

# Formal semantics of the query-language Cypher

Victor Marsault<sup>1</sup>

joint work with

Nadime Francis<sup>2</sup>

Leonid Libkin<sup>1</sup>

Mats Rydberg<sup>3</sup>

Andrés Taylor<sup>3</sup>

Alastair Green<sup>3</sup>

Tobias Lindaaaker<sup>3</sup>

Petra Selmer<sup>3</sup>

Paolo Guagliardo<sup>1</sup>

Stefan Plantikow<sup>3</sup>

Martin Schuster<sup>1</sup>

1. University of Edinburgh
2. Université Paris-Est Marne-la-vallée
3. NeoTechnology

Séminaire de l'équipe Valda

DI, ENS, Paris

2018-15-05

- 1 Introduction
- 2 Cypher by example
- 3 General principles of the semantics
- 4 Comparison Cypher/UCRPQ
- 5 The MATCH clause
- 6 Conclusion and future work

Most databases use the relational model

- Relational algebra in theory
- The language SQL in practice

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

## Data model

Property graphs, RDF

## Query languages

Cypher, Gremlin, PGQL, SparQL

## Engines

JanusGraph, Jena, Neo4j, Virtuoso

## Domain

Fraud detection, Investigative journalism

## Data model

Property graphs, RDF

## Query languages

Cypher, Gremlin, PGQL, SparQL

## Engines

JanusGraph, Jena, Neo4j, Virtuoso

## Domain

Fraud detection, Investigative journalism

- Language for querying and updating *property graphs*
- Invented by Neo Technology
- Originally, part of engine Neo4j → commercial success
- Now, in multiple datagraph engines (e.g., SAP HANA Graph, Redis Graph, Agens Graph)

- Language for querying and updating *property graphs*

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

## The openCypher project

- Since 2015
- Seeks to make of Cypher a standard (SQL for property graphs?)
  - Community-led evolution
  - Complete specification

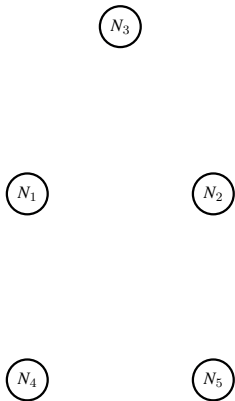


## Our goal

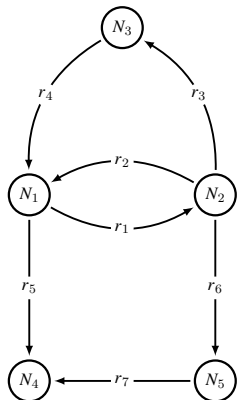
Full denotational semantics for the language Cypher.

- Industrial partnership Neo Technology/University of Edinburgh
- Reverse engineering and formalisation from Neo4j
- Today: Semantics of the “core fragment” of Cypher

- 1 Introduction
- 2 Cypher by example
- 3 General principles of the semantics
- 4 Comparison Cypher/UCRPQ
- 5 The MATCH clause
- 6 Conclusion and future work



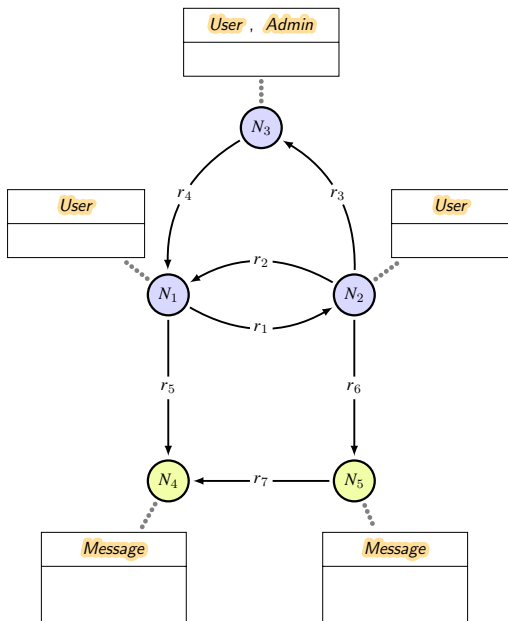
- Nodes :  $N_1, N_2, \dots, N_5$



■ Nodes :  $N_1, N_2, \dots, N_5$

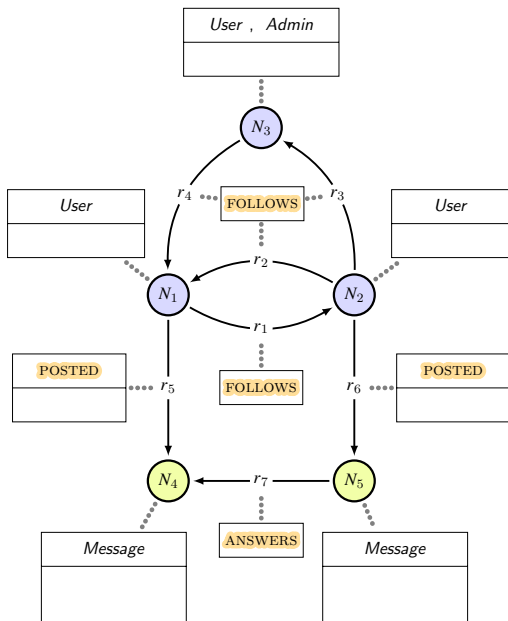
■ Relationships :

$r_1, r_2, \dots, r_7$



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

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



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

- Relationships :

$r_1, r_2, \dots, r_7$

- Labels (de nœuds) :

- *User*

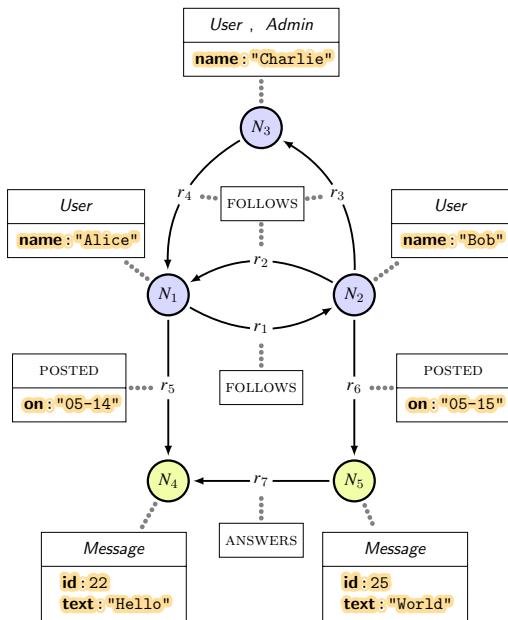
- *Message*

- Types (of relationships) :

- FOLLOWS

- POSTED

- ANSWERS



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

- Labels (de nœuds) :

- *User*
- *Message*

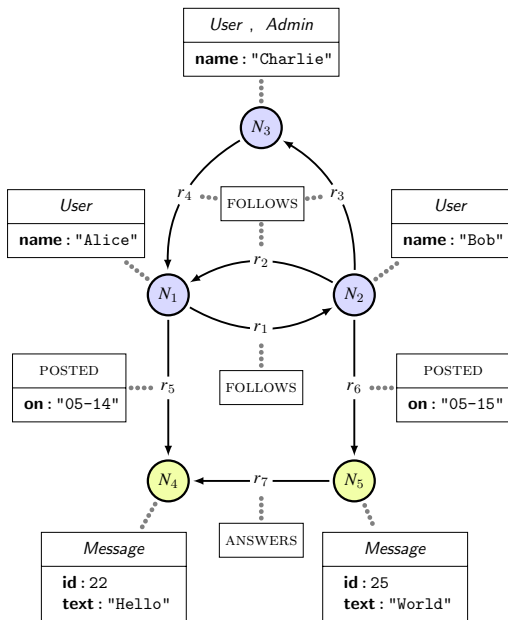
- Types (of relationships) :

- FOLLOWS
- POSTED
- ANSWERS

- Properties

(i.e. Key/Value pairs) :

- name: "Alice"
- id: 22
- text: "Hello"



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

- Relationships :

$r_1, r_2, \dots, r_7$

- Labels (de nœuds) :

- User*
- Message*

- Types (of relationships) :

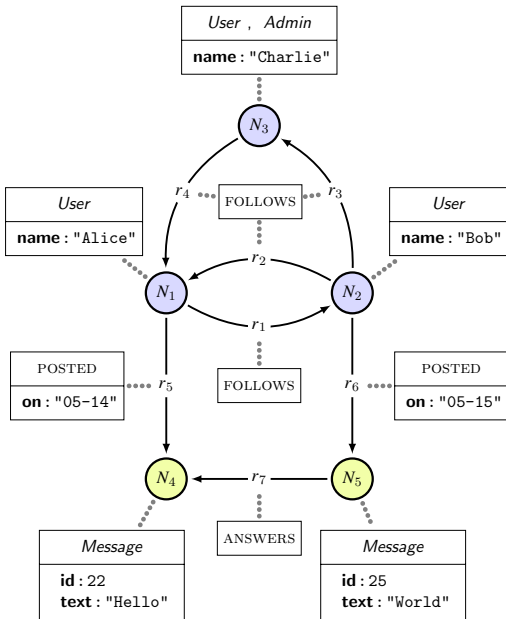
- FOLLOWS
- POSTED
- ANSWERS

- Properties

(i.e. Key/Value pairs) :

- name** : "Alice"
- id** : 22
- text** : "Hello"



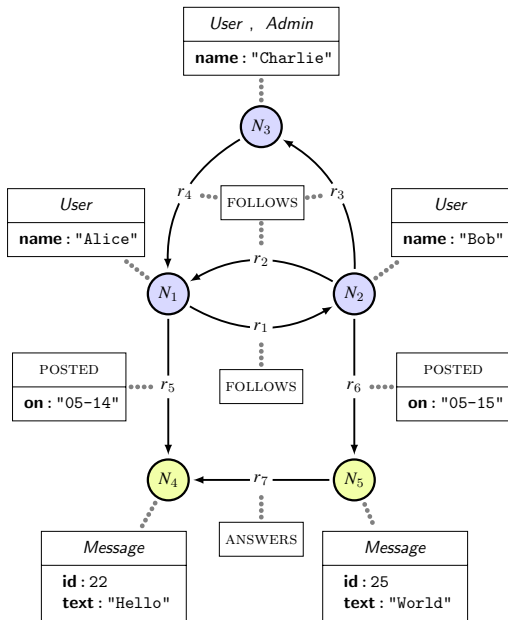


Example of a Cypher query :

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

A Cypher statement :

- is a sequence of *clauses*



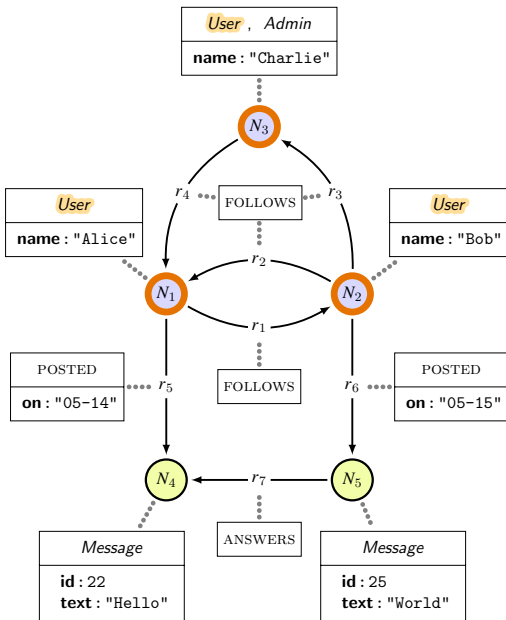
Example of a Cypher query :

```
MATCH (u1)-[p1:POSTE]->(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

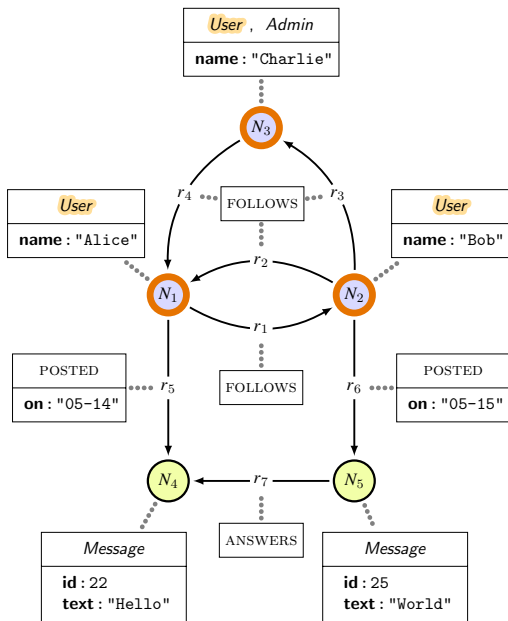
# Matching nodes (1)



Query :

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

# Matching nodes (1)



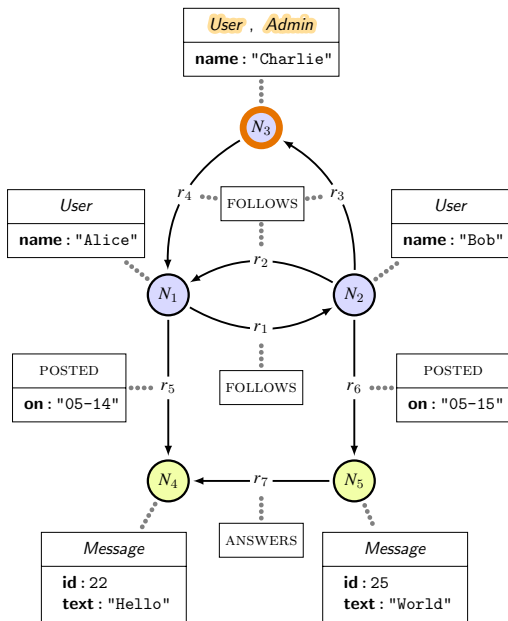
Query :

**MATCH** (u1:User)

Result :

u1  
N<sub>1</sub>  
N<sub>2</sub>  
N<sub>3</sub>

# Matching nodes (2)



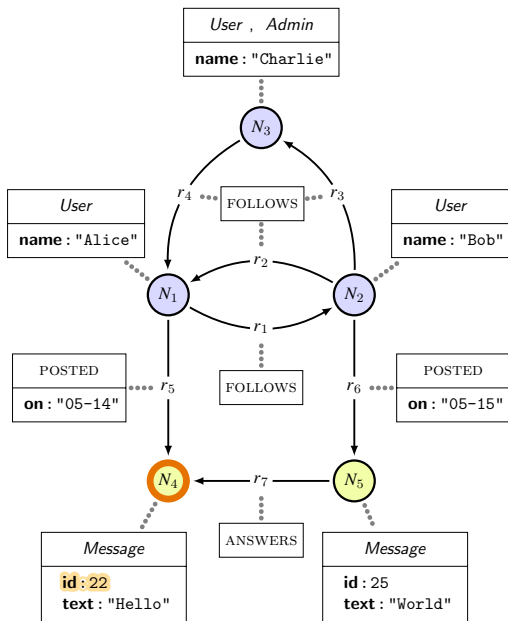
Query :

**MATCH** (u1:User:Admin)

Result :

u1  
N3

# Matching nodes (3)

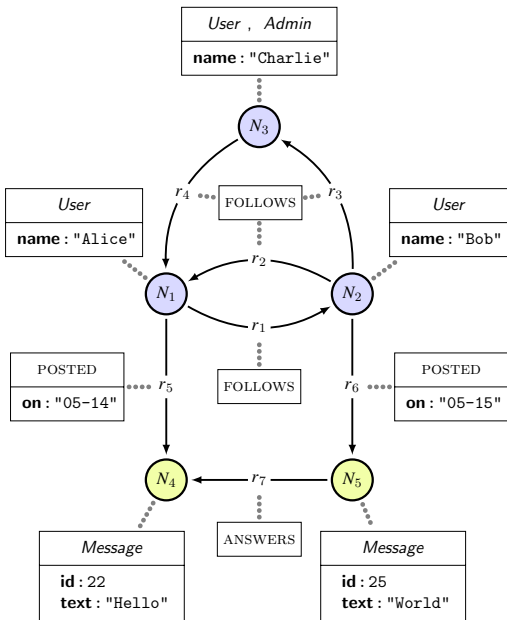


Query :

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

Result :

u1
$N_4$

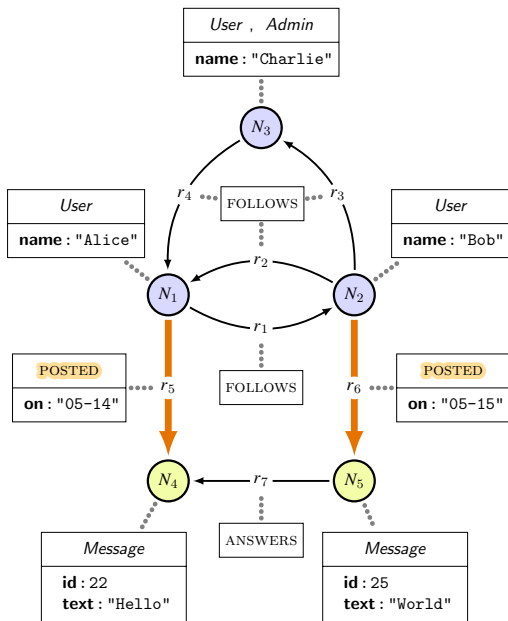


Query :

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

Result :

p1  
r<sub>1</sub>  
r<sub>2</sub>  
r<sub>3</sub>  
r<sub>4</sub>  
r<sub>5</sub>  
r<sub>6</sub>  
r<sub>7</sub>



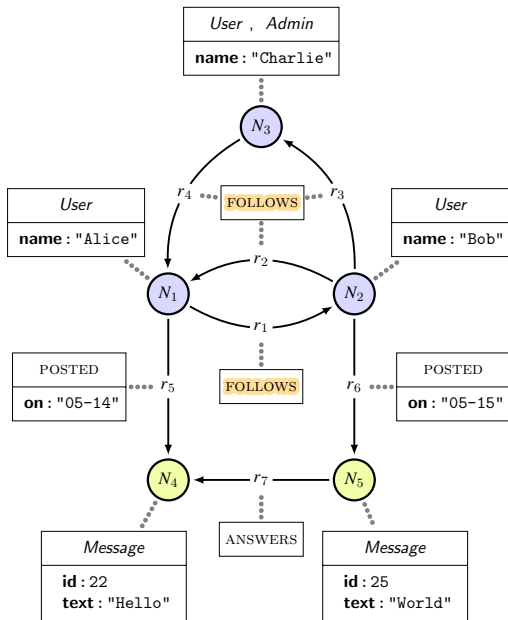
Query :

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

Result :

u1	p1	m1
$N_1$	$r_5$	$N_4$
$N_2$	$r_6$	$N_5$





Query :

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

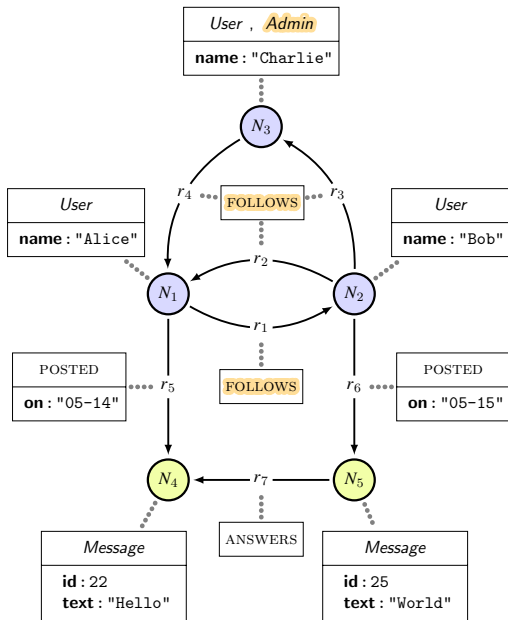
Result :

u1
$N_1$
$N_2$
$N_2$
$N_3$

Bag semantics :

$N_2$  has two outgoing FOLLOWS rel.  $\Rightarrow$  two lines  $N_2$

# Matching relationships (4)



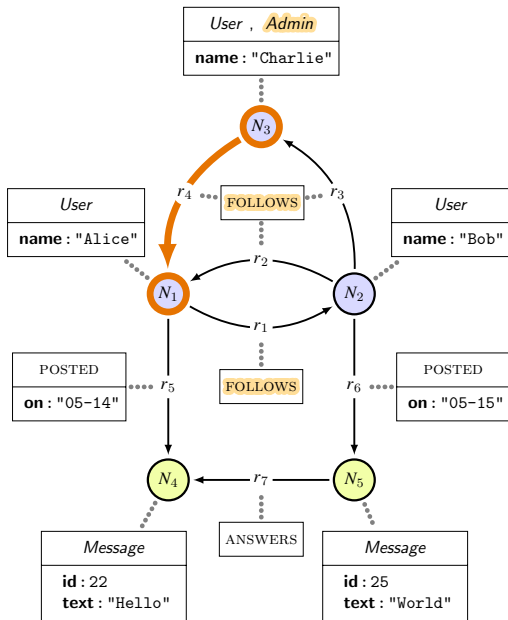
Query :

```
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 relationships (4)

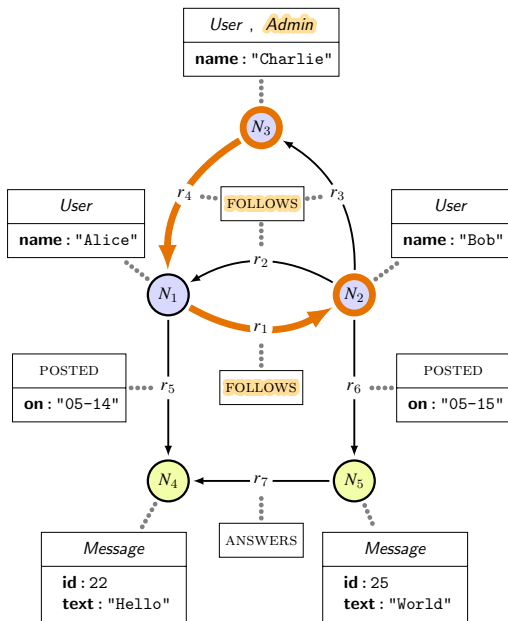


Query :

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

Result :

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



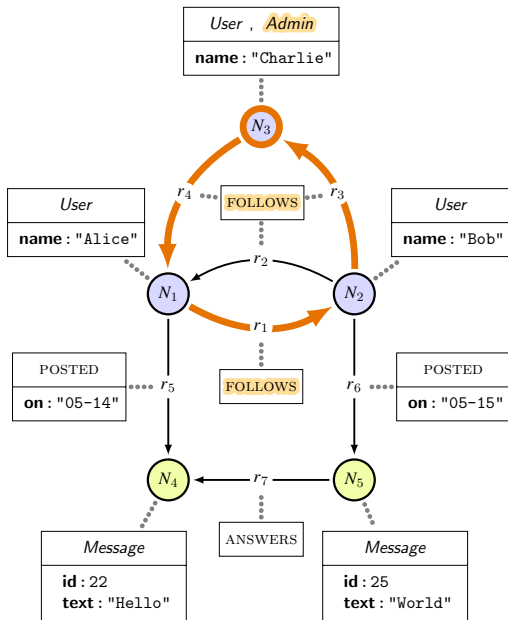
Query :

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

Result :

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

# Matching relationships (4)

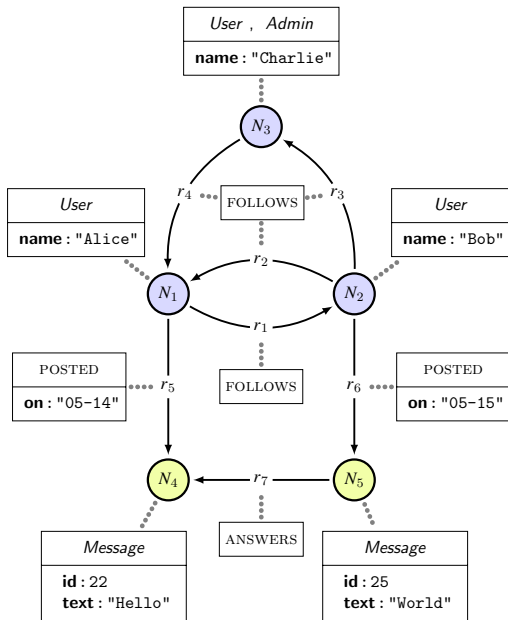


Query :

```
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 :

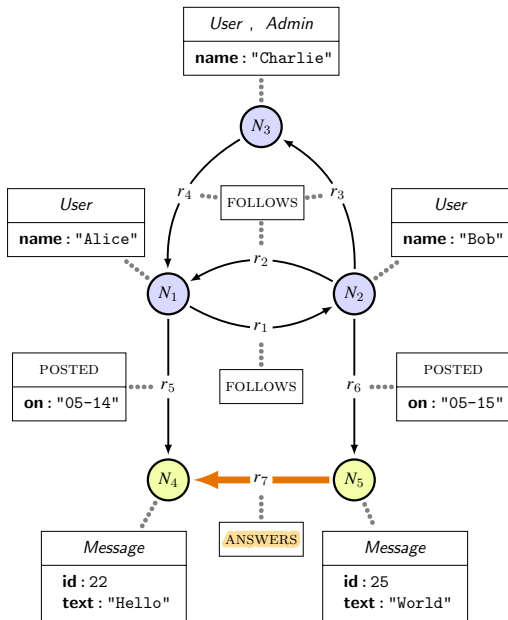
```
MATCH (u1:Admin)
      -[l1:FOLLOWS*]->(m1)
```

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  $\leq 1$  time
- $\Rightarrow$  Finitely many results

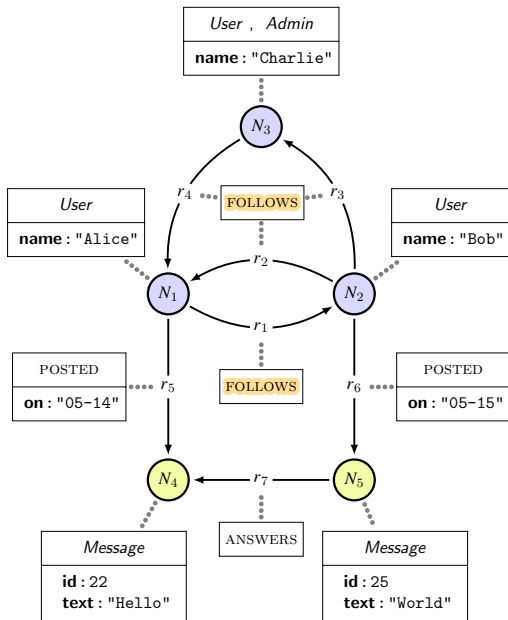


Query :

**MATCH** (m1)-[:ANSWERS]-(m2)

Result :

m1	m2
$N_4$	$N_5$
$N_5$	$N_4$



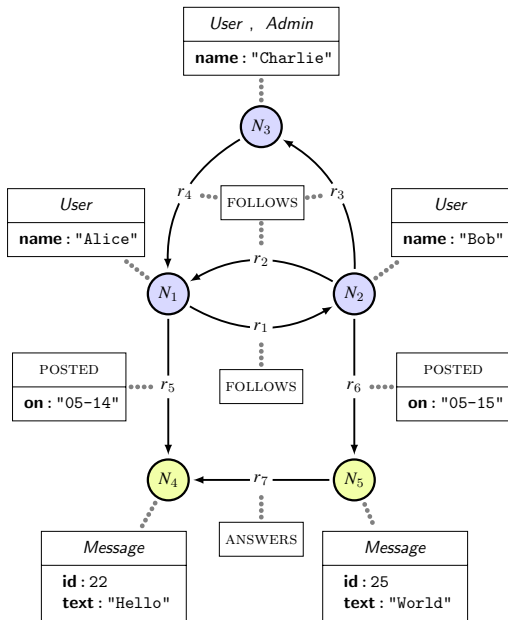
Query :

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

Result :

<u>u1</u>
$N_1$
$N_1$
$N_2$
$N_2$
$N_3$



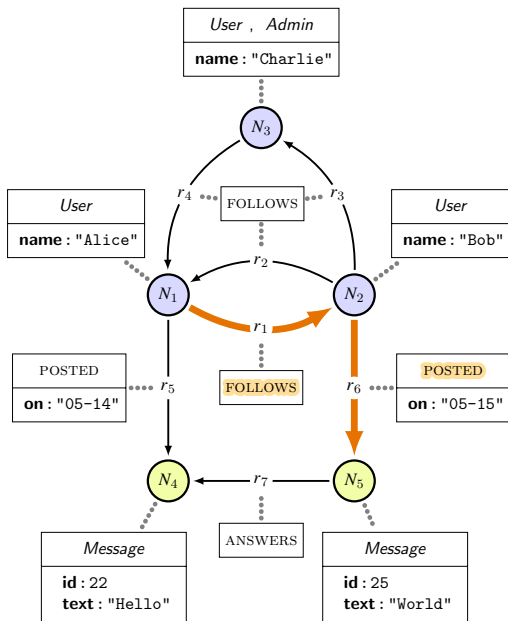


Query :

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

Result :

u1	m1
$N_1$	$N_5$
$N_2$	$N_4$
$N_3$	$N_5$

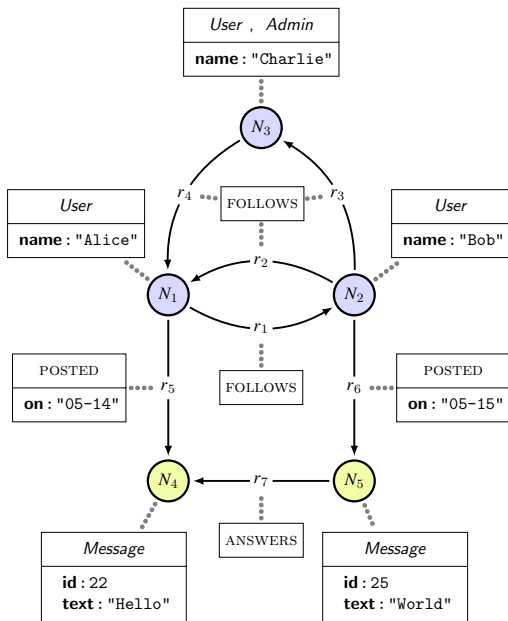


Query :

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

Result :

u1	m1
$N_1$	$N_5$
$N_2$	$N_4$
$N_3$	$N_5$



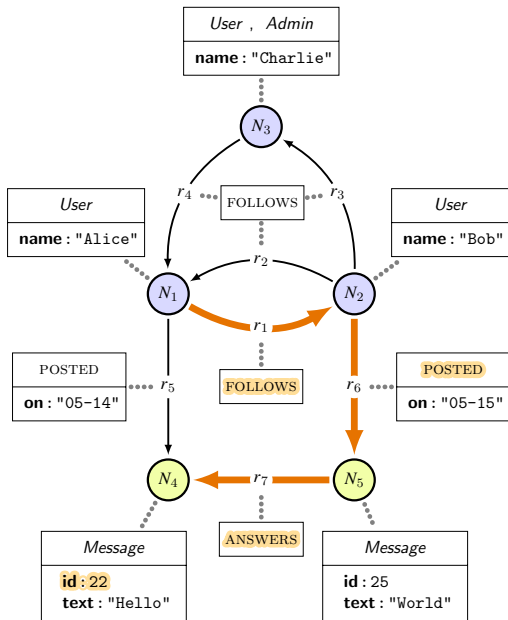
Query :

```

MATCH (u1)-[:FOLLOWS]->()
      -[:POSTED]->(m1)
      -[:ANSWERS]->({id:22})
    
```

Result :

u1	m1
$N_1$	$N_5$



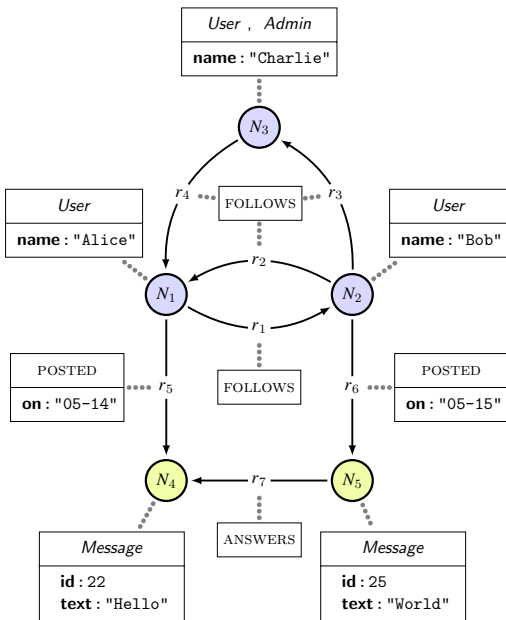
Query :

```

MATCH (u1)-[:FOLLOWS]->()
      -[:POSTED]->(m1)
      -[:ANSWERS]->({id:22})
    
```

Result :

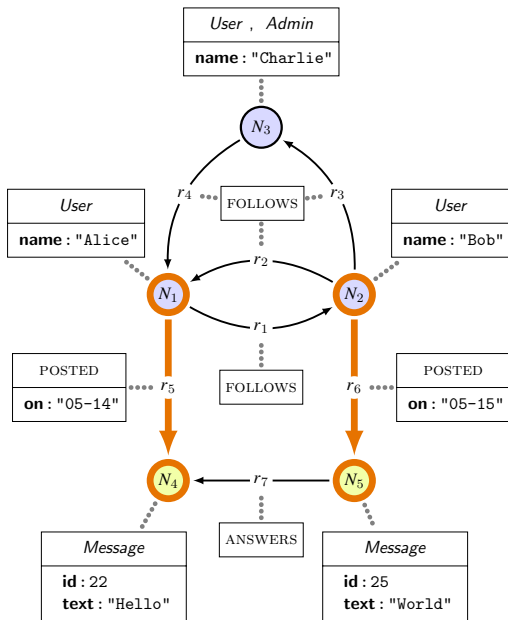
u1	m1
$N_1$	$N_5$



Query :

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

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

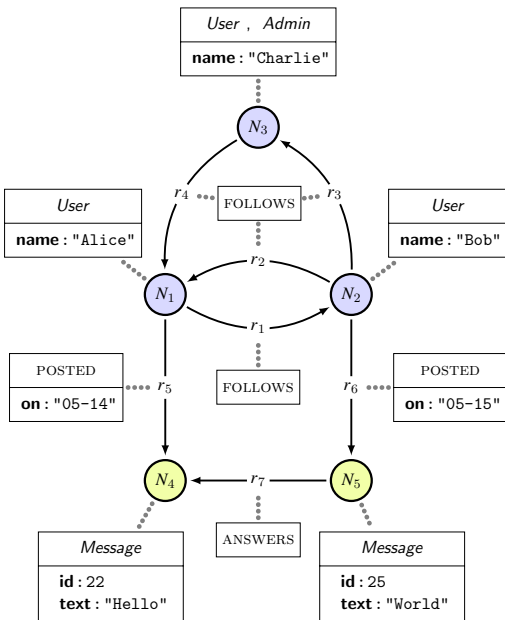


Query :

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

After first MATCH :

u1	m1
$N_1$	$N_4$
$N_2$	$N_5$



Query :

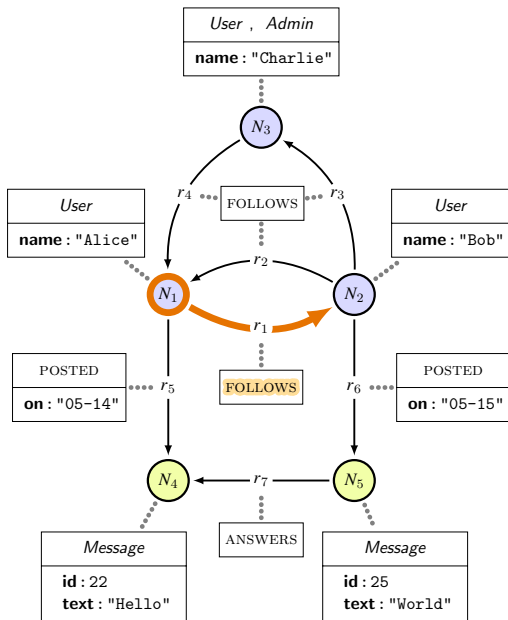
```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
      -[:FOLLOWS]->(u3)
```

After first MATCH :

u1	m1
$N_1$	$N_4$
$N_2$	$N_5$

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)
```

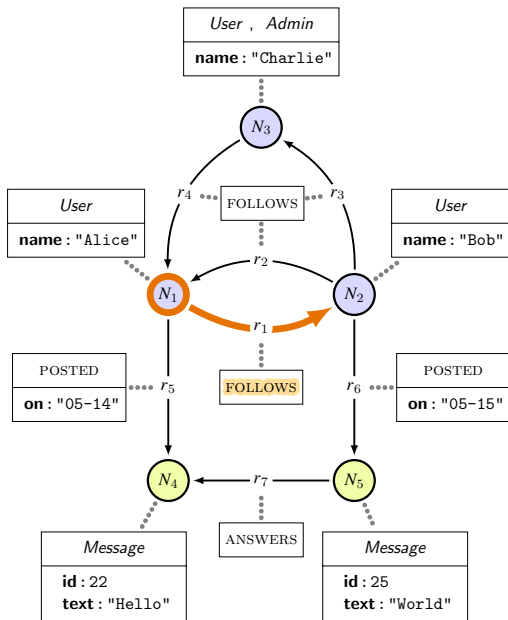
After first MATCH :

u1	m1
$N_1$	$N_4$
$N_2$	$N_5$

After second MATCH :

u1	m1	u2	u3
$N_1$	$N_4$	.	.
$N_2$	$N_5$	.	.





Query :

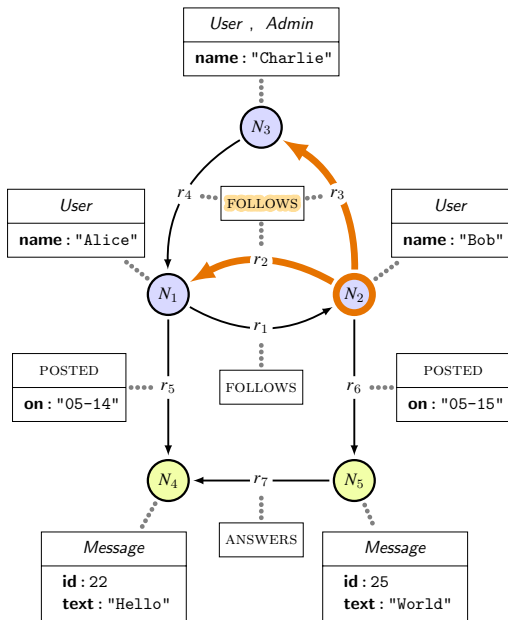
```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
      -[:FOLLOWS]->(u3)
```

After first MATCH :

u1	m1
$N_1$	$N_4$
$N_2$	$N_5$

After second MATCH :

u1	m1	u2	u3
<del><math>N_1</math></del>	<del><math>N_4</math></del>		
$N_2$	$N_5$	.	.



Query :

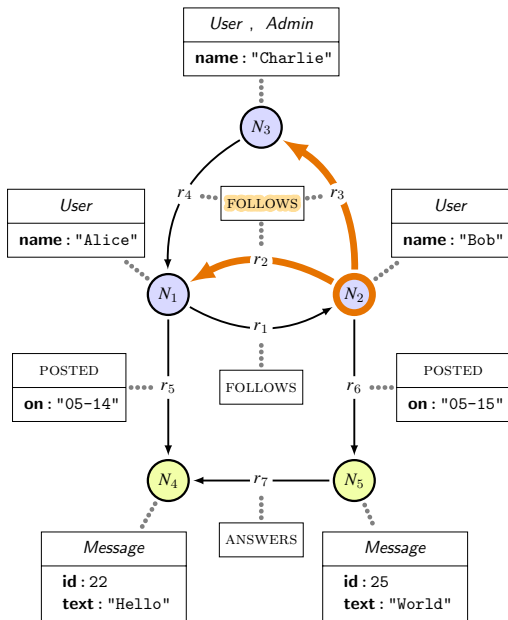
```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
      -[:FOLLOWS]->(u3)
```

After first MATCH :

u1	m1
$N_1$	$N_4$
$N_2$	$N_5$

After second MATCH :

u1	m1	u2	u3
<del><math>N_1</math></del>	<del><math>N_4</math></del>	.	.
$N_2$	$N_5$	.	.



Query :

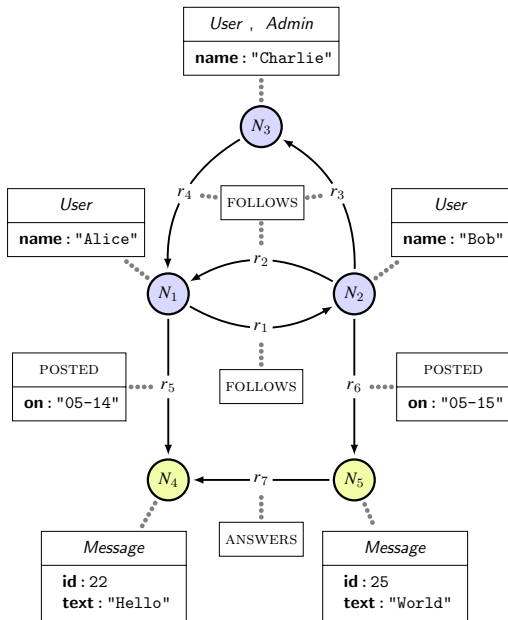
```
MATCH (u1)-[:POSTED]->(m1)
MATCH (u2)<-[:FOLLOWS]-(u1)
      -[:FOLLOWS]->(u3)
```

After first MATCH :

u1	m1
$N_1$	$N_4$
$N_2$	$N_5$

After second MATCH :

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



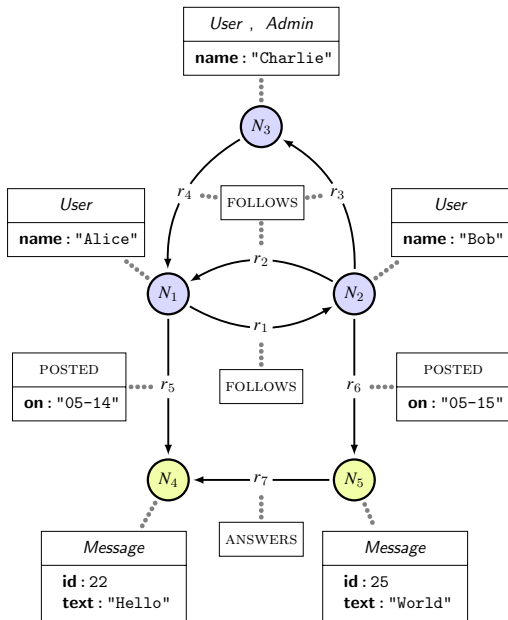
Query :

```
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 :

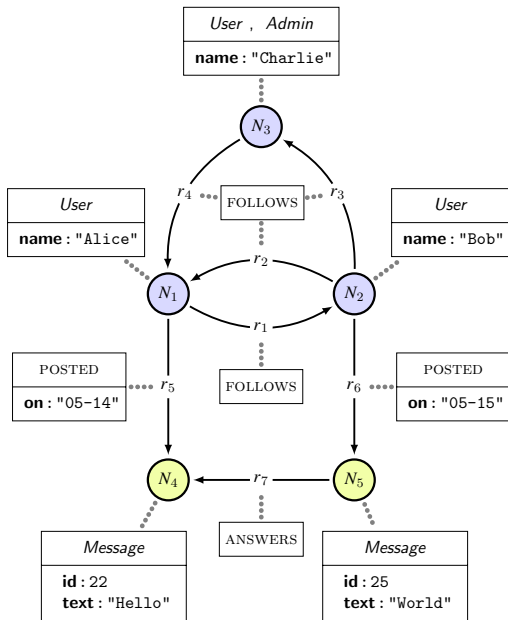
```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

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
$N_1$	$r_5$	
$N_2$	$r_6$	



Query :

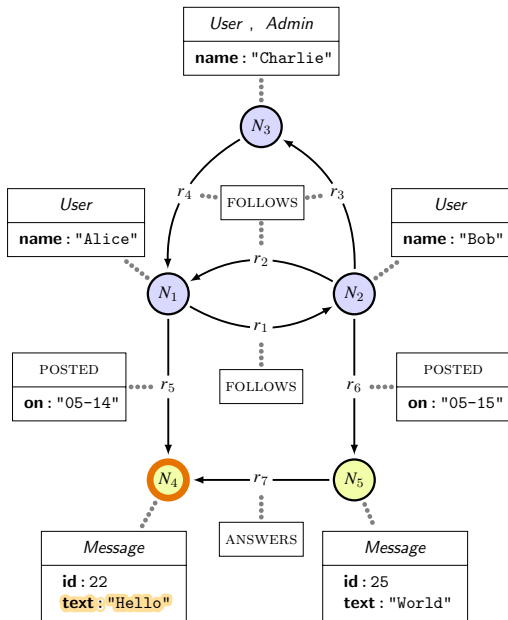
```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

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
$N_1$	$r_5$	
$N_2$	$r_6$	



Query :

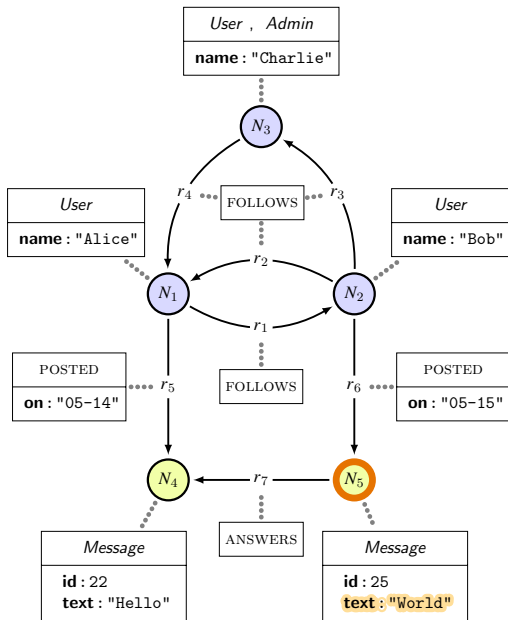
```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

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
```

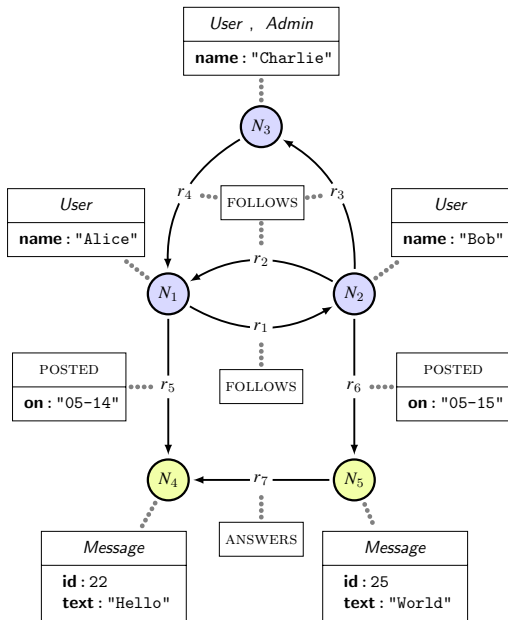
After the MATCH clause :

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"





Query :

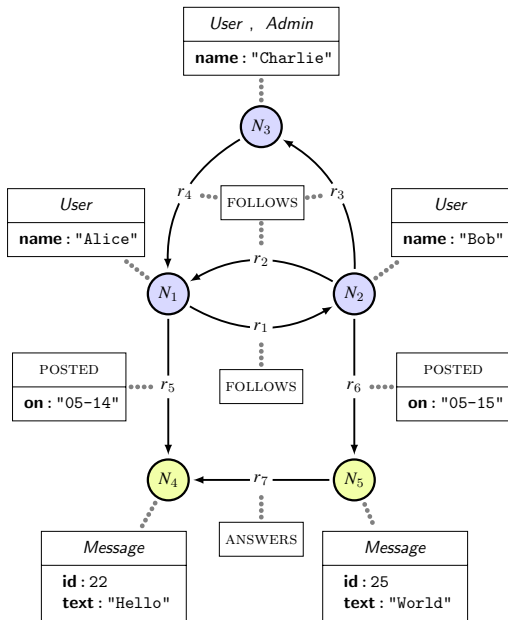
```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
```

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"

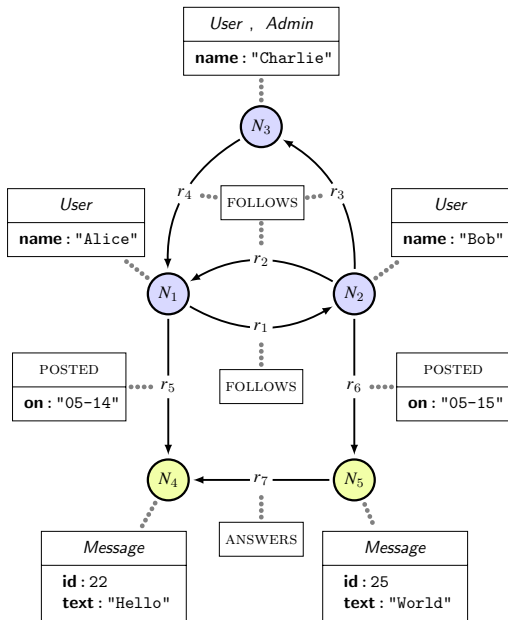


Query :

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

Après le WITH :

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



Query :

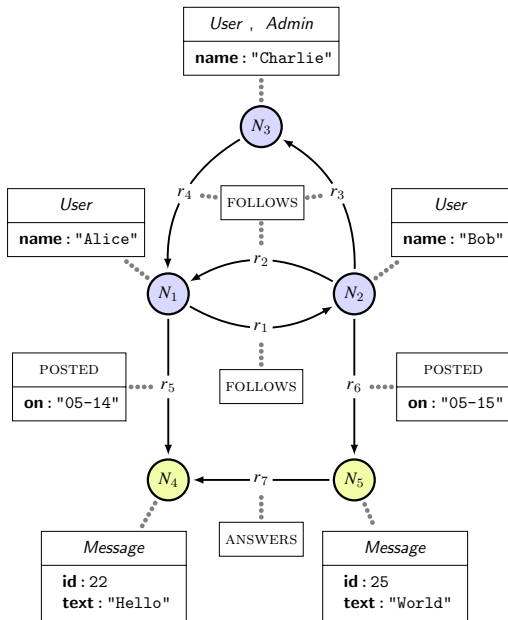
```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
WHERE t1 = "Hello"
```

Après le WITH :

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"
<del><math>N_2</math></del>	<del><math>r_6</math></del>	<del>"World"</del>



Query :

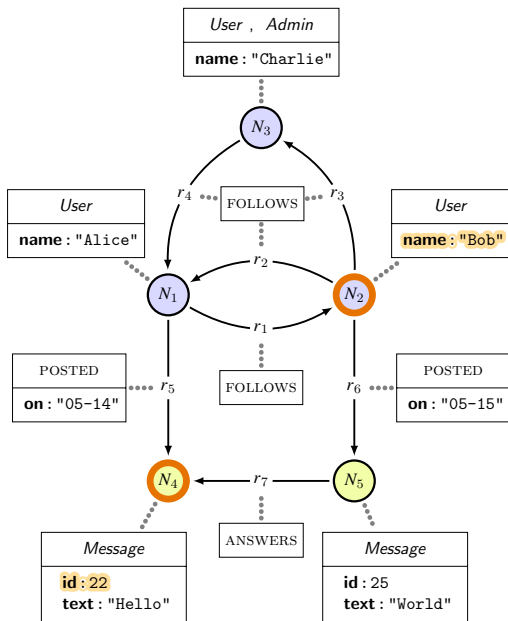
```
MATCH (u1)-[p1:POSTED]->(m1)
WITH u1, p1, m1.text AS t1
WHERE t1 = "Hello"
```

Après le WITH :

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

Final result :

u1	p1	t1
$N_1$	$r_5$	"Hello"

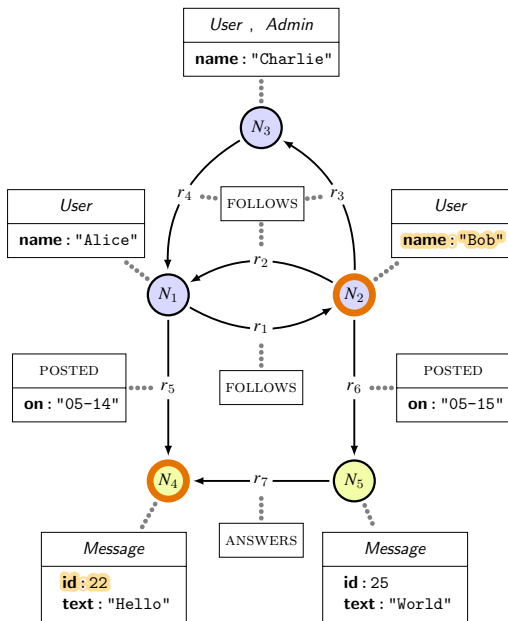


Query :

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

Question :

What does this computes ?



Query :

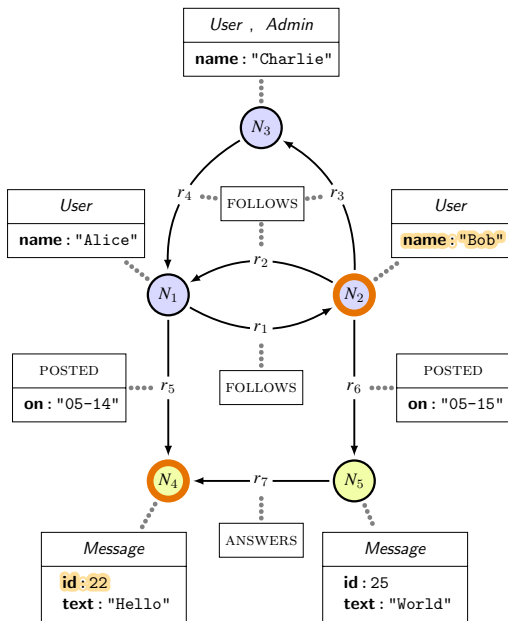
```
MATCH (a{name:"Bob"})
      -[*]->({id:22})
      <-[*]-(a)
```

Question :

What does this computes ?

Answer :

The # of pairs of disjoint paths between  $N_6$  and  $N_2$ .



Query :

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

Question :

What does this computes ?

Answer :

The # of pairs of disjoint paths between  $N_6$  and  $N_2$ .

⇒ Evaluation of one constant  
MATCH is NP-HARD

- 1 Introduction
- 2 Cypher by example
- 3 General principles of the semantics
- 4 Comparison Cypher/UCRPQ
- 5 The MATCH clause
- 6 Conclusion and future work



## Record (table row)

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

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

## Record (table row)

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

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

## Table

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

Example:

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

## Record (table row)

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

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

## Table

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

Example:

<hr/>			<hr/>	
$x$	$y$	=	$x$	$y$
<hr/>			<hr/>	
"Bob"	1		999	"Alice"
"Alice"	999		1	"Bob"
"Bob"	1		1	"Bob"
<hr/>			<hr/>	

$G$ : a graph

## Semantics of expressions

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

## Semantics of clauses

$\llbracket \cdot \rrbracket_G : \text{clause} \mapsto (\text{function: Tables} \rightarrow \text{Tables})$

## Semantics of queries

$\llbracket \cdot \rrbracket_G : \text{query} \mapsto (\text{function: Tables} \rightarrow \text{Tables})$

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

$G$ : a graph

$Q$ : a query

To compute the output of  $Q$

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

$G$ : 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  $\llbracket C_1 \rrbracket_G$ ,  $\llbracket C_2 \rrbracket_G$ ,  $\dots$ ,  $\llbracket C_n \rrbracket_G$

$G$ : 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  $\llbracket C_1 \rrbracket_G$ ,  $\llbracket C_2 \rrbracket_G$ ,  $\dots$ ,  $\llbracket C_n \rrbracket_G$
- Let  $\llbracket Q \rrbracket_G = \llbracket C_n \rrbracket_G \circ \cdots \circ \llbracket C_2 \rrbracket_G \circ \llbracket C_1 \rrbracket_G$

$G$ : 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  $\llbracket C_1 \rrbracket_G, \llbracket C_2 \rrbracket_G, \dots, \llbracket C_n \rrbracket_G$
- Let  $\llbracket Q \rrbracket_G = \llbracket C_n \rrbracket_G \circ \cdots \circ \llbracket C_2 \rrbracket_G \circ \llbracket C_1 \rrbracket_G$
- $\text{output}(G, Q) = \llbracket Q \rrbracket_G(T_{\text{unit}})$

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



$$\blacksquare \llbracket \text{WHERE } e \rrbracket_G(T) = \left\{ u \in T \mid \llbracket e \rrbracket_{G,u} = \text{true} \right\}$$

$$\blacksquare \llbracket \text{MATCH } \bar{\pi} \rrbracket_G(T) = \bigcup_{u \in T} \{u \cdot u' \mid u' \in \text{match}(\bar{\pi}, G, u)\}$$

$$\blacksquare \llbracket \text{MATCH } \bar{\pi} \text{ WHERE } e \rrbracket_G(T) = \llbracket \text{WHERE } e \rrbracket \left( \llbracket \text{MATCH } \bar{\pi} \rrbracket_G(T) \right)$$

$$\blacksquare \llbracket \text{OPTIONAL MATCH } \bar{\pi} \text{ WHERE } e \rrbracket_G(T) \\ = \bigcup_{u \in T} \left\{ \begin{array}{ll} \llbracket \text{MATCH } \bar{\pi} \text{ WHERE } e \rrbracket_G(\{u\}) & \text{if } \llbracket \text{MATCH } \bar{\pi} \text{ WHERE } e \rrbracket_G(\{u\}) \neq \emptyset \\ (u, (\text{free}(u, \bar{\pi}) : \text{null})) & \text{otherwise} \end{array} \right.$$

$$\blacksquare \llbracket \text{OPTIONAL MATCH } \bar{\pi} \rrbracket_G(T) = \llbracket \text{OPTIONAL MATCH } \bar{\pi} \text{ WHERE true} \rrbracket_G(T)$$

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

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

- 1 Introduction
- 2 Cypher by example
- 3 General principles of the semantics
- 4 Comparison Cypher/UCRPQ
- 5 The MATCH clause
- 6 Conclusion and future work

$A$ : alphabet of relationship types

$N$ : nodes in the graph

## Regular Path Query (RPQ)

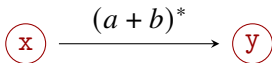
$E$ : a regular expression over  $A$

$x, y$ : two variables

$(x, E, y)$  is an RPQ

$\text{Answer}((x, E, y))$

$$= \left\{ F : \{x, y\} \rightarrow N \mid \exists u \in L(E), F(x) \xrightarrow{u} F(y) \right\}$$



$A$ : alphabet of relationship types

$N$ : nodes in the graph

### Conjunction of RPQs (CRPQ)

$R_1, R_2, \dots, R_n$ : RPQs over  $A$

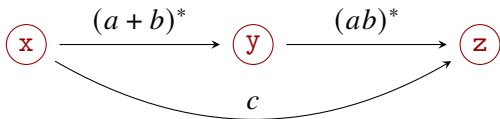
$(R_1 \wedge R_2 \wedge \dots \wedge R_n)$  is a CRPQ

$V_i$ : variables of  $R_i$

$V = \cup_{i=1}^n V_i$ : variables in  $C$

$\text{Answer}(R_1 \wedge R_2 \wedge \dots \wedge R_n)$

$$= \{ F : V \rightarrow N \mid \forall i, F|_{V_i} \in \text{Answer}(Q_i) \}$$



$A$ : alphabet of relationship type

$N$ : nodes in the graph

### Union of CRPQs (UCRPQ)

$C_1, C_2, \dots, C_n$ : CRPQs with the same free variables

$Q = (C_1 \cup C_2 \cup \dots \cup C_n)$  is a CRPQ

$$\text{Answer}(C_1 \cup C_2 \cup \dots \cup C_n) = \bigcup_{i=1}^n \text{Answer}(C_i)$$

## 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  
→  $(ab)^3 + (ab)^2cd + ab(cd)^2 + (cd)^3$

- 1 Introduction
- 2 Cypher by example
- 3 General principles of the semantics
- 4 Comparison Cypher/UCRPQ
- 5 The MATCH clause
- 6 Conclusion and future work



$$\langle \text{node\_pattern} \rangle ::= ( \langle \text{name} \rangle? \langle \text{labels} \rangle? \langle \text{properties} \rangle? )$$

## Examples

- |                                     |   |
|-------------------------------------|---|
| <code>()</code>                     | Any node  |
| <code>(a)</code>                    | Any node; verify/map id to column <i>a</i>  |
| <code>(:User)</code>                | Nodes bearing label <i>User</i>   |
| <code>{name:"Alice"}</code>         | Nodes with the property <b>name</b> set to "Elin"   |
| <code>(a:User{name:"Alice"})</code> | Nodes of type <i>User</i> and with a property <b>name</b> set to "Elin"; verify/map id to column <i>a</i> . |

`<pattern> ::= [ <name>? <types>? <iter>? <properties>? ]`

## Example

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

```
⟨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\*]-(())

## 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

<code>(a) &lt;- [] -&gt; () - [{id:220}] -&gt; ()</code>	Rigid
<code>() - [] -&gt; () - [*42..42] -&gt; (:User)</code>	Rigid
<code>() - [*2..3] -&gt; (b)    and    (a) &lt;- [] - (b) - [*] - (a)</code>	Flexible

## Definition

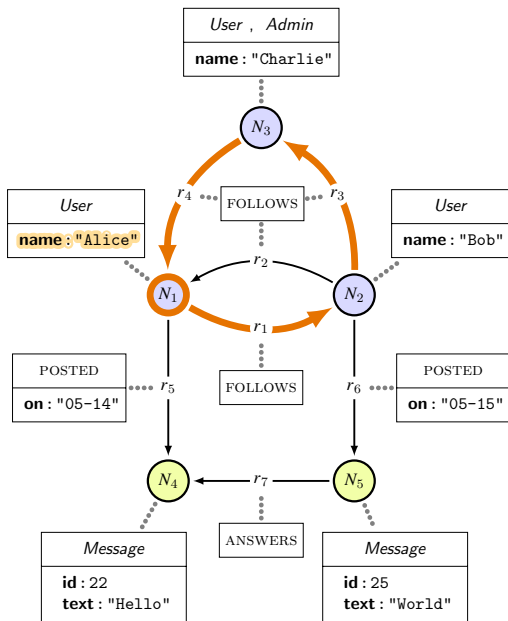
A path pattern is *Rigid* if its length is fixed  
 (i.e. all `<iter>` are absent or derive to `*i..i` for some  $i \in \mathbb{N}$ )

## Examples

<code>(a) &lt;- [] -&gt; () - [{id:220}] -&gt; ()</code>	Rigid
<code>() - [] -&gt; () - [*42..42] -&gt; (:User)</code>	Rigid
<code>() - [*2..3] -&gt; (b)    and    (a) &lt;- [] - (b) - [*] - (a)</code>	Flexible

## Property

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

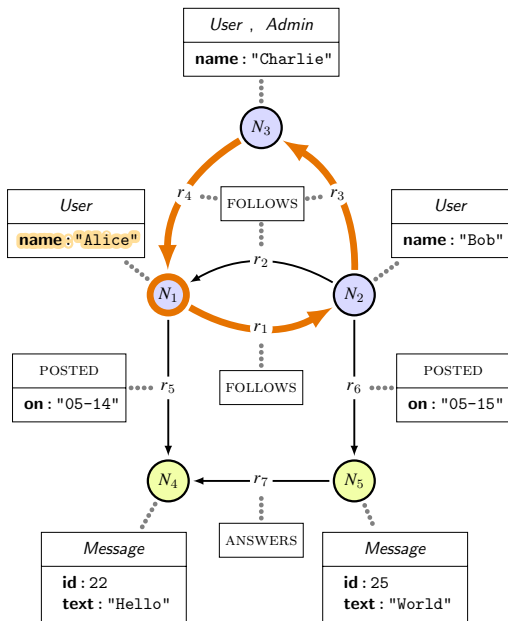


Query :

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

The path

$N_1 \xrightarrow{r_1} N_2 \xrightarrow{r_3} N_3 \xrightarrow{r_4} N_1$   
satisfies the pattern.



Query :

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

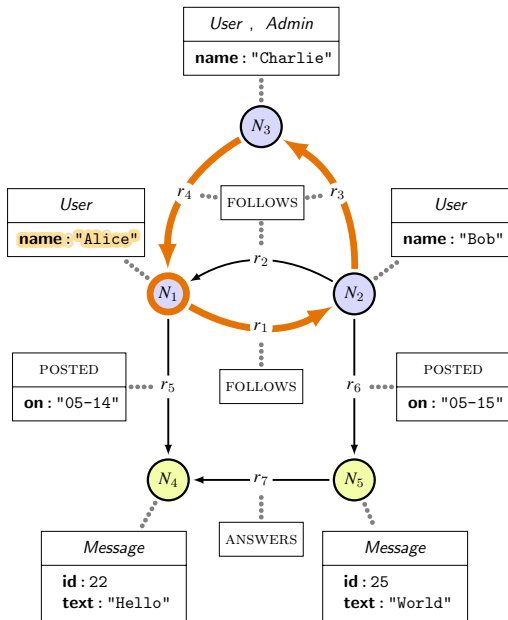
The path

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

satisfies the pattern.

Variables are uniquely set

x	mid
$N_1$	$N_2$



Query :

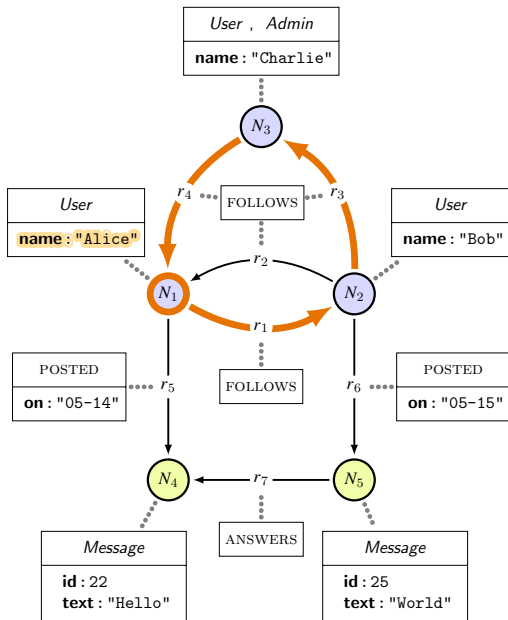
```
MATCH (x {name:"Alice"})
      -[:FOLLOWS*1..2]->(mid)
      -[:FOLLOWS*1..2]->(x)
```

The path

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

satisfies this second pattern.





Query :

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

The path

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

satisfies this second pattern.

Two possible assignments

x	mid
$N_1$	$N_2$
$N_1$	$N_3$

$u$ : a record

$p$ : a path

$\pi$ : a rigid path-pattern

$n$ : length of  $\pi$

## 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  $\pi$   
(variable, labels, properties)
- $\forall i < p$ , the  $i$ -th relationship of  $p$  satisfies the  $i$ -th relationship pattern of  $\pi$   
(variable, types, properties)

$\text{rigid}(\pi) = \text{set of all rigid patterns subsumed by } \pi$

### Examples

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

$\text{rigid}(\pi) = \text{set of all rigid patterns subsumed by } \pi$

### Examples

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

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

$\text{rigid}(\pi) = \text{set of all rigid patterns subsumed by } \pi$

## Examples

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

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

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

$\text{rigid}(\pi)$  = set of all rigid patterns subsumed by  $\pi$

### Examples

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

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

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

WLOG:  $\text{rigid}(\pi)$  contains no pattern longer than the graph size

## Semantics of MATCH

$$\llbracket \text{MATCH } \pi \rrbracket_G : T \mapsto \bigcup_{u \in T} \text{expand}_{G, \pi}(u)$$

## Semantics of MATCH

$$\llbracket \text{MATCH } \pi \rrbracket_G : T \mapsto \bigcup_{u \in T} \text{expand}_{G, \pi}(u)$$

## Table-row expansion

$\pi$ : a path-pattern

$G$ : a graph

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

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



- 1 Introduction
- 2 Cypher by example
- 3 General principles of the semantics
- 4 Comparison Cypher/UCRPQ
- 5 The MATCH clause
- 6 Conclusion and future work

## Cypher

- Relatively recent query-language for graph databases
- More and more used in industry
- Community-led development since 2015 (openCypher project)

## Our goal

Full denotational semantic for Cypher.

## Status

- |                               |                  |
|-------------------------------|------------------|
| ■ Core of the language        | Done (SIGMOD'18) |
| ■ Remainder of read-only part | In finalisation  |
| ■ Updates                     | Going well       |

## Features

- **CREATE** / **DELETE** clause: add/remove nodes or relationships.
- **SET** clause: change labels and properties.
- **MERGE** clause: “match else create”.

## Features

- **CREATE** / **DELETE** clause: add/remove nodes or relationships.
- **SET** clause: change labels and properties.
- **MERGE** clause: “match else create”.
- Mixture of read-only and update clauses  
*E.g.*, **MATCH** (a:Student) **CREATE** (a)-[]->(:Project)
- Query evaluation is ACID ...supposedly  
(Atomicity, Consistency, Isolation, Durability)

## Features

- **CREATE** / **DELETE** clause: add/remove nodes or relationships.
- **SET** clause: change labels and properties.
- **MERGE** clause: “match else create”.
- Mixture of read-only and update clauses  
E.g., **MATCH** (a:**Student**) **CREATE** (a)-[]->(:**Project**)
- Query evaluation is ACID ...supposedly  
(Atomicity, Consistency, Isolation, Durability)

## A few hiccups

- Inconsistent syntax accross the update clauses
- Graph in inconsistent state in the middle of query
- Row-order dependence of **MERGE** and **SET**
-

- 1 Introduction
- 2 Cypher by example
  - ▶ Property graphs
  - ▶ Pattern matching
  - ▶ Chaining clauses
  - ▶ Column manipulation
  - ▶ Line filtering
- 3 General principles of the semantics
  - ▶ Modelling tables
  - ▶ Principle of the semantics
- 4 Comparison Cypher/UCRPQ
- 5 The MATCH clause
  - ▶ Path patterns
  - ▶ Rigid path patterns
  - ▶ General case
- 6 Conclusion and future work
- 7 Remainder of the core fragment
  - ▶ Caveat on the MATCH clause
  - ▶ The OPTIONAL MATCH clause
  - ▶ The WHERE subclause
  - ▶ The WITH subclause
  - ▶ The UNWIND clause
  - ▶ The RETURN clause
  - ▶ UNION of queries
- 8 Ongoing additions
  - ▶ Aggregation
  - ▶ Line order
- 9 Oddities and inconsistencies

Queries are essentially sequences of clauses

```
⟨query⟩ ::=  ⟨query⟩ UNION [ALL]? ⟨query⟩  
           |  [⟨clause⟩]* ⟨return⟩
```

A clause is a main statement followed by subclauses

```
⟨clause⟩ ::=  ⟨match⟩ [⟨where⟩]?  
            |  ⟨with⟩ [⟨where⟩]?  
            |  ⟨unwind⟩
```

## 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.



# Caveat on the MATCH clause (2)

## Semantics of MATCH

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

$G$ : a graph

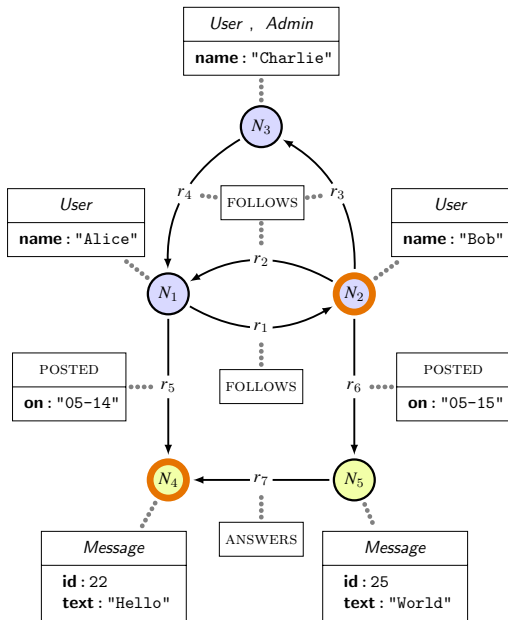
$\text{rigid}(\bar{\pi}) = \text{rigid}(\pi_1) \times \text{rigid}(\pi_2) \times \dots \times \text{rigid}(\pi_n)$

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

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

$$\llbracket \text{MATCH } \bar{\pi} \rrbracket_G : T \mapsto \bigcup_{u \in T} \text{expand}_{G, \bar{\pi}}(u)$$

# OPTIONAL MATCH – Example

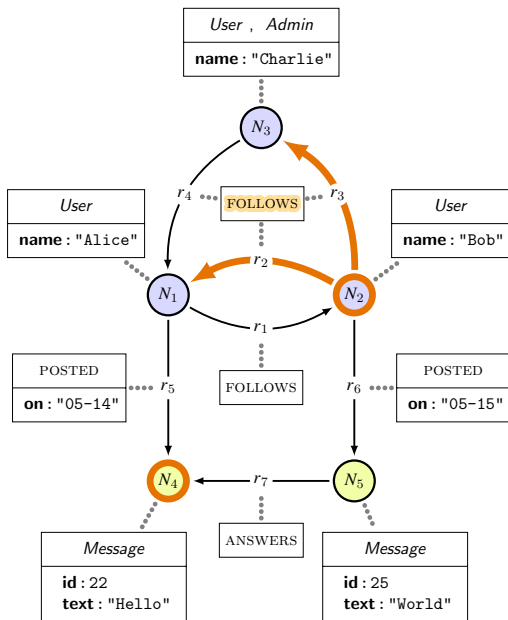

$$\frac{x}{\frac{N_2}{N_4}}$$


Query :

OPTIONAL MATCH

$(x) - [y : \text{FOLLOWS}] \rightarrow ()$

# OPTIONAL MATCH – Example

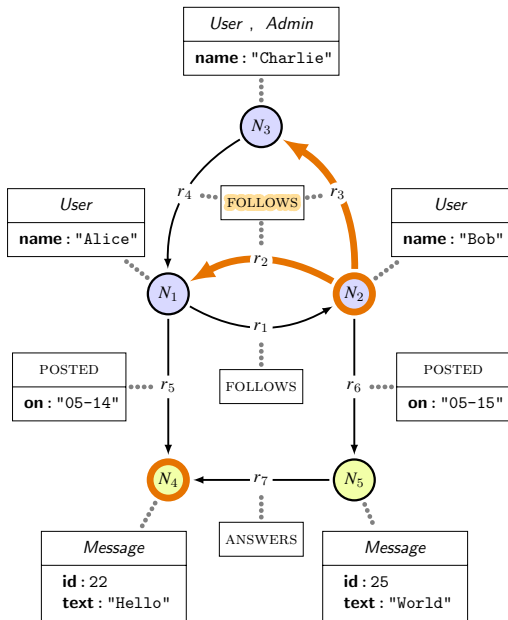

$$\frac{x}{N_2}$$
$$\frac{N_4}{N_4}$$


Query :

OPTIONAL MATCH

$(x) - [y : \text{FOLLOWS}] -> ()$

# OPTIONAL MATCH – Example



x
$N_2$
$N_4$



Query :

OPTIONAL MATCH

$(x) - [y : \text{FOLLOWS}] -> ()$



x	y
$N_2$	$r_2$
$N_2$	$r_3$
$N_4$	null

$u$ : a record

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

$\text{cimpl}(u, \bar{\pi})$  is the record:

- $\text{dom}(\text{cimpl}(u, \bar{\pi})) = \text{dom}(u) \cup \text{var}(\bar{\pi})$
- $a \mapsto \begin{cases} u(a) & \text{if } a \in \text{dom}(u) \\ \text{null} & \text{otherwise} \end{cases}$

$$\llbracket \text{OPTIONAL MATCH } \bar{\pi} \rrbracket_G (T) = \bigsqcup_{u \in T} \begin{cases} T_u & \text{if } T_u \neq \emptyset \\ \{\text{cimpl}(u)\} & \text{otherwise} \end{cases}$$

where  $T_u = \llbracket \text{MATCH } \bar{\pi} \rrbracket_G (\{u\})$

## Syntax

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

## Semantics

$$\llbracket \text{WHERE } e \rrbracket : T \mapsto \bigcup_{u \in T} \begin{cases} \{u\} & \text{if } \llbracket e \rrbracket_{G,u} = \text{true} \\ \emptyset & \text{otherwise} \end{cases}$$

## Combined semantics

- $\llbracket \text{MATCH } \bar{\pi} \text{ WHERE } e \rrbracket_G (T) = \llbracket \text{WHERE } e \rrbracket (\llbracket \text{MATCH } \bar{\pi} \rrbracket_G (T))$
- $\llbracket \text{WITH } \bar{\pi} \text{ WHERE } e \rrbracket_G (T) = \llbracket \text{WHERE } e \rrbracket (\llbracket \text{WITH } \bar{\pi} \rrbracket_G (T))$
- $\llbracket \text{OPTIONAL MATCH } \bar{\pi} \text{ WHERE } e \rrbracket_G (T)$   
$$= \bigcup_{u \in T} \begin{cases} T_u & \text{if } T_u \neq \emptyset \\ \{\text{cml}(u)\} & \text{otherwise} \end{cases}$$

where  $T_u = \llbracket \text{MATCH } \bar{\pi} \text{ WHERE } e \rrbracket_G (\{u\})$   
(and  $\text{cml}(u, \bar{\pi})$  is the completion of  $u$  w.r.t. variables of  $\bar{\pi}$ )

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

| WITH  $\langle \text{expr} \rangle$  [AS  $\langle \text{name} \rangle$ ]? , ... ,  $\langle \text{expr} \rangle$  [AS  $\langle \text{name} \rangle$ ]?

| WITH \* ,  $\langle \text{expr} \rangle$  [AS  $\langle \text{name} \rangle$ ]? , ... ,  $\langle \text{expr} \rangle$  [AS  $\langle \text{name} \rangle$ ]?

- 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”.



## Example

<i>a</i>	<i>b</i>	<i>c</i>
0	1	2
1	1	1
2	1	0



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



<i>a</i>	<i>b</i>	<i>c</i>	'a+b'	order
0	1	2	1	true
1	1	1	2	false
2	1	0	3	false

## Semantics

- $\llbracket \text{WITH } * \rrbracket_G(T) = T$
- $$\llbracket \text{WITH } e_1 \text{ AS } a_1, \dots, e_m \text{ AS } a_m \rrbracket_G(T) = \bigcup_{u \in T} \left\{ (a_1 : \llbracket e_1 \rrbracket_{G,u}, \dots, a_m : \llbracket e_m \rrbracket_{G,u}) \right\}$$
- $$\llbracket \text{WITH } e_1 [\text{AS } a_1]?, \dots, e_m [\text{AS } a_m]? \rrbracket_G(T) = \llbracket \text{WITH } e_1 \text{ AS } a'_1, \dots, e_m \text{ AS } a'_m \rrbracket_G(T)$$

where  $a'_i$  equals  $a_i$  if it is given, or arbitrary otherwise.

- $\llbracket \text{WITH } *, \dots \rrbracket_G(T) = \llbracket \text{WITH } b_1 \text{ AS } b_1, \dots, b_q \text{ AS } b_q, \dots \rrbracket_G(T)$

where  $b_1, \dots, b_q$  are the column names of  $T$

$\langle \text{unwind} \rangle ::= \text{UNWIND } \langle \text{expr} \rangle \text{ AS } \langle \text{name} \rangle$

- 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.

## Example

<i>n</i>	<i>list</i>
0	["Hello", "World"]
1	["singleton"]
2	"not_a_list"



UNWIND list AS x



<i>n</i>	<i>list</i>	x
0	["Hello", "World"]	"Hello"
0	["Hello", "World"]	"World"
1	["singleton"]	"singleton"
2	"not_a_list"	"not_a_list"

$$\llbracket \text{UNWIND } e \text{ AS } a \rrbracket_G(T) = \bigcup_{u \in T} \bigcup_{v \in E_u} \{(u, a : v)\},$$

$$\text{where } E_u = \begin{cases} \biguplus_{0 \leq i < m} \{v_i\} & \text{if } \llbracket e \rrbracket_{G,u} = \text{list}(v_1, \dots, v_m) \\ \{\} & \text{if } \llbracket e \rrbracket_{G,u} = \text{list}() \\ \{\llbracket e \rrbracket_{G,u}\} & \text{otherwise} \end{cases}$$

RETURN and WITH clauses have the exact same semantics.

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.

## Syntax

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

- 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.



## Syntax

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

- 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

- $\llbracket Q_1 \text{ UNION ALL } Q_2 \rrbracket_G(T) = \llbracket Q_1 \rrbracket_G(T) \cup \llbracket Q_2 \rrbracket_G(T)$
- $\llbracket Q_1 \text{ UNION } Q_2 \rrbracket_G(T) = \text{distinct} \left( \llbracket Q_1 \rrbracket_G(T) \cup \llbracket Q_2 \rrbracket_G(T) \right)$

## Principle

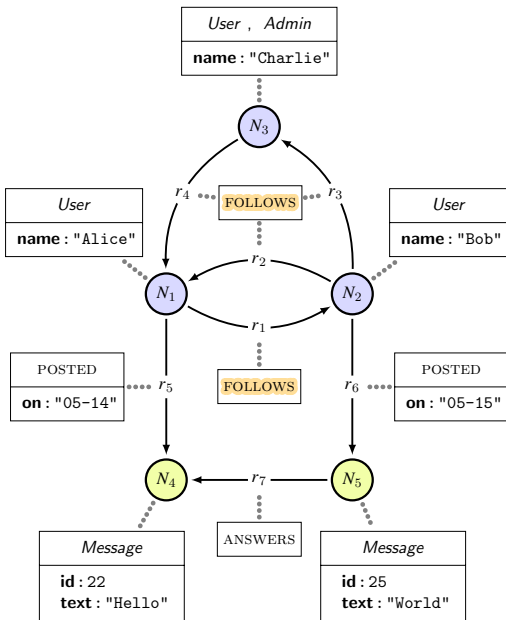
- Group lines according to some criteria
- Some column of the output table contains aggregated values.

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



Query :

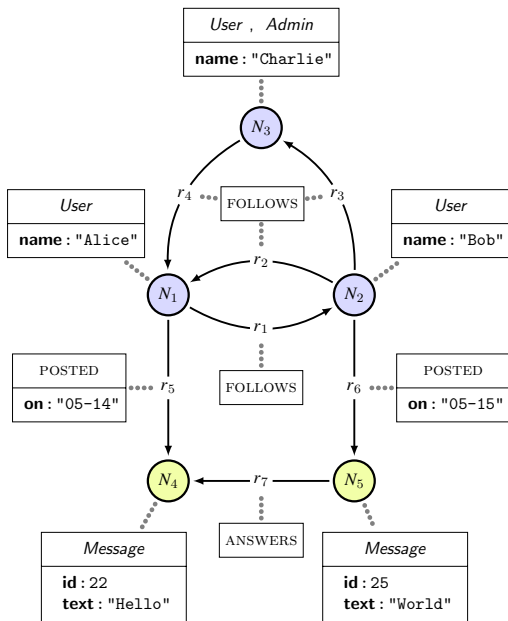
```
MATCH (a)-[r:FOLLOWS]->( )
WITH a.name AS b, count(r) AS c
```

Aggregate

- over expression a.name
- and count them

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

# Aggregation (3) — Example



Query :

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

d
2

## 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  $\langle \text{expr} \rangle$

$\max(e_3)$  is an  $\langle \text{aggexpr} \rangle$

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

- We partition the input table as subtables  $T_1, \dots, T_k$
- 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  $\langle \text{expr} \rangle$

$\max(e_3)$  is an  $\langle \text{aggexpr} \rangle$

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

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

## 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  $\langle \text{expr} \rangle$

$\max(e_3)$  is an  $\langle \text{aggexpr} \rangle$

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

- We partition the input table as subtables  $T_1, \dots, T_k$
- 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:

## 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  $\langle \text{expr} \rangle$

$\max(e_3)$  is an  $\langle \text{aggexpr} \rangle$

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

- We partition the input table as subtables  $T_1, \dots, T_k$
- 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  $V_i = \left\{ \llbracket e_3 \rrbracket_{G,v} \mid v \in T_i \right\}$  and let  $u \in T_i$ .



## 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  $\langle \text{expr} \rangle$

$\max(e_3)$  is an  $\langle \text{aggexpr} \rangle$

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

- We partition the input table as subtables  $T_1, \dots, T_k$
- 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  $V_i = \left\{ \llbracket e_3 \rrbracket_{G,v} \mid v \in T_i \right\}$  and let  $u \in T_i$ .
- The  $i$ -th line is  $\left( a_1 : \llbracket e_1 \rrbracket_{G,u}, a_2 : \llbracket e_2 \rrbracket_{G,u}, a_3 : \max(V_i) \right)$

## 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

## 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

## 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

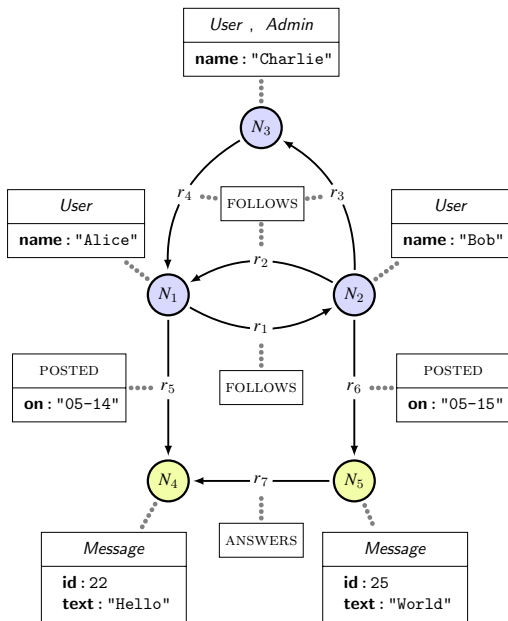
## Up to now

- 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



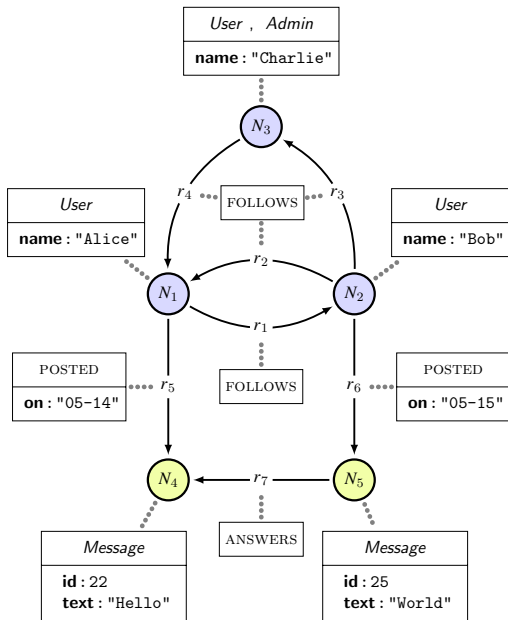
("Alice" < "Bob" < "Charlie")

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"

```
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
```