-[]-(b)-[*]-(a)`	Flexible

Rigid path-patterns

Definition

A path pattern is *Rigid* if its length is fixed
(*i.e.* all ⟨iter⟩ are absent or derive to $*i..i$ for some $i \in \mathbb{N}$)

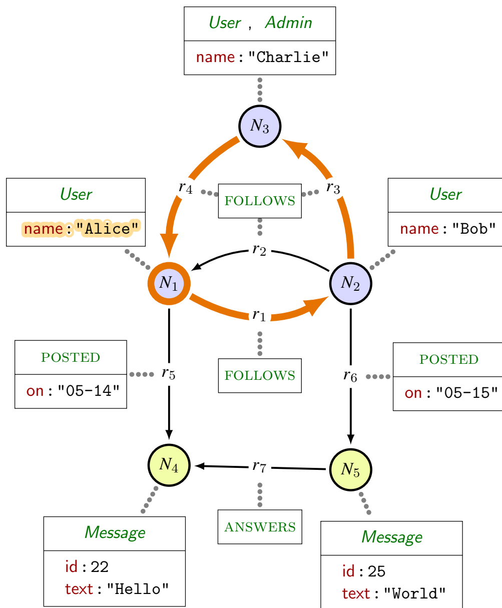
Examples

`(a)<-[]->()-[{id:220}]->()`	Rigid
`()-[]->()-[*42..42]->(:User)`	Rigid
`()-[*2..3]->(b)` and `(a)<-[]-(b)-[*]-(a)`	Flexible

Property

A path has only one way to satisfy a rigid path pattern.

Rigid path vs flexible path (1)



Query:

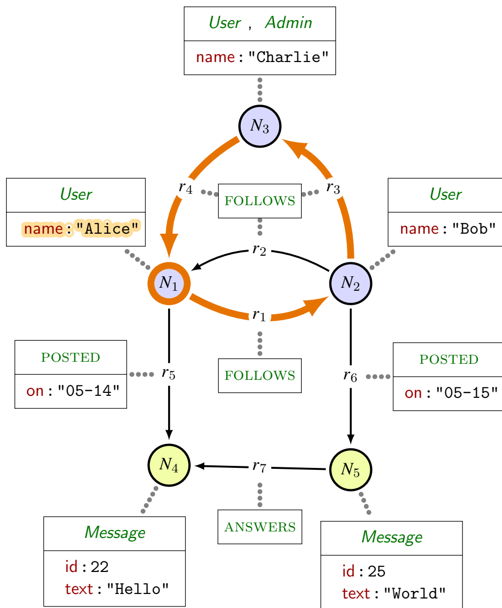
```
MATCH (x {name:Alice})
      -[:FOLLOWS*1..1]->(mid)
      -[:FOLLOWS*2..2]->(x)
```

The path

$$N_1 \xrightarrow{r_1} N_2 \xrightarrow{r_3} N_3 \xrightarrow{r_4} N_1$$

satifies the pattern.

Rigid path vs flexible path (1)



Query:

```
MATCH (x {name:Alice})
      -[:FOLLOWS*1..1]->(mid)
      -[:FOLLOWS*2..2]->(x)
```

The path

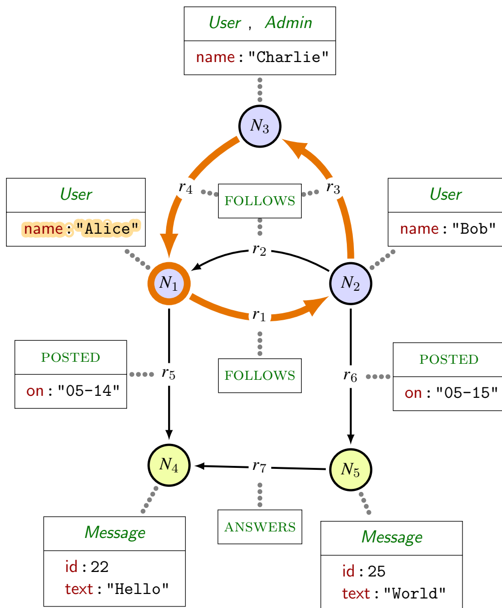
$$N_1 \xrightarrow{r_1} N_2 \xrightarrow{r_3} N_3 \xrightarrow{r_4} N_1$$

satifies the pattern.

Variables are uniquely set

x	mid
N_1	N_2

Rigid path vs flexible path (2)



Query:

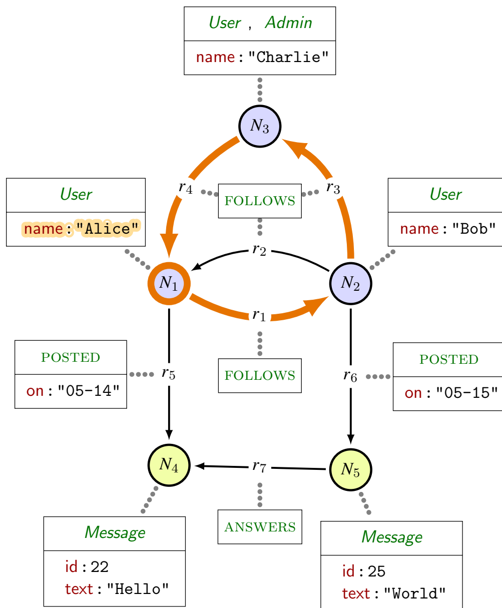
```
MATCH (x {name:"Alice"})
      -[:FOLLOWS*1..2]->(mid)
      -[:FOLLOWS*1..2]->(x)
```

The path

$$N_1 \xrightarrow{r_1} N_2 \xrightarrow{r_3} N_3 \xrightarrow{r_4} N_1$$

satifies this second pattern.

Rigid path vs flexible path (2)



Query:

```
MATCH (x {name:"Alice"})
      -[:FOLLOWS*1..2]->(mid)
      -[:FOLLOWS*1..2]->(x)
```

The path

$$N_1 \xrightarrow{r_1} N_2 \xrightarrow{r_3} N_3 \xrightarrow{r_4} N_1$$

satifies this second pattern.

Two possible assignments

x	mid
N_1	N_2
N_1	N_3

Path satisfying a rigid path-pattern

u: a record

p: a path

π: a rigid path-pattern

n: length of π

Definition

$G, p, u \models \pi$ if

- p is of length n
- $\forall i \leq p$, the i-th node of p satisfies the i-th node pattern of π (variable, labels, properties)
- $\forall i < p$, the i-th relationship of p satisfies the i-th relationship pattern of π (variable, types, properties)

rigid(π) = set of all rigid patterns subsumed by π

Examples

`()-[*]->()` $\mapsto$ { `()-[*1..1]->()` , `()-[*2..2]->()` , $\cdots$ }

Rigid path-pattern subsumed by a flexible path-pattern

rigid(π) = set of all rigid patterns subsumed by π

Examples

`()-[*]->()` $\mapsto$ { `()-[*1..1]->()` , `()-[*2..2]->()` , $\cdots$ }

`()-[*2..3]-(b)` $\mapsto$ { `()-[*2..2]-(b)` , `()-[*3..3]-(b)` }

Rigid path-pattern subsumed by a flexible path-pattern

rigid(π) = set of all rigid patterns subsumed by π

Examples

`()-[*]->()` $\mapsto$ {`()-[*1..1]->()` , `()-[*2..2]->()` , ⋯}

`()-[*2..3]-(b)` $\mapsto$ {`()-[*2..2]-(b)` , `()-[*3..3]-(b)`}

`({name:"Alice"})-[*]->()-[:CITES*]->()` $\mapsto$
{`({name:"Alice"})-[*1..1]->()-[:CITES*1..1]->()` ,
`({name:"Alice"})-[*1..1]->()-[:CITES*2..2]->()` ,
`({name:"Alice"})-[*2..2]->()-[:CITES*1..1]->()` , ⋯}

rigid(π) = set of all rigid patterns subsumed by π

Examples

`()-[*]->()` ↦ { `()-[*1..1]->()` , `()-[*2..2]->()` , ⋯ }

`()-[*2..3]-(b)` ↦ { `()-[*2..2]-(b)` , `()-[*3..3]-(b)` }

```
({name:"Alice"})-[*]->()-[:CITES*]->() ↦
{({name:"Alice"})-[*1..1]->()-[:CITES*1..1]->() ,
({name:"Alice"})-[*1..1]->()-[:CITES*2..2]->() ,
({name:"Alice"})-[*2..2]->()-[:CITES*1..1]->() , ⋯}
```

WLOG: rigid(π) contains no pattern longer than the graph size

The MATCH clause

Semantics of MATCH

$$[\![\texttt{MATCH}\ \pi]\!]_G : \quad T \quad \longmapsto \quad \biguplus_{u \in T} \mathsf{expand}_{G,\pi}(u)$$

The MATCH clause

Semantics of MATCH

$$[\![\texttt{MATCH}\ \pi]\!]_G : \quad T \longmapsto \biguplus_{u \in T} \text{expand}_{G,\pi}(u)$$

Table-row expansion

π: a path-pattern

G: a graph

$$\text{expand}_{G,\pi} : \quad \text{Records} \longrightarrow \text{Tables}$$

$$u \longmapsto \biguplus_{\substack{\text{paths } p \text{ in } G \\ \text{patterns } \pi' \text{ in rigid}(\pi)}} \left\{ \text{records } v \;\middle|\; \begin{array}{l} \blacksquare\ \text{dom}(v) = \text{dom}(u) \cup \text{var}(\pi') \\ \blacksquare\ v \text{ extends } u \\ \blacksquare\ G, p, v \models \pi' \end{array} \right\}$$

The core fragment

Queries are essentially sequences of clauses

$$\langle\text{query}\rangle ::= \langle\text{query}\rangle \;\; \texttt{UNION} \; [\texttt{ALL}]^{?} \;\; \langle\text{query}\rangle \;\; \mid \;\; [\langle\text{clause}\rangle]^{*} \;\; \langle\text{return}\rangle$$

A clause is a main statement followed by subclauses

$$\langle\text{clause}\rangle ::= \langle\text{match}\rangle \;\; [\langle\text{where}\rangle]^{?} \;\; \mid \;\; \langle\text{with}\rangle \;\; [\langle\text{where}\rangle]^{?} \;\; \mid \;\; \langle\text{unwind}\rangle$$

Caveat on the MATCH clause (1)

Syntax

⟨match⟩ ::= [OPTIONAL]? MATCH ⟨pattern_tuple⟩

⟨pattern_tuple⟩ ::= ⟨pattern⟩ [, ⟨pattern⟩]*

- MATCH may search for a tuple of path-pattern
  - Cypher-morphism applies to the whole tuple
- MATCH may be "optional" :
  - if MATCH succeeds → no changes
  - if MATCH fails → free variables are set to null instead of deleting the line.

Semantics of MATCH

$\bar{\pi} = (\pi_1, \pi_2, \ldots, \pi_n)$: a path-pattern tuple
G: a graph

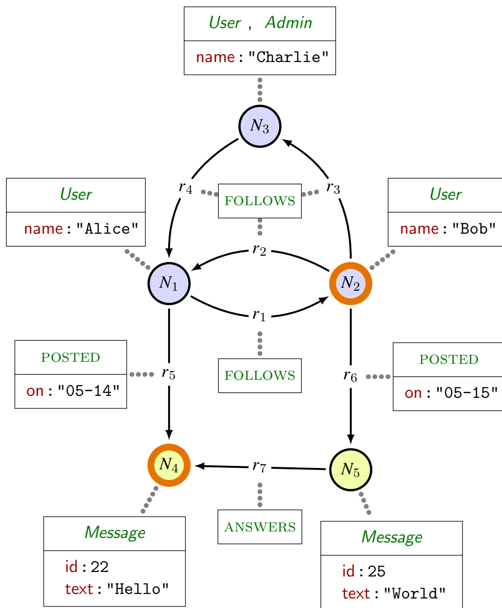
$$\text{rigid}(\bar{\pi}) = \text{rigid}(\pi_1) \times \text{rigid}(\pi_2) \times \cdots \times \text{rigid}(\pi_n)$$

$$\text{expand}_{G,\bar{\pi}} : \quad \text{Records} \longrightarrow \text{Tables}$$

$$u \longmapsto \biguplus_{\substack{\text{paths } \bar{p} \text{ in } G \\ \text{patterns } \bar{\pi}' \text{ in rigid}(\pi)}} \left\{ \text{records } v \;\middle|\; \begin{array}{l} \blacksquare\ \text{dom}(v) = \\ \quad \text{dom}(u) \cup \text{var}(\bar{\pi}') \\ \blacksquare\ v \text{ extends } u \\ \blacksquare\ G, \bar{p}, v \models \bar{\pi}' \end{array} \right\}$$

$$[\![\texttt{MATCH } \bar{\pi}]\!]_G : \quad T \longmapsto \biguplus_{u \in T} expand_{G,\pi}(u)$$

OPTIONAL MATCH – Example



x
N_2
N_4

↓

Query:

```
OPTIONAL MATCH
        (x)-[y:FOLLOWS]->()
```

OPTIONAL MATCH – Example

User, *Admin*
name : "Charlie"

N_3

User
name : "Alice"

r_4 FOLLOWS r_3

User
name : "Bob"

r_2

N_1 N_2

r_1

POSTED
on : "05-14"

r_5 FOLLOWS r_6

POSTED
on : "05-15"

N_4 r_7 N_5

Message
id : 22
text : "Hello"

ANSWERS

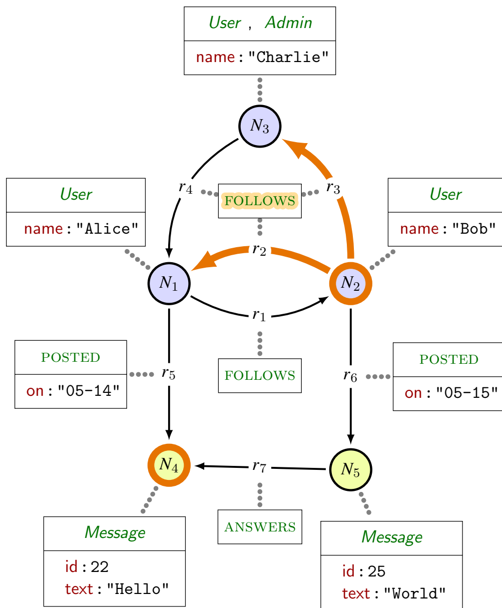
Message
id : 25
text : "World"

x
N_2
N_4

Query:

```
OPTIONAL MATCH
        (x)-[y:FOLLOWS]->()
```

OPTIONAL MATCH – Example



x
N_2
N_4

↓

Query:

```
OPTIONAL MATCH
        (x)-[y:FOLLOWS]->()
```

↓

x	y
N_2	r_2
N_2	r_3
N_4	null

OPTIONAL MATCH – Semantics

u: a record

$\bar{\pi}$: a path-pattern tuple

$\mathsf{compl}(u, \bar{\pi})$ is the record:

- $\mathsf{dom}\Big(\mathsf{cmpl}(u, \bar{\pi})\Big) = \mathsf{dom}(u) \cup \mathsf{var}(\bar{\pi})$
- $a \longmapsto \begin{cases} u(a) & \text{if } a \in \mathsf{dom}(u) \\ \texttt{null} & \text{otherwise} \end{cases}$

$$\Big[\!\!\Big[\texttt{OPTIONAL MATCH } \bar{\pi}\Big]\!\!\Big]_G(T) = \biguplus_{u \in T} \begin{cases} T_u & \text{if } T_u \neq \emptyset \\ \{\mathsf{cmpl}(u)\} & \text{otherwise} \end{cases}$$

where $T_u = \Big[\!\!\Big[\texttt{MATCH } \bar{\pi}\Big]\!\!\Big]_G(\{u\})$

The WHERE subclause (1)

Syntax

$$\langle \text{where} \rangle ::= \quad \texttt{WHERE} \quad \langle \text{expr} \rangle$$

Semantics

$$\left[\!\left[\texttt{WHERE}\ e \right]\!\right] : \quad T \quad \longmapsto \quad \biguplus_{u \in T} \begin{cases} \{u\} & \text{if } \left[\!\left[e \right]\!\right]_{G,u} = \texttt{true} \\ \emptyset & \text{otherwise} \end{cases}$$

Combined semantics

- $[\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_G(T) = [\![\texttt{WHERE}\ e]\!]\big([\![\texttt{MATCH}\ \bar{\pi}]\!]_G(T)\big)$

- $[\![\texttt{WITH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_G(T) = [\![\texttt{WHERE}\ e]\!]\big([\![\texttt{WITH}\ \bar{\pi}]\!]_G(T)\big)$

- $[\![\texttt{OPTIONAL MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_G(T)$

$$= \biguplus_{u \in T} \begin{cases} T_u & \text{if } T_u \neq \emptyset \\ \{\text{cmpl}(u)\} & \text{otherwise} \end{cases}$$

where $T_u = [\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_G(\{u\})$
(and $\text{cmpl}(u, \bar{\pi})$ is the completion of u w.r.t. variables of $\bar{\pi}$)

$$\langle\text{with}\rangle ::=$$

| WITH ⟨expr⟩ [AS ⟨name⟩]? , · · · , ⟨expr⟩ [AS ⟨name⟩]?

| WITH * , ⟨expr⟩ [AS ⟨name⟩]? , · · · , ⟨expr⟩ [AS ⟨name⟩]?

- Each expression/name pair will be one column in the output table.
- If the name is missing, a default name is provided (implementation dependent).
- * means "every column previously in the table".

The WITH clause (2)

Example

a	b	c
0	1	2
1	1	1
2	1	0

```
WITH  *,  a+b,  a<c AS order
```

a	b	c	‘a+b‘	order
0	1	2	1	true
1	1	1	2	false
2	1	0	3	false

Semantics

- $\left[\!\left[\texttt{WITH}\ * \right]\!\right]_G (T) = T$

- $\left[\!\left[\texttt{WITH}\ e_1\ \texttt{AS}\ a_1, \ldots, e_m\ \texttt{AS}\ a_m \right]\!\right]_G (T)$
$$= \biguplus_{u \in T} \left\{ (a_1 : \left[\!\left[e_1 \right]\!\right]_{G,u}, \ldots, a_m : \left[\!\left[e_m \right]\!\right]_{G,u}) \right\}$$

- $\left[\!\left[\texttt{WITH}\ e_1\ [\texttt{AS}\ a_1]?, \ldots, e_m\ [\texttt{AS}\ a_m]? \right]\!\right]_G (T)$
$$= \left[\!\left[\texttt{WITH}\ e_1\ \texttt{AS}\ a'_1, \ldots, e_m\ \texttt{AS}\ a'_m \right]\!\right]_G (T)$$
where a'_i equals a_i if it is given, or arbitrary otherwise.

- $\left[\!\left[\texttt{WITH}\ *, \ldots \right]\!\right]_G (T) = \left[\!\left[\texttt{WITH}\ b_1\ \texttt{AS}\ b_1, \ldots, b_q\ \texttt{AS}\ b_q, \ldots \right]\!\right]_G (T)$
where $b_1, \ldots, b_q$ are the column names of T

The UNWIND clause (1)

Syntax

$\langle$unwind$\rangle$ ::= `UNWIND` $\langle$expr$\rangle$ `AS` $\langle$name$\rangle$

- Given expression is supposed to be evaluated to lists
- Each line of the input table is expanded as multiple lines in the output, one for each element of the list.

The UNWIND clause (2)

Example

n	*list*
0	["Hello", "World"]
1	["singleton"]
2	"not_a_list"

```
UNWIND list AS x
```

n	*list*	x
0	["Hello","World"]	"Hello"
0	["Hello","World"]	"World"
1	["singleton"]	"singleton"
2	"not_a_list"	"not_a_list"

$$\left[\!\left[\texttt{UNWIND}\ e\ \texttt{AS}\ a \right]\!\right]_G (T) \quad = \quad \biguplus_{u \in T} \biguplus_{v \in E_u} \{(u, a : v)\}\ ,$$

$$\text{where} \quad E_u = \begin{cases} \biguplus_{0 \le i < m} \{v_i\} & \text{if } \left[\!\left[e \right]\!\right]_{G,u} = \text{list}(v_1, \ldots, v_m) \\ \{\} & \text{if } \left[\!\left[e \right]\!\right]_{G,u} = \text{list}() \\ \left\{ \left[\!\left[e \right]\!\right]_{G,u} \right\} & \text{otherwise} \end{cases}$$

The `RETURN` clause

`RETURN` and `WITH` clauses have the exact same semantics.

The RETURN clause

RETURN and WITH clauses have the exact same semantics.

RETURN clause is the “end-marker” of a clause sequence:

- it is mandatory;
- it cannot appear earlier.

UNION and UNION ALL

Syntax

$$\langle\text{query}\rangle ::= \langle\text{query}\rangle\ \texttt{UNION}\ [\texttt{ALL}]?\ \langle\text{query}\rangle$$

- The left and right queries are assumed to have the same column-tables.
- `UNION` is the set-union of tables.
- `UNION ALL` is the bag-union of tables.

UNION and UNION ALL

Syntax

$$\langle\text{query}\rangle ::= \langle\text{query}\rangle\ \texttt{UNION}\ [\texttt{ALL}]?\ \langle\text{query}\rangle$$

- The left and right queries are assumed to have the same column-tables.
- `UNION` is the set-union of tables.
- `UNION ALL` is the bag-union of tables.

Semantics

- $[\![Q_1 \texttt{ UNION ALL } Q_2]\!]_G(T) = [\![Q_1]\!]_G(T) \cup [\![Q_2]\!]_G(T)$
- $[\![Q_1 \texttt{ UNION } Q_2]\!]_G(T) = \text{distinct}\left([\![Q_1]\!]_G(T) \cup [\![Q_2]\!]_G(T) \right)$

Aggregation (1)

Principle

- Group lines according to some criteria
- Some column of the output table contains aggregated values.

Aggregation functions

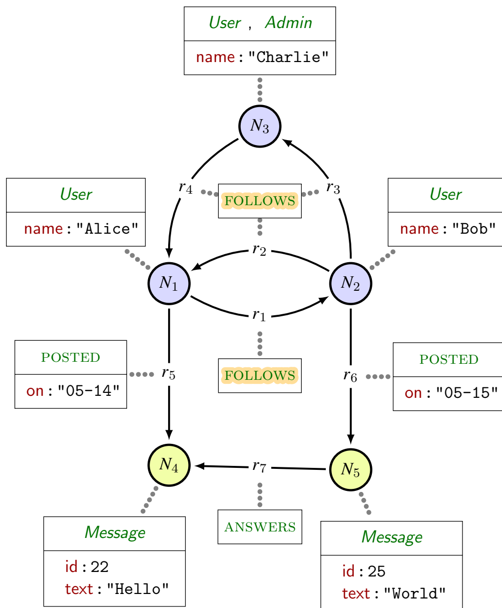
$\mathcal{G}$: a set of functions that map value bags to single values

E.g., `count`, `max` $\in \mathcal{G}$

- In `WITH` clauses, we allow *aggregate expression*:

  $(\mathit{WITH}\ \ \langle\text{aggexpr}\rangle\ [\texttt{AS}\ \ \langle\text{name}\rangle]?)$
- Aggregate expressions are of the form: $g(\langle\text{expr}\rangle)$ where $g \in \mathcal{G}$

Aggregation (2) — Example



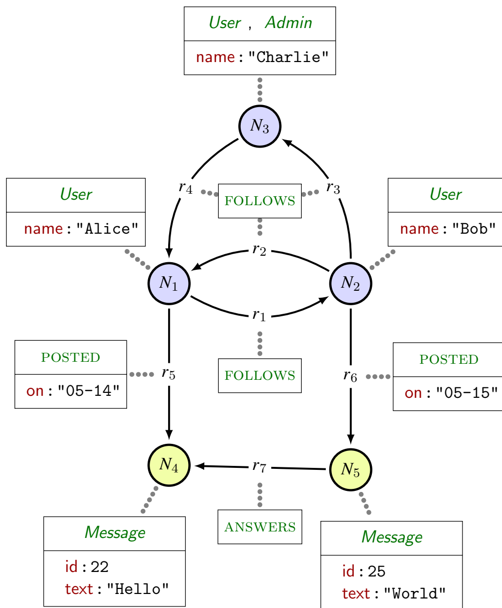
Query:

```
MATCH (a)-[r:FOLLOWS]->()
WITH a.name AS b, count(r) AS c
```

Aggregate

- over expression `a.name`
- and count them

b	c
"Alice"	1
"Bob"	1
"Charlie"	2



Query:

```
MATCH (a)-[r:FOLLOWS]->()
WITH a.name AS b, count(r) AS c
WITH max(c) AS d
```

d
2

Example: WITH e_1 AS a_1, e_2 AS a_2, $max(e_3)$ AS a_3

e_1, e_2 and e_3 are ⟨expr⟩
$max(e_3)$ is an ⟨aggexpr⟩

We group the lines according to the expressions e_1, e_2:

- We partition the input table as subtables $T_1, \ldots, T_k$
- For each $u, v \in T_i$, $\llbracket e_j \rrbracket_{G,u} = \llbracket e_j \rrbracket_{G,v}$ (for $j \in \{1, 2\}$)

Example: `WITH` e_1 `AS` a_1, e_2 `AS` a_2, $max(e_3)$ `AS` a_3

e_1, e_2 and e_3 are ⟨expr⟩
$max(e_3)$ is an ⟨aggexpr⟩

We group the lines according to the expressions e_1, e_2:

- We partition the input table as subtables $T_1, \ldots, T_k$
- For each $u, v \in T_i$, $[\![e_j]\!]_{G,u} = [\![e_j]\!]_{G,v}$ (for $j \in \{1, 2\}$)

Aggregation (4)

Semantics on an example

Example: `WITH` e_1 `AS` a_1, e_2 `AS` a_2, $max(e_3)$ `AS` a_3

e_1, e_2 and e_3 are ⟨expr⟩
$max(e_3)$ is an ⟨aggexpr⟩

We group the lines according to the expressions e_1, e_2:

- We partition the input table as subtables $T_1, \ldots, T_k$
- For each $u, v \in T_i$, $[\![e_j]\!]_{G,u} = [\![e_j]\!]_{G,v}$ (for $j \in \{1, 2\}$)

The output table has 3 columns and k lines; the i-th line is:

Example: `WITH` e_1 `AS` a_1, e_2 `AS` a_2, $max(e_3)$ `AS` a_3

e_1, e_2 and e_3 are ⟨expr⟩
$max(e_3)$ is an ⟨aggexpr⟩

We group the lines according to the expressions e_1, e_2:

- We partition the input table as subtables $T_1, \ldots, T_k$
- For each $u, v \in T_i$, $\llbracket e_j \rrbracket_{G,u} = \llbracket e_j \rrbracket_{G,v}$ (for $j \in \{1, 2\}$)

The output table has 3 columns and k lines; the i-th line is:

- For $i < k$, let $V_i = \left\{ \llbracket e_3 \rrbracket_{G,v} \middle| v \in T_i \right\}$ and let $u \in T_i$.

Aggregation (4)

Semantics on an example

Example: `WITH` e_1 `AS` a_1, e_2 `AS` a_2, $max(e_3)$ `AS` a_3

e_1, e_2 and e_3 are ⟨expr⟩
$max(e_3)$ is an ⟨aggexpr⟩

We group the lines according to the expressions e_1, e_2:

- We partition the input table as subtables $T_1, \ldots, T_k$
- For each $u, v \in T_i$, $\quad [\![e_j]\!]_{G,u} = [\![e_j]\!]_{G,v} \quad$ (for $j \in \{1, 2\}$)

The output table has 3 columns and k lines; the i-th line is:

- For $i < k$, let $V_i = \left\{ [\![e_3]\!]_{G,v} \middle| v \in T_i \right\}$ and let $u \in T_i$.
- The i-th line is $\left(a_1 : [\![e_1]\!]_{G,u},\ a_2 : [\![e_2]\!]_{G,u},\ a_3 : max(V_i)\right)$

Line order (1)

Principle

- `ORDER BY` clause: provides an order
- `LIMIT` subclause: keeps only the first lines of the table
- `SKIP` subclause: removes the first lines of the table

Line order (1)

Principle

- `ORDER BY` clause: provides an order
- `LIMIT` subclause: keeps only the first lines of the table
- `SKIP` subclause: removes the first lines of the table

Up to now

- In tables, columns and lines are not ordered

- Semantics of queries
- Semantics of clauses
- Semantics of subclauses

} are functions from tables to tables

Line order (1)

Principle

- `ORDER BY` clause: provides an order
- `LIMIT` subclause: keeps only the first lines of the table
- `SKIP` subclause: removes the first lines of the table

Up to now

- In tables, columns and lines are not ordered

- Semantics of queries
- Semantics of clauses
- Semantics of subclauses

} are functions from tables to tables

From now on

- `ORDER BY` and its subclause propagate an order

Line order (2) — Semantics through example

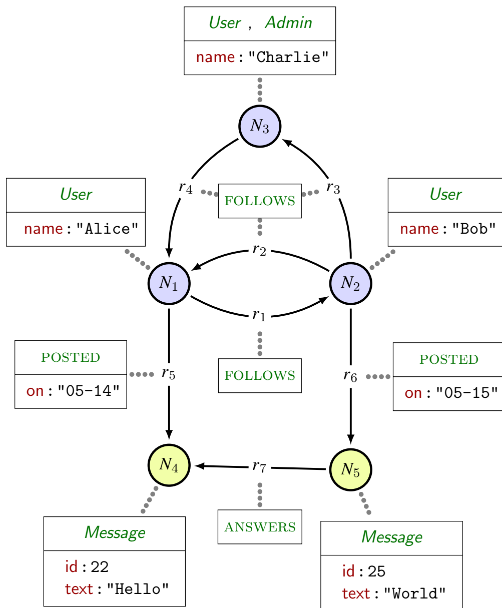
("Alice" < "Bob" < "Charlie")

Query:

```
MATCH (a:User)
ORDER BY a.name
  SKIP 1
  LIMIT 1
RETURN a.name AS b
```

b
"Bob"

Line order (3) — Semantics through example



(`false` < `true`)

Query:

```
MATCH (a:User)
ORDER BY a:Admin
  SKIP 1
  LIMIT 1
RETURN a.name AS b
```

In this case:

- Non-determinism
- 1 column and 1 line
- Cell contains either `"Alice"` or `"Bob"`

Oddities and inconsistencies


```
WHERE []
```

```
WHERE [1] = 1
```

```
MATCH (a)-[r*]->(b)-[r*]->(c)
```

```
MATCH (a)
OPTIONAL MATCH p=(a),(b:LabelAbsentFromTheGraph)
```

```
UNWIND [[1,2],3] AS a
UNWIND a AS b
```

```
UNWIND [1,[2,3]] AS a
UNWIND a AS b
```

```
WITH a + b + count(c) AS x
```

```
WITH a + count(c) + b AS x
```

```
MATCH (n)-[r]-() AS a
DELETE n
```

```
MATCH (n)-[r]-() AS a
DELETE n
  [Arbitrary clauses]
DELETE r
```