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

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.

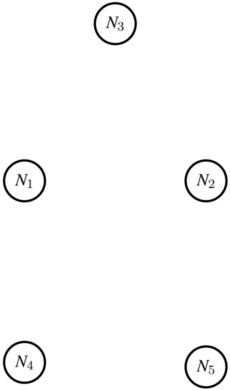
- Industrial partnership Neo4j/University of Edinburgh
- Reverse engineering and formalisation from Neo4j
- Done semantics of the "core fragment' [Francis et al'18]
- Soon: semantics of the "update clauses

Long term goal

Design a standard language for querying property graphs: GQL.

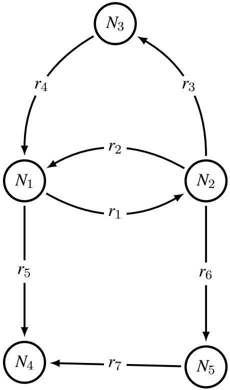
- Merging Cypher, PGQL, G-Core.
- Involvement of Neo Technology, Oracle and LDBC.

Outline



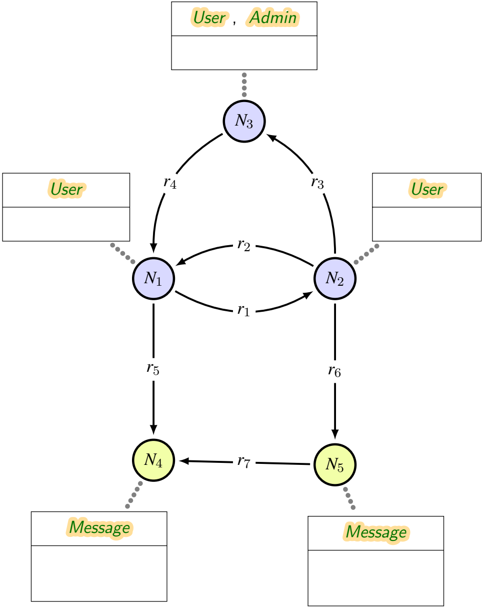
- Nodes : $N_1, N_2, \cdots, N_5$

Property graph



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

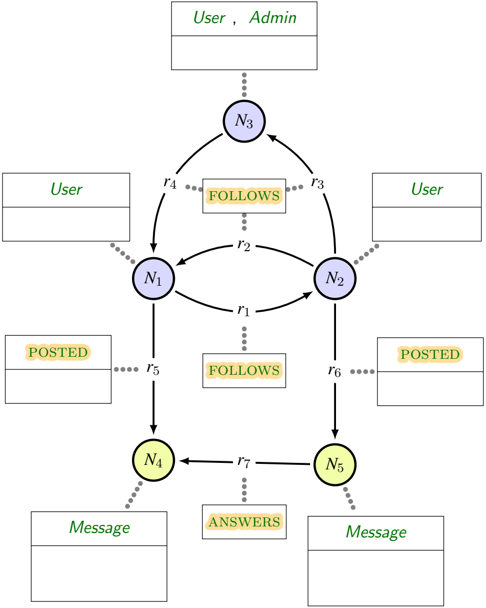
Property graph



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

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

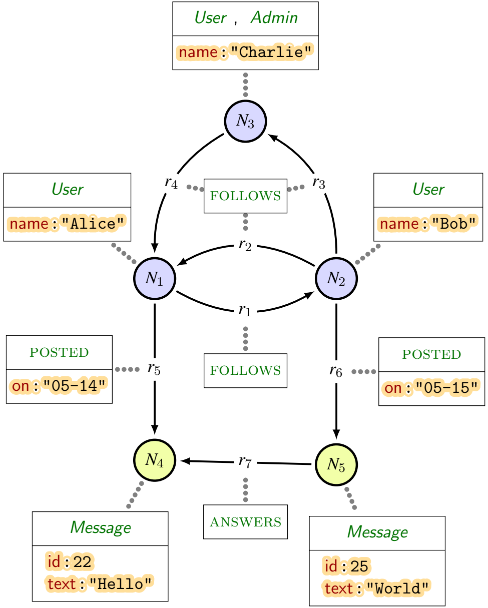
Property graph



- 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

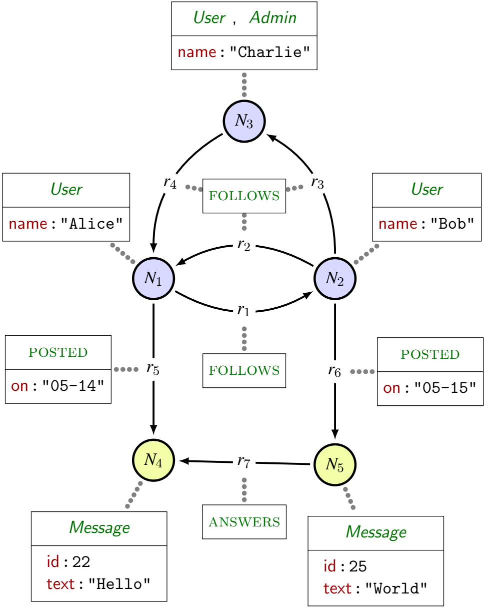


- Nodes : $N_1, N_2, \cdots, N_5$
- Relationships : $r_1, r_2, \cdots, r_7$

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

Outline

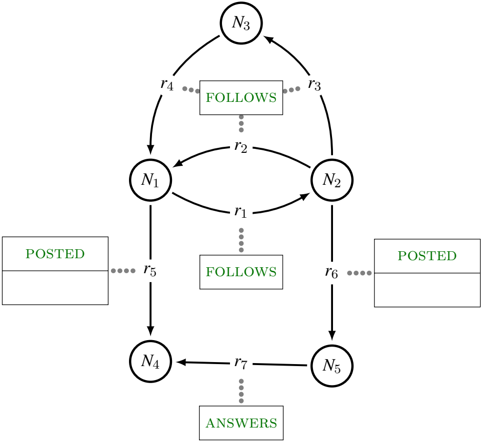
Data model for theoretical query languages



Graph database

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

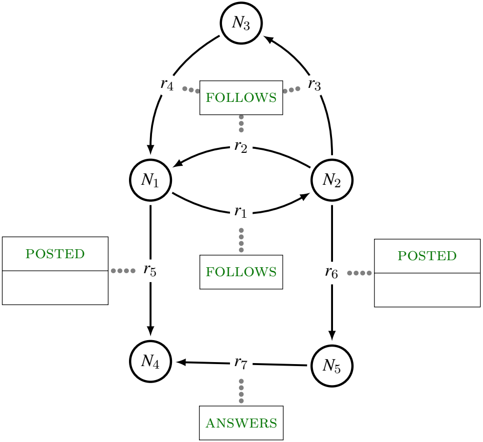
Data model for theoretical query languages



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"

Regular Path Query (RPQ) – Definition

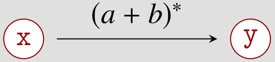
A: finite alphabet (of relation types)
N: nodes in the graph

Definition (RPQ R):

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

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

Example:

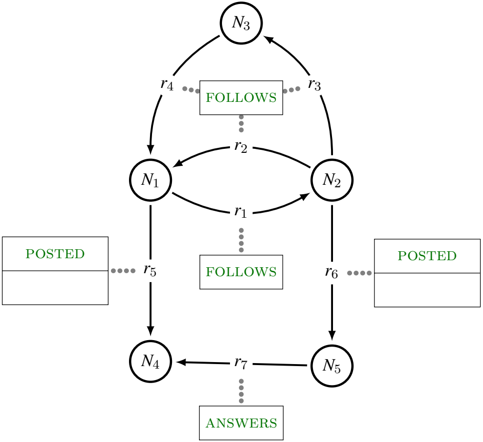


Definition (Answer to R):

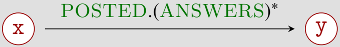
Set of the functions

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

Regular Path Query (RPQ) – Example 1



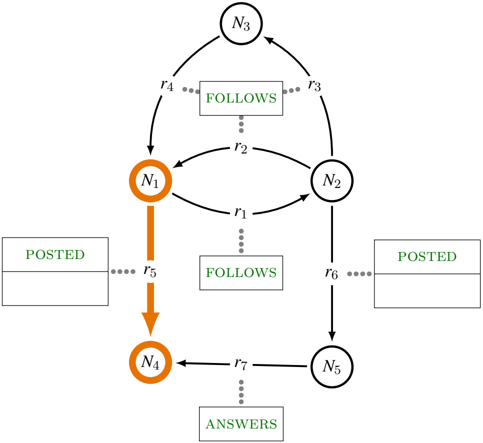
Query:



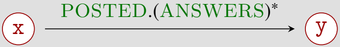
Answers:

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

Regular Path Query (RPQ) – Example 1



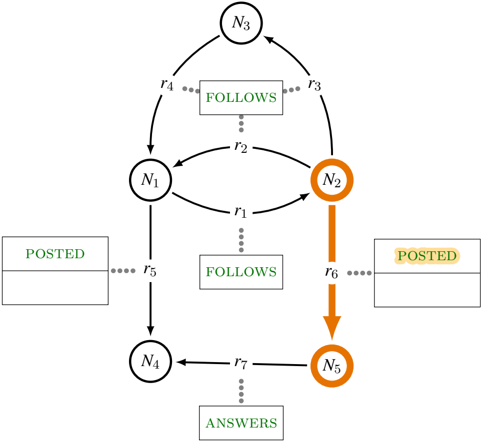
Query:



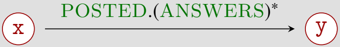
Answers:

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

Regular Path Query (RPQ) – Example 1



Query:



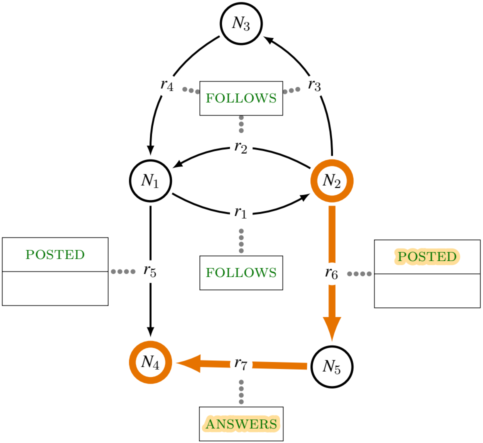
Answers:

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

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

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

Regular Path Query (RPQ) – Example 1



Query:

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

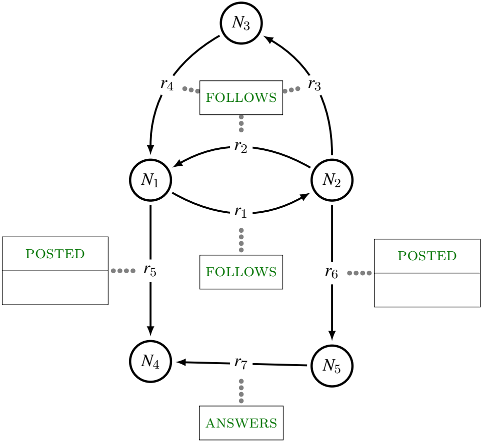
Answers:

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

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

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

Regular Path Query (RPQ) – Example 1



Query:

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

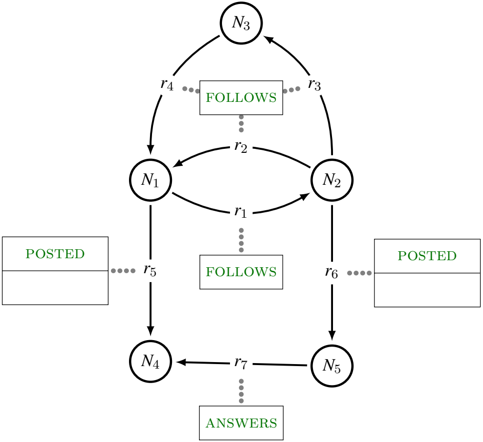
Answers:

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

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

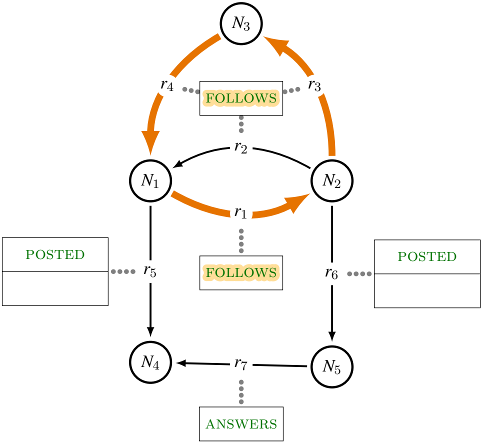
$F_3: \quad x \mapsto N_2 \quad y \mapsto N_4$

Regular Path Query (RPQ) – Example 2



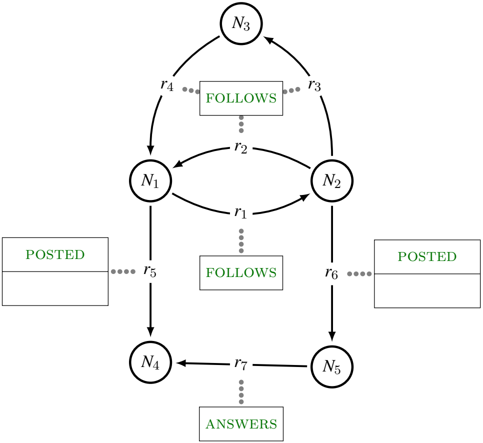
Query:

Regular Path Query (RPQ) – Example 2



Query:

Regular Path Query (RPQ) – Example 2



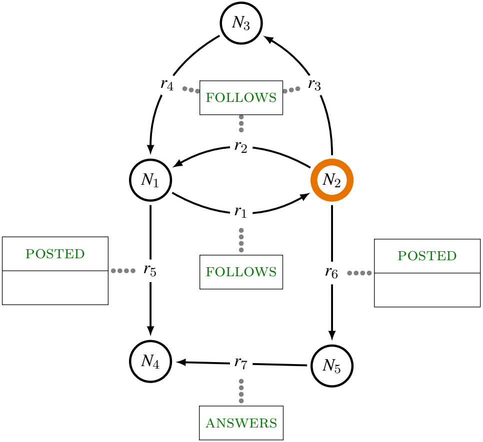
Query:

$x \xrightarrow{(\text{FOLLOWS})^*} y$

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

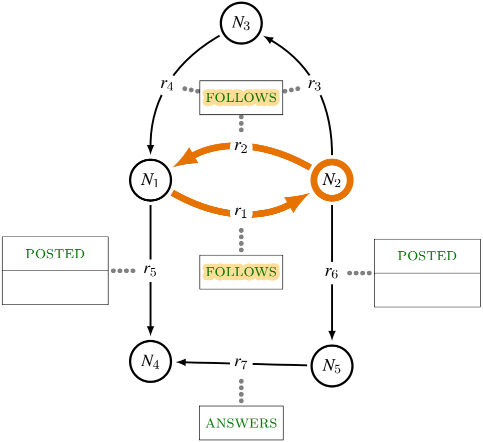


Query:

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:

x —(FOLLOWS)*→ y

Answers:

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

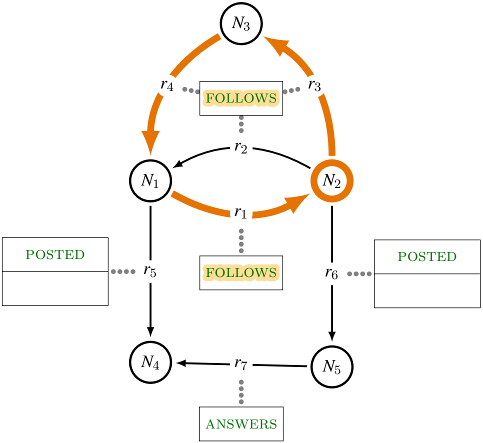
$F_2: \quad x \mapsto N_1 \quad y \mapsto N_2$

$F_3: \quad x \mapsto N_1 \quad y \mapsto N_3$

$\vdots \quad \vdots \quad \vdots$

$F_9: \quad x \mapsto N_3 \quad y \mapsto N_3$

Regular Path Query (RPQ) – Example 2



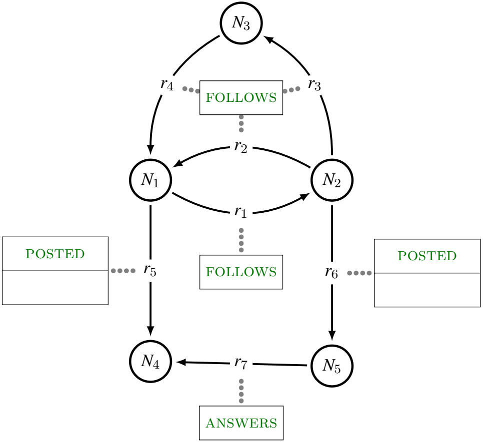
Query:

$x \xrightarrow{\text{(FOLLOWS)}^*} y$

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:

x —(FOLLOWS)*→ y

Answers:

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

→ ∞-many path realise F_1

→ RPQ follows set-semantics

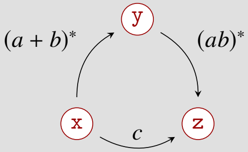
A: alphabet (of relation types)
N: nodes in the graph

Definition (a CRPQ C):

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

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

Example:

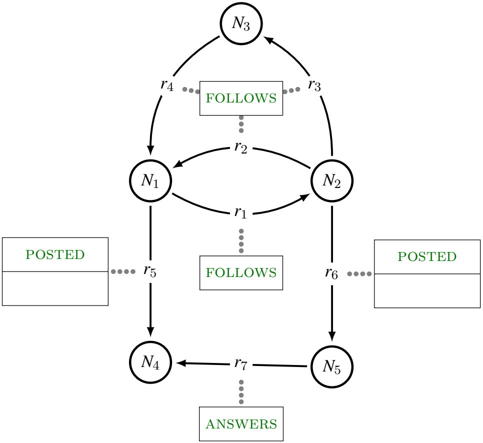


Answer to C:

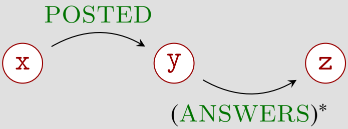
Set of the functions $F : \mathsf{var}(C) \longrightarrow N$

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
	$\downarrow$	$\downarrow$	$\downarrow$
F_1 :	N_1	N_4	N_4
F_2 :	N_2	N_5	N_4
F_3 :	N_2	N_5	N_5

Union of CRPQs (UCRPQ) – Defintion

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:

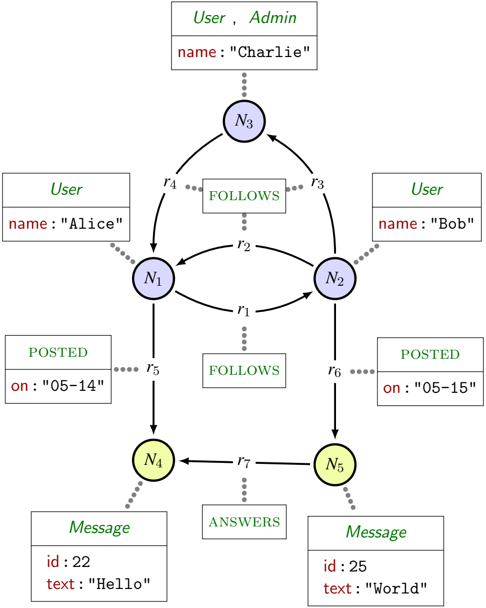
Set of partial functions: $\big(\mathsf{var}(C_1) \cup \cdots \cup \mathsf{var}(C_n)\big) \to N$

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

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

Outline

Evaluating a Cypher query



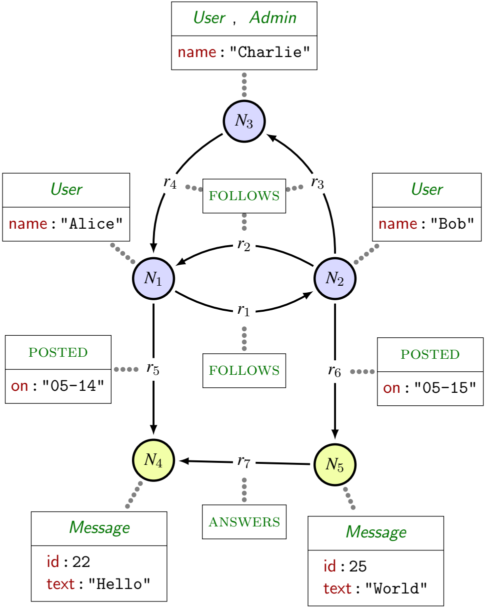
Example of Cypher query:

```
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



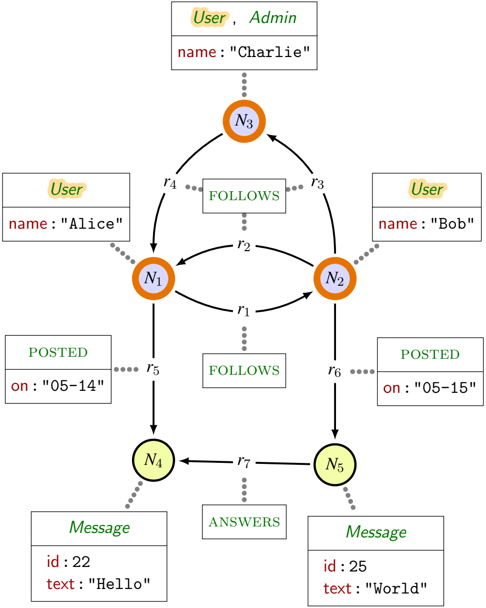
Example of Cypher query:

```
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

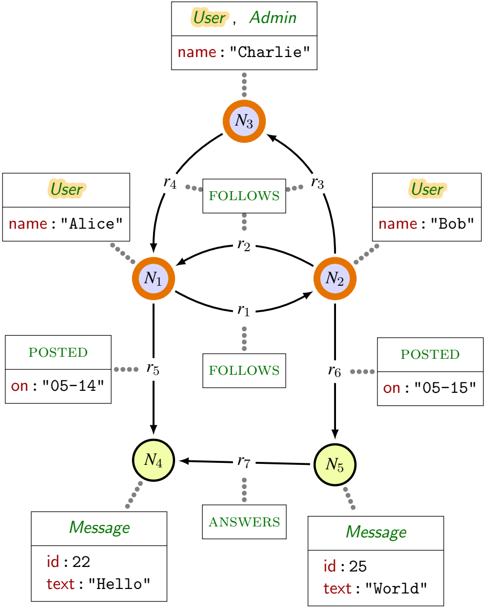
Matching nodes (1)



Query:

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

Matching nodes (1)



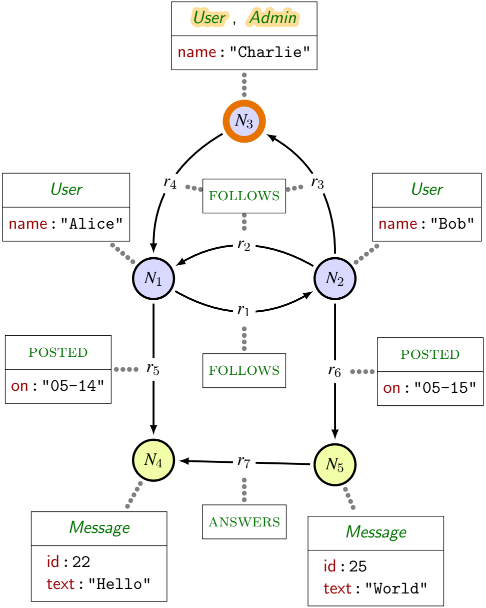
Query:

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

Result:

u1
N_1
N_2
N_3

Matching nodes (2)



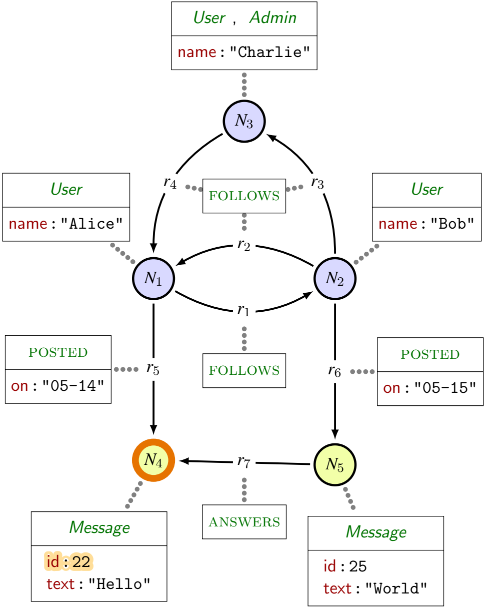
Query:

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

Result:

u1
N_3

Matching nodes (3)



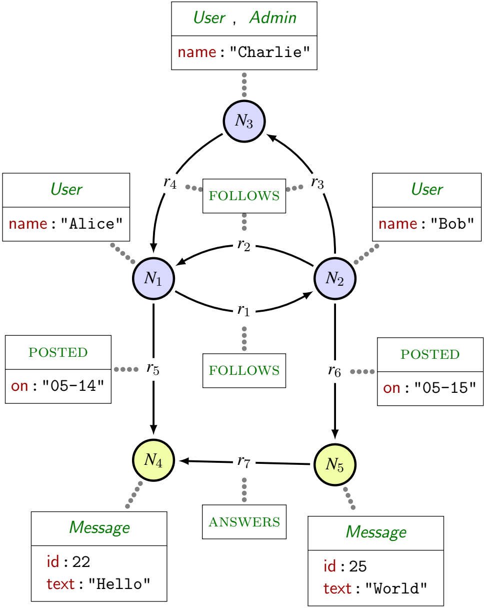
Query:

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

Result:

u1
N_4

Matching relations (1)



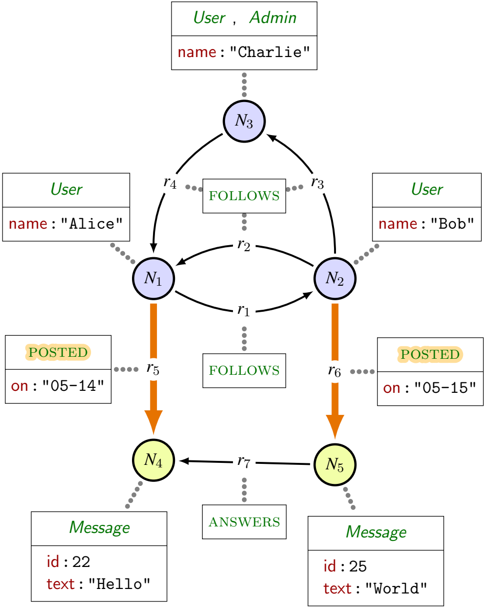
Query:

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

Result:

p1
r_1
r_2
r_3
r_4
r_5
r_6
r_7

Matching relation (2)



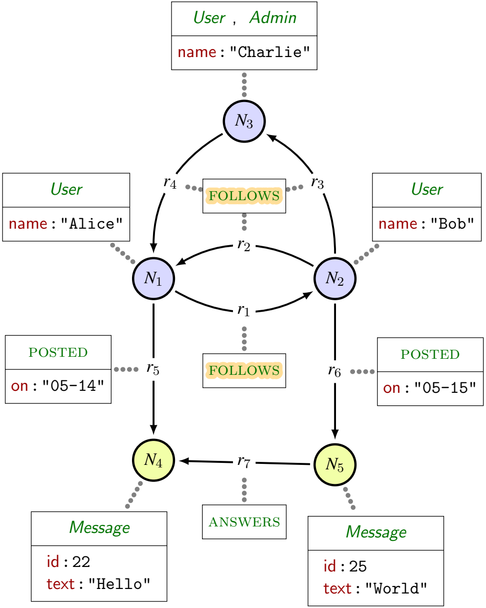
Query:

```
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]->()
```

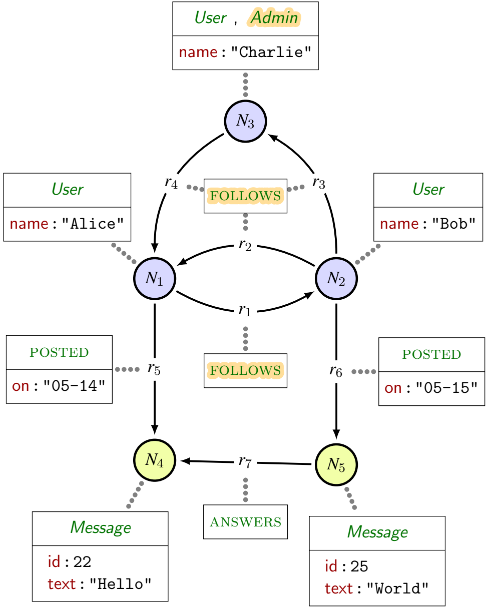
Result:

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)



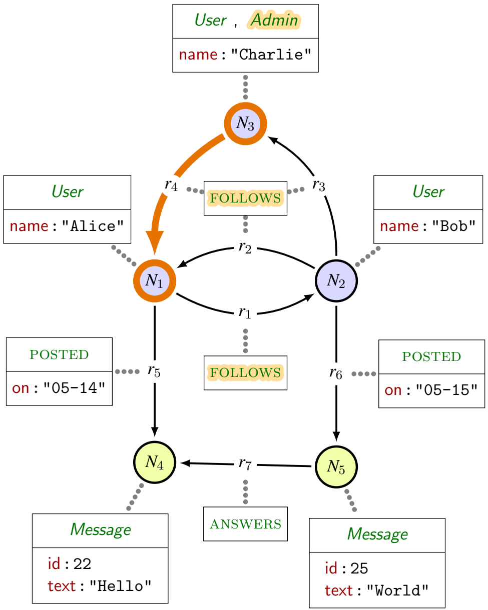
Query:

```
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)



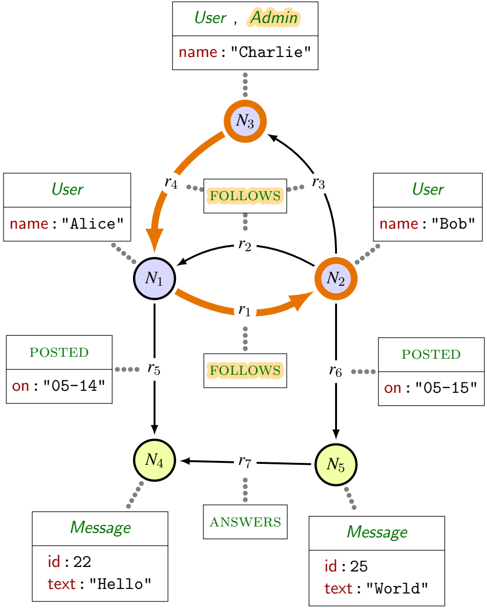
Query:

```
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)



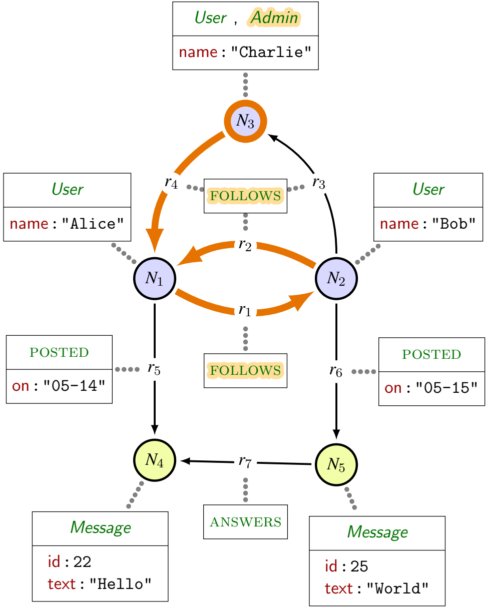
Query:

```
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)



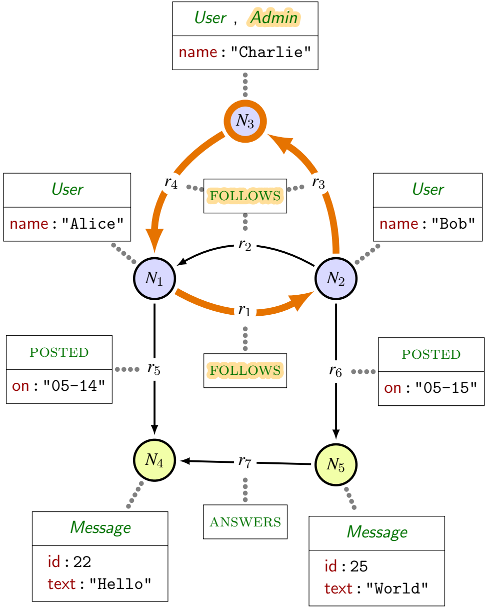
Query:

```
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)



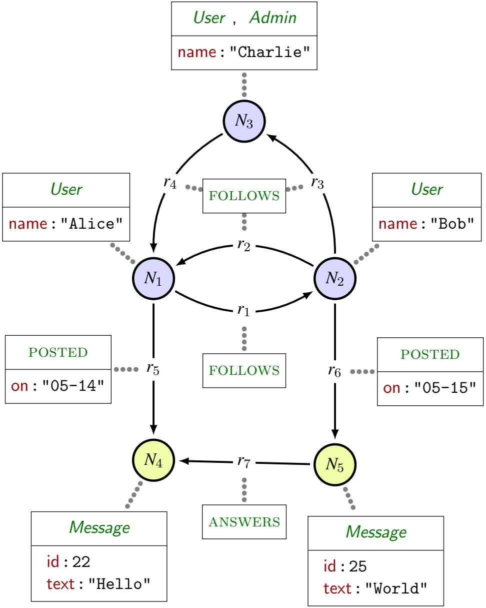
Query:

```
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

Matching paths (1)



Query:

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

Result:

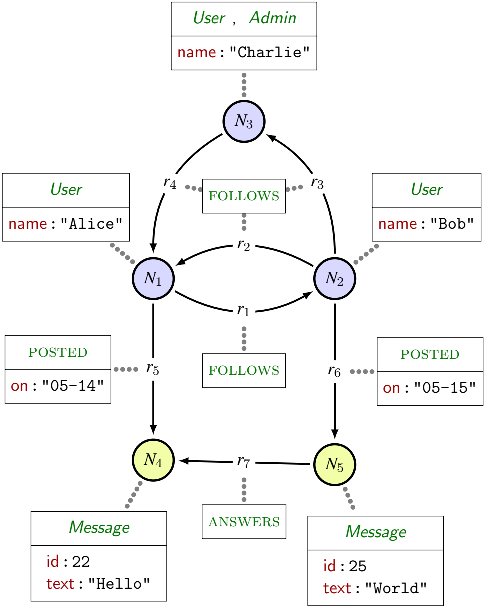
u1	l1	m1
N_3	$[r_4]$	N_1
N_3	$[r_4, r_1]$	N_2
N_3	$[r_4, r_1, r_2]$	N_1
N_3	$[r_4, r_1, r_3]$	N_3

Cypher-Morphism

- Each rel. matched ≤ 1 time

$\Rightarrow$ Finitely many results

Matching paths (2)



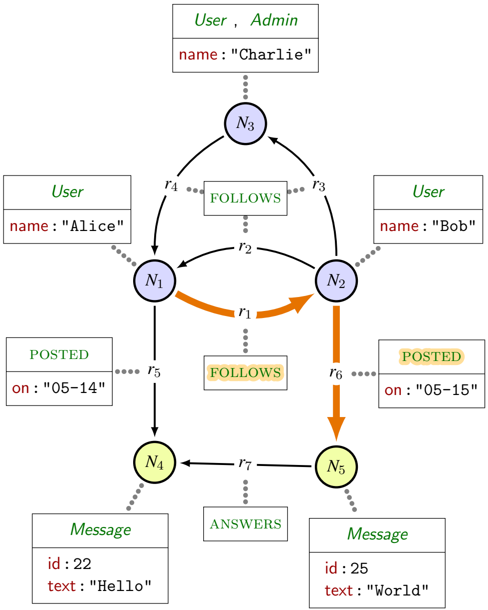
Query:

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

Result:

u1	m1
N_1	N_5
N_2	N_4
N_3	N_5

Matching paths (2)



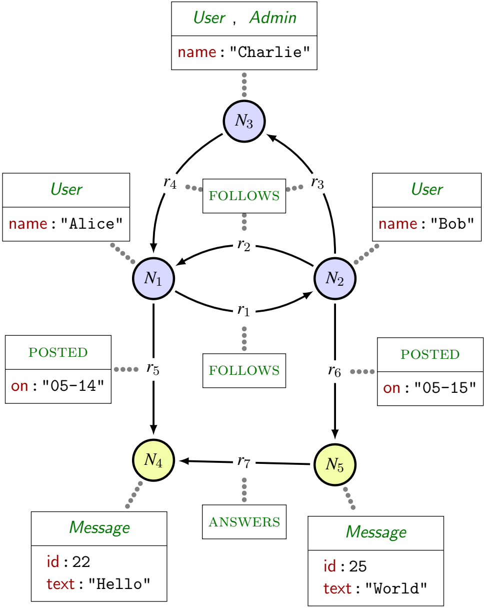
Query:

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

Result:

u1	m1
N_1	N_5
N_2	N_4
N_3	N_5

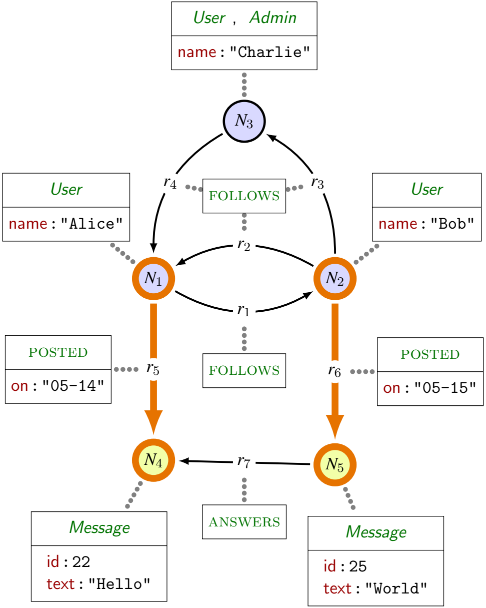
Chaining clauses



Query:

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

Chaining clauses



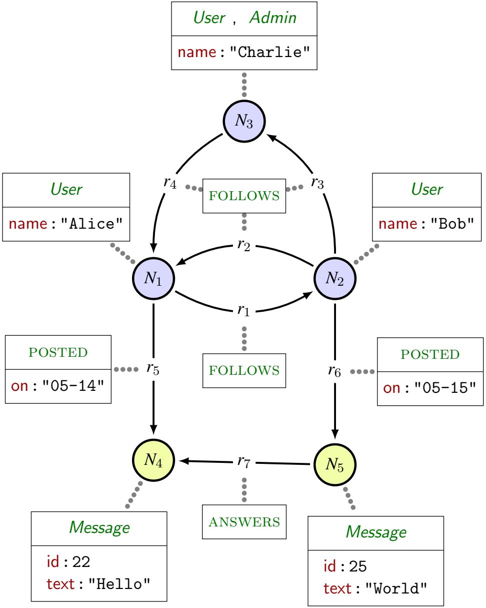
Query:

```
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:

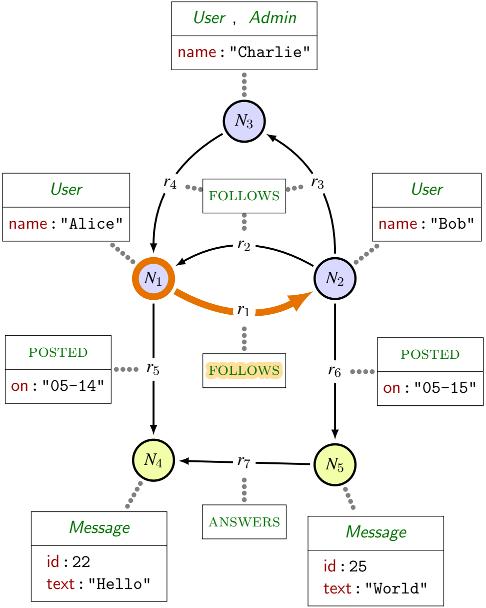
```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
                -[:FOLLOWS]->(u3)
```

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	·	·



Query:

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

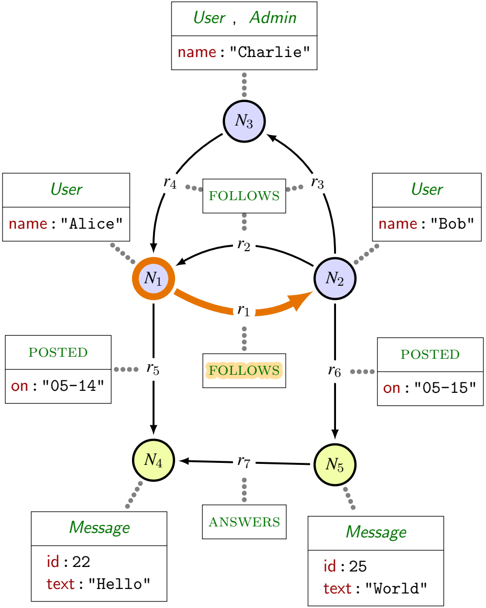
Table after first MATCH:

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)
```

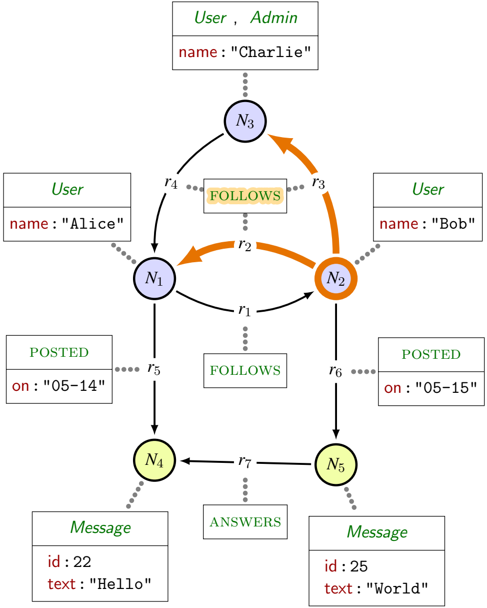
Table after first MATCH:

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)
```

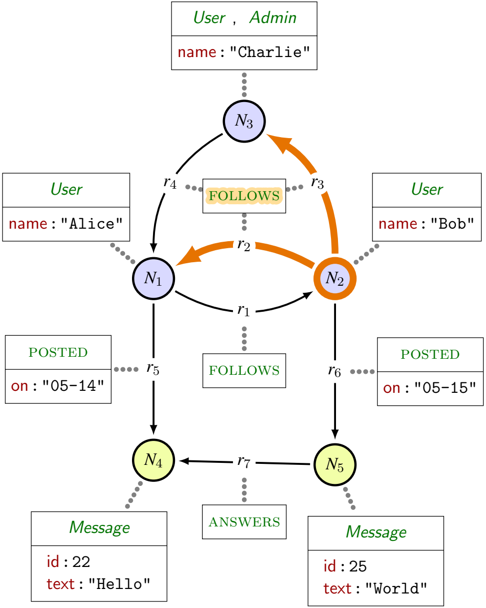
Table after first MATCH:

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:

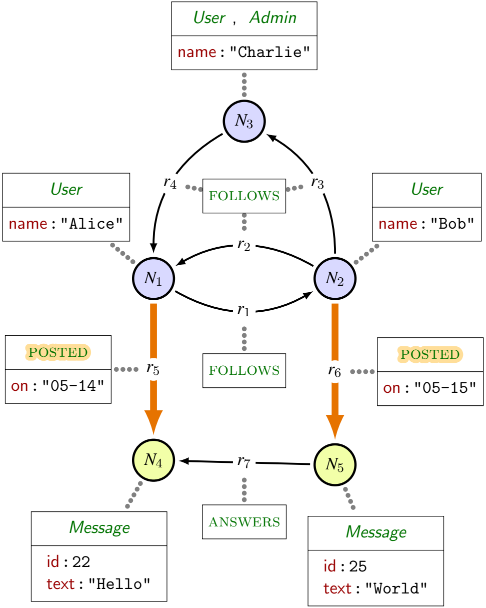
```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
            -[:FOLLOWS]->(u3)
```

Table after first MATCH:

u1	m1
N_1	N_4
N_2	N_5

Table after second MATCH:

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



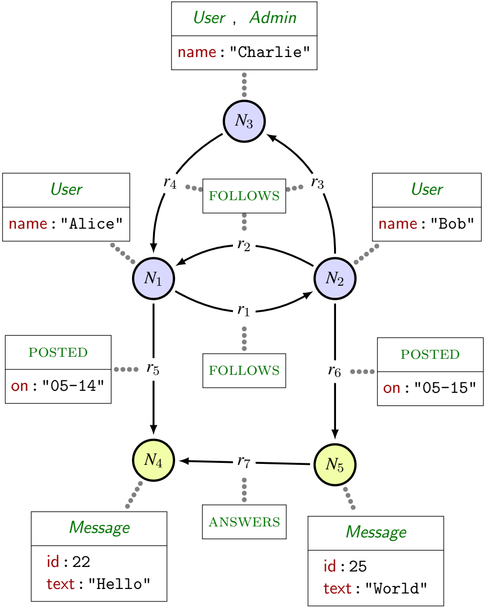
Query:

```
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
```

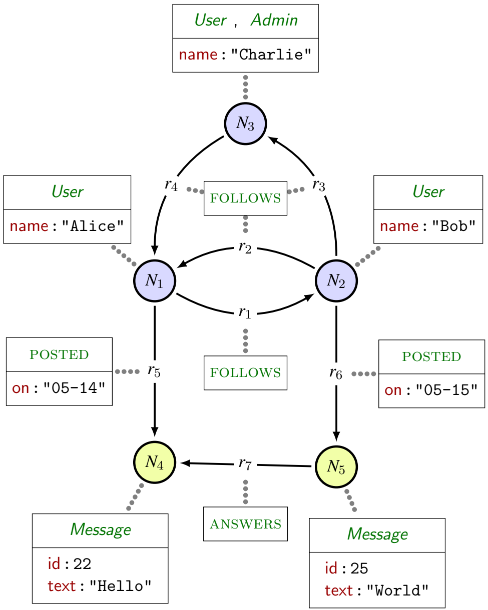
After the MATCH clause

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
```

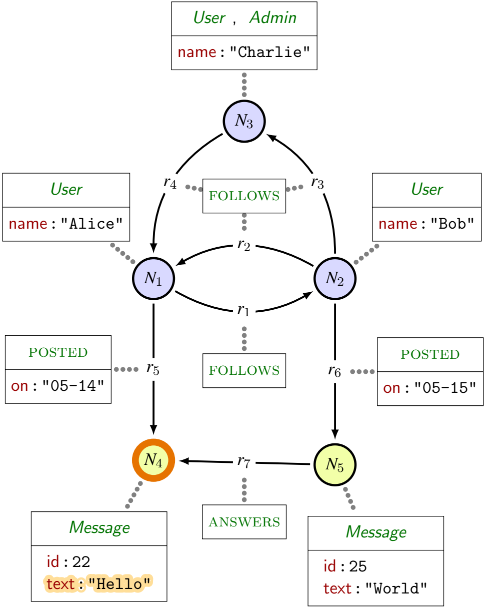
After the MATCH clause

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
```

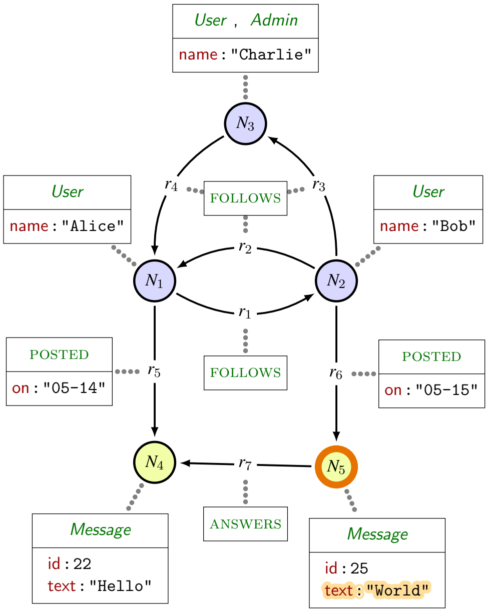
After the MATCH clause

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	

Column manipulation (WITH clause)



Query:

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

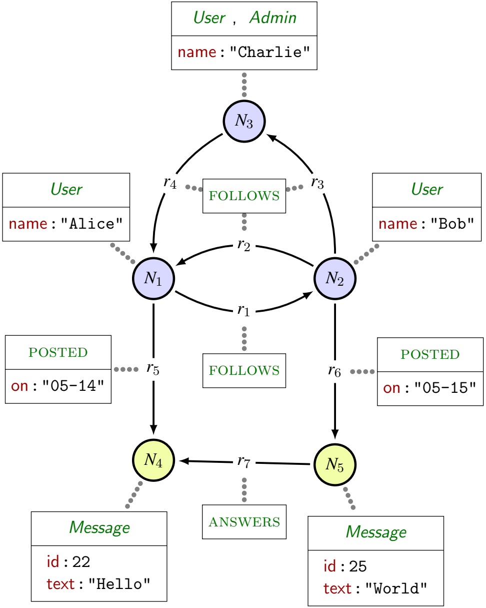
After the MATCH clause

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	"World"

Column manipulation (WITH clause)



Query:

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

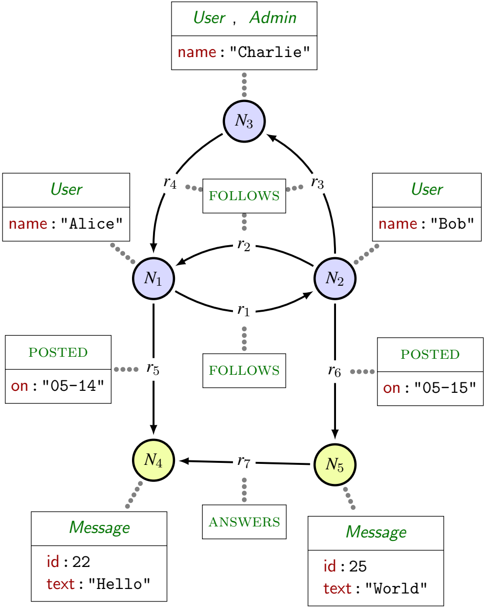
After the MATCH clause

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)



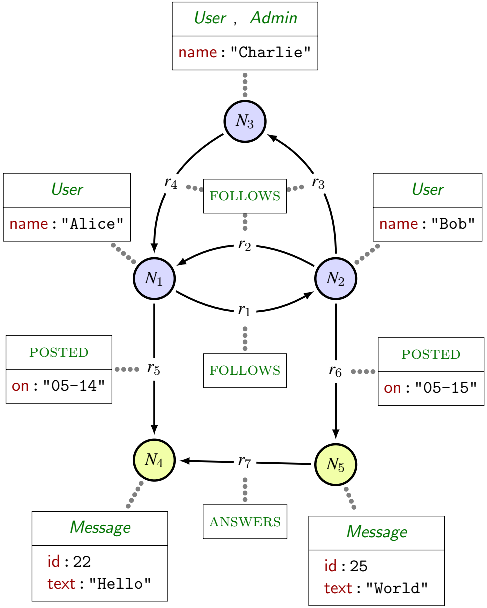
Query:

```
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"
```

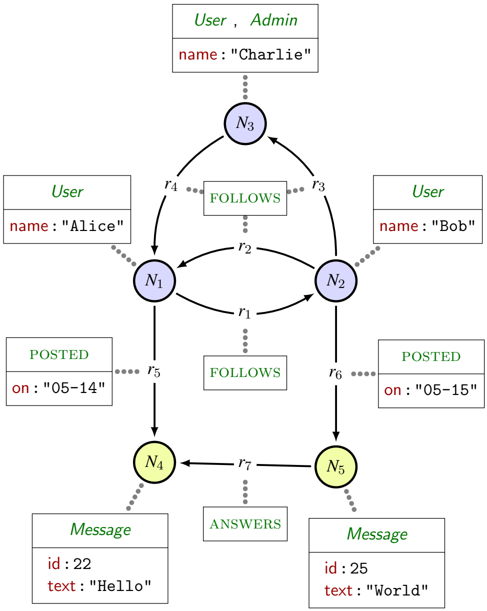
After the WITH clause

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"
```

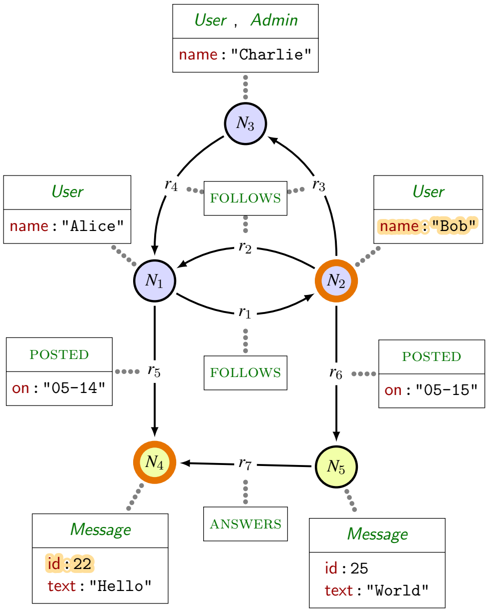
After the WITH clause

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



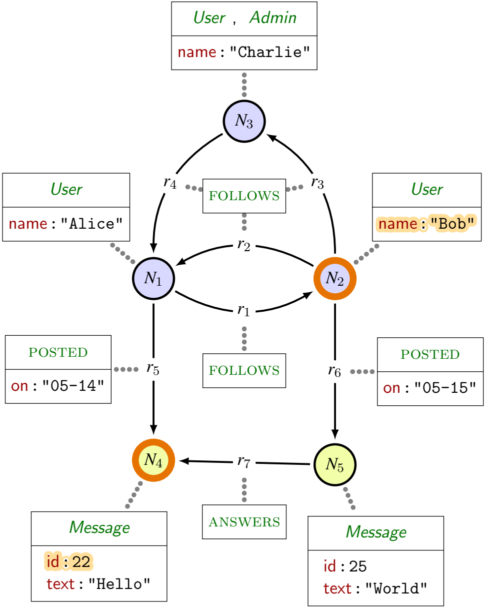
Query:

```
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)
```

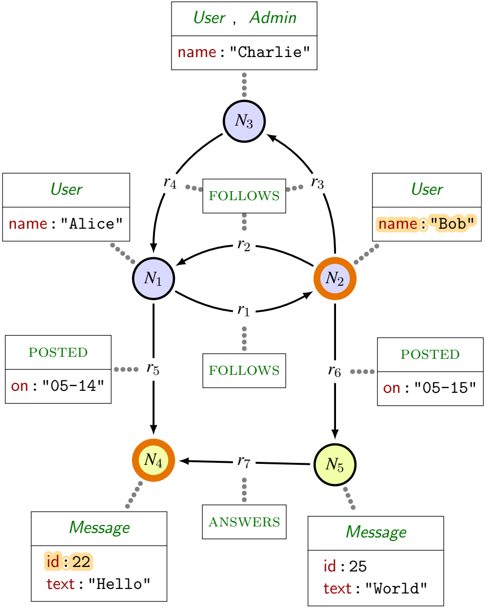
Question

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)
```

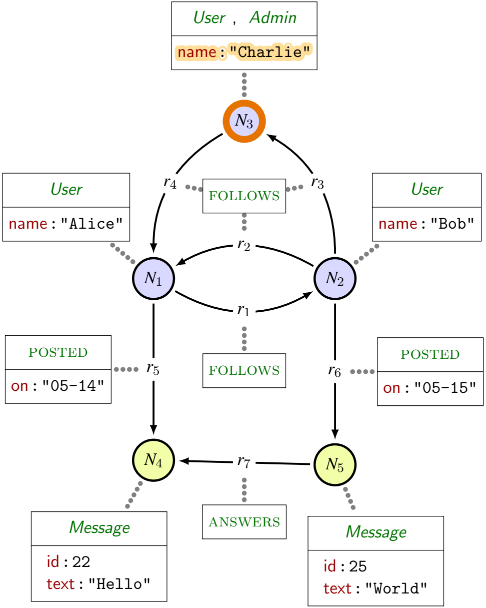
Question

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



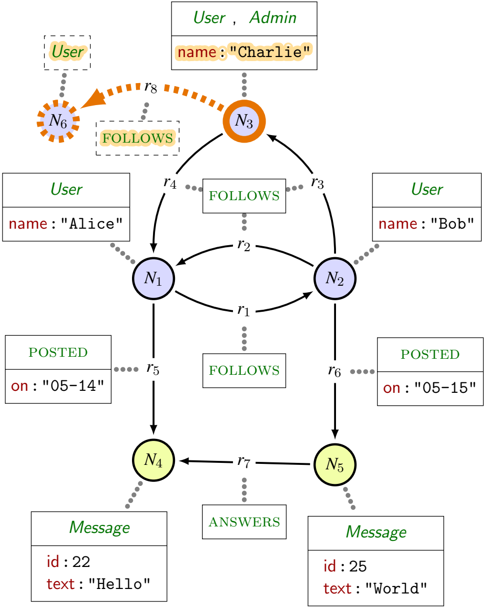
Query:

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

Table after MATCH clause:

a
N_3

Node and relation creation (CREATE)



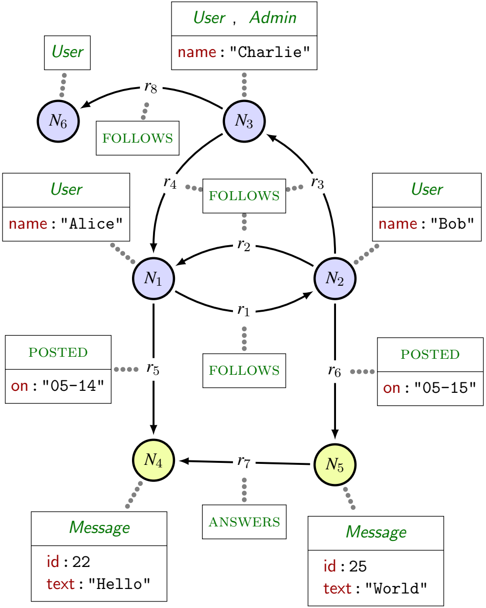
Query:

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

Table after MATCH clause:

a
N_3

Node and relation creation (CREATE)



Query:

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

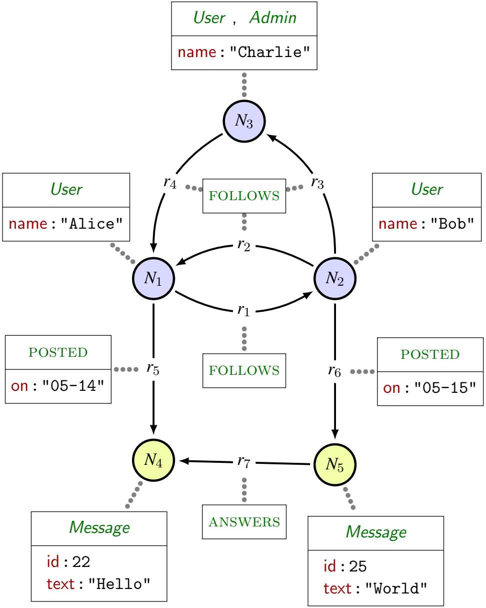
Table after MATCH clause:

a
N_3

Table after CREATE clause:

a	b
N_3	N_6

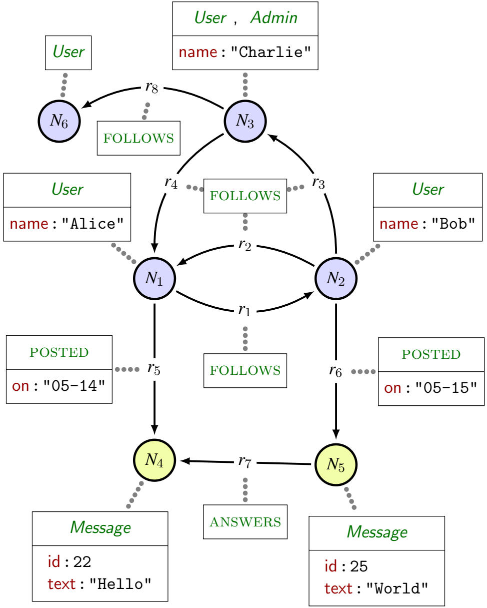
The example graph stored as CREATE clauses



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)



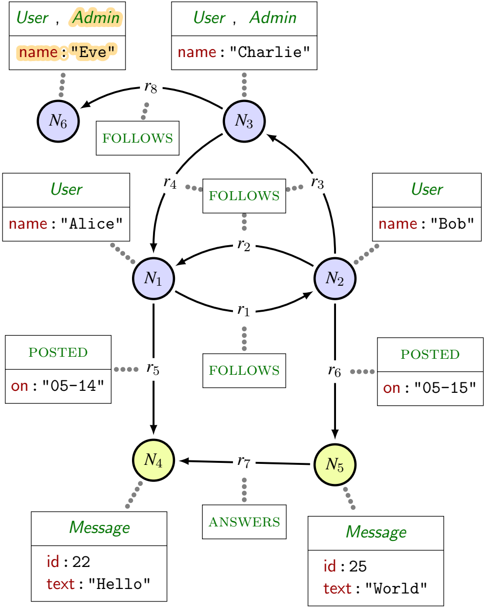
Query:

```
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)



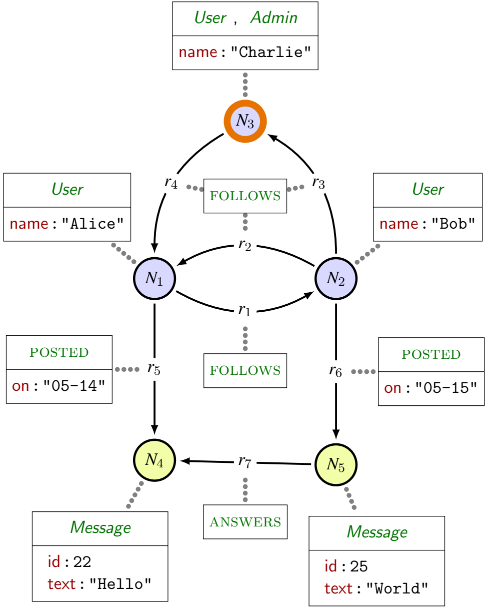
Query:

```
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"

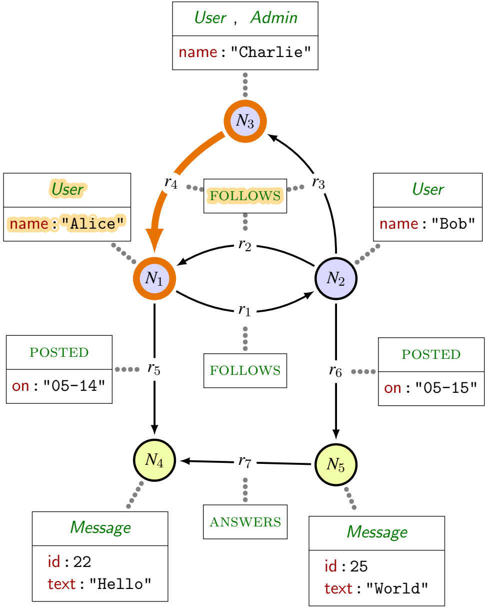
Query:

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

Output table:

a	n	c
N_3	"Alice"	
N_3	"Eve"	

The MERGE clause : MATCH else CREATE



Input table:

a	n
N_3	"Alice"
N_3	"Eve"

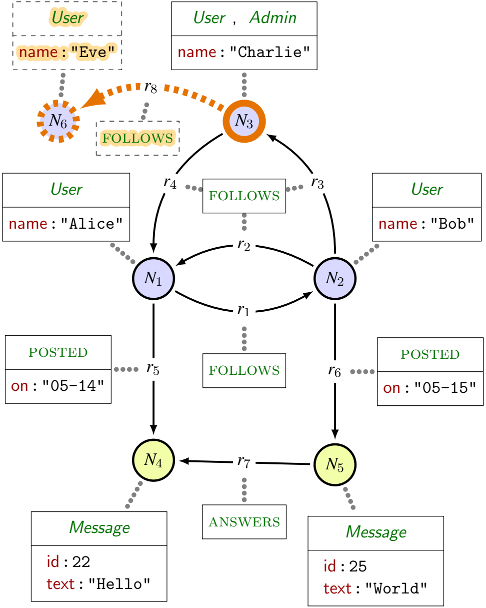
Query:

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

Output table:

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

The MERGE clause : MATCH else CREATE



Input table:

a	n
N_3	"Alice"
N_3	"Eve"

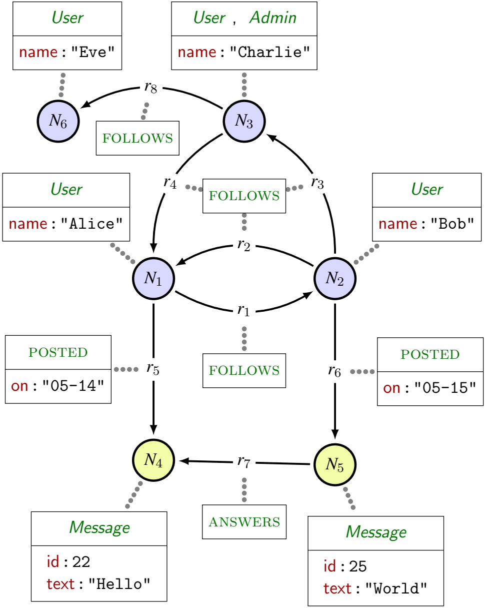
Query:

```
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

Record (table row)

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

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

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

$\llbracket \cdot \rrbracket_{u,G}$: expression $\longmapsto$ value (where u is a record)

Semantics of clauses

$\llbracket \cdot \rrbracket_{G}$: clause $\longmapsto$ (function: Tables $\to$ Tables)

Semantics of queries

$\llbracket \cdot \rrbracket_{G}$: 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 $[\![Q]\!]_G = [\![C_n]\!]_G \circ \cdots \circ [\![C_2]\!]_G \circ [\![C_1]\!]_G$
- output$(G, Q) = [\![Q]\!]_G\left(T_{\text{unit}}\right)$

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)

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

- $[\![\texttt{UNWIND } e \texttt{ AS } a]\!]_G(T) = \biguplus_{u \in T} \biguplus_{v \in E_u} \{(u, a : v)\},$
  $$\text{with } \ E_u = \begin{cases} \biguplus_{0 \leq i < m} \{v_i\} & \text{if } [\![e]\!]_{G,u} = \text{list}(v_0, \ldots, v_{m-1}) \\ \{\} & \text{if } [\![e]\!]_{G,u} = \text{list}() \\ \left\{ [\![e]\!]_{G,u} \right\} & \text{otherwise} \end{cases}$$

Semantics brackets – Read-Write

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

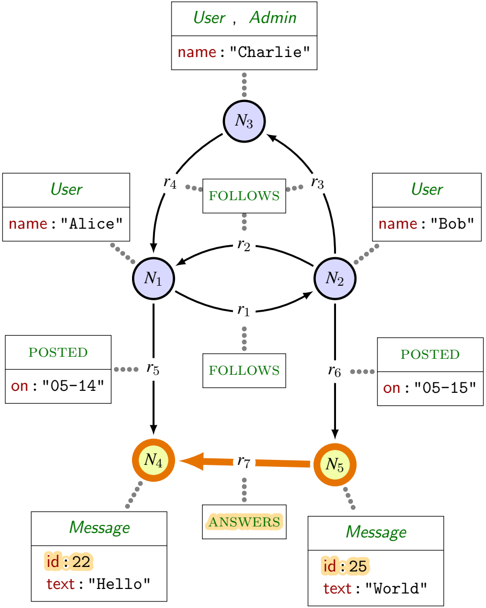
General scheme of the semantics if a clause is:

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



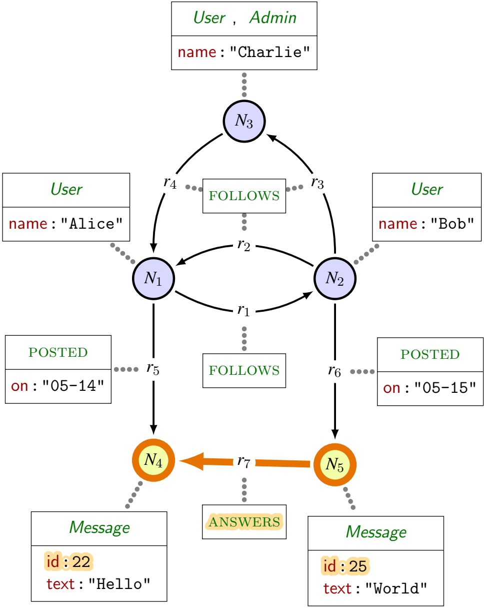
Query:

```
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
```

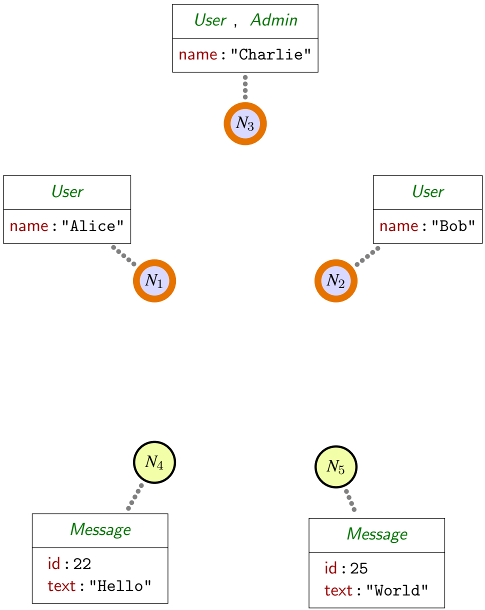
Table before SET clause:

m1	m2
N_5	N_4

Neo4j sets both id to 25.

Semantics exchange id values

In Neo4j, MERGE is highly non-deterministic



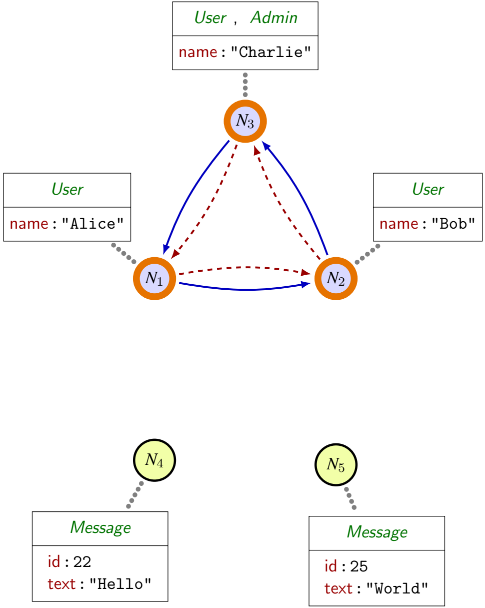
Input Table:

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:

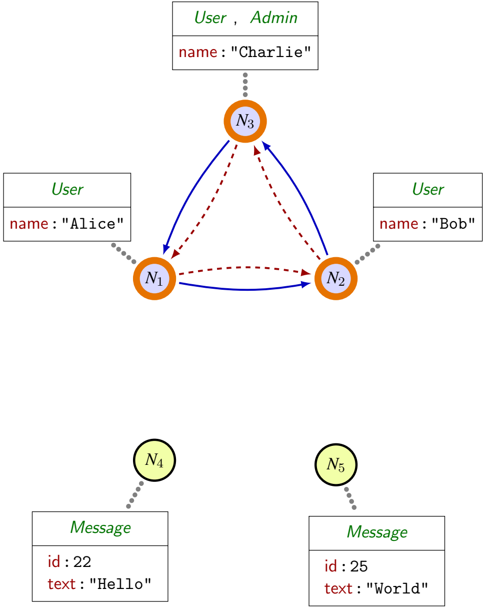
a	b	c
N_1	N_2	N_3
N_2	N_3	N_1
N_3	N_1	N_2

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

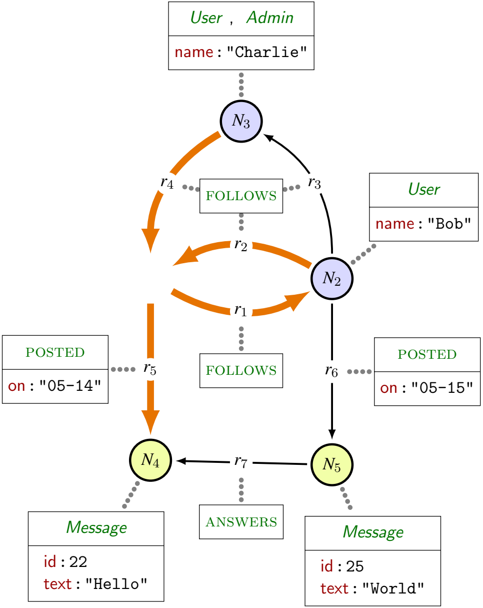
Query:

```
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

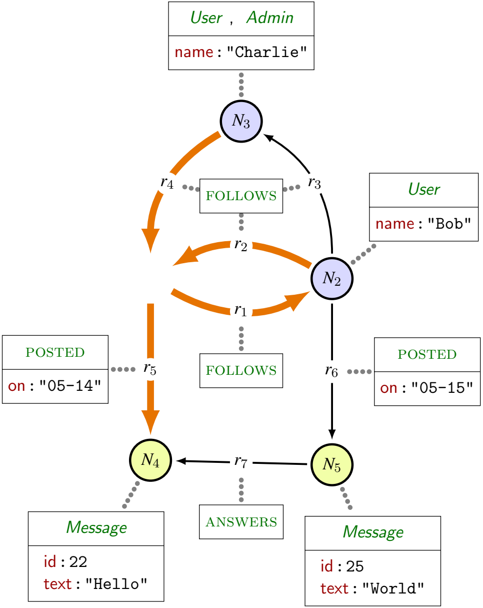


Query:

```
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

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
  $\longrightarrow (ab)^3+(ab)^2cd+ab(cd)^2+(cd)^3$

PGQL (Property Graph Query Language)

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

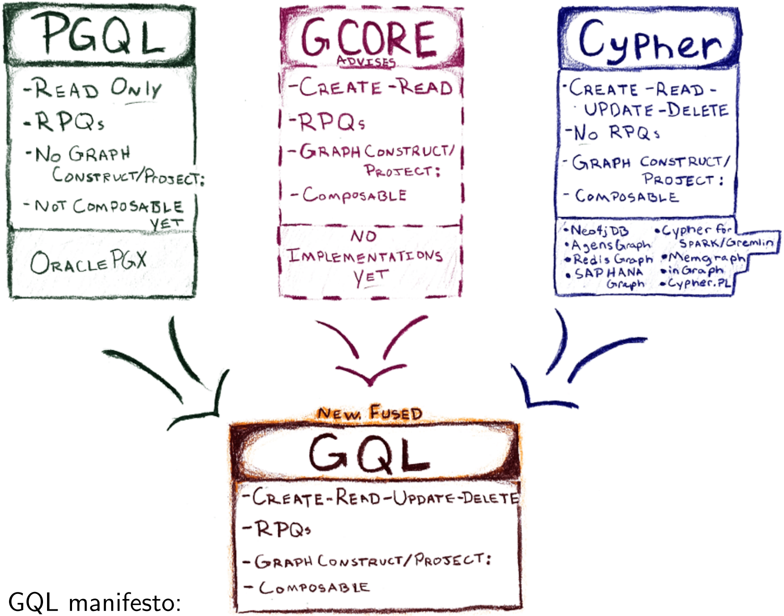
Example: (from http://pgql-lang.org/)

```
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.       */
)
```

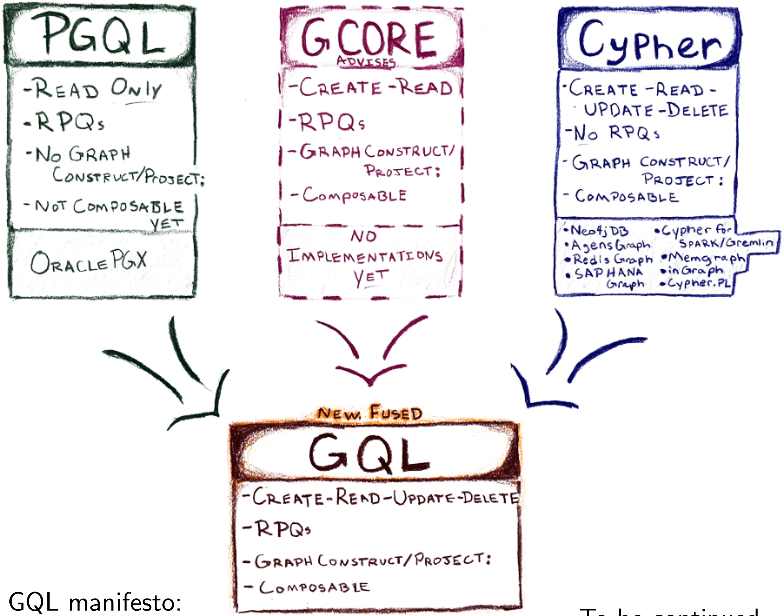
- Designed by LDBC
- Experimental
- Support full UCRPQ
- ASCII-art representation of patterns (similar to Cypher)
- Support Multiple graphs, graph views, and paths cost
- Multiple semantics

Example:

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



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



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

To be continued...