Formal semantics of the query-language Cypher

Victor Marsault[1]

joint work with

Nadime Francis[2]
Alastair Green[3]
Paolo Guagliardo[1]
Leonid Libkin[1]
Tobias Lindaaker[3]
Stefan Plantikow[3]
Mats Rydberg[3]
Petra Selmer[3]
Martin Schuster[1]
Andrés Taylor[3]

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

Séminaire Automate
IRIF, Paris
2018-04-06

Outline

Representing data as graphs

Most of database use the relational model

- Relational algebra in theory
- The language SQL in practice

Representing data as graphs

Most of database 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

Graph databases are becoming more widespread

Data model

Property graphs, RDF

Query languages

Cypher, Gremlin, PGQL, SparQL

Engines

JanusGraph, Jena, Neo4j, Virtuoso

Domain

Fraud detection, Investigative journalism

Graph databases are becoming more widespread

Data model

Property graphs, RDF

Query languages

Cypher, Gremlin, PGQL, SparQL

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 $\rightarrow$ 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 $\rightarrow$ 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 data graphs?)
  - Community-led evolution
  - Complete specification

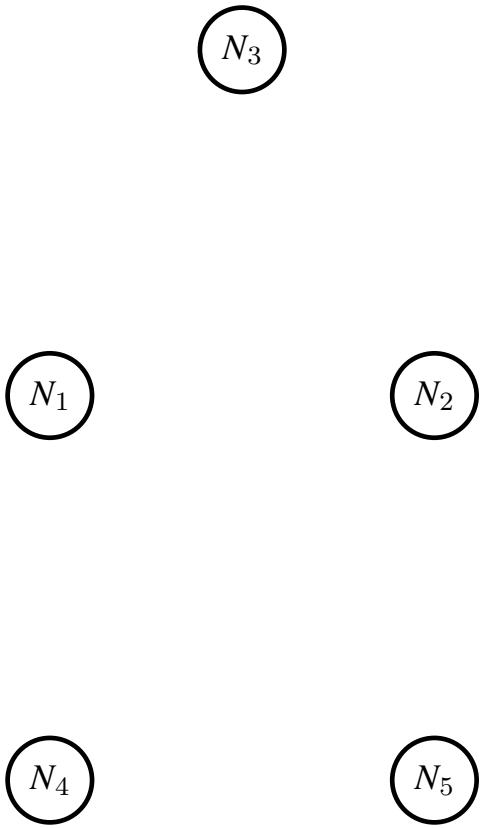
Our goal

Full denotational semantics for the language Cypher.

- Industrial partnership Neo Technology/University of Edinburgh
- Reverse engineering and formalisation from Neo4j
- What I present here: Semantics of the "core fragment" of Cypher

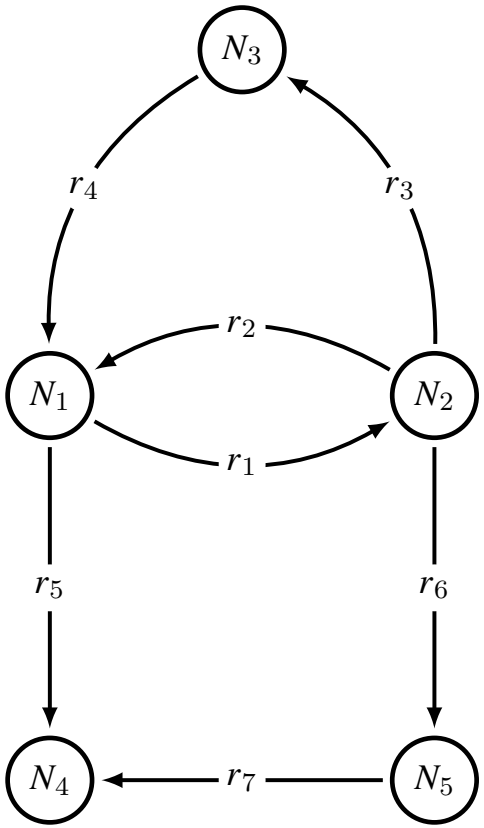
Outline

Property graph



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

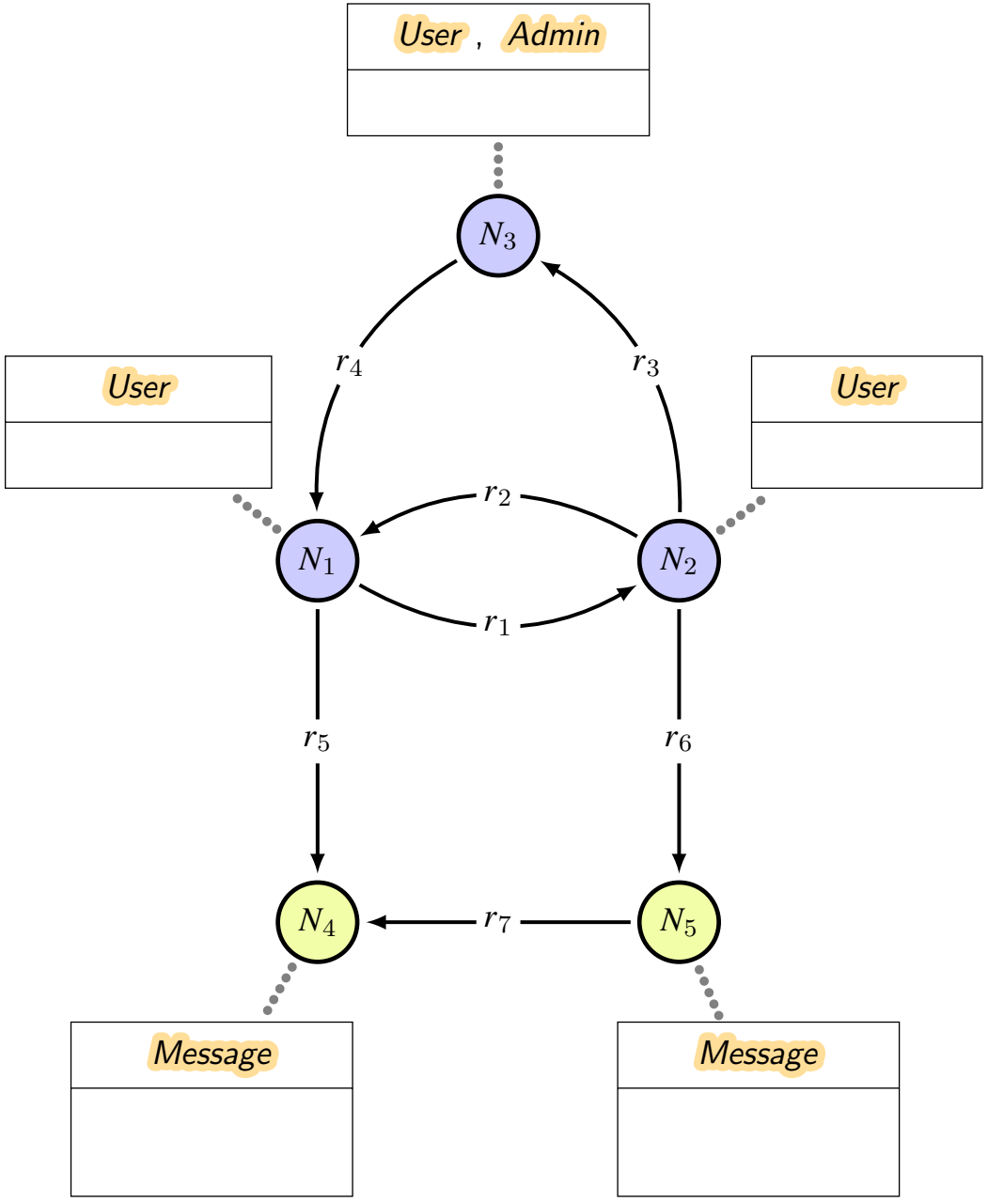
Property graph



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

  $r_1, r_2, \cdots, r_7$

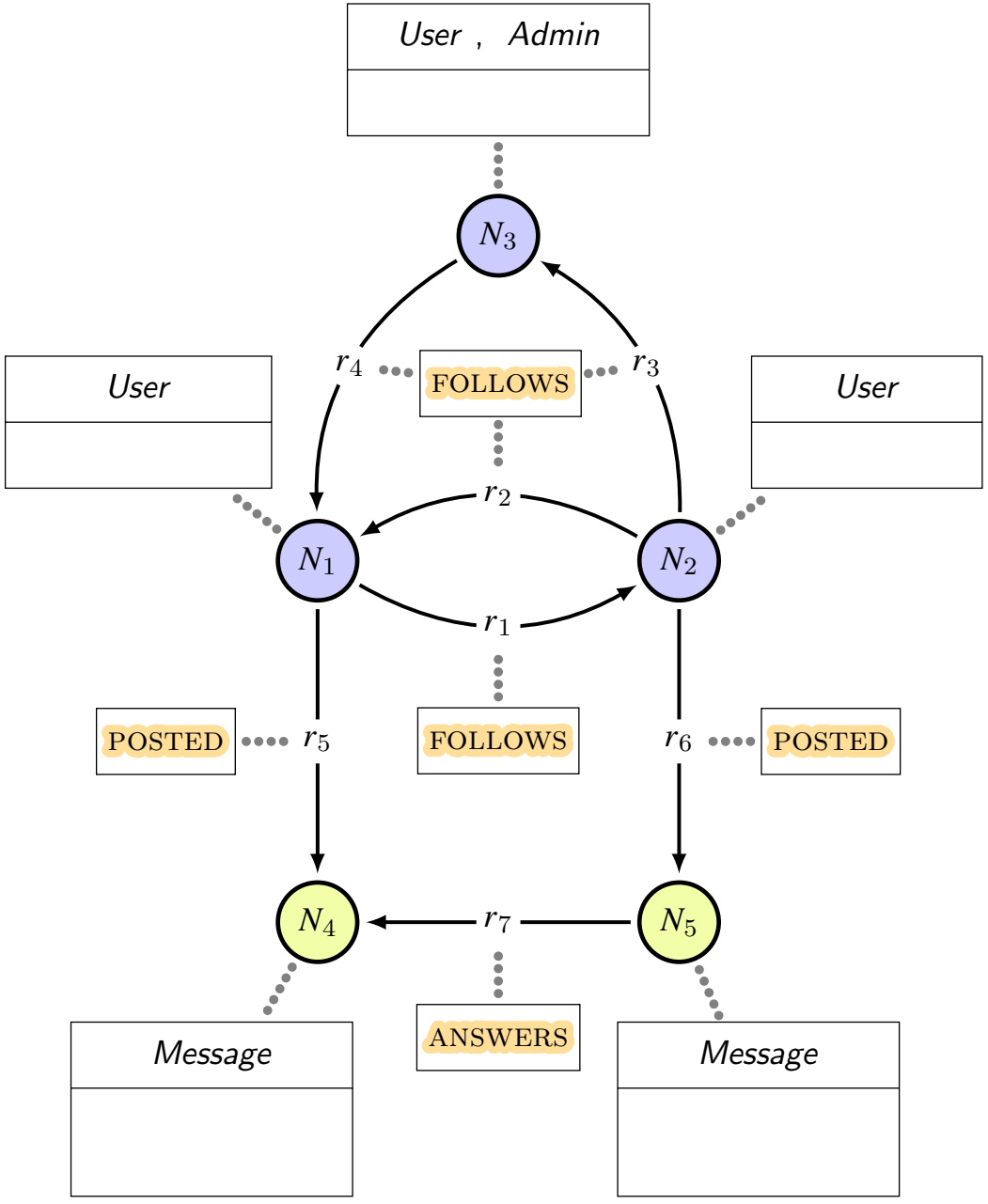
Property graph



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

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

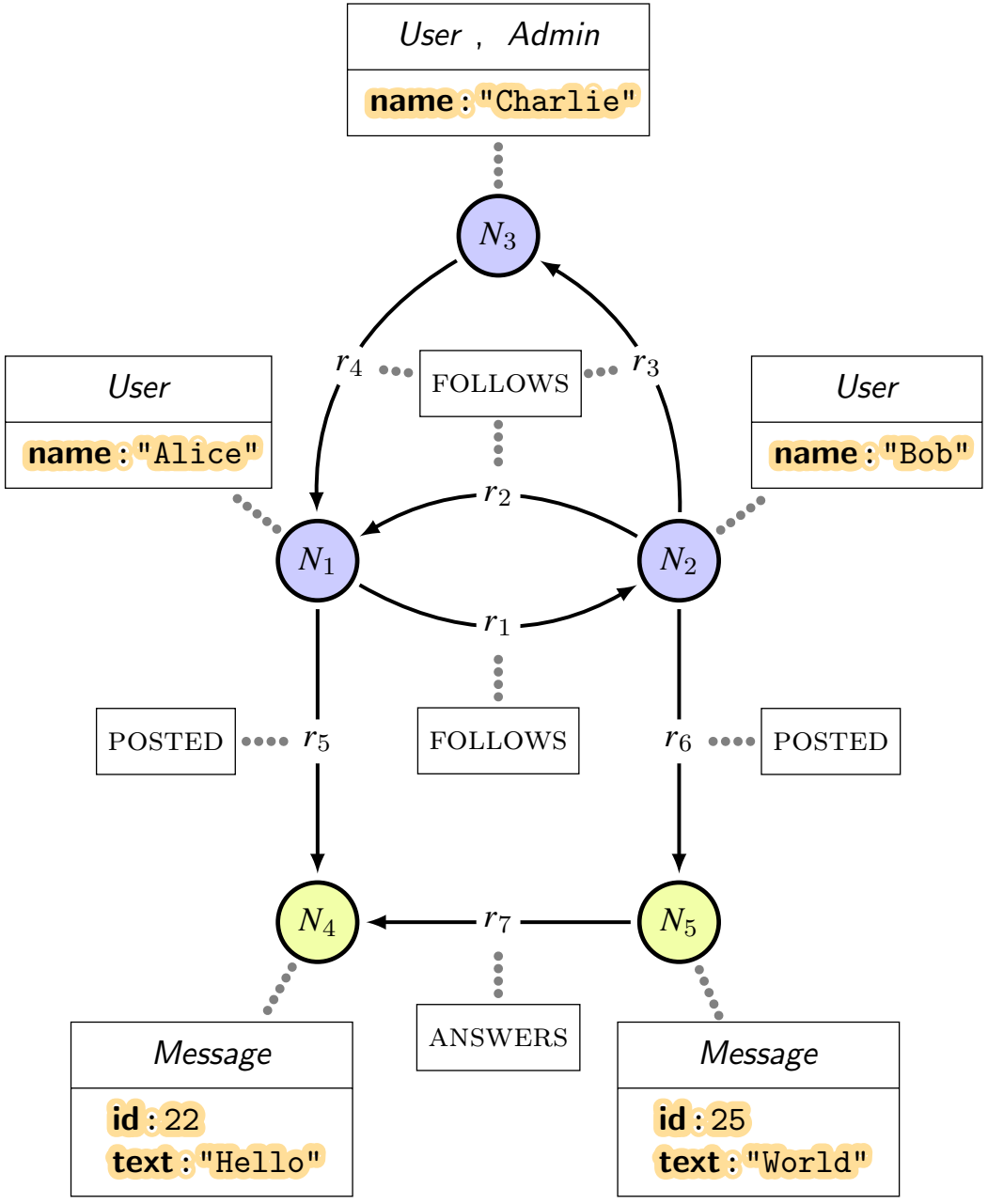
Property graph



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

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

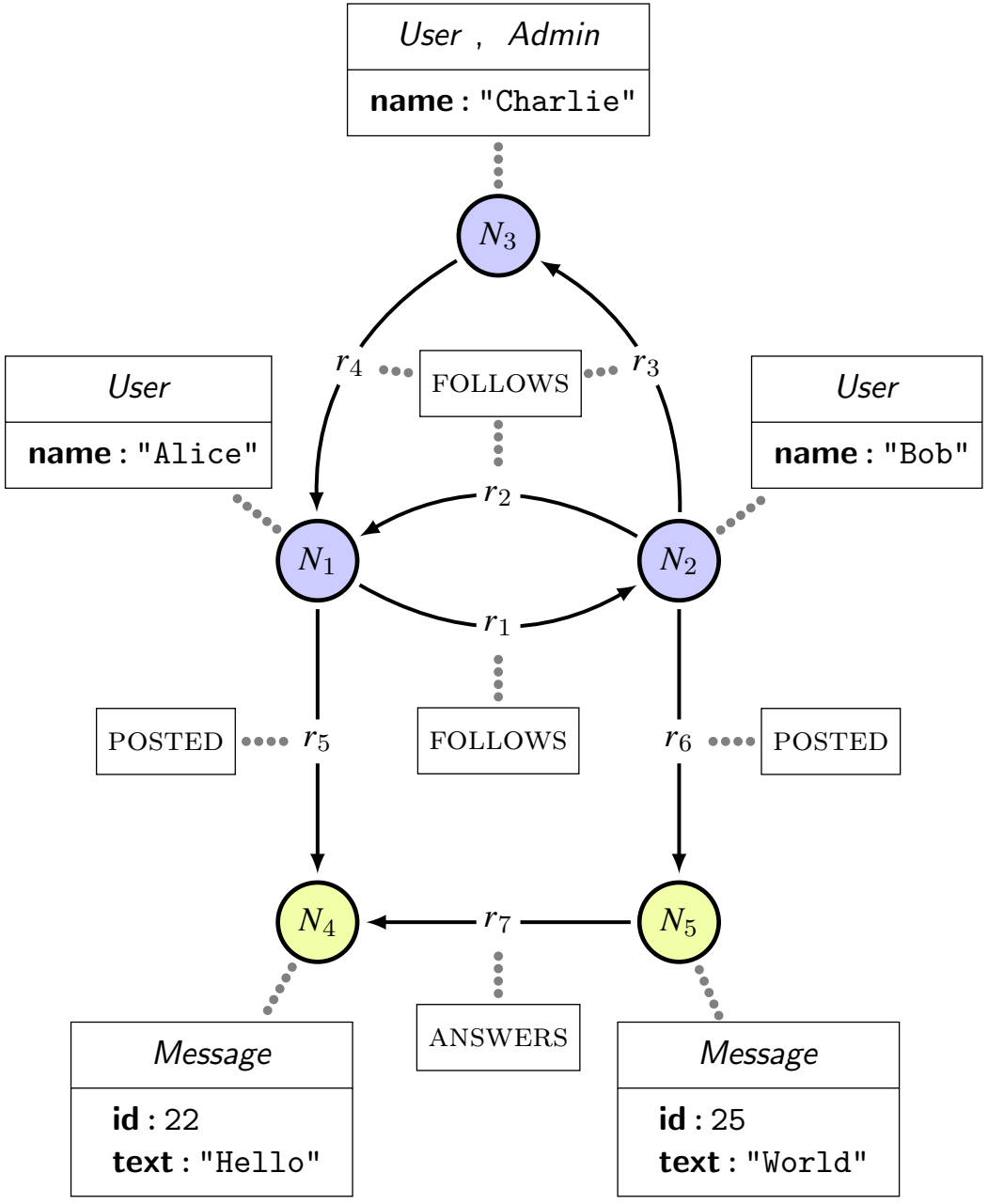
Property graph



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

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

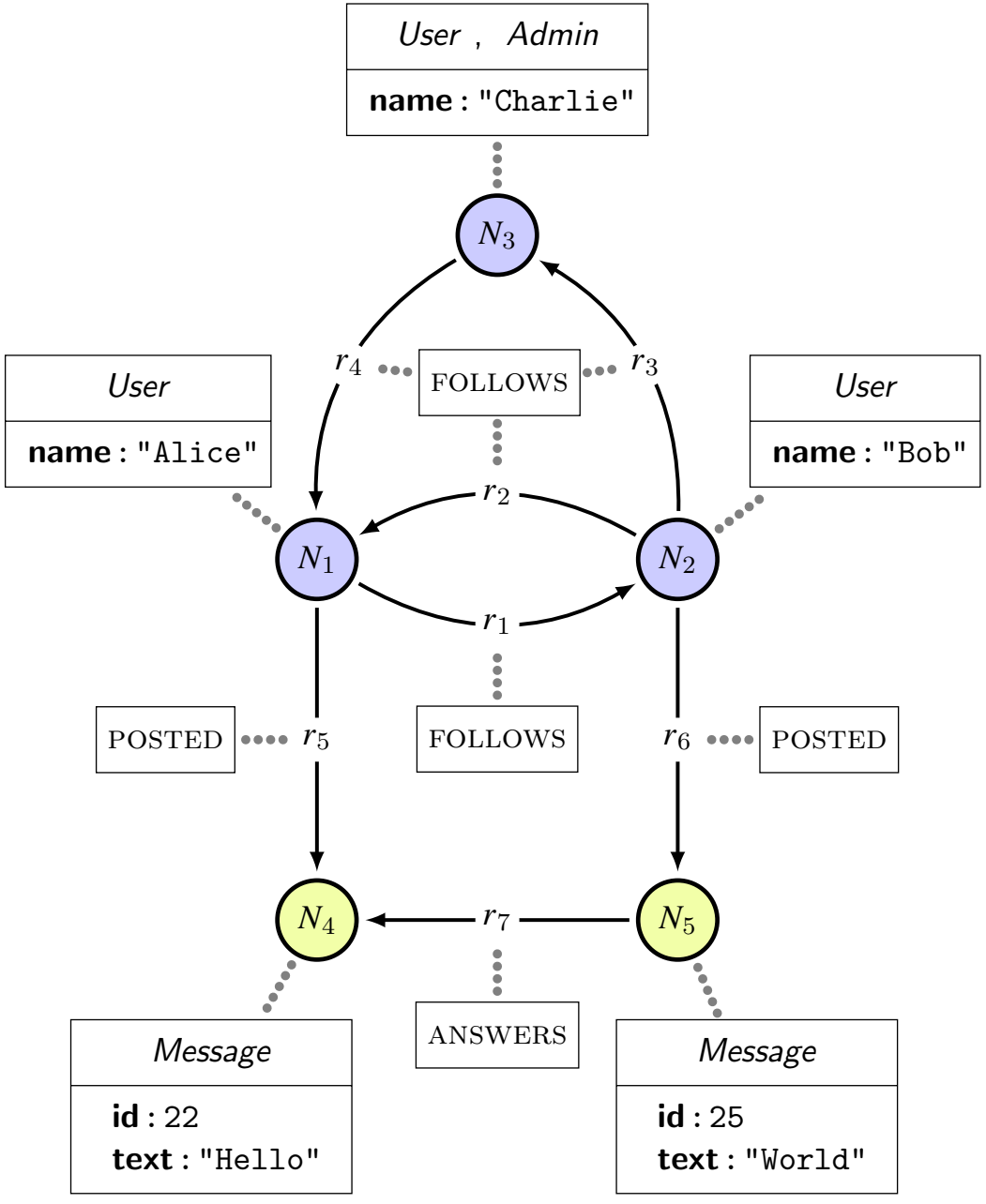
Property graph



- 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"`

Computing a Cypher query



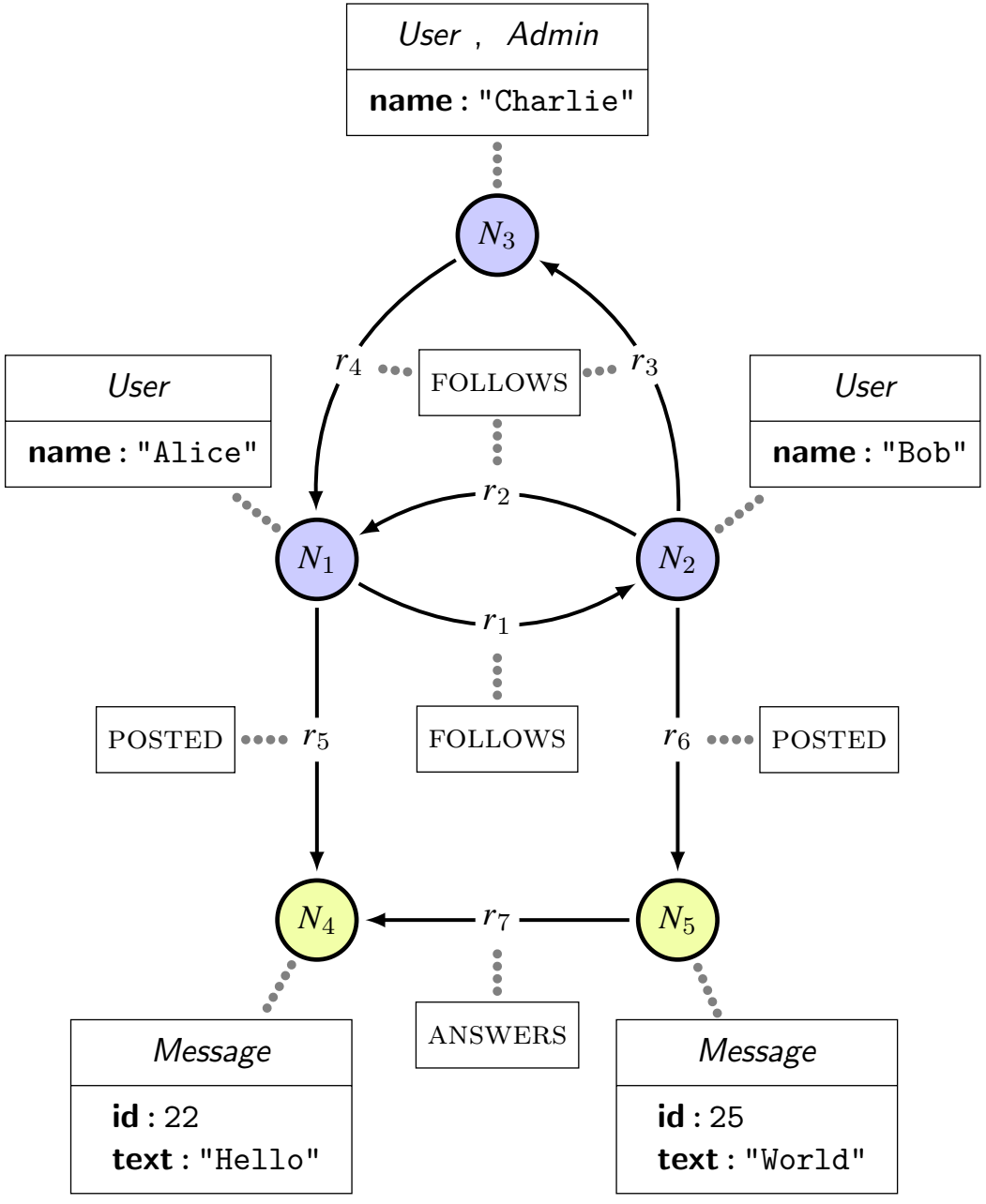
Example of a Cypher query :

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

A Cypher statement :

- is a sequence of *clauses*

Computing a Cypher query



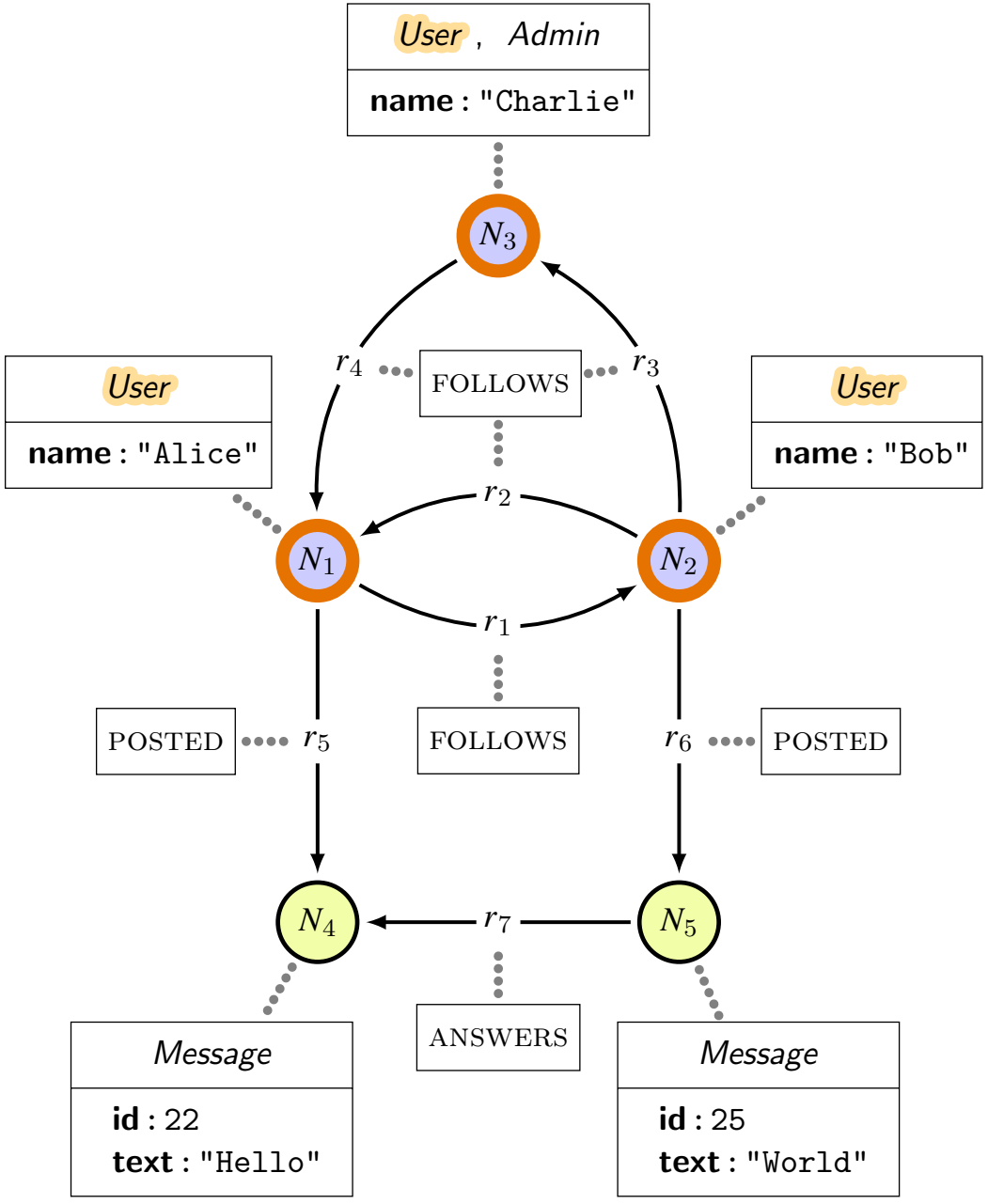
Example of a Cypher query :

```
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

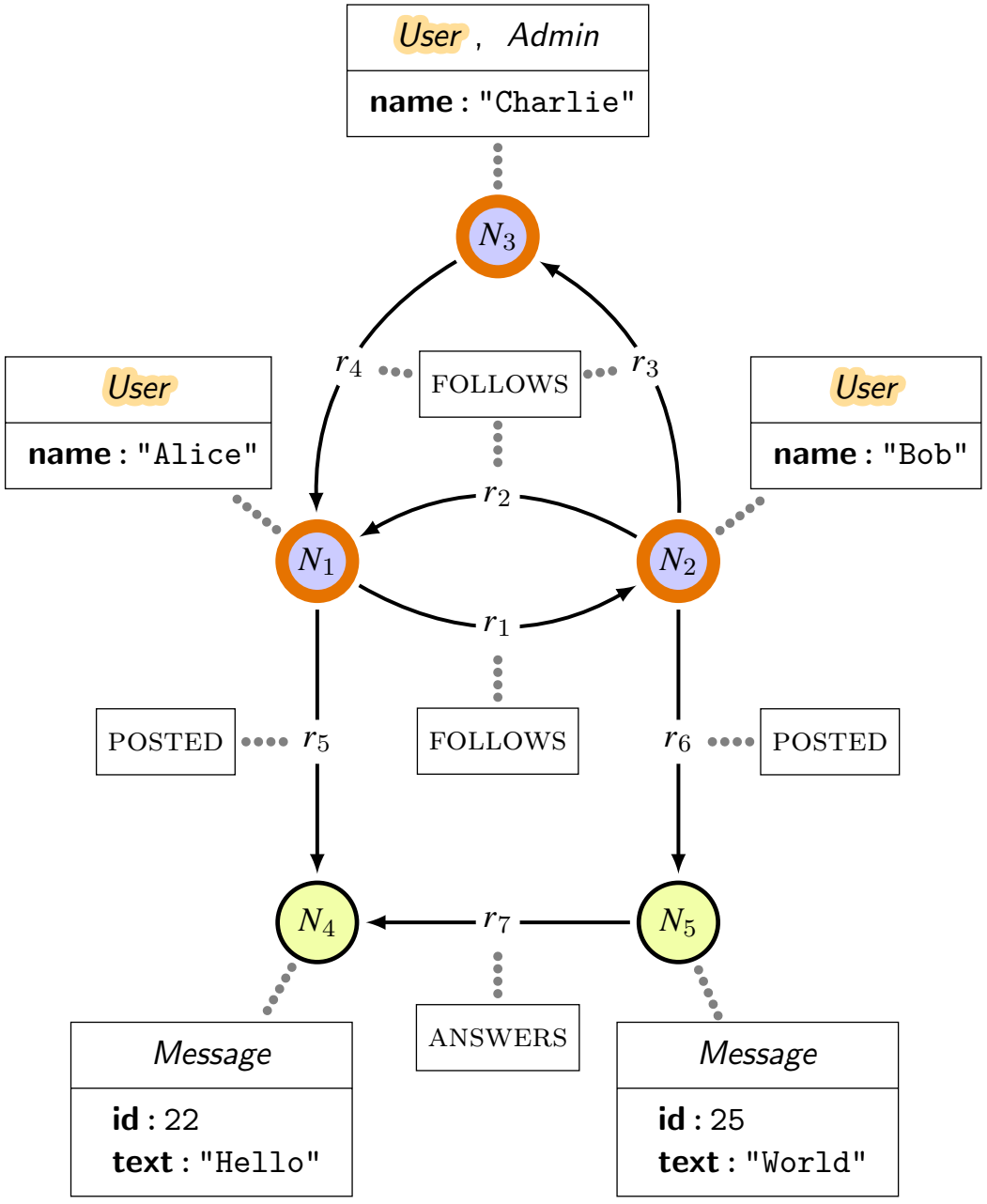
Matching nodes (1)



Query :

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

Matching nodes (1)



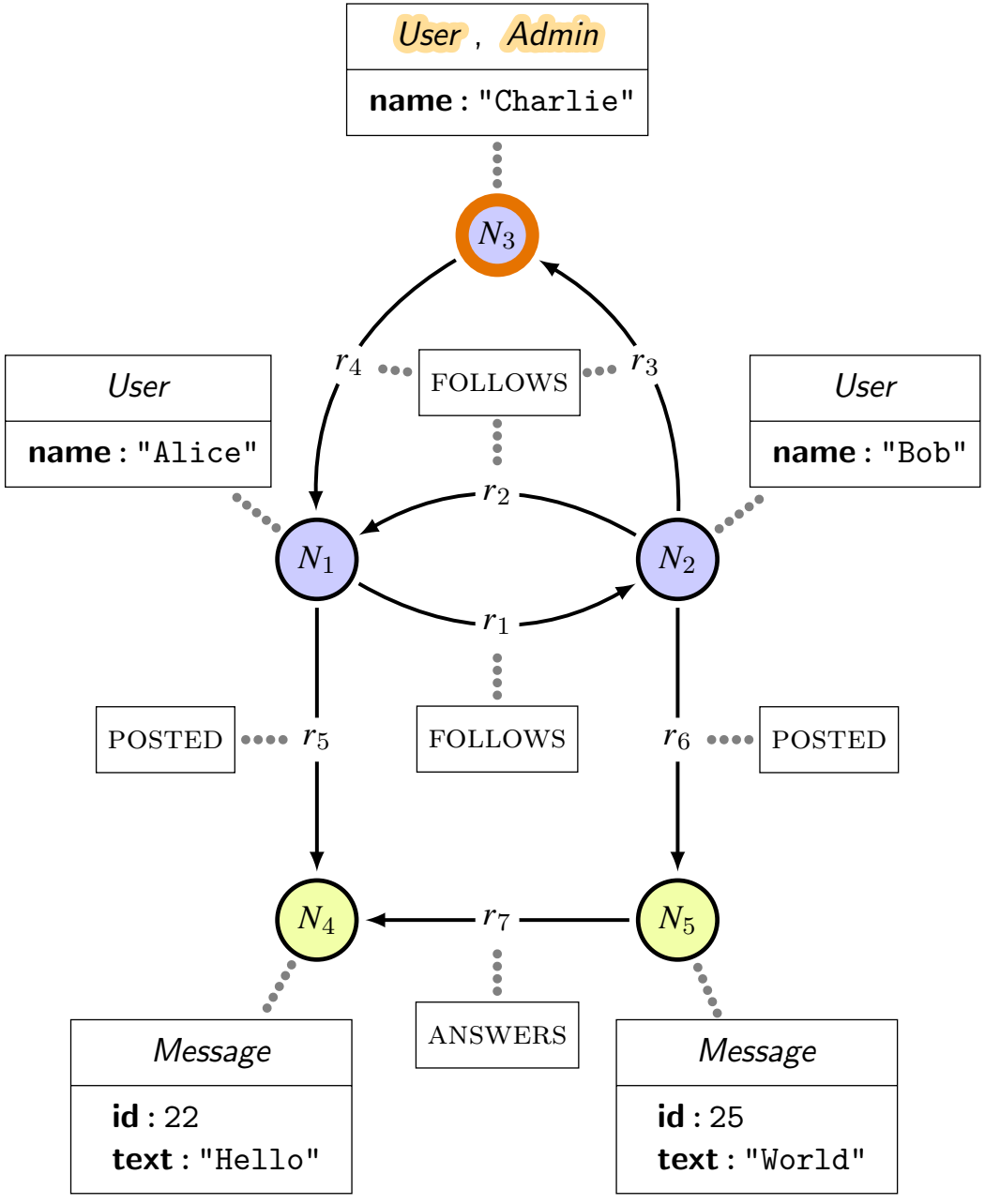
Query :

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

Result :

u1
N_1
N_2
N_3

Matching nodes (2)



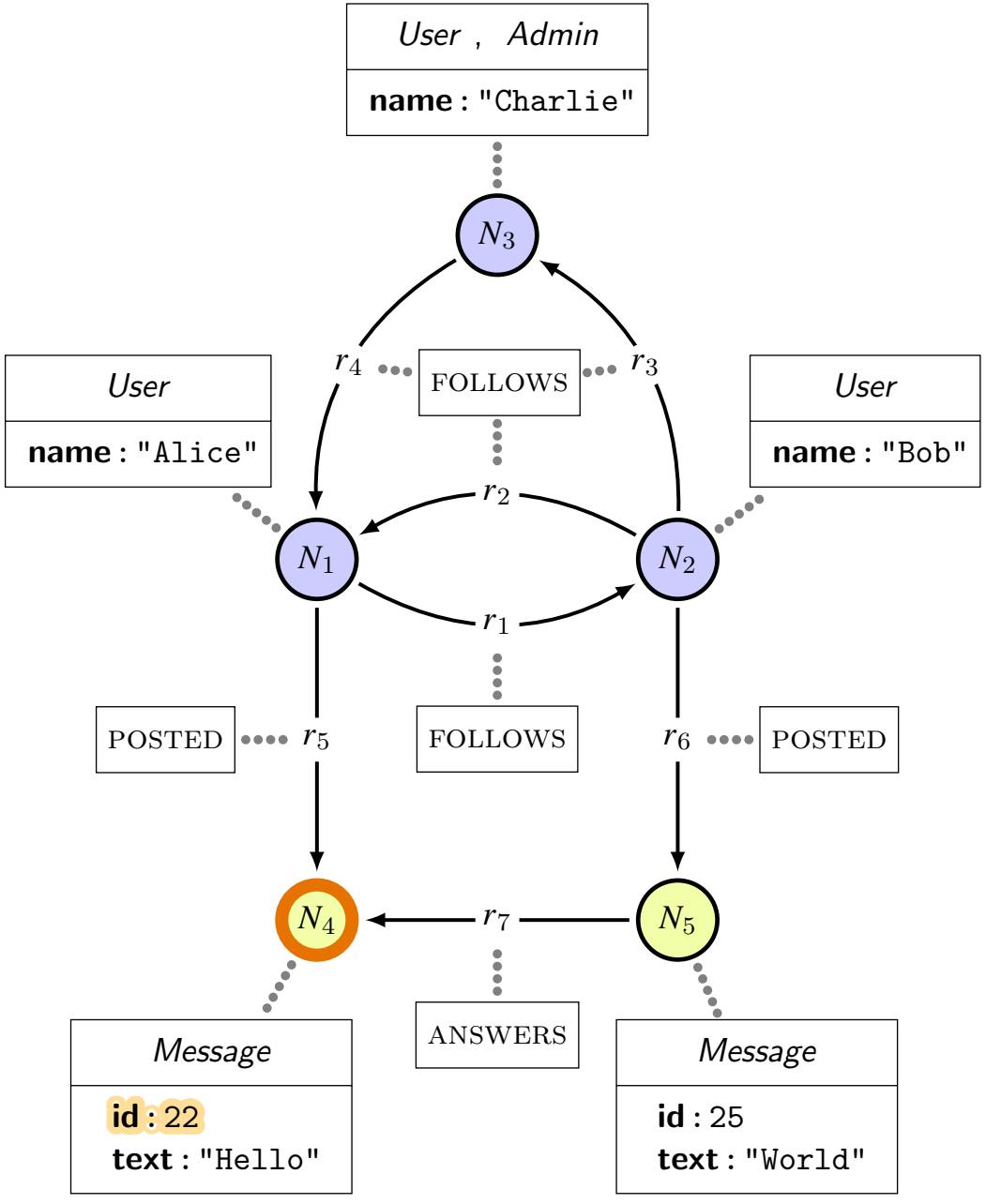
Query :

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

Result :

u1
N_3

Matching nodes (3)



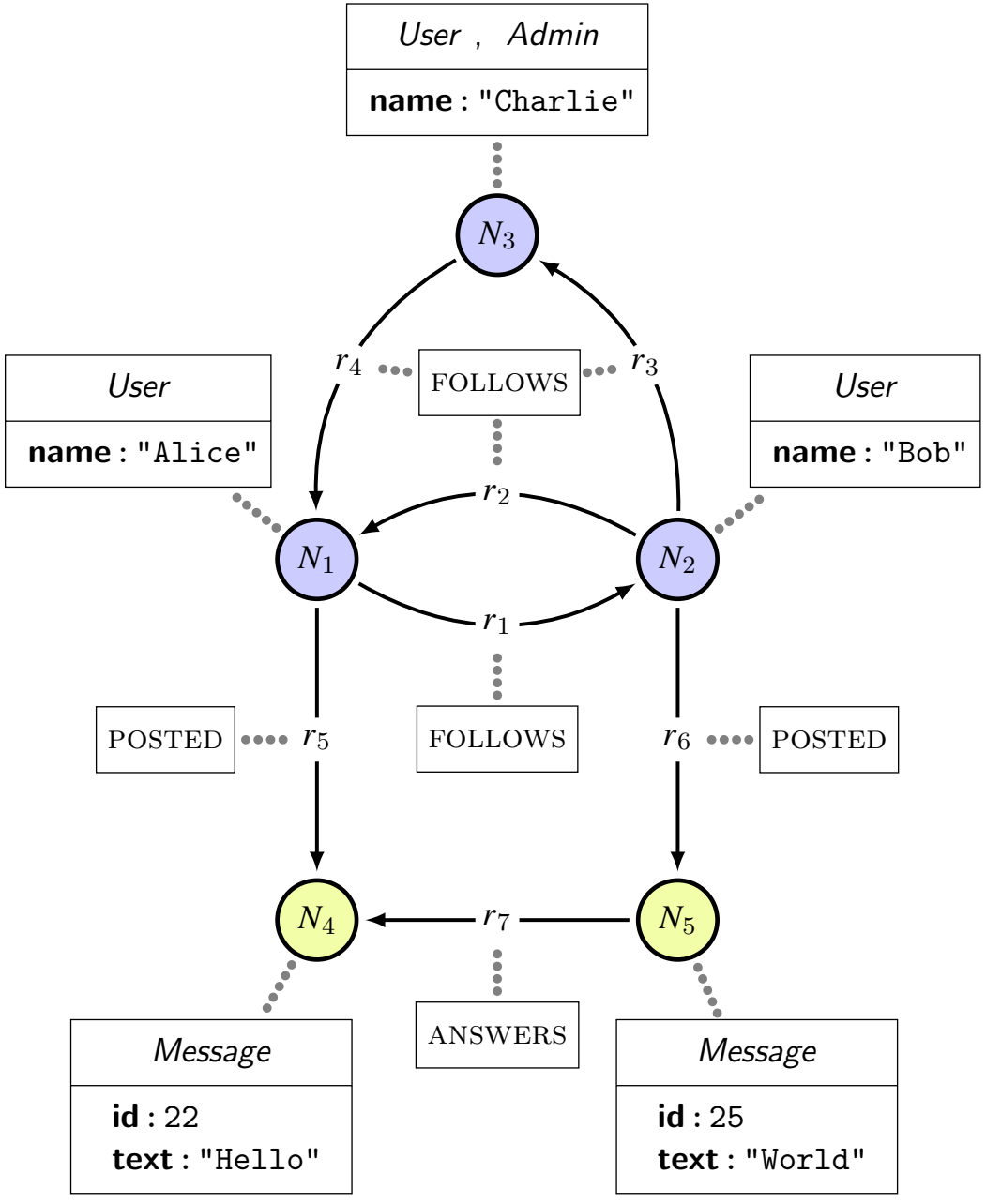
Query :

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

Result :

u1
N_4

Matching relationships (1)



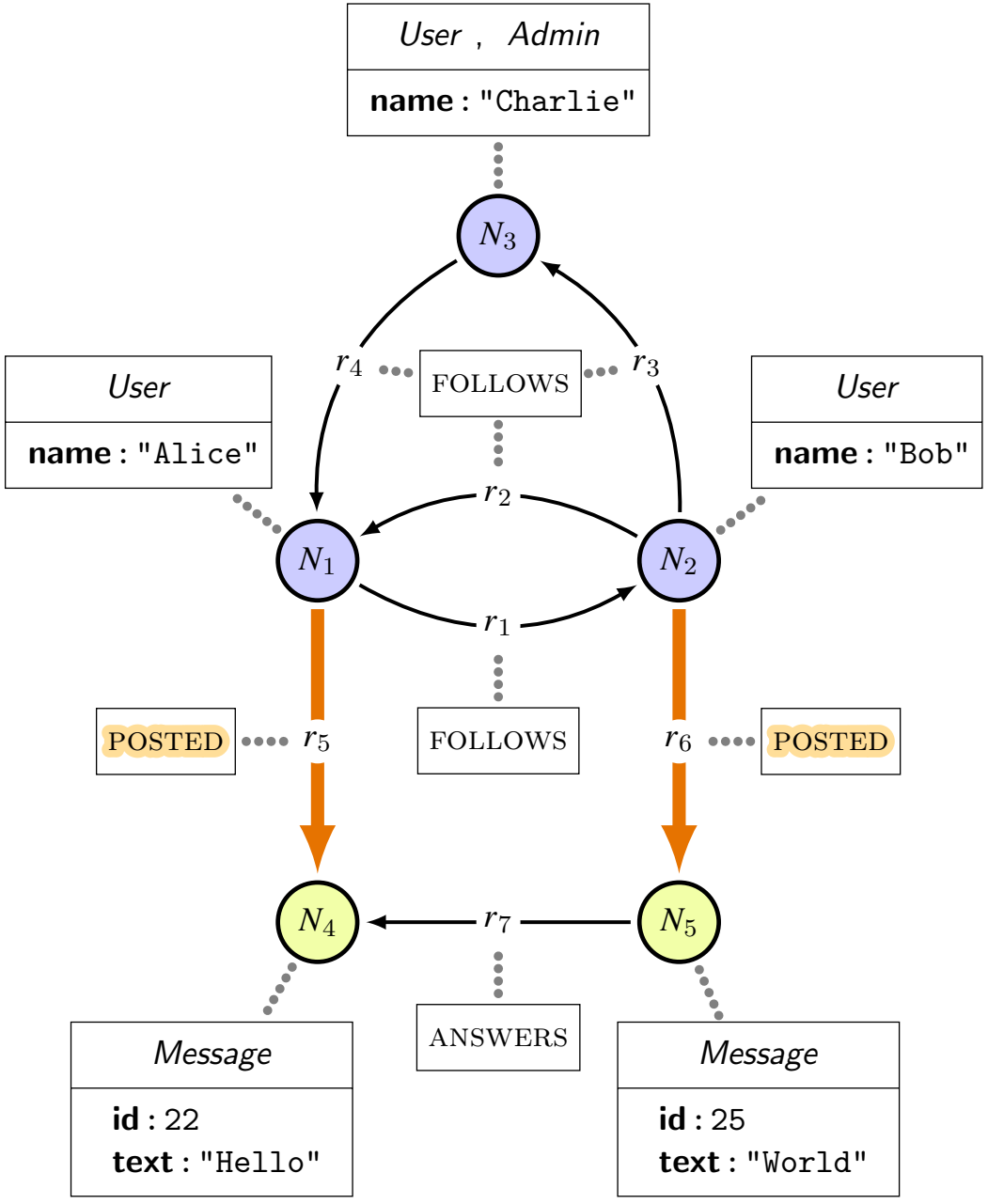
Query :

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

Result :

p1
r_1
r_2
r_3
r_4
r_5
r_6
r_7

Matching relationships (2)



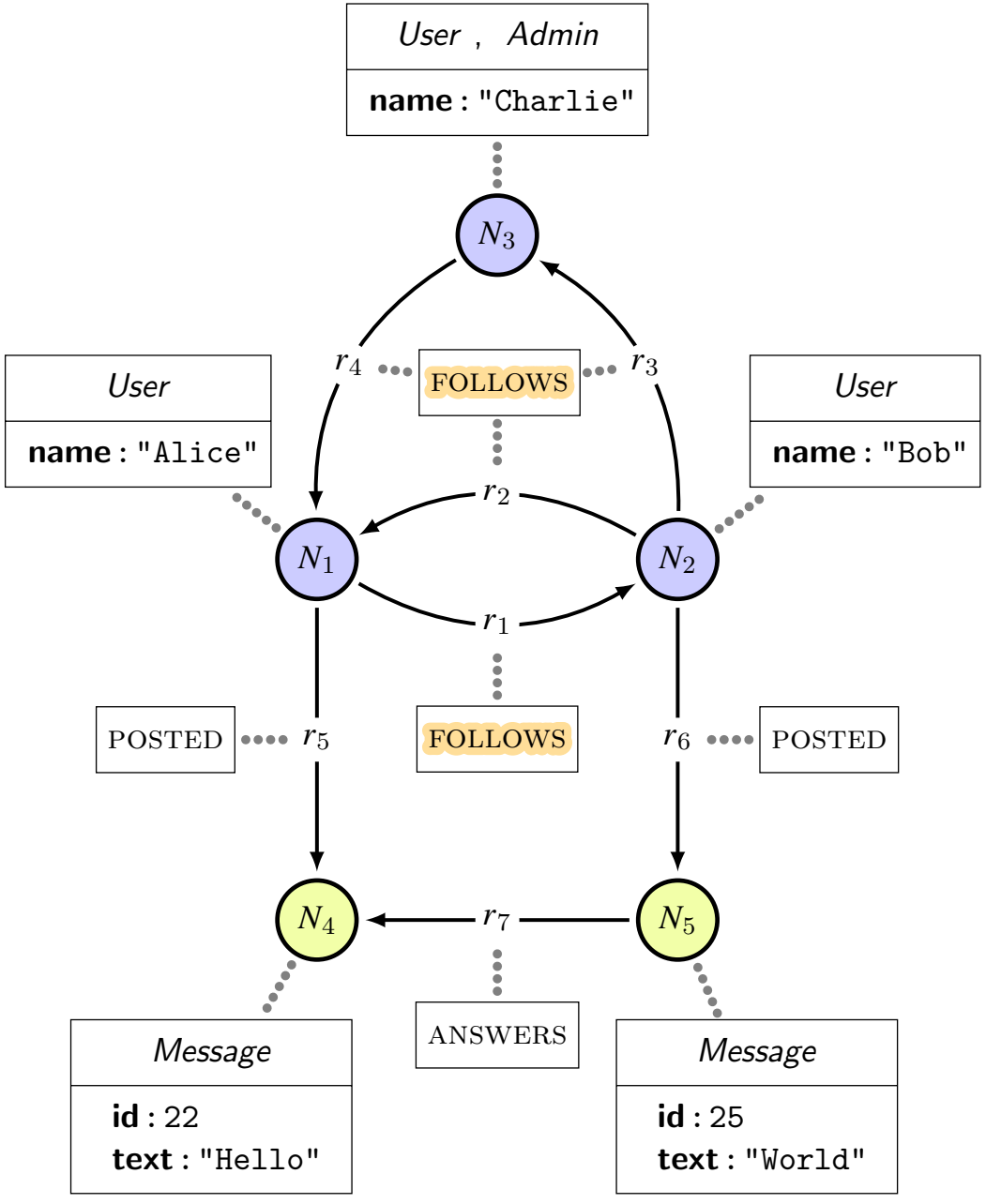
Query :

```
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]->()
```

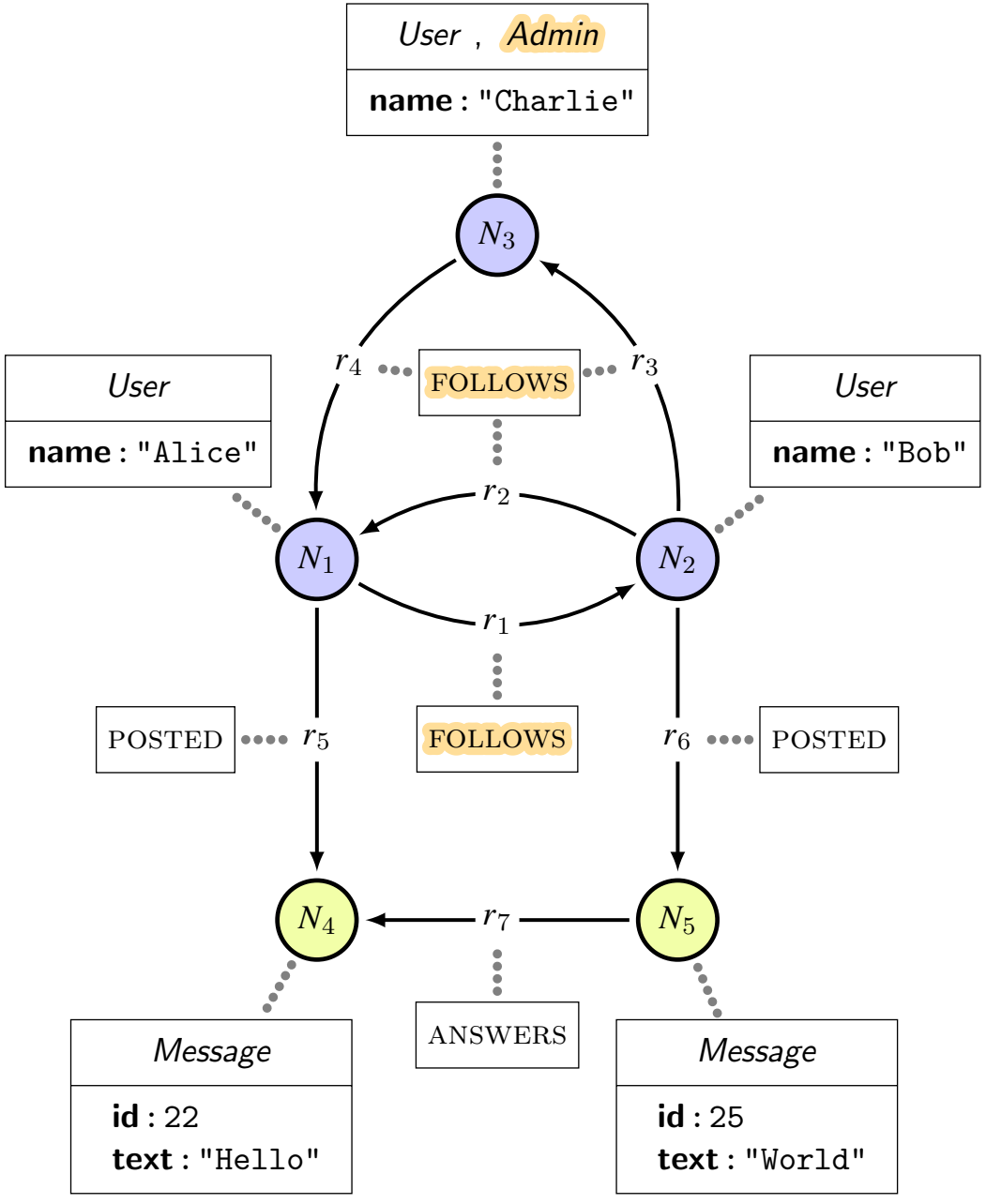
Result :

u1
N_1
N_2
N_2
N_3

Bag semantics :

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

Matching relationships (4)



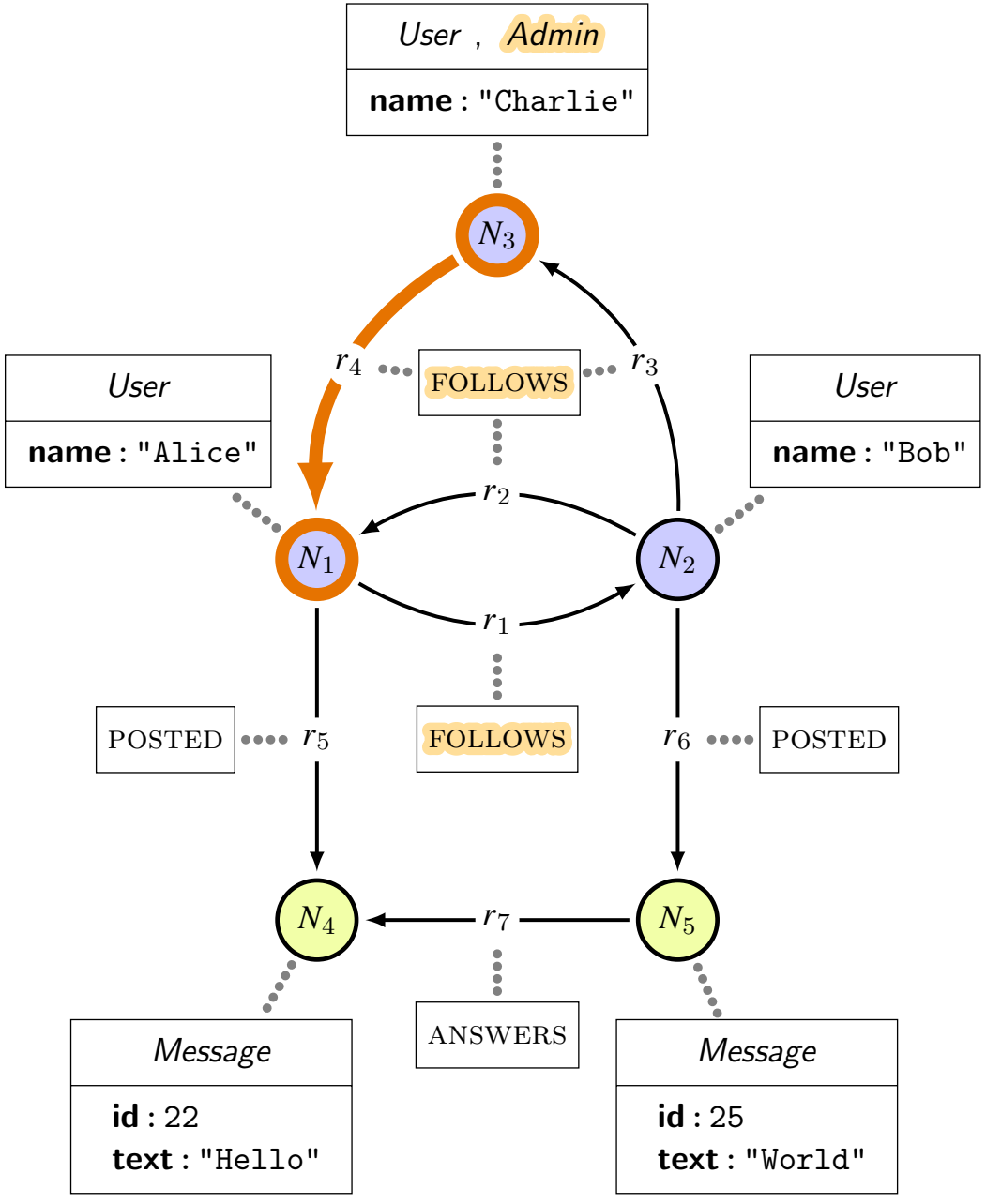
Query :

```
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)



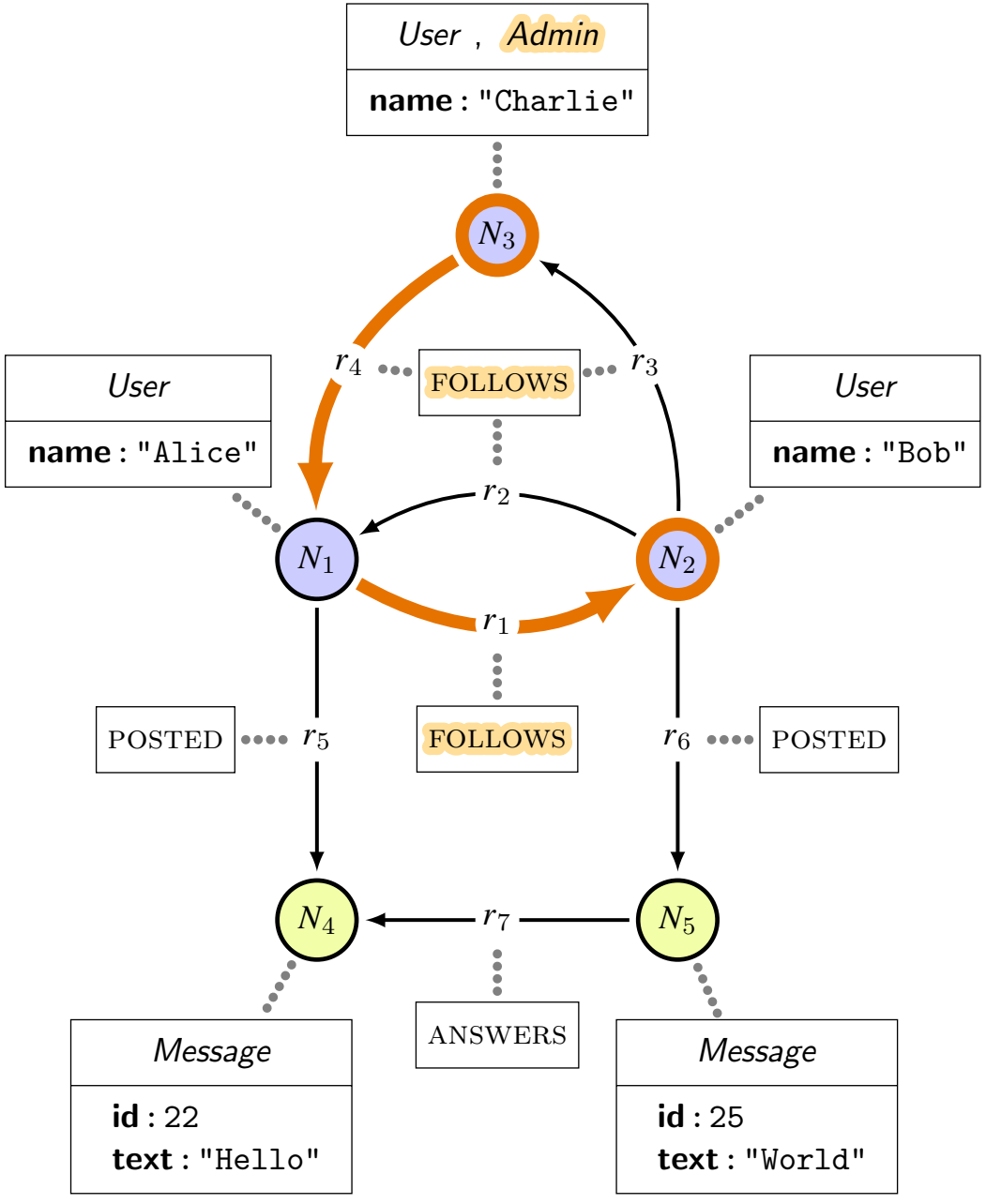
Query :

```
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

Matching relationships (4)



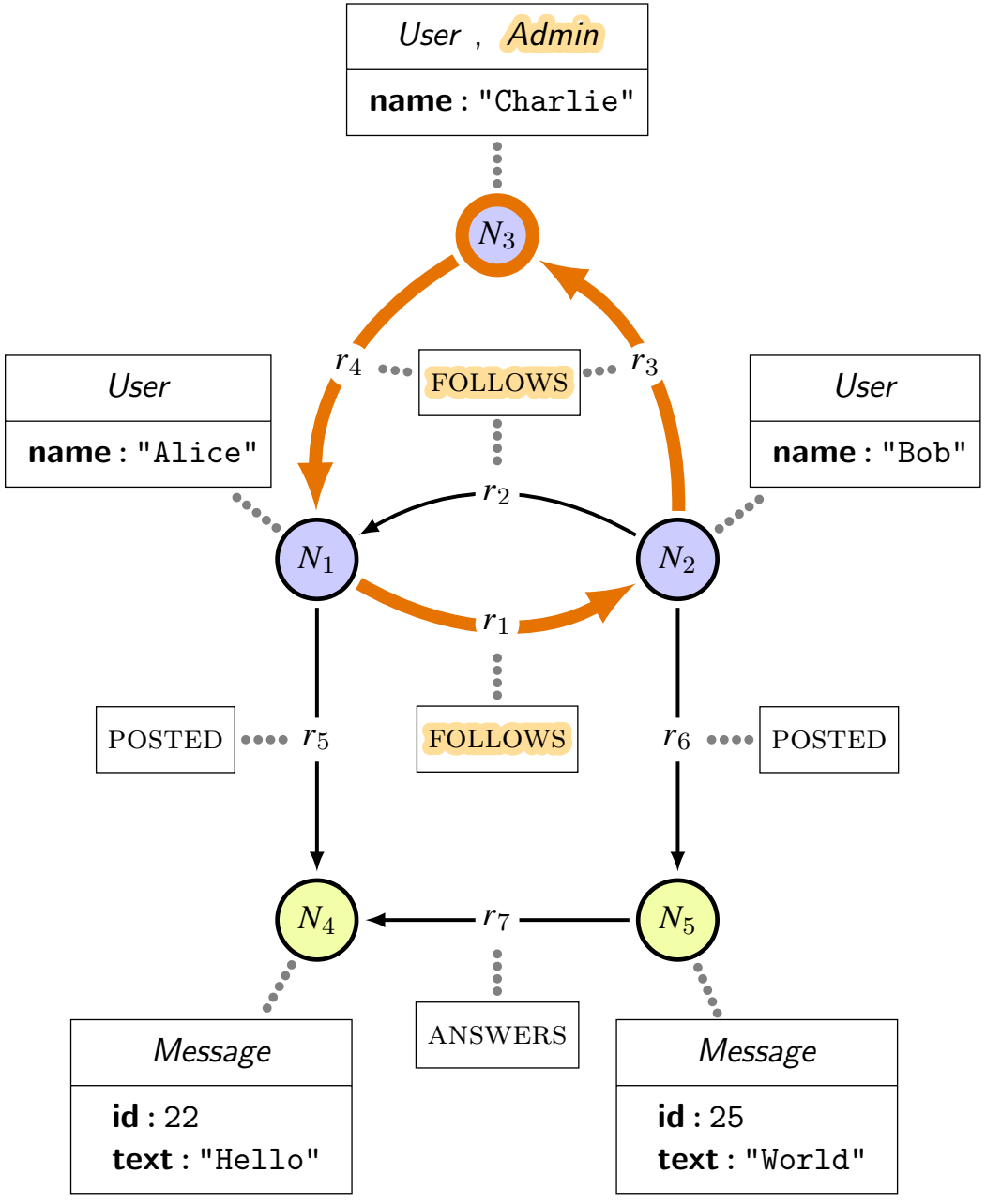
Query :

```
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)



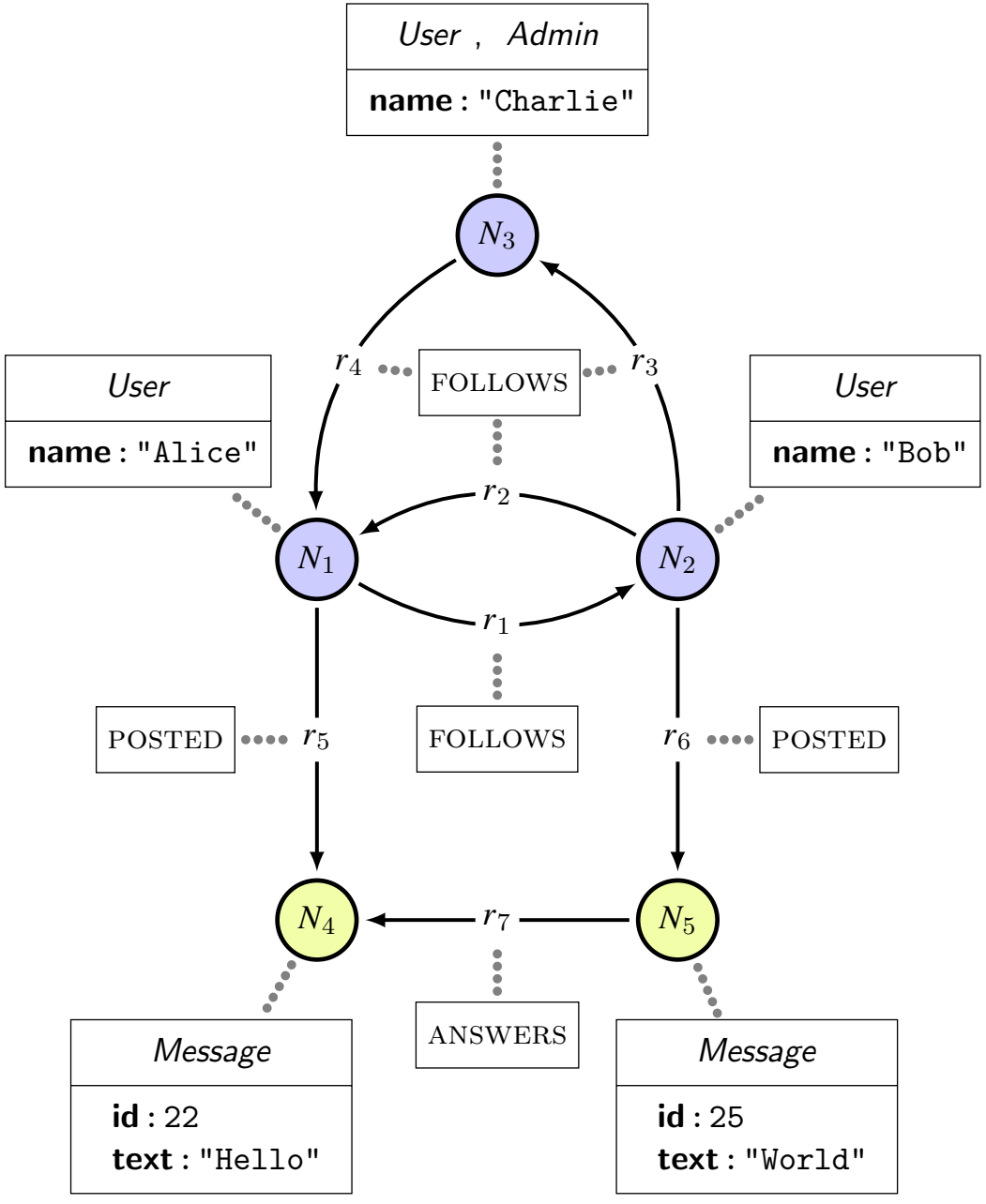
Query :

```
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

Matching relationships (4)



Query :

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

Result :

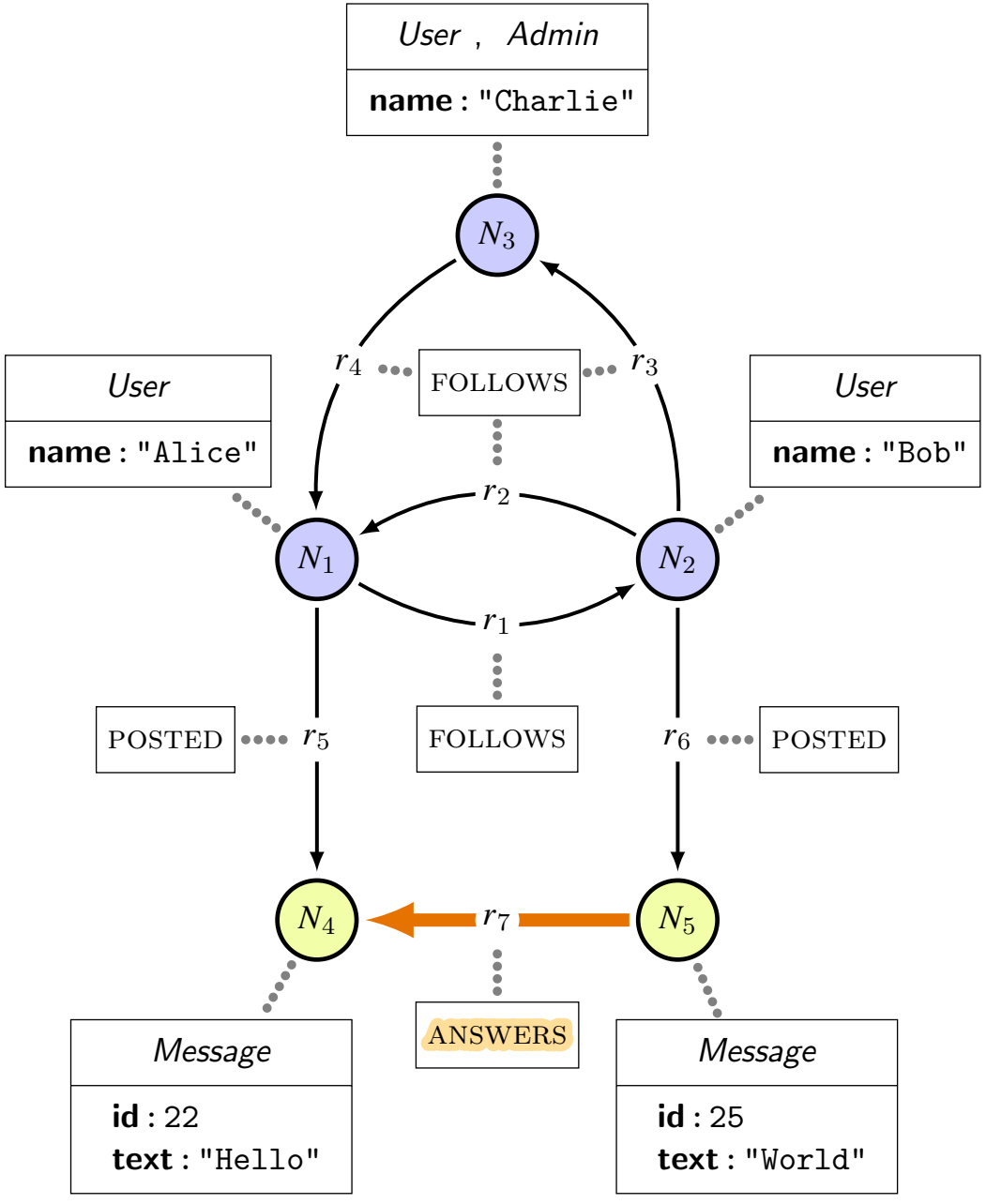
u1	l1	m1
N_3	$[r_4]$	N_1
N_3	$[r_4, r_1]$	N_2
N_3	$[r_4, r_1, r_2]$	N_1
N_3	$[r_4, r_1, r_3]$	N_3

Cypher-Morphism :

- Each rel. matched ≤ 1 time

$\Rightarrow$ Finitely many results

Matching relationships (5)



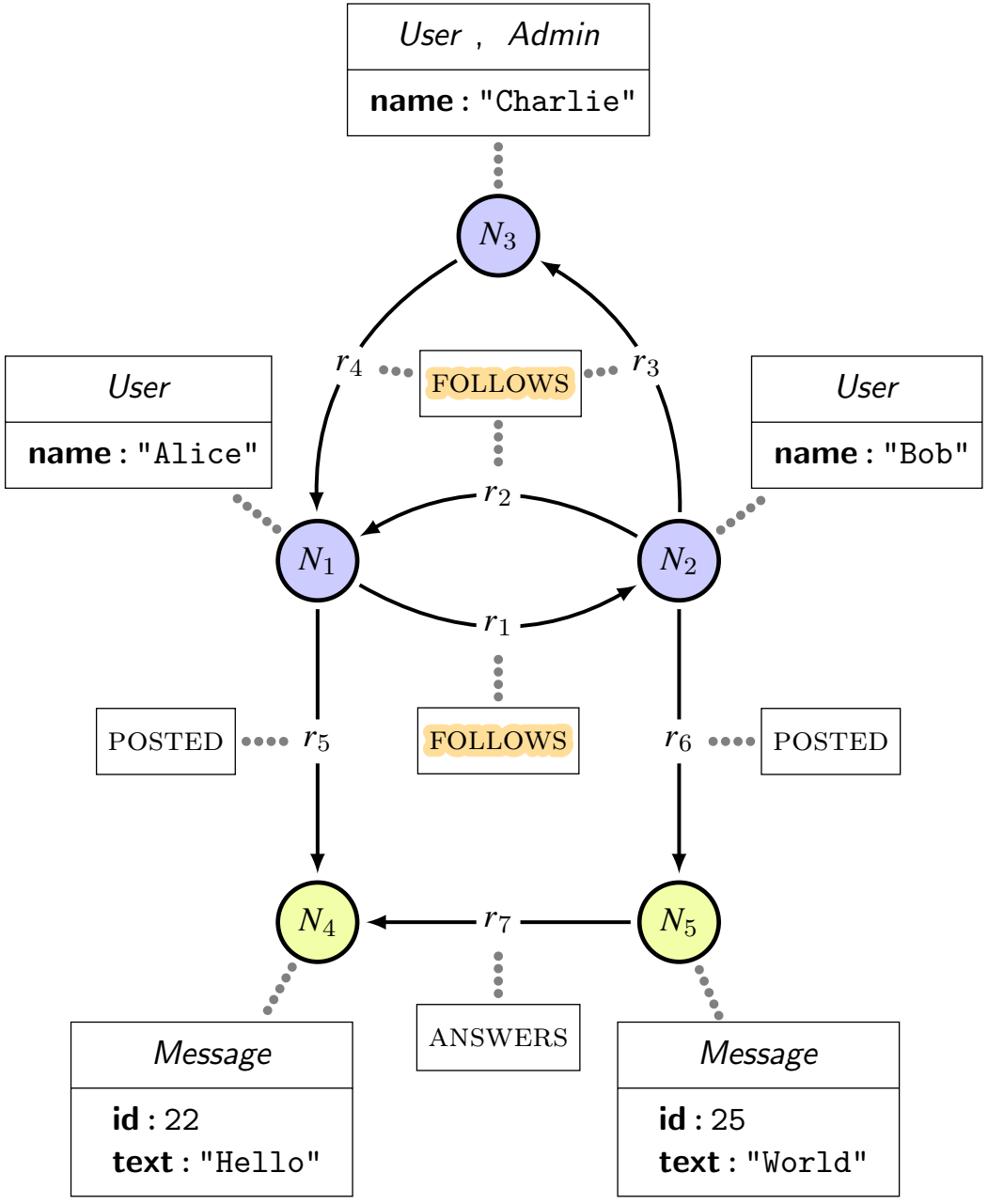
Query :

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

Result :

m1	m2
N_4	N_5
N_5	N_4

Matching relationships (6)



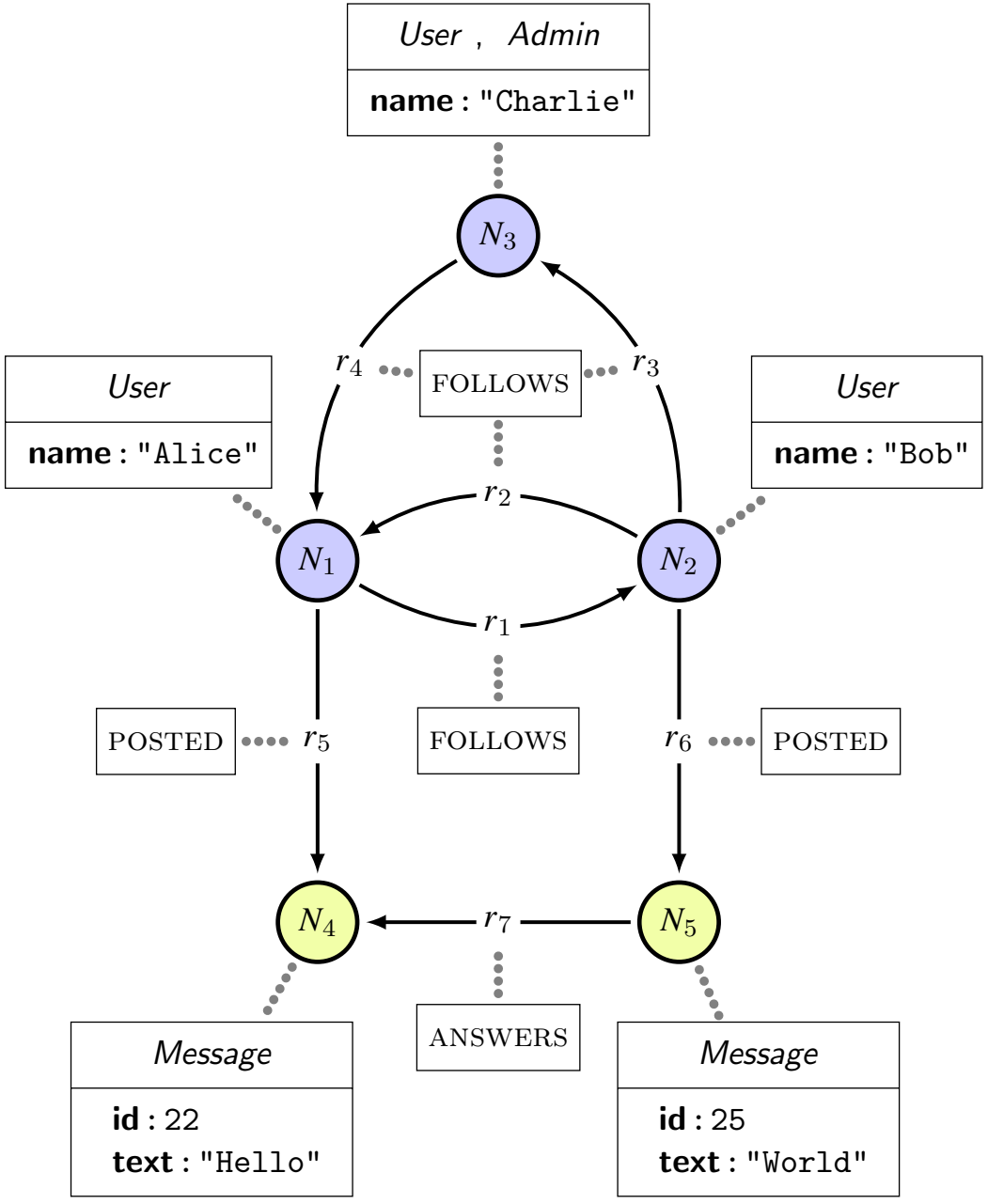
Query :

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

Result :

u1
N_1
N_1
N_2
N_2
N_3

Matching paths (1)



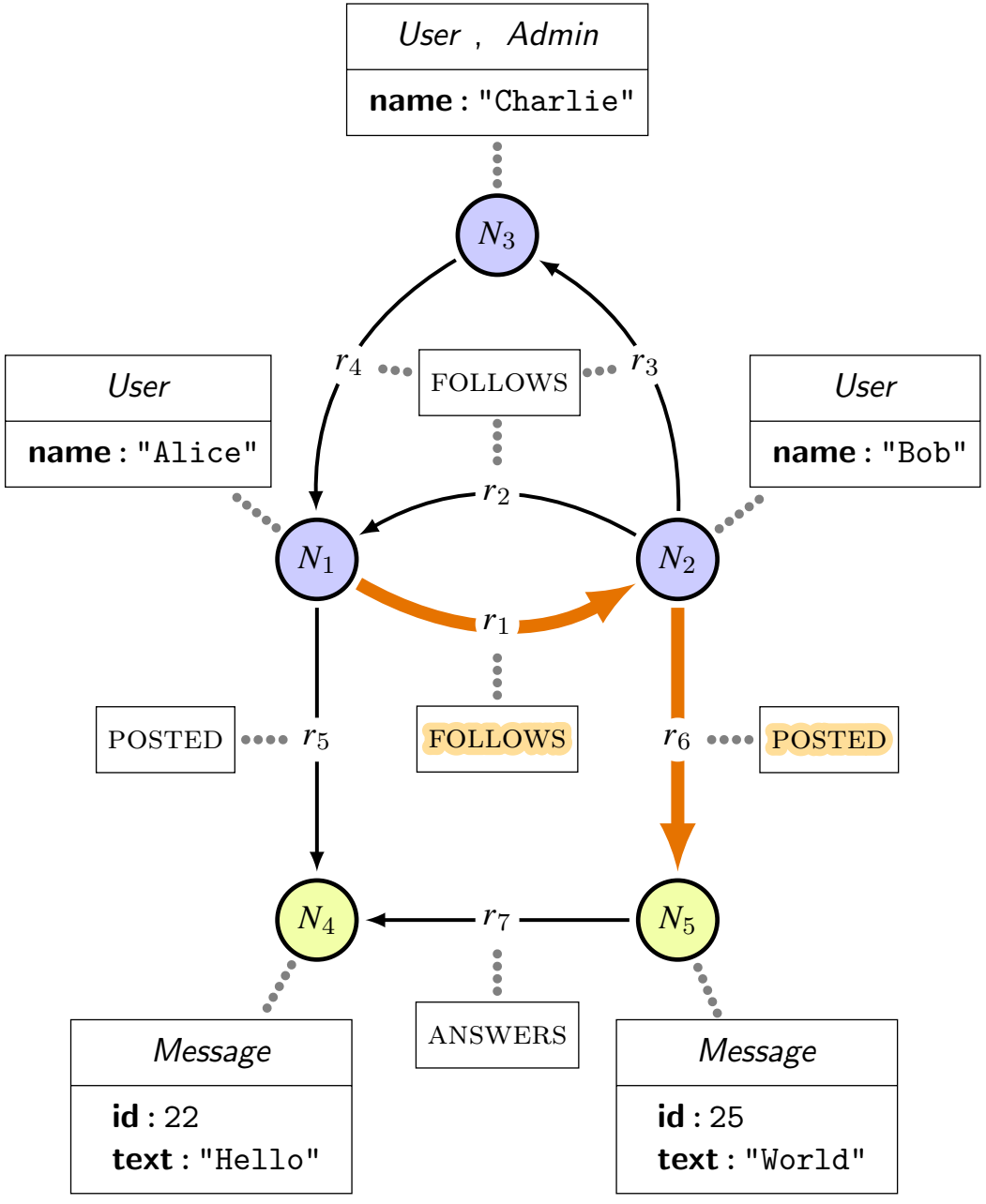
Query :

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

Result :

u1	m1
N_1	N_5
N_2	N_4
N_3	N_5

Matching paths (1)



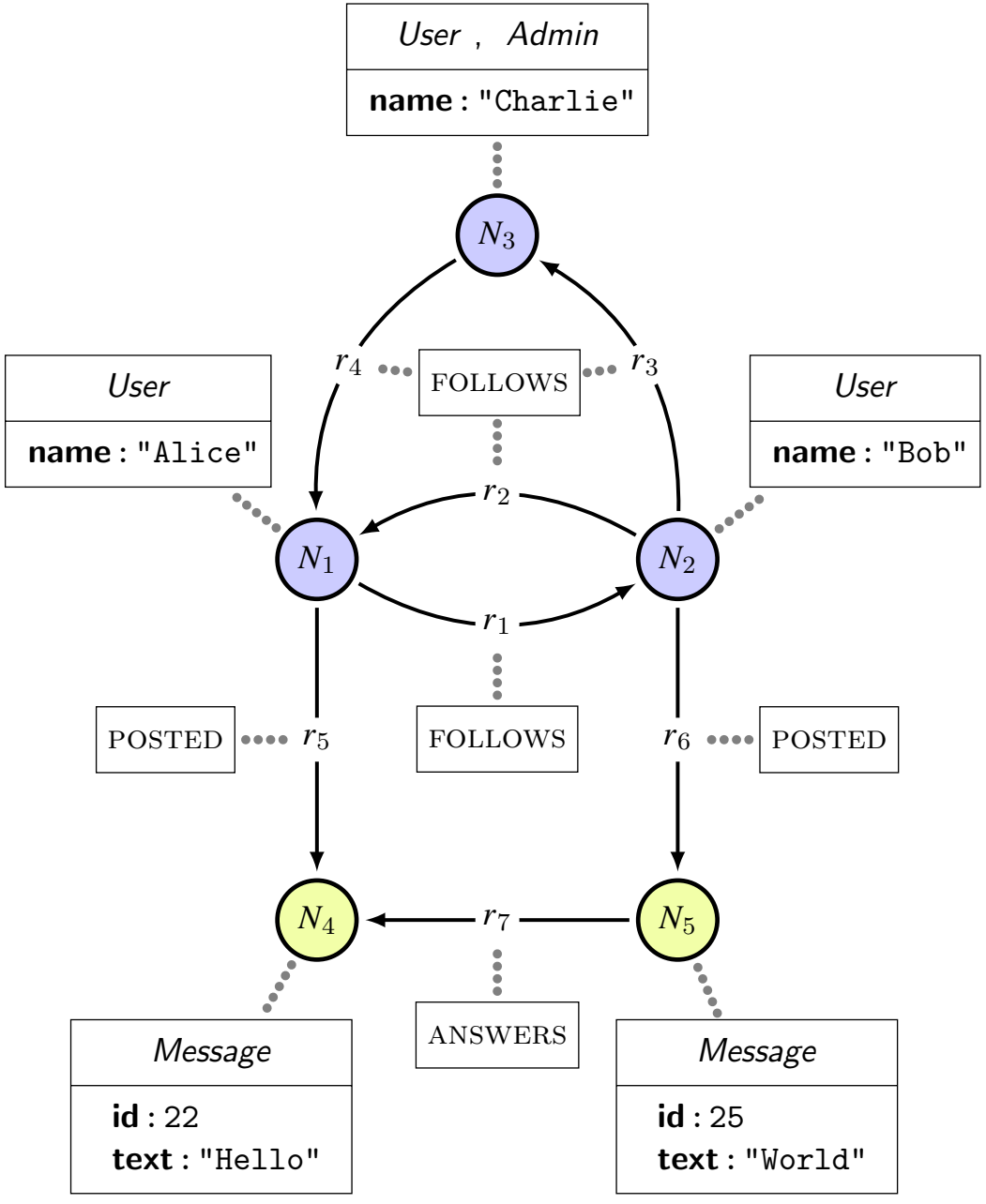
Query :

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

Result :

u1	m1
N_1	N_5
N_2	N_4
N_3	N_5

Matching paths (2)



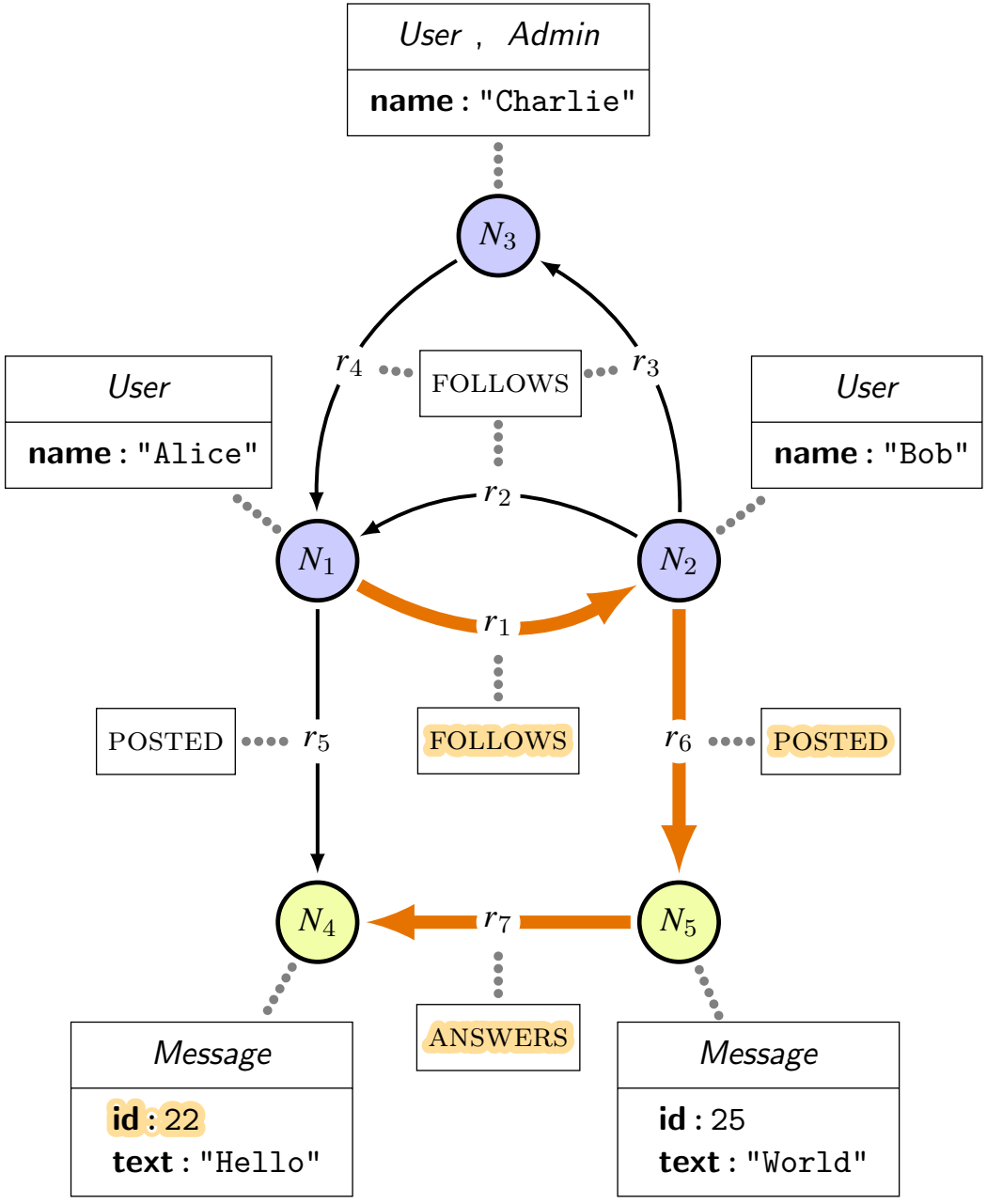
Query :

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

Result :

u1	m1
N_1	N_5

Matching paths (2)



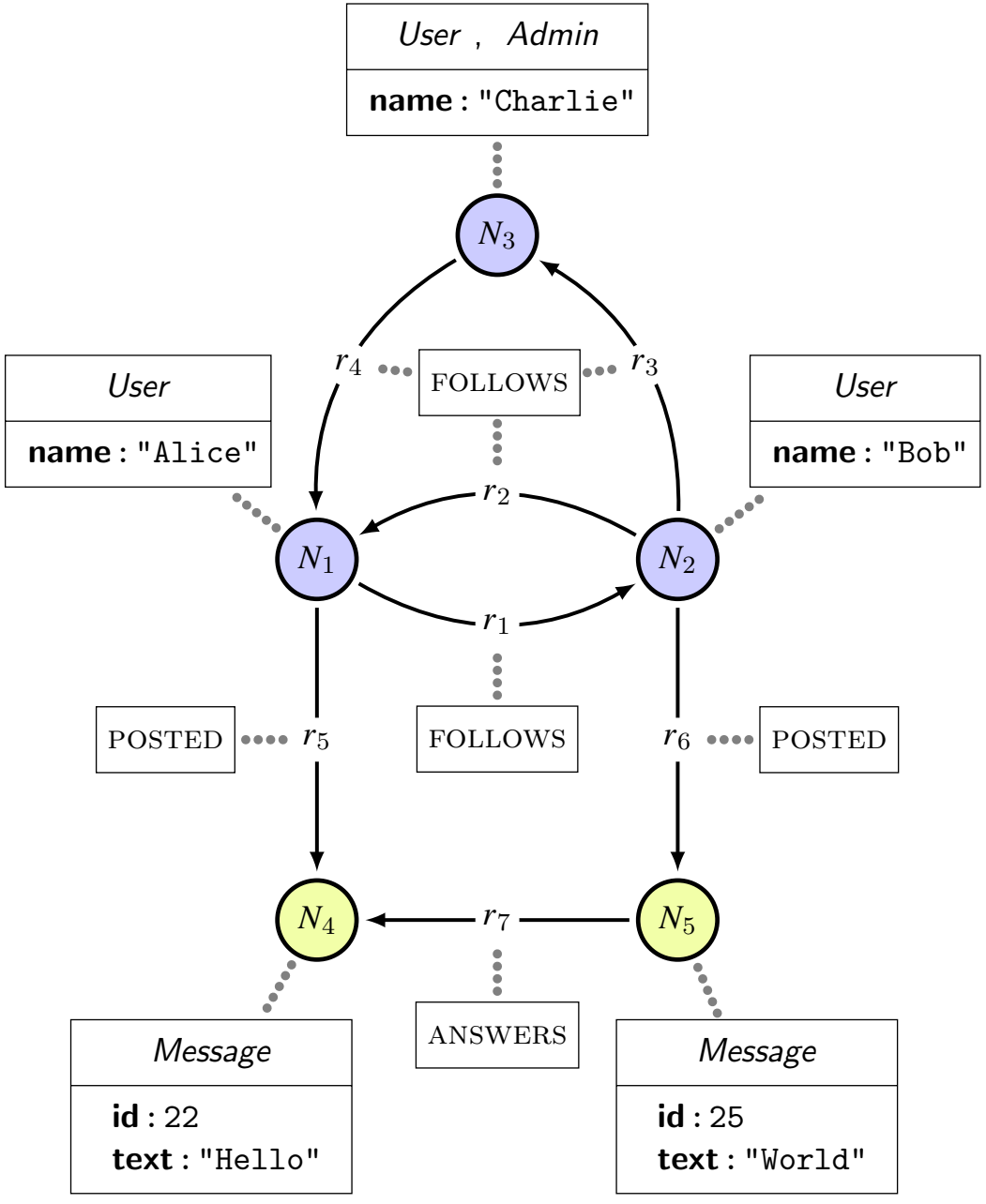
Query :

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

Result :

u1	m1
N_1	N_5

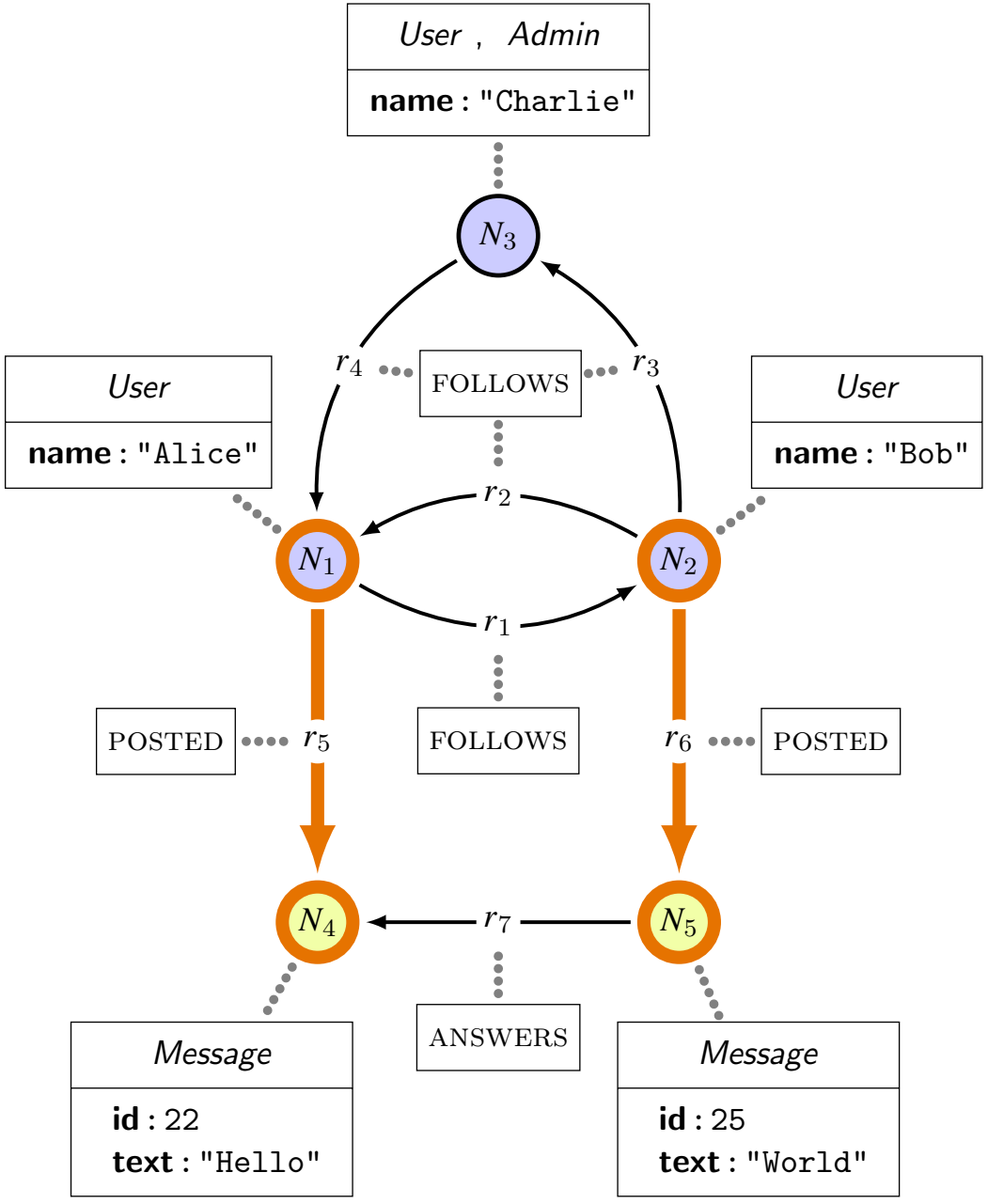
Chaining clauses



Query :

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

Chaining clauses



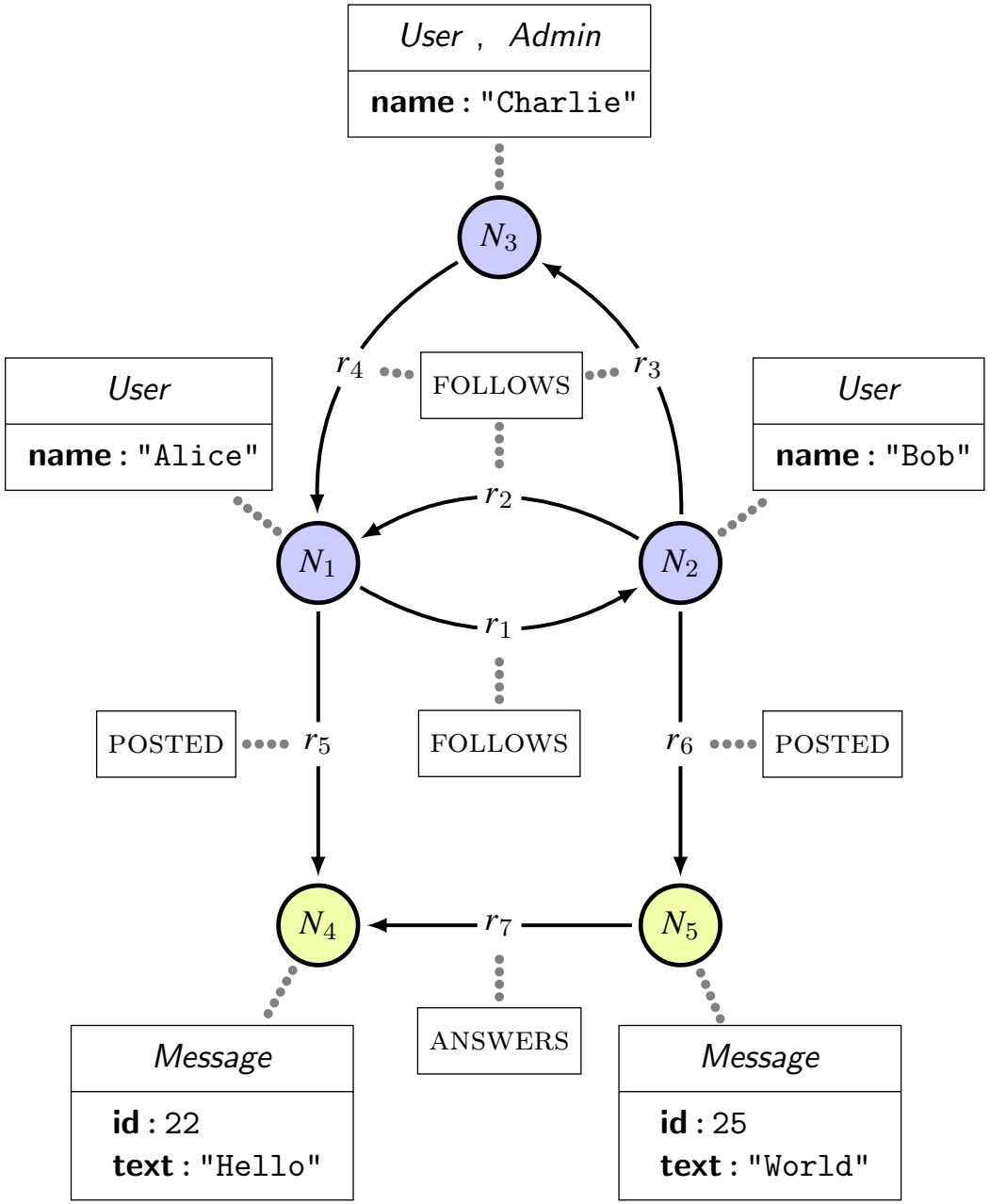
Query :

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

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)
```

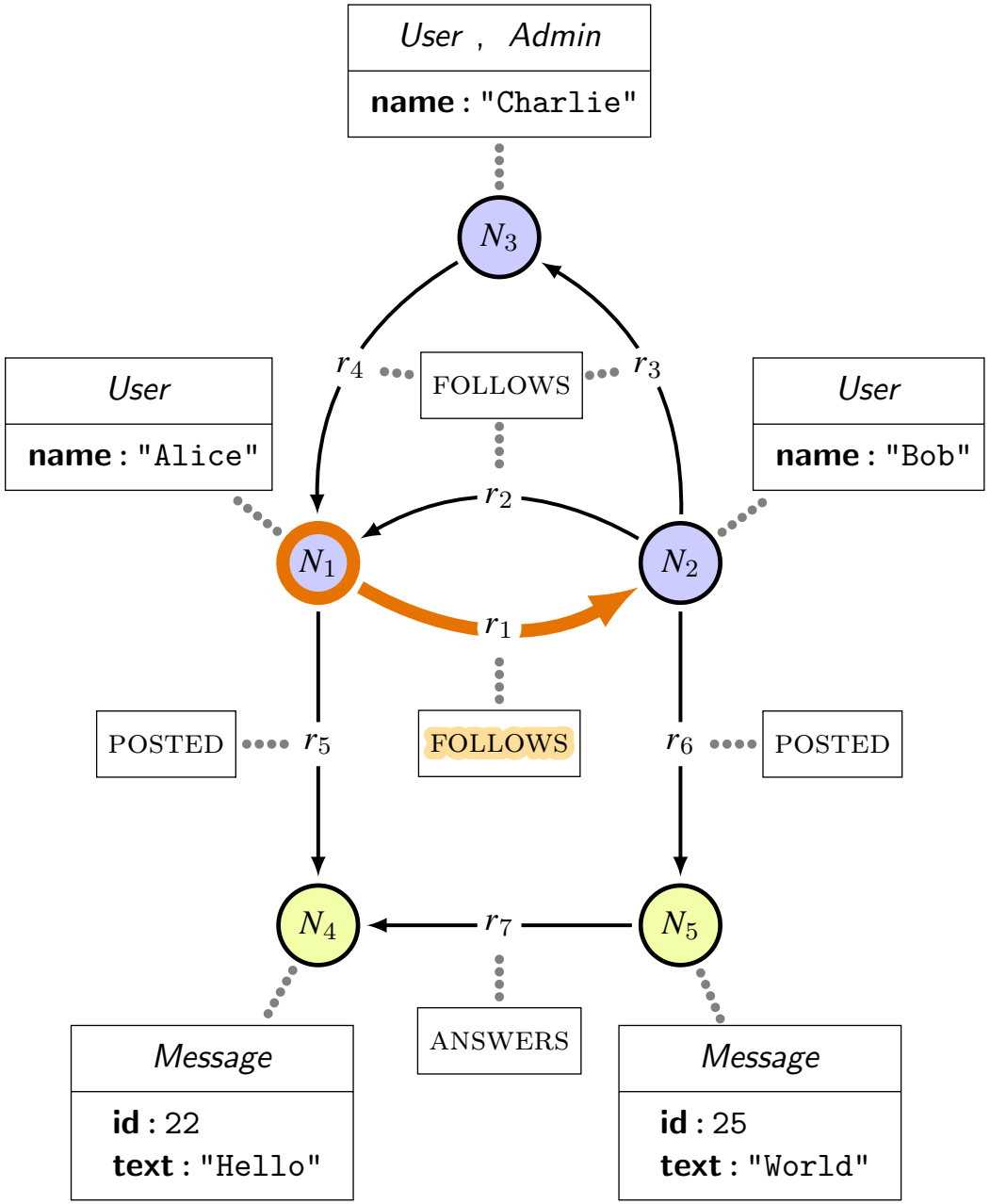
After first `MATCH` :

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	·	·

Chaining clauses



Query :

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

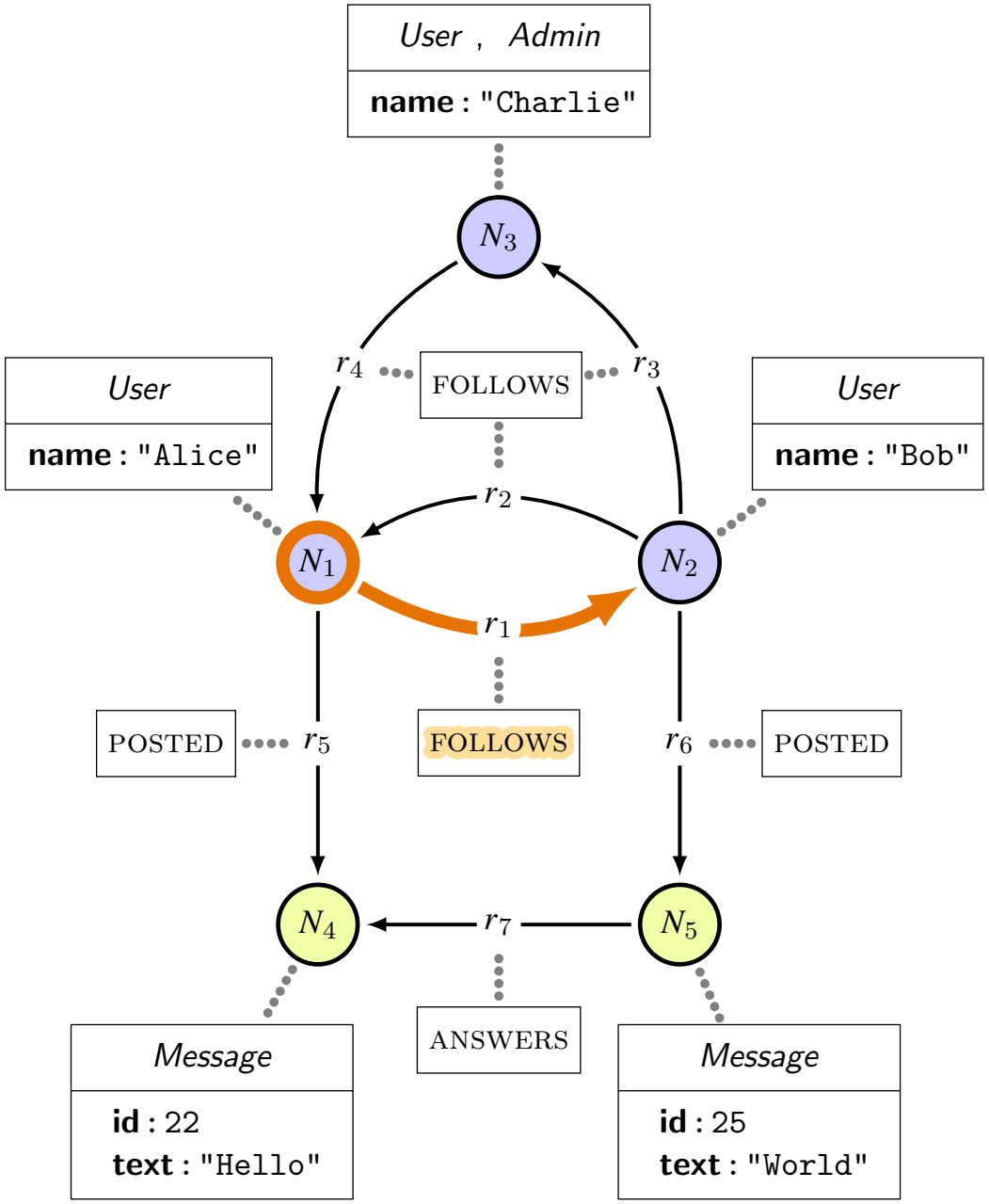
After first `MATCH` :

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	·	·

Chaining clauses



Query :

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

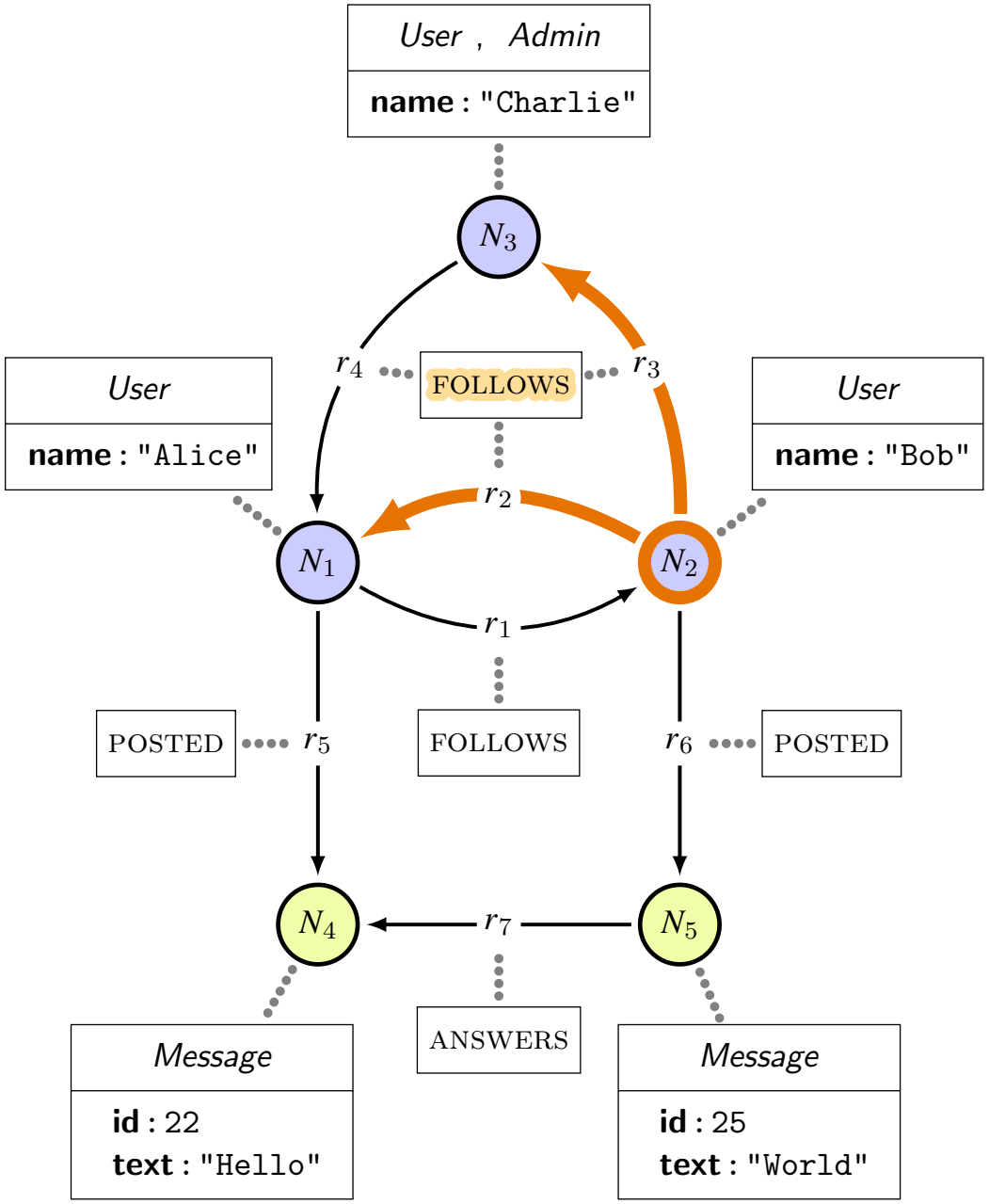
After first `MATCH` :

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	.	.

Chaining clauses



Query :

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

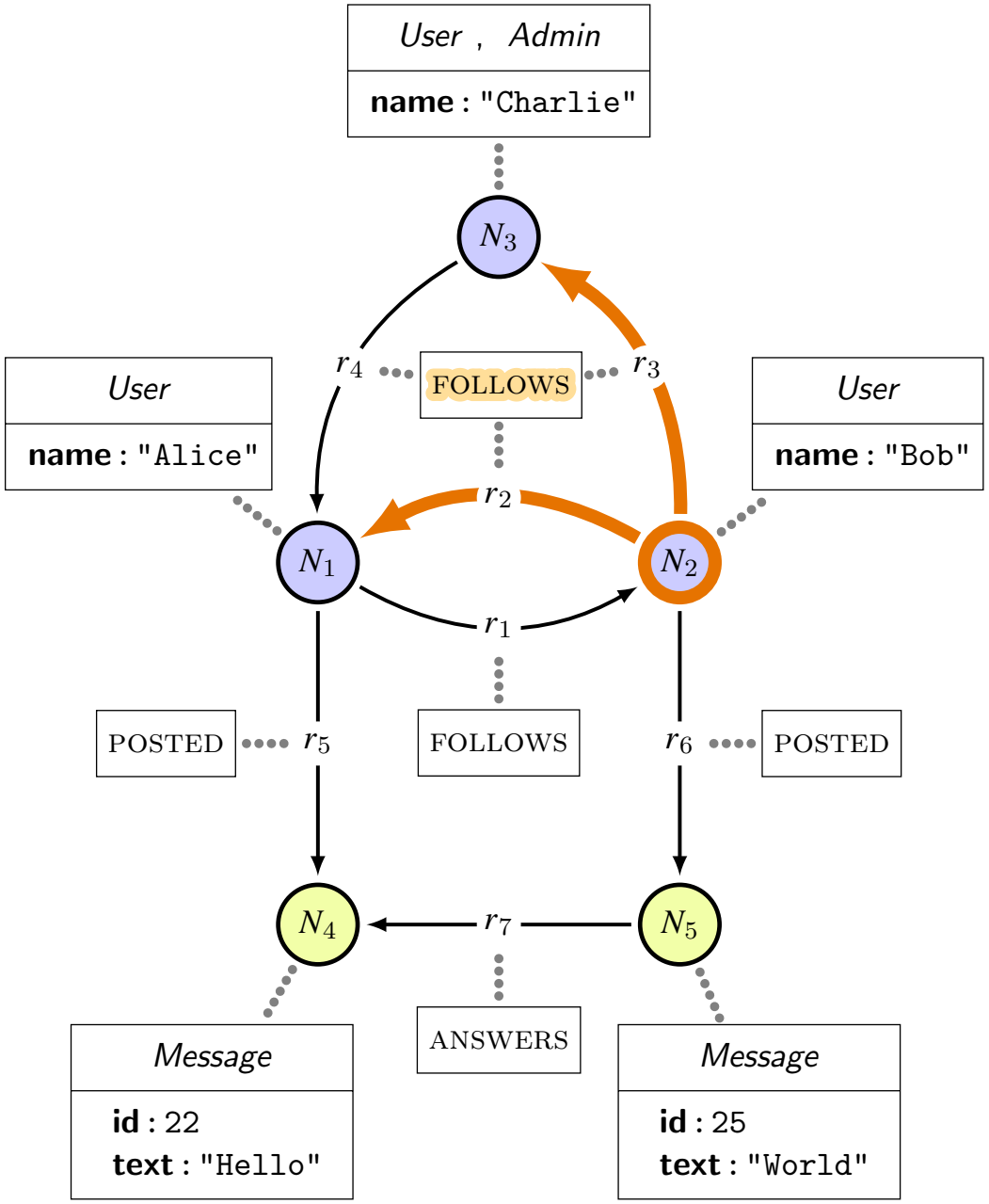
After first `MATCH` :

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	·	·

Chaining clauses



Query :

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

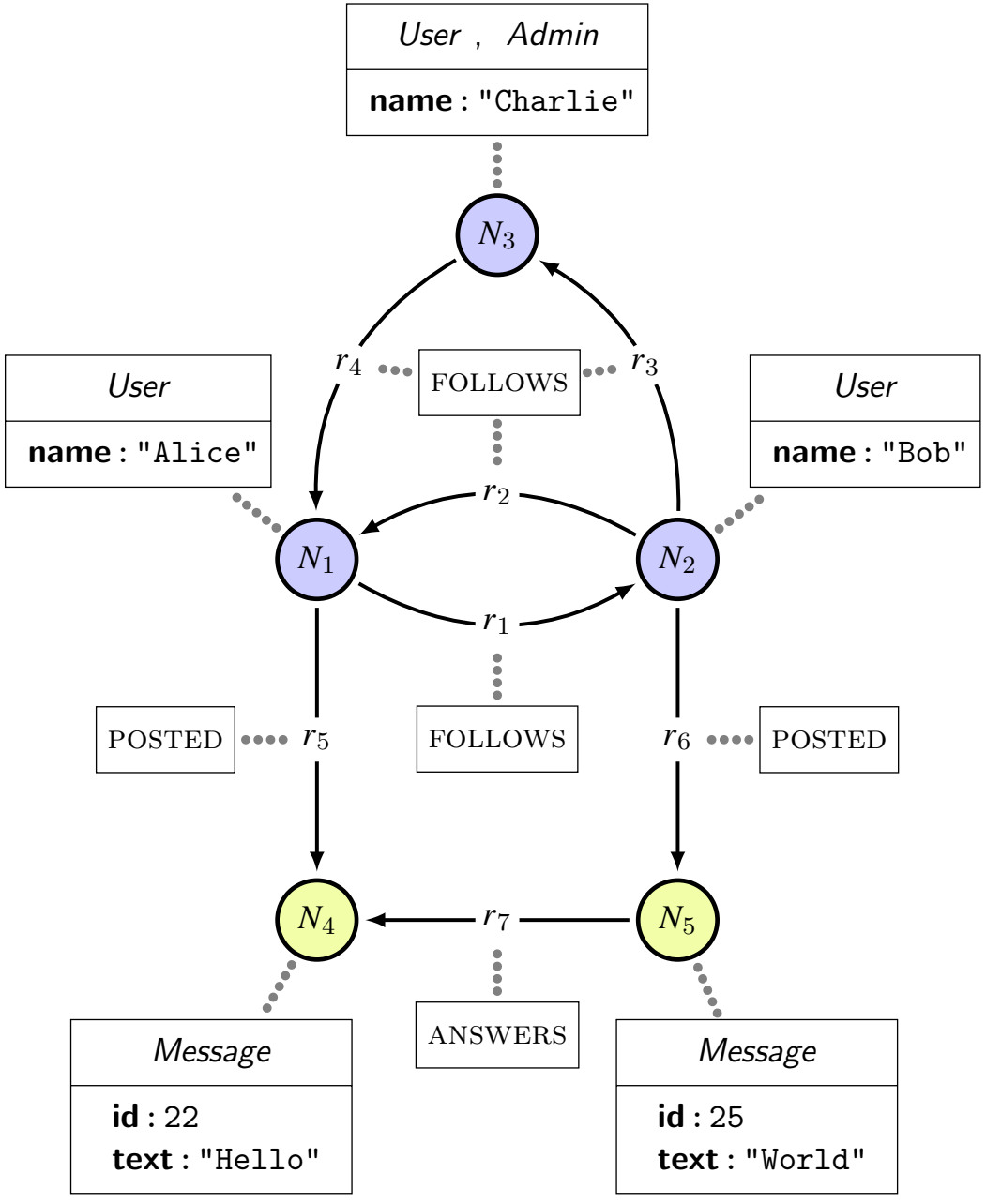
After first `MATCH` :

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

Column manipulation (WITH clause)



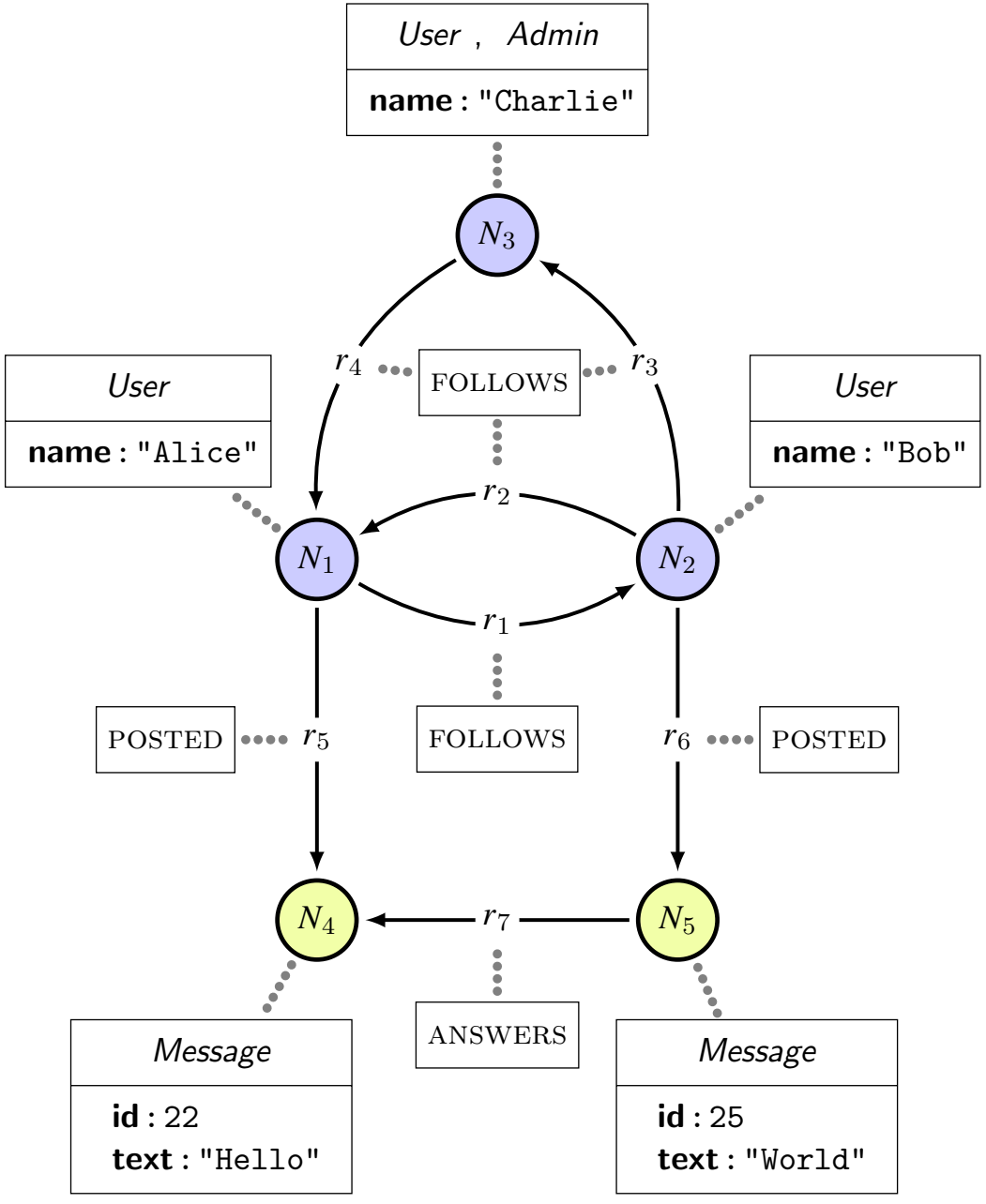
Query :

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

After the MATCH clause :

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Column manipulation (WITH clause)



Query :

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

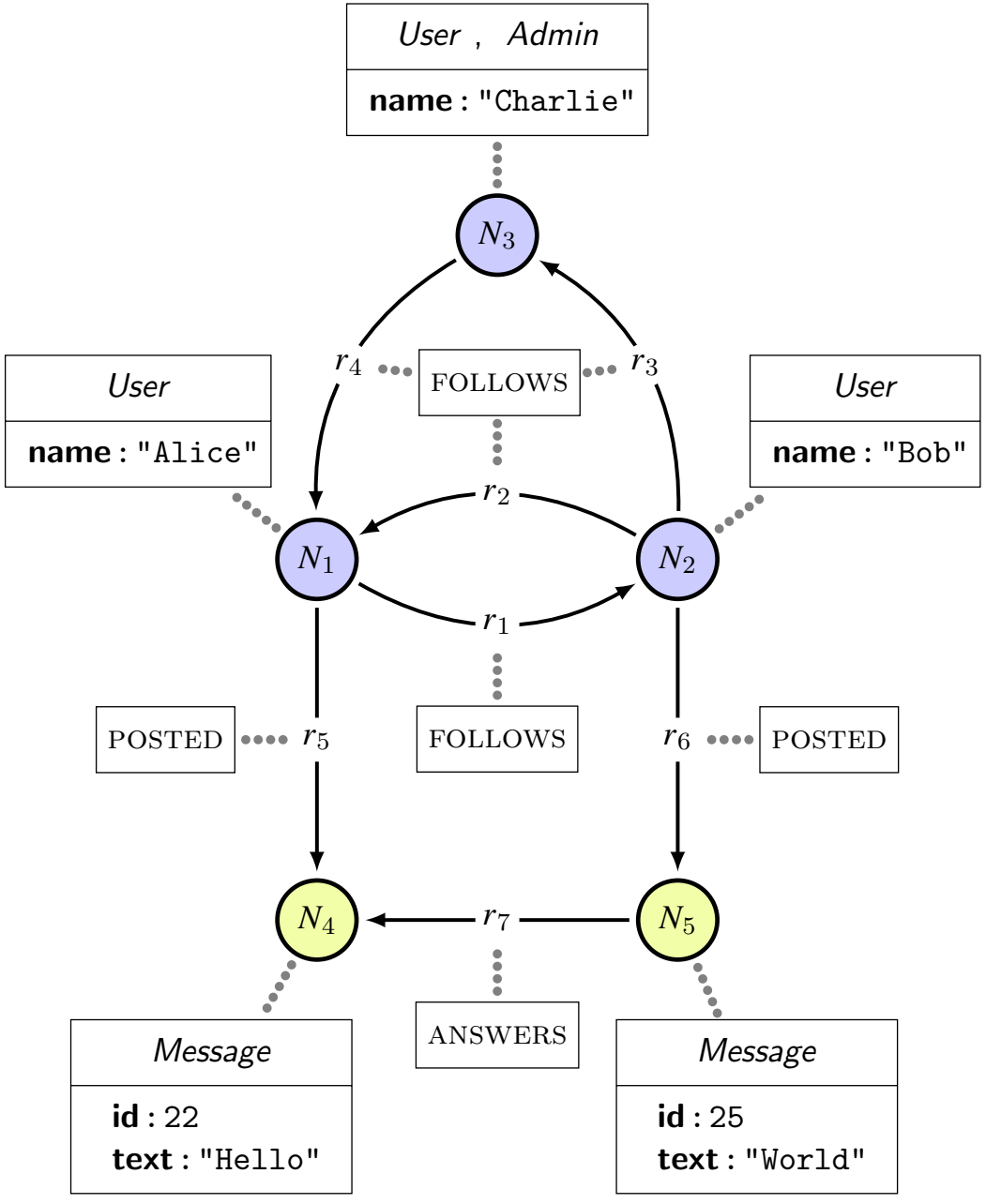
After the MATCH clause :

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Execution of the WITH clause :

u1	p1	t1
N_1	r_5	
N_2	r_6	

Column manipulation (WITH clause)



Query :

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

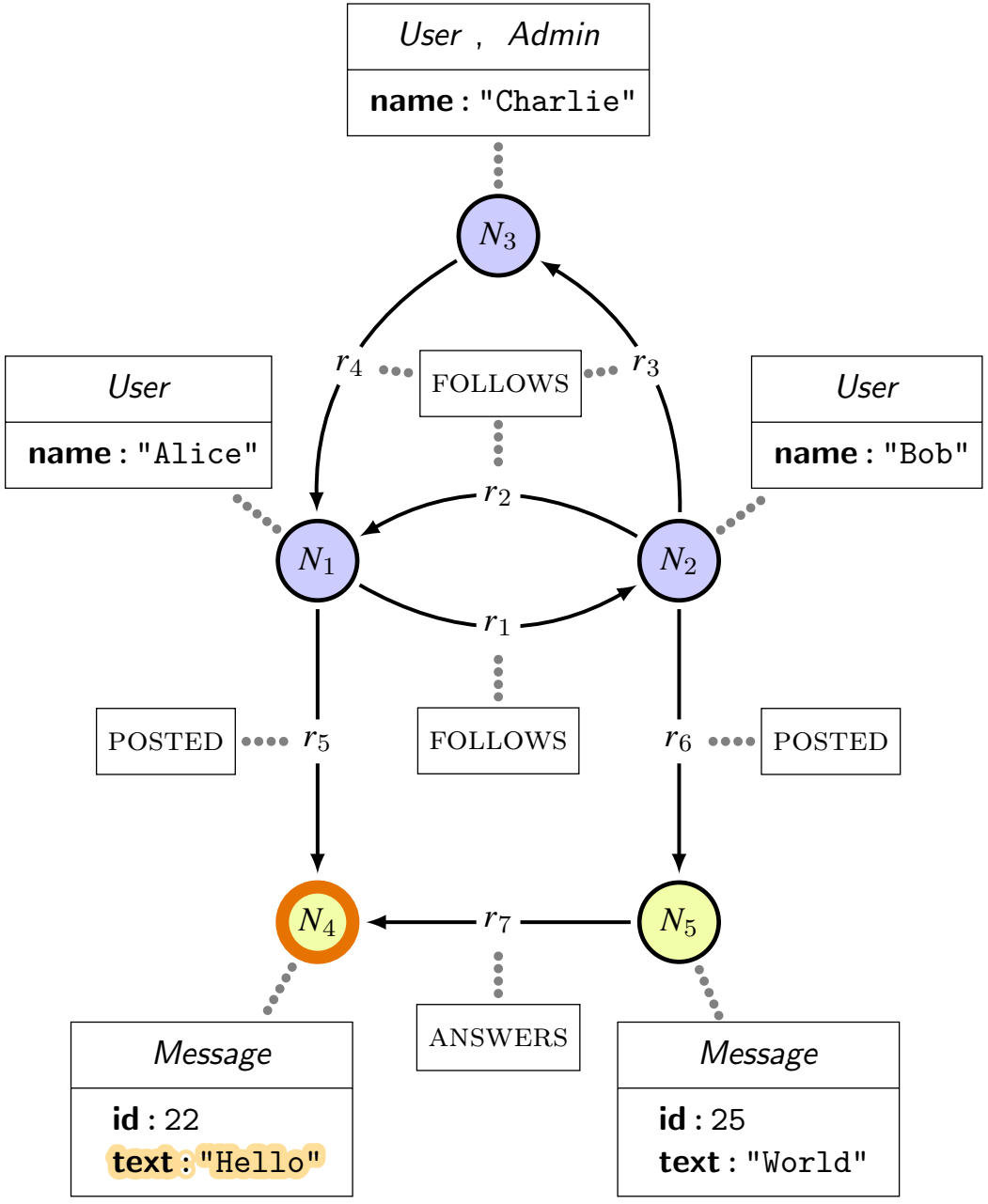
After the MATCH clause :

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Execution of the WITH clause :

u1	p1	t1
N_1	r_5	
N_2	r_6	

Column manipulation (WITH clause)



Query :

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

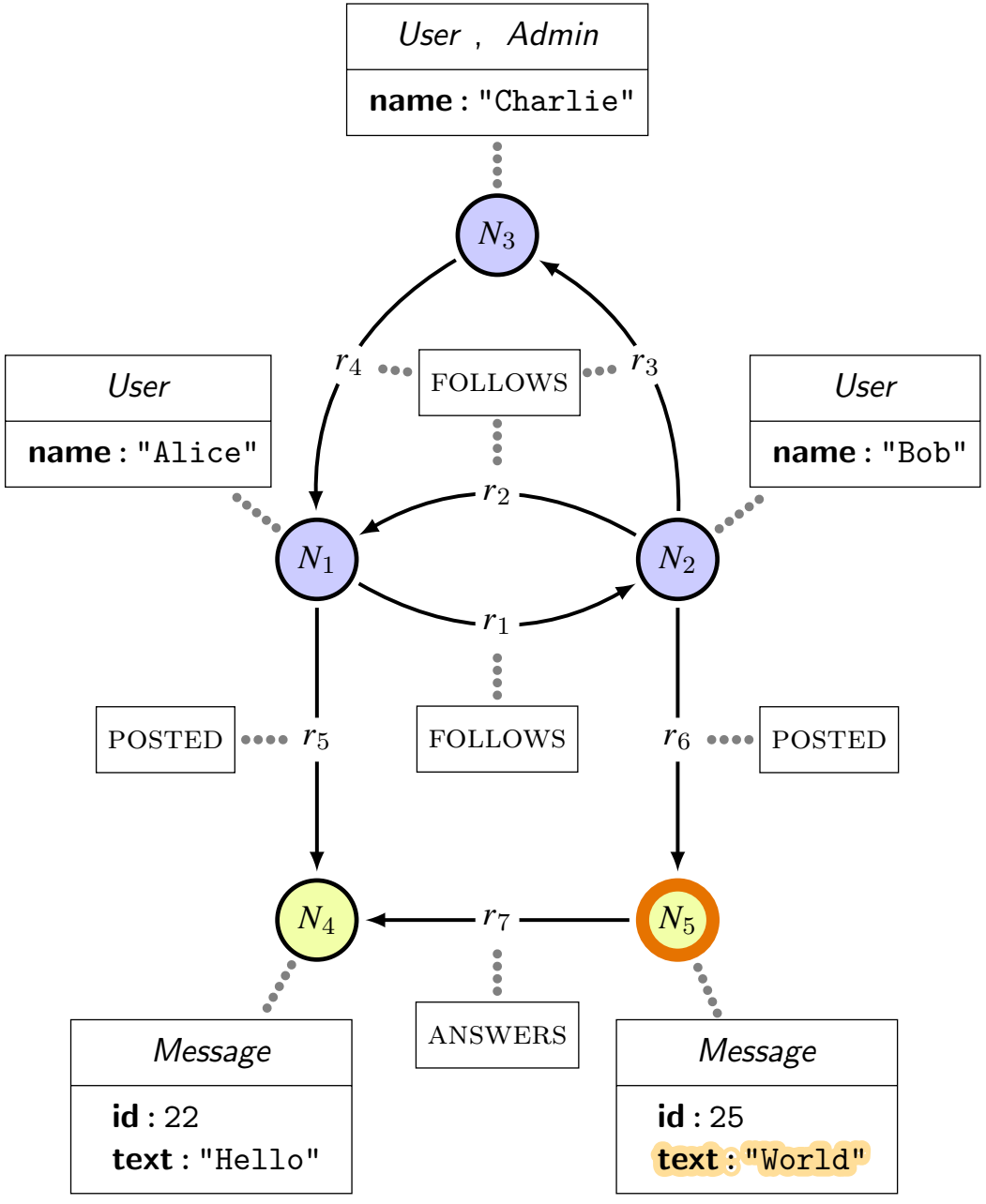
After the MATCH clause :

u1	p1	m1
N_1	r_5	N_4
N_2	r_6	N_5

Execution of the WITH clause :

u1	p1	t1
N_1	r_5	"Hello"
N_2	r_6	

Column manipulation (WITH clause)



Query :

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

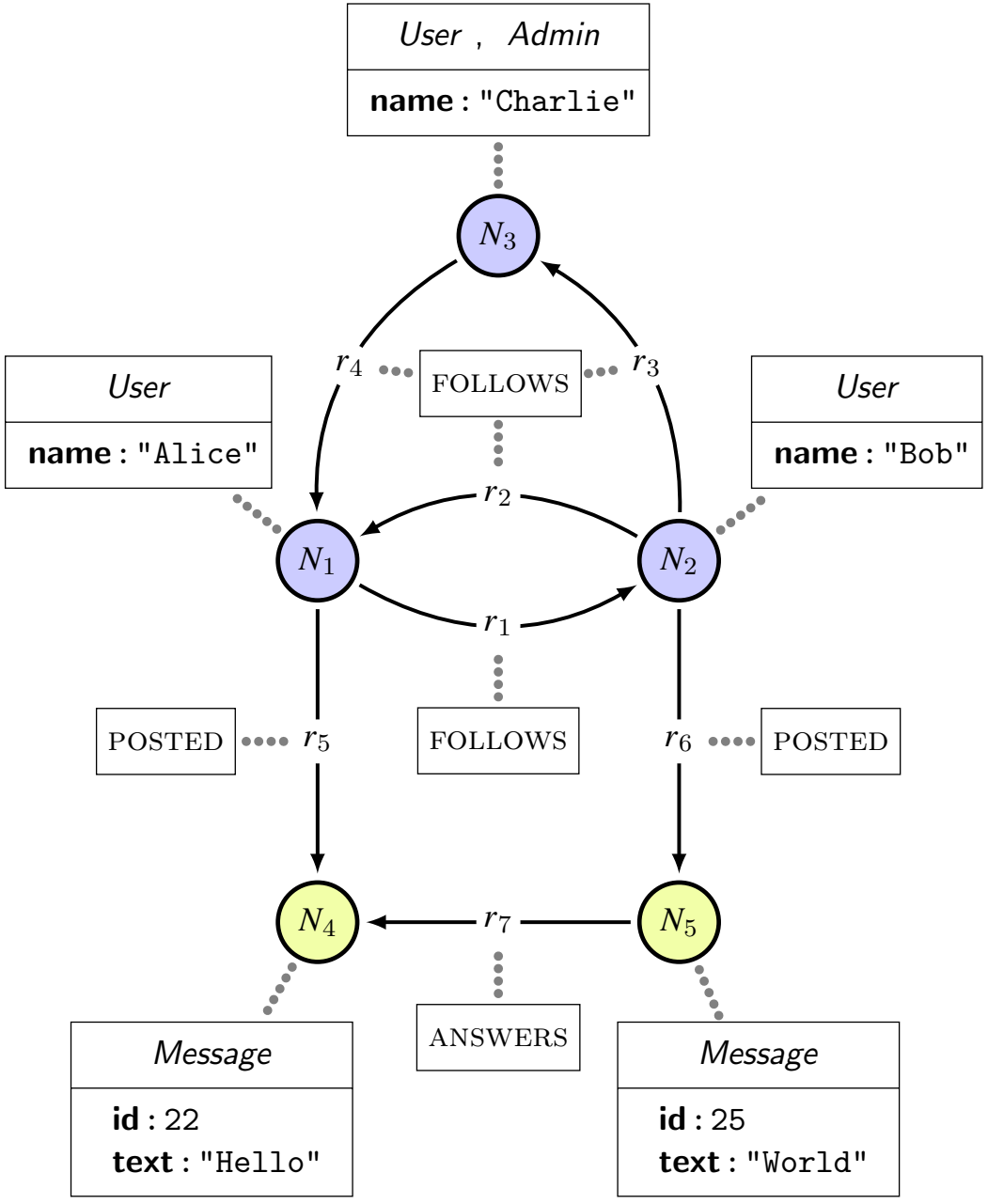
After the MATCH clause :

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
```

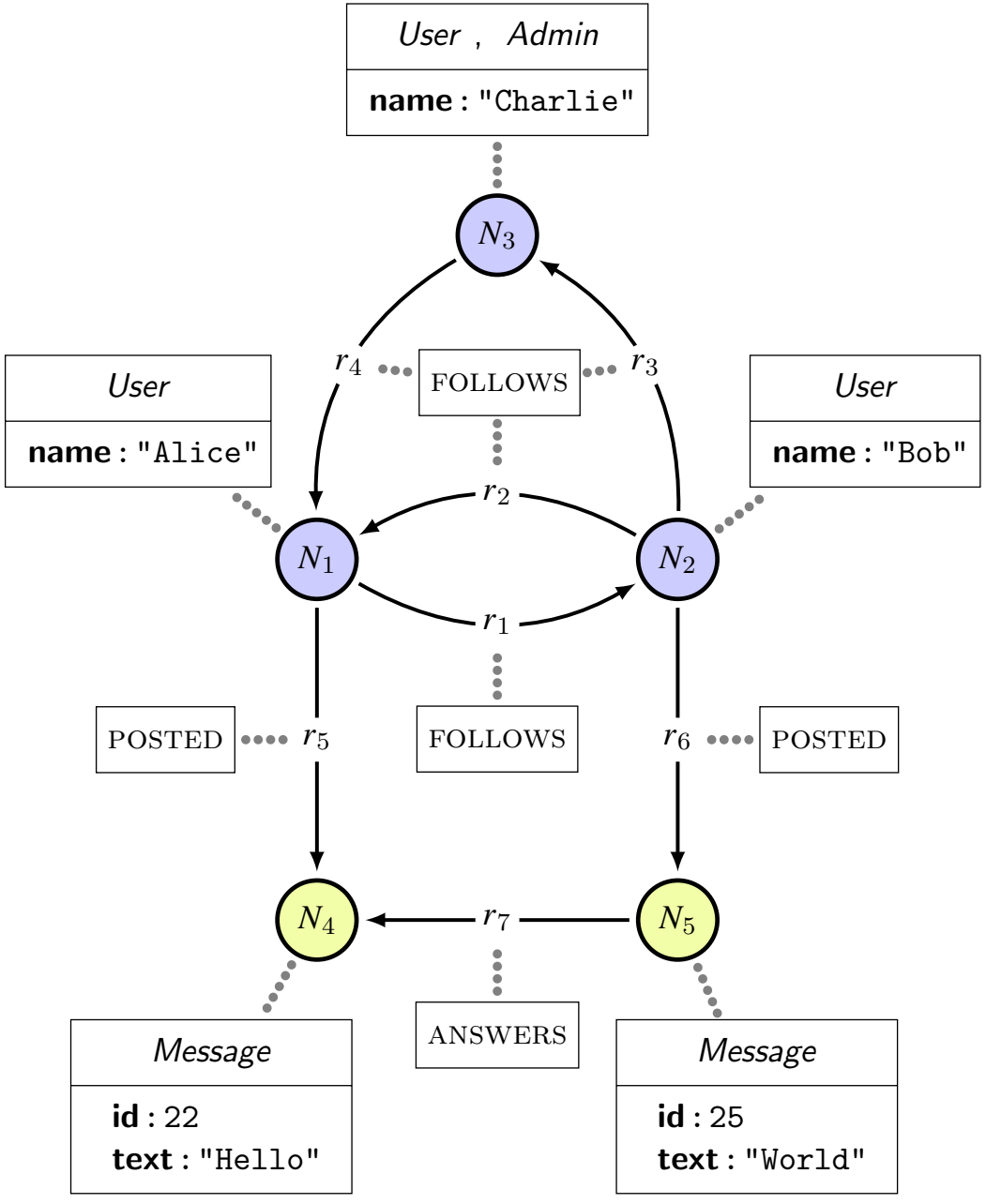
After the MATCH clause :

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)



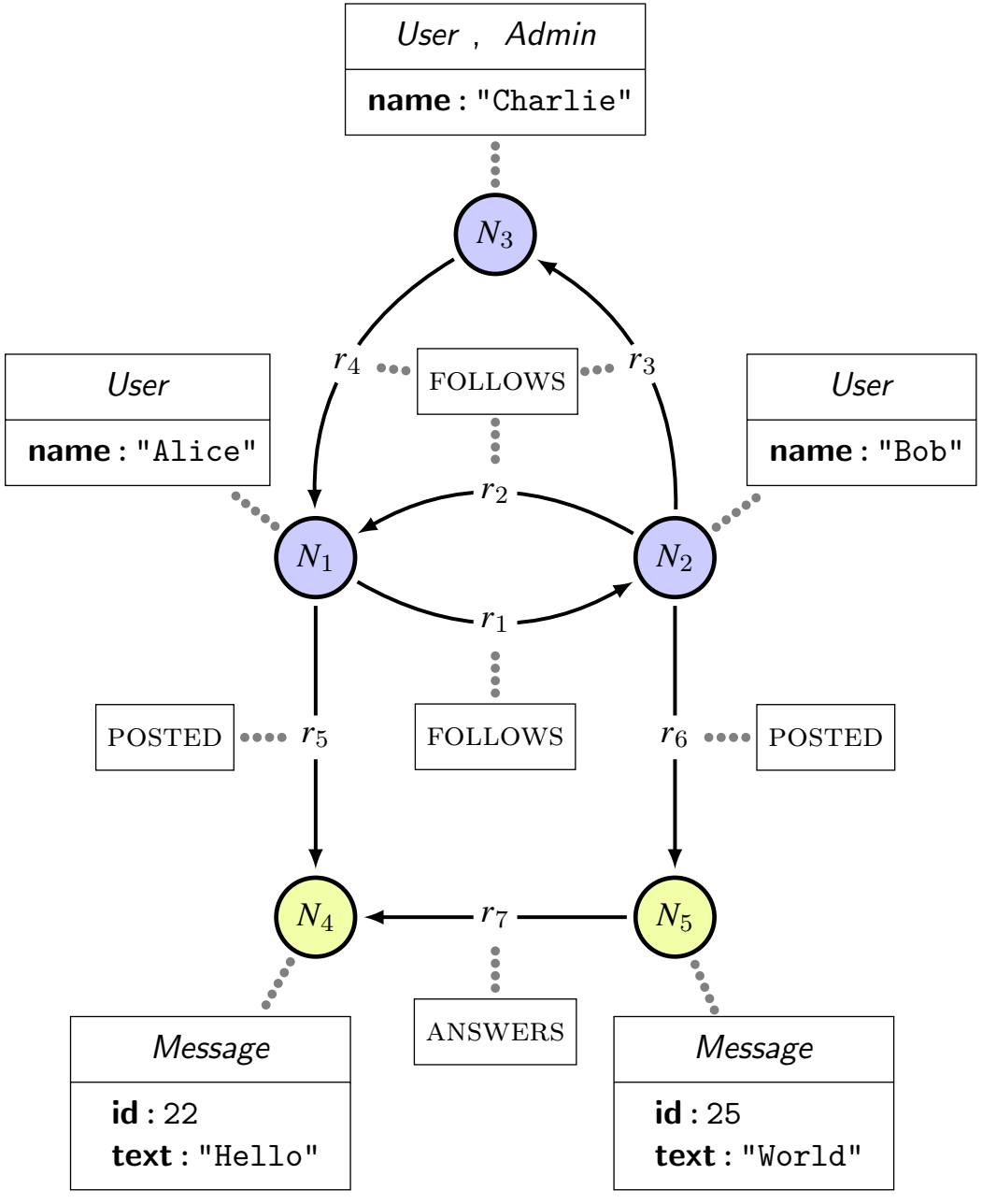
Query :

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

Après le WITH :

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"
```

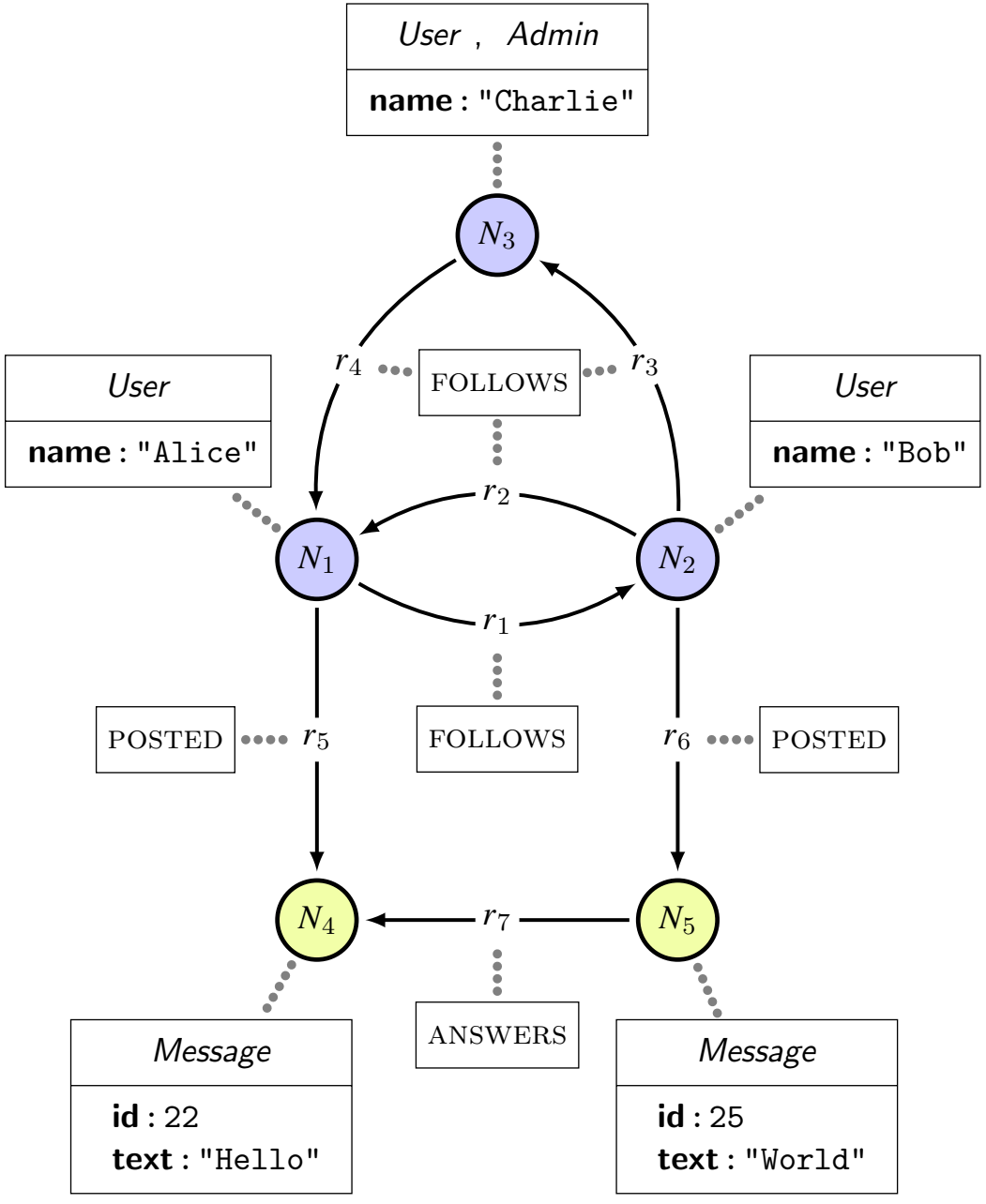
Après le WITH :

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"
~~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"
```

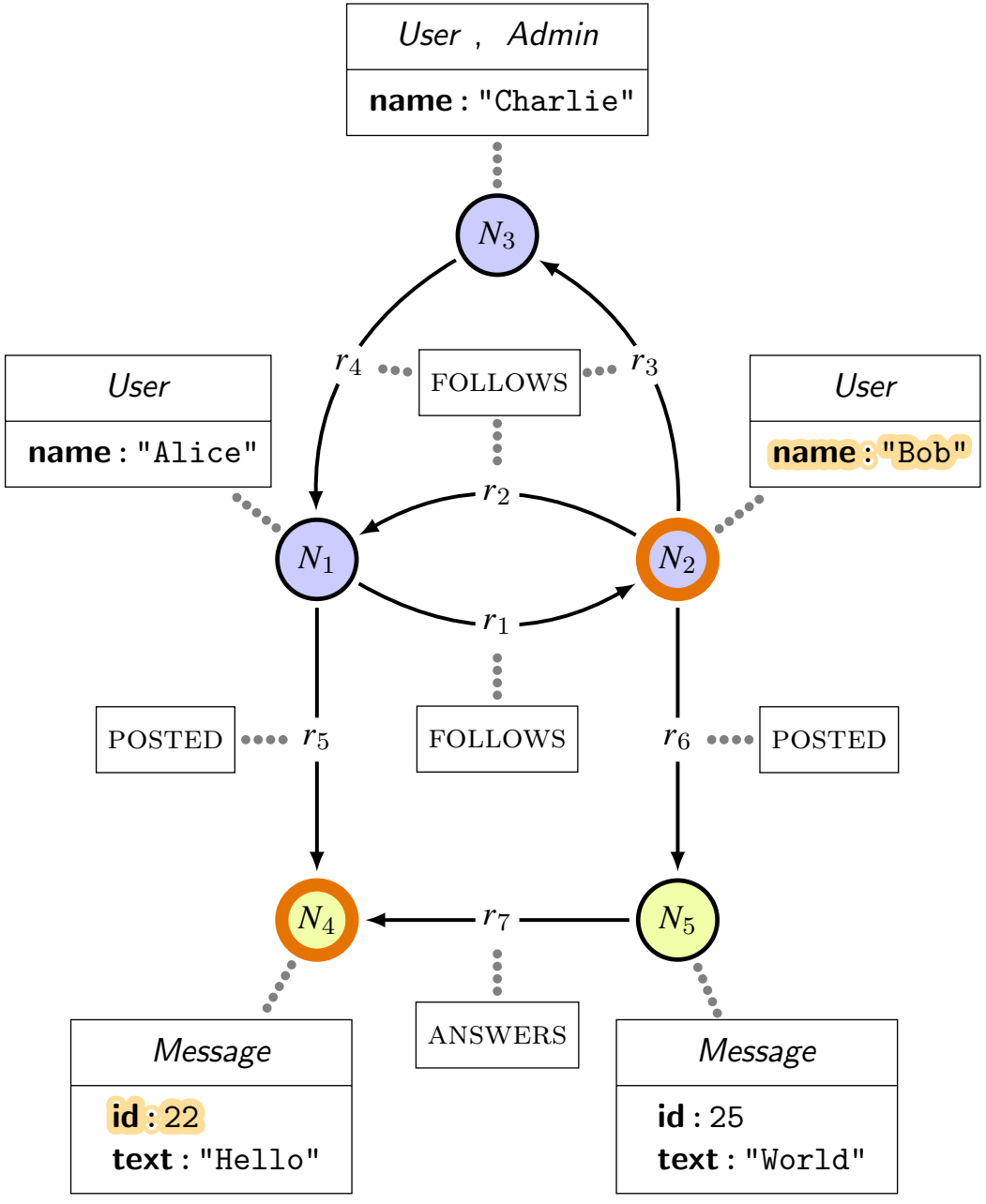
Après le WITH :

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

Final result :

u1	p1	t1
N_1	r_5	"Hello"

A last example



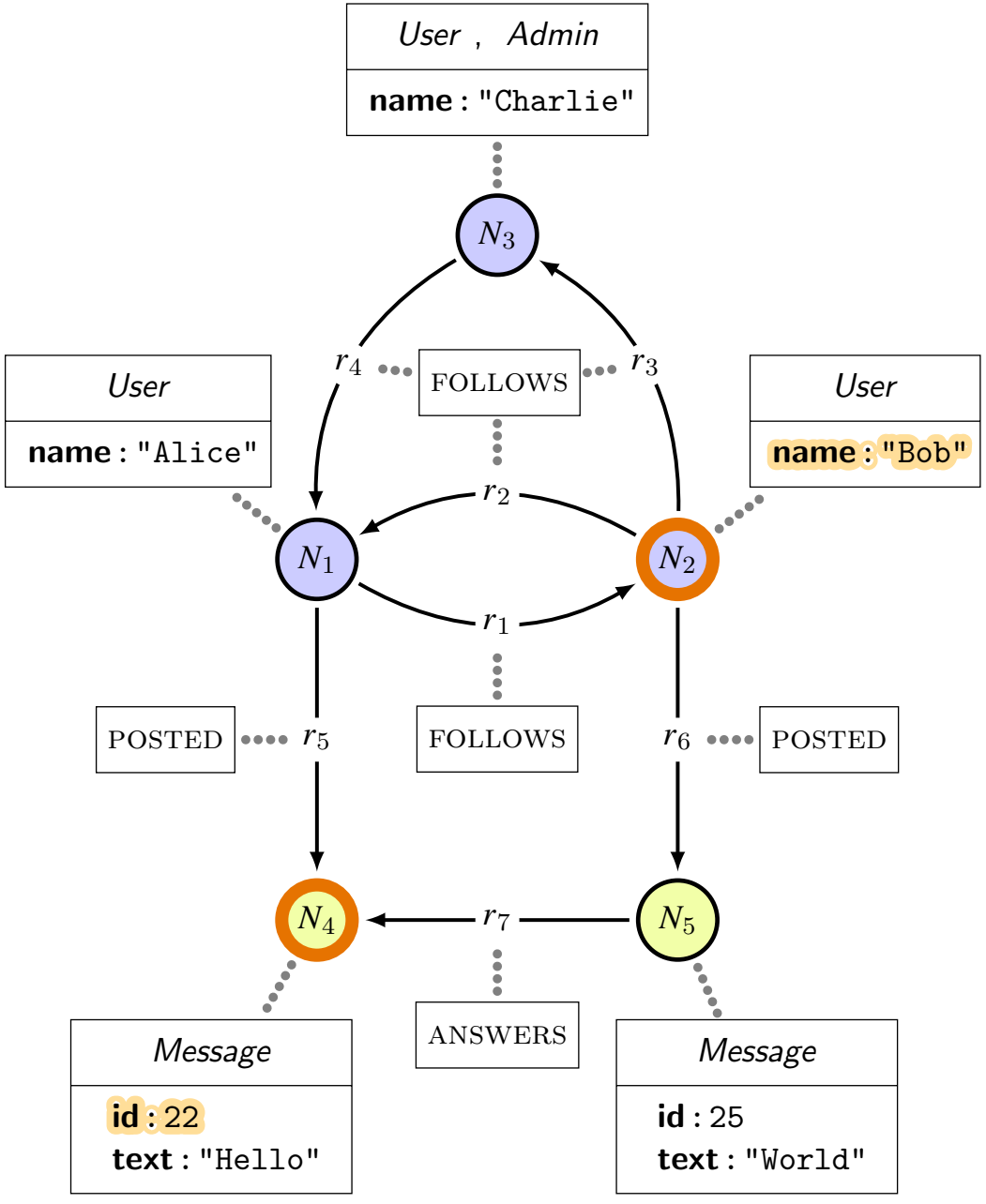
Query :

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

Question :

What does this computes ?

A last example



Query :

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

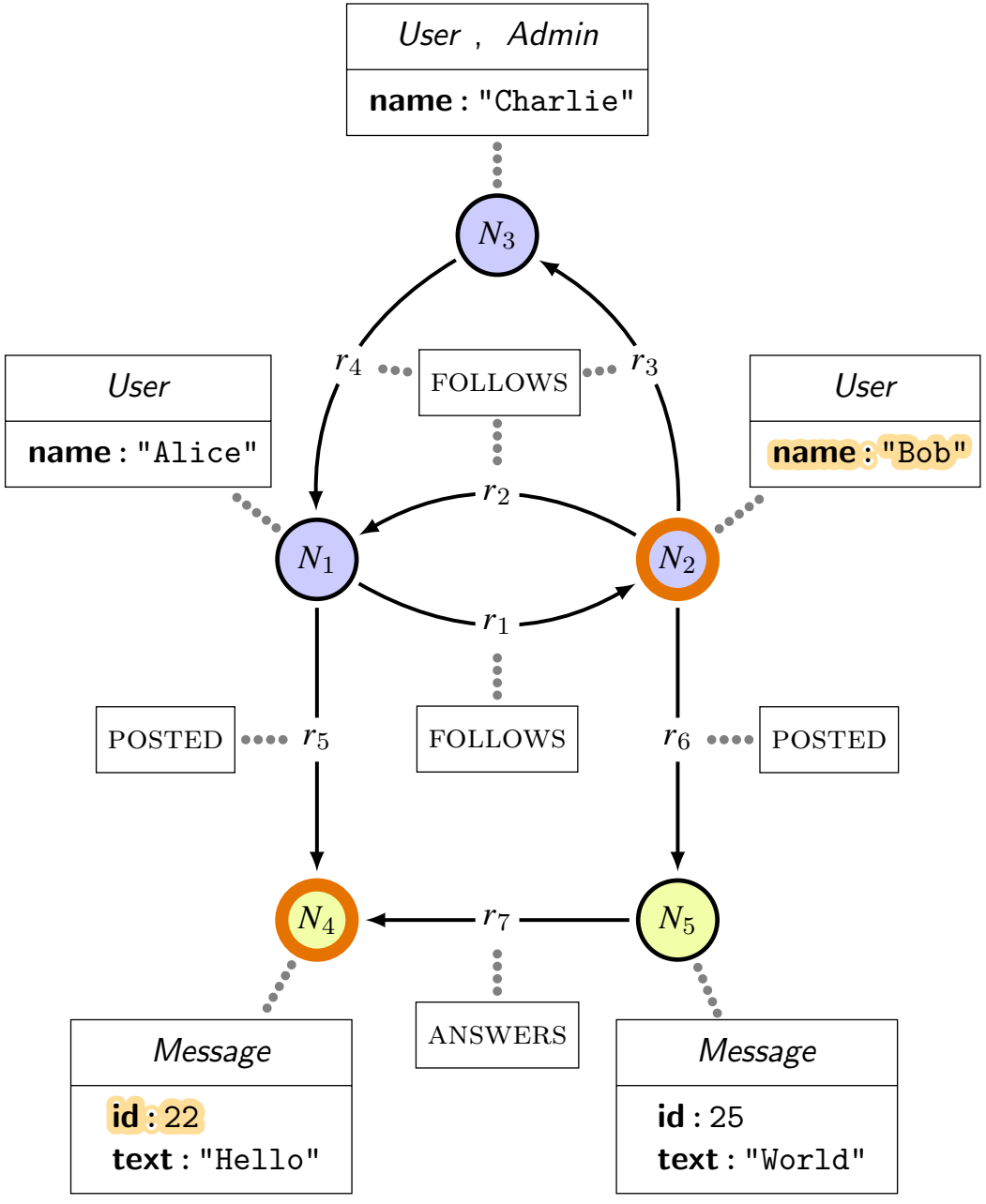
Question :

What does this computes ?

Answer :

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

A last example



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.

$\Rightarrow$ Evaluation of Cypher queries is NP-HARD

Outline

Record (table row)

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

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

Record (table row)

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

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

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

Table

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

Example:

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

=

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

Semantics brackets

G: a graph

Semantics of expressions

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

Semantics of clauses

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

Semantics of queries

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

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

Principle of the 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 the 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 $\llbracket C_1 \rrbracket_G$, $\llbracket C_2 \rrbracket_G$, $\ldots$, $\llbracket C_n \rrbracket_G$

Principle of the 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 $[\![C_1]\!]_G\,,\quad [\![C_2]\!]_G\,,\quad \cdots\,,\quad [\![C_n]\!]_G$
- Let $[\![Q]\!]_G = [\![C_n]\!]_G \circ \cdots \circ [\![C_2]\!]_G \circ [\![C_1]\!]_G$

Principle of the 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 $[\![C_1]\!]_G\,,\quad [\![C_2]\!]_G\,,\quad \ldots\,,\quad [\![C_n]\!]_G$
- 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.

Outline

Node patterns

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

Examples

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

Relationship patterns

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

Example

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

Path patterns

⟨pattern⟩ ::=
⟨node_pattern⟩ [<-? ⟨rel_pattern⟩ ->? ⟨node_pattern⟩]*

Examples

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

Rigid path-patterns

Definition

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

Examples

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

Rigid path-patterns

Definition

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

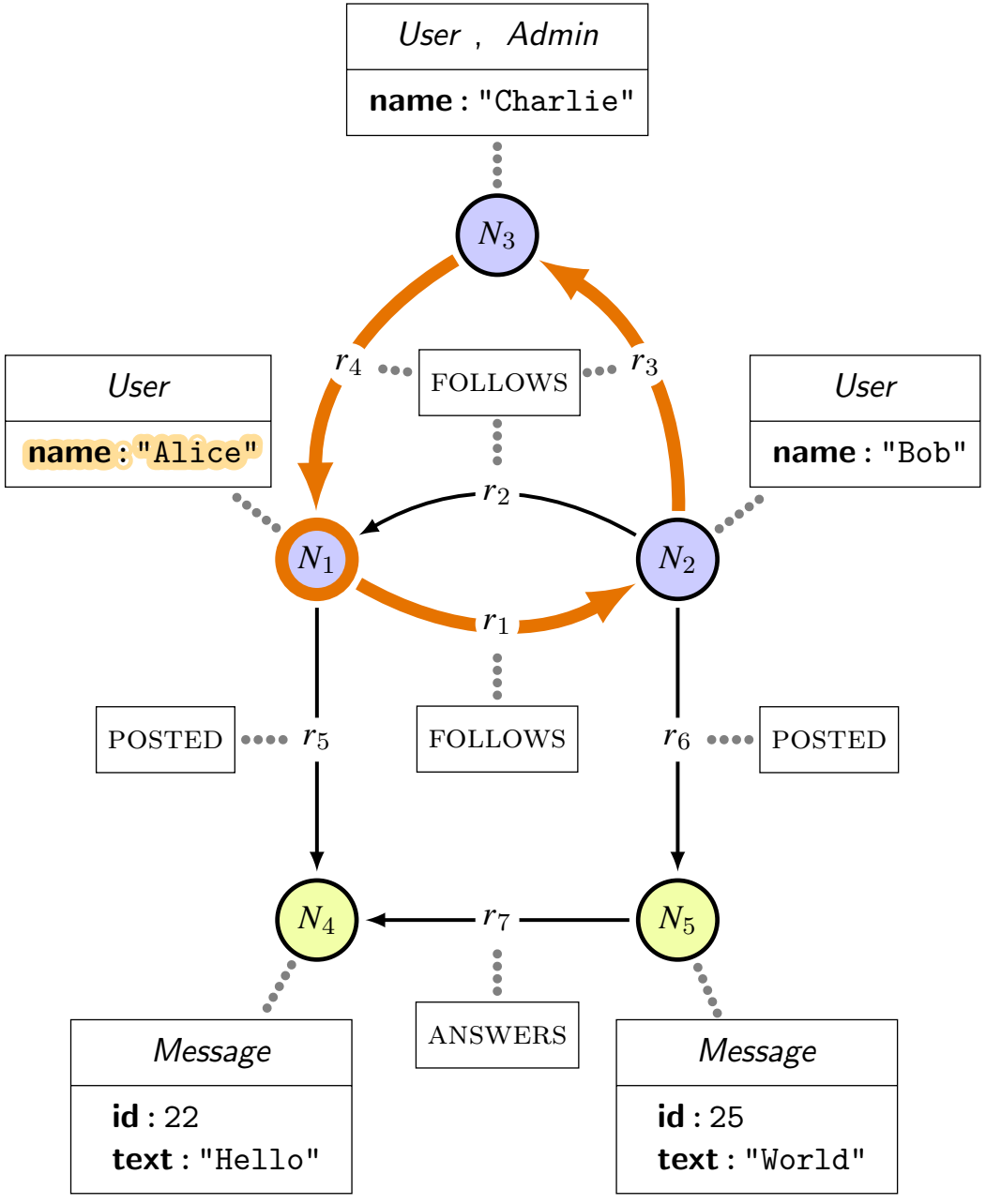
Examples

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

Property

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

Rigid vs flexible path (1)



Query :

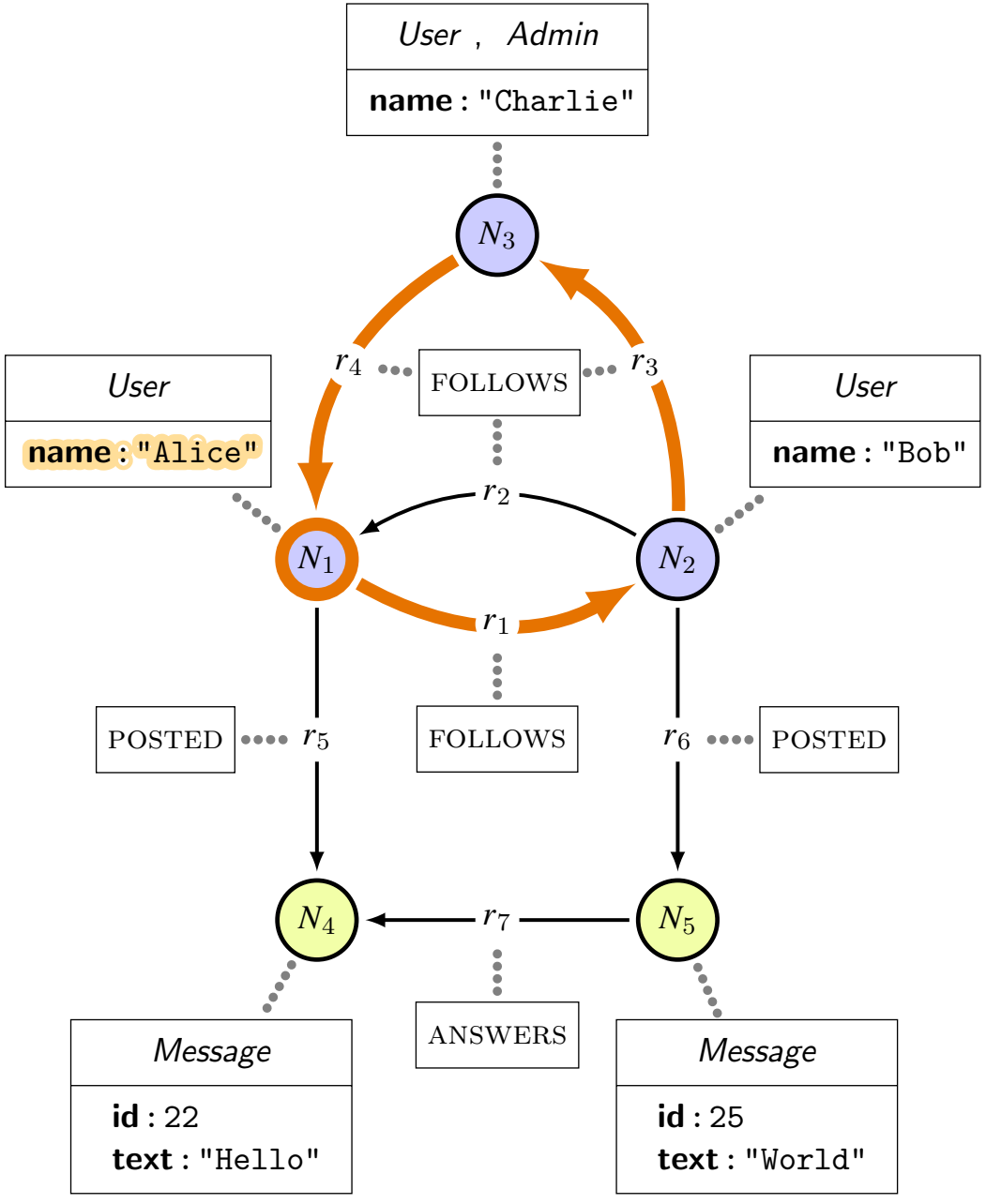
```
MATCH (x {name:Alice})
      -[:FOLLOWS*1..1]->(mid)
      -[:FOLLOWS*2..2]->(x)
```

The path

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

satifies the pattern.

Rigid vs flexible path (1)



Query :

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

The path

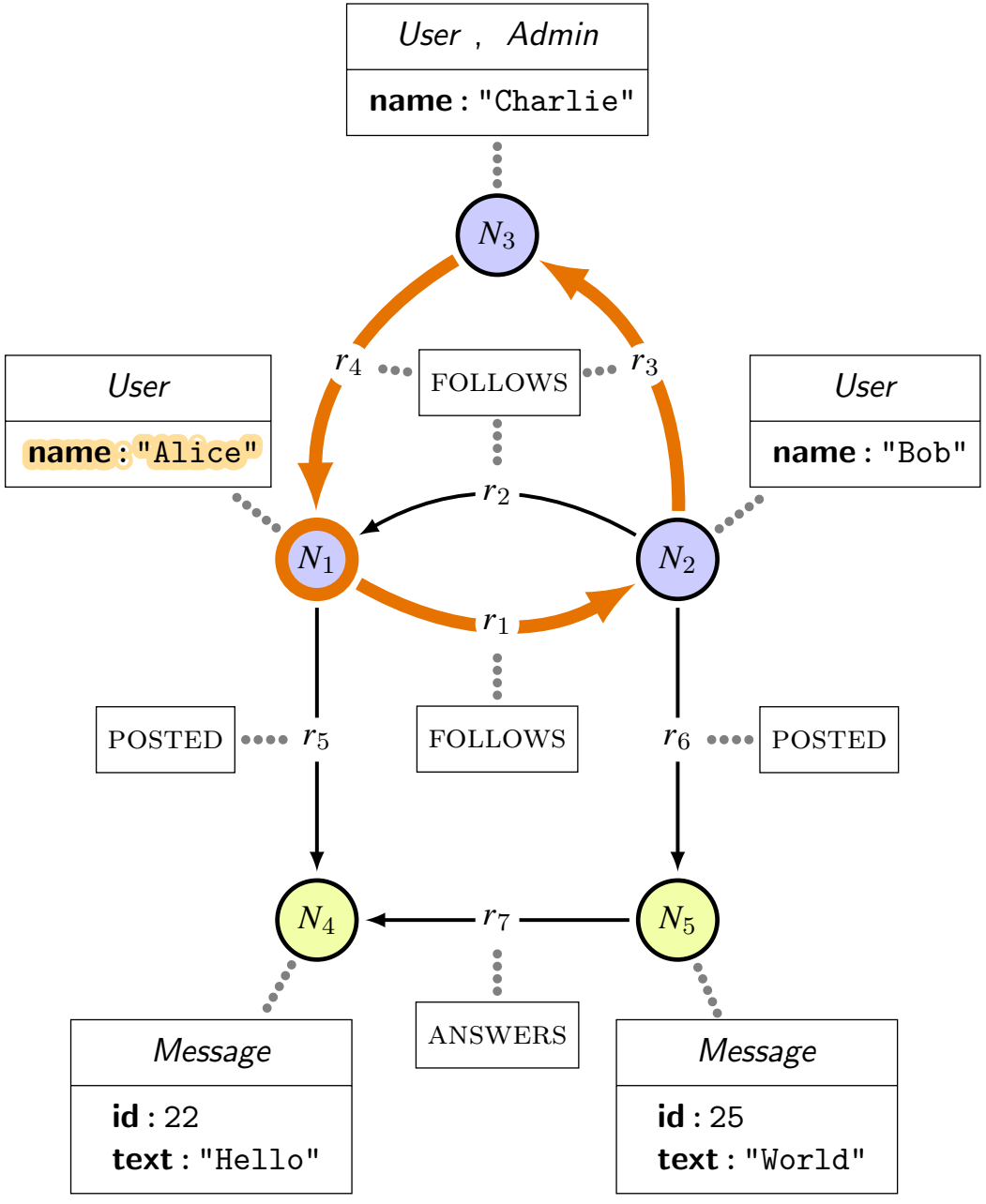
$$N_1 \xrightarrow{r_1} N_2 \xrightarrow{r_3} N_3 \xrightarrow{r_4} N_1$$

satifies the pattern.

Variables are uniquely set

x	mid
N_1	N_2

Rigid vs flexible path (2)



Query :

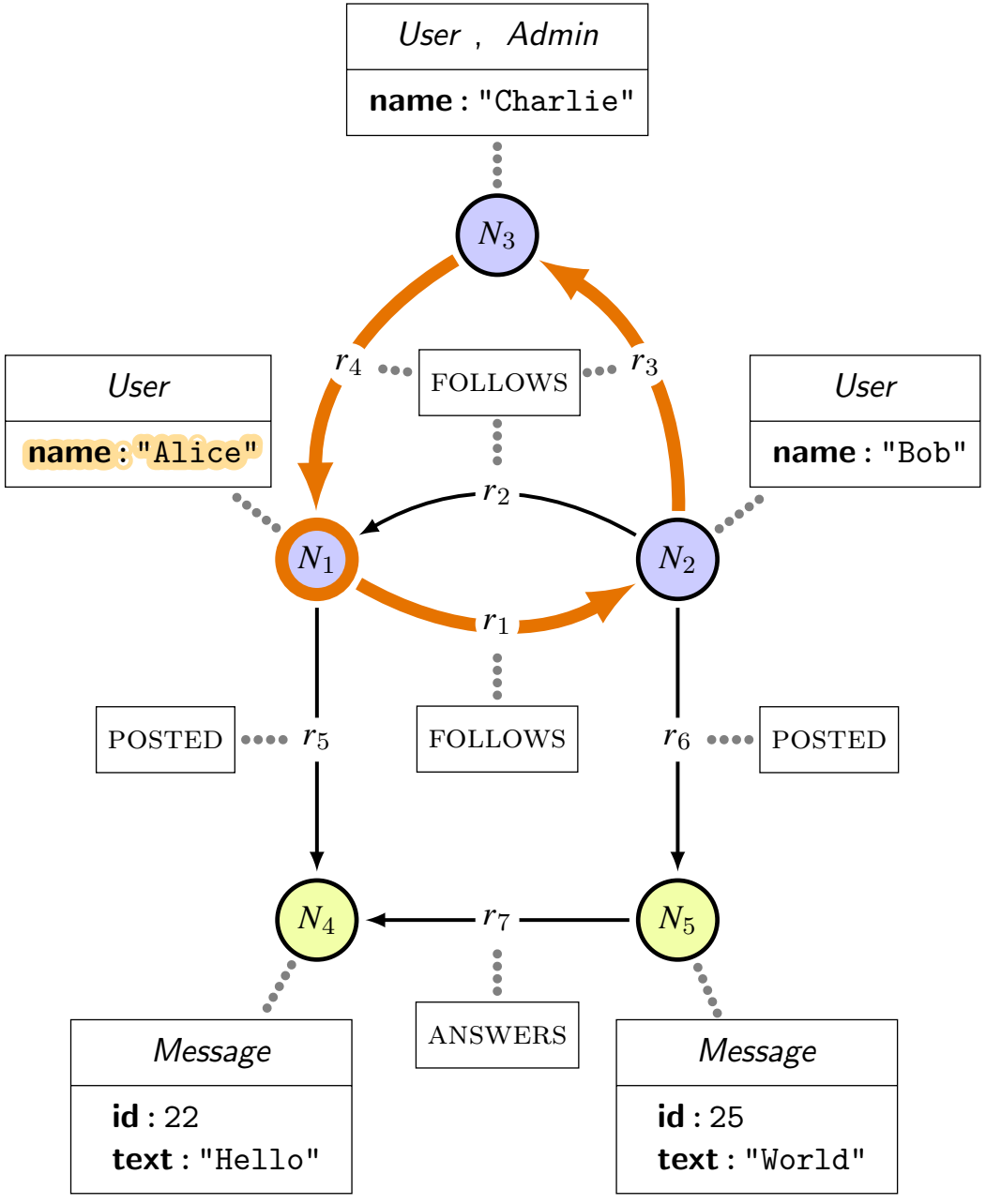
```
MATCH (x {name:"Alice"})
      -[:FOLLOWS*1..2]->(mid)
      -[:FOLLOWS*1..2]->(x)
```

The path

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

satifies this second pattern.

Rigid vs flexible path (2)



Query :

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

The path

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

satifies this second pattern.

Two possible assignments

x	mid
N_1	N_2
N_1	N_3

Path satisfying a rigid path-pattern

u: a record
p: a path
π: a rigid path-pattern
n: length of π

Definition

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

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

Rigid path-pattern subsumed by a flexible path-pattern

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

Examples

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

Rigid path-pattern subsumed by a flexible path-pattern

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

Examples

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

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

Rigid path-pattern subsumed by a flexible path-pattern

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

Examples

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

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

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

Rigid path-pattern subsumed by a flexible path-pattern

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

Examples

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

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

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

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

The MATCH clause

Semantics of MATCH

$$\llbracket \texttt{MATCH}\ \pi \rrbracket_G : \quad T \quad \longmapsto \quad \biguplus_{u \in T} \mathsf{expand}_{G,\pi}(u)$$

The MATCH clause

Semantics of MATCH

$$\llbracket \texttt{MATCH}\ \pi \rrbracket_G : \quad T \quad \longmapsto \quad \biguplus_{u \in T} \mathsf{expand}_{G,\pi}(u)$$

Table-row expansion

π: a path-pattern
G: a graph

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

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

Outline

The core fragment

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⟩

Caveat on the MATCH clause (1)

Syntax

⟨match⟩ ::= [OPTIONAL]$^{?}$ MATCH ⟨pattern_tuple⟩

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

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

Caveat on the MATCH clause (2)

Semantics of MATCH

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

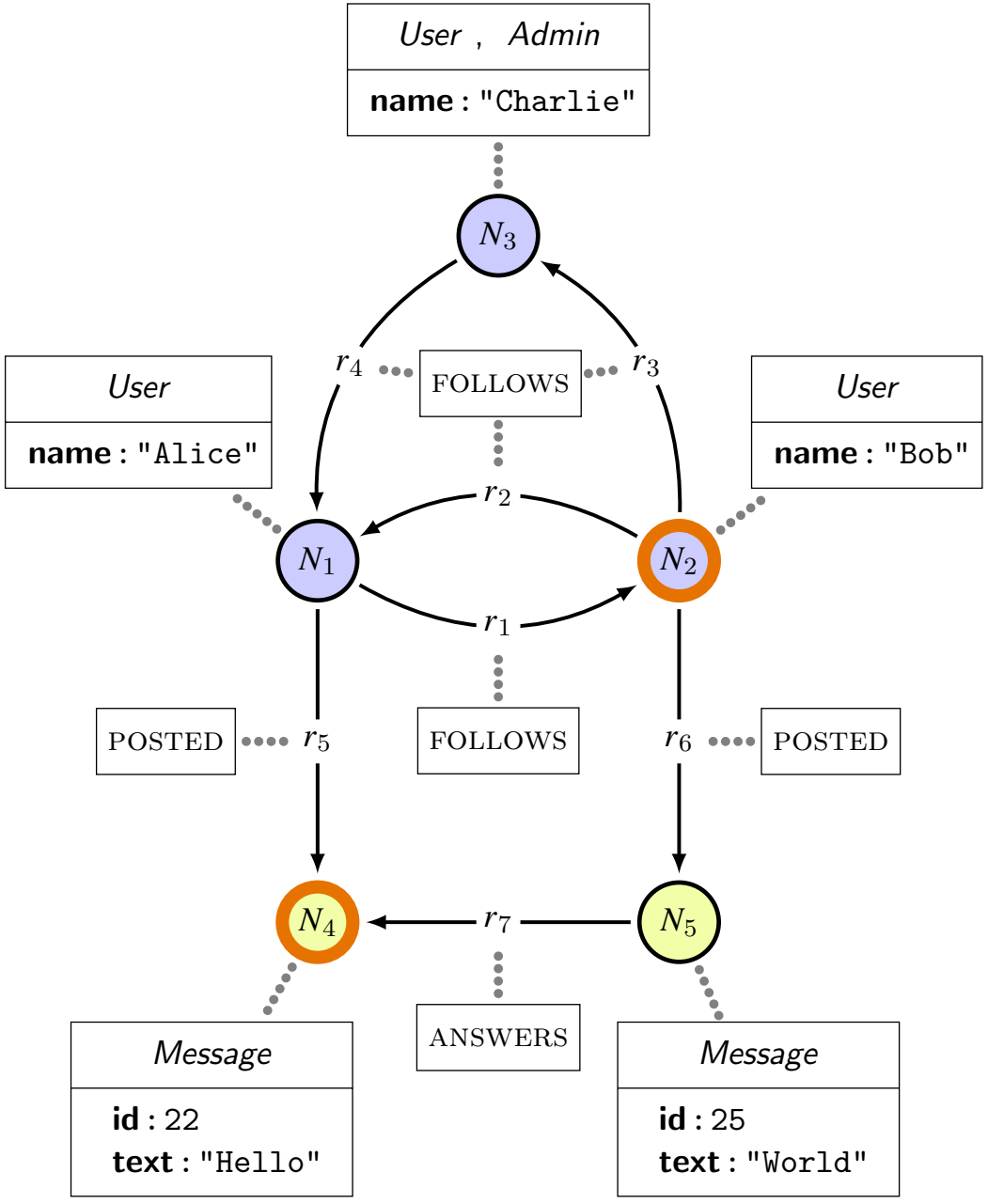
$$\text{rigid}(\bar{\pi}) = \text{rigid}(\pi_1) \times \text{rigid}(\pi_2) \times \cdots \times \text{rigid}(\pi_n)$$

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

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

$$\llbracket \texttt{MATCH } \bar{\pi} \rrbracket_G : \quad T \quad \longmapsto \quad \biguplus_{u \in T} expand_{G,\pi}(u)$$

Example of OPTIONAL MATCH



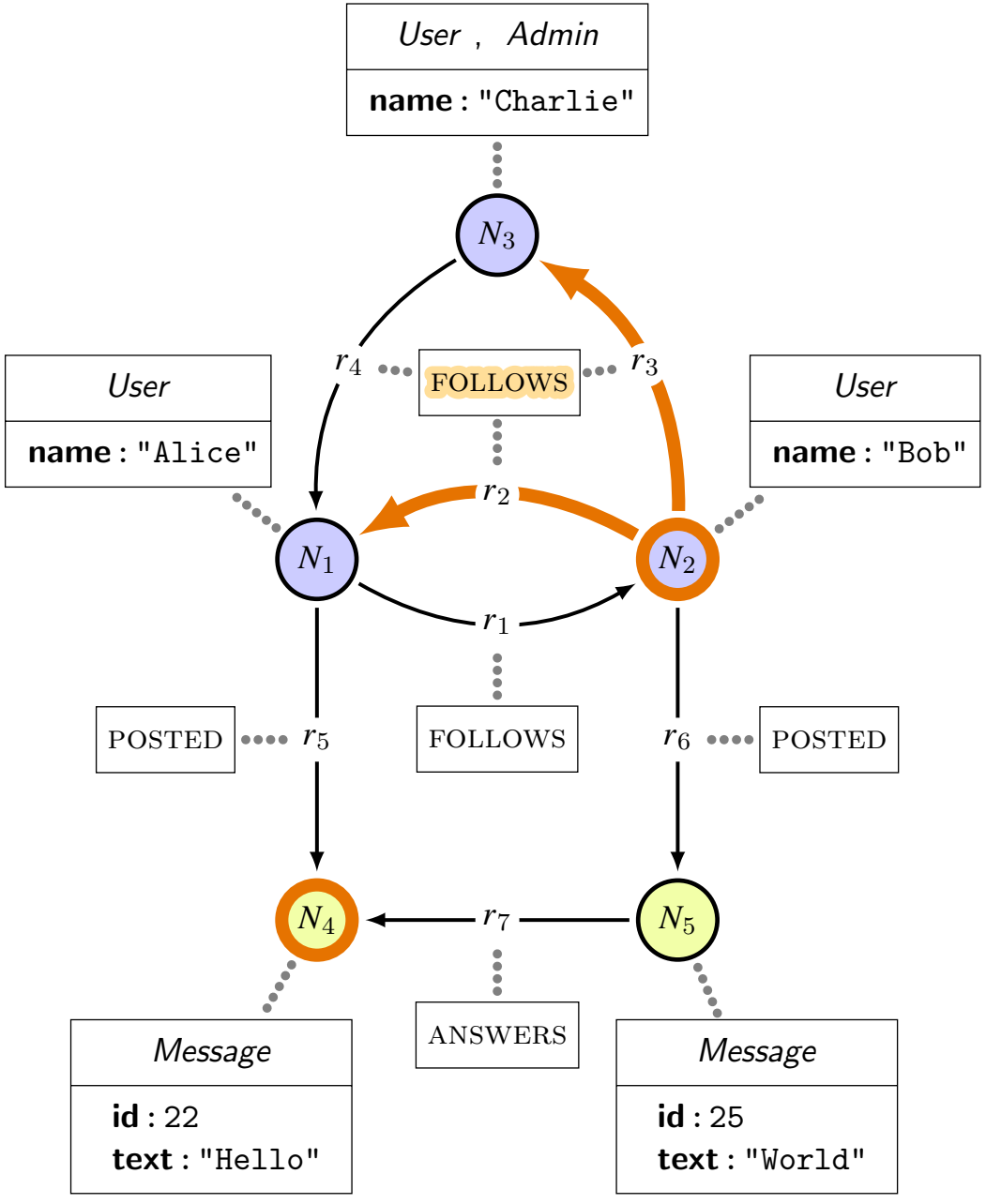
x
N_2
N_4

↓

Query :

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

Example of OPTIONAL MATCH



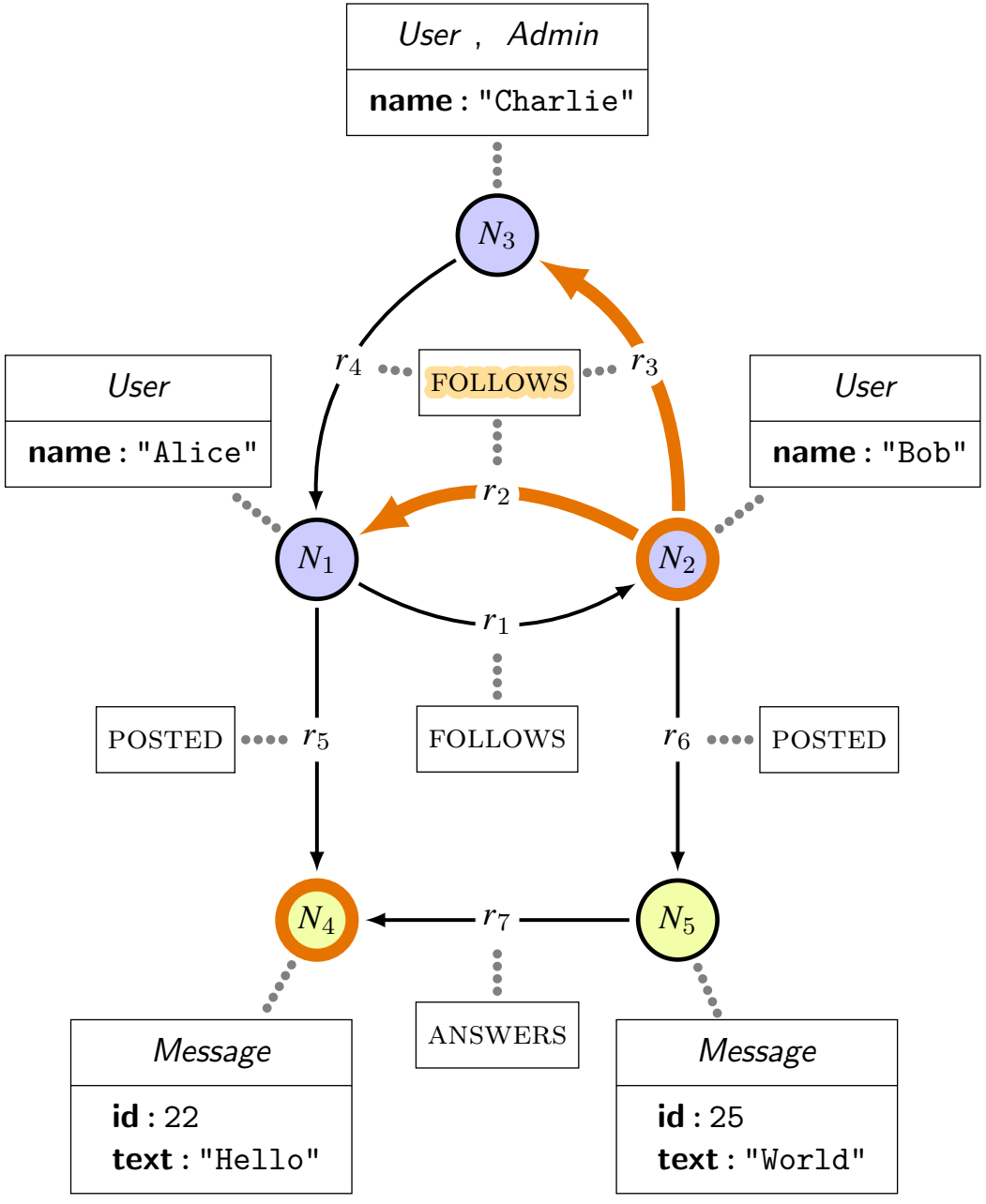
x
N_2
N_4

↓

Query :

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

Example of OPTIONAL MATCH



x
N_2
N_4

↓

Query :

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

↓

x	y
N_2	r_2
N_2	r_3
N_4	null

Caveat on the MATCH clause (3)

Semantics of OPTIONAL MATCH

u: a record

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

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

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

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

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

The WHERE subclause (1)

Syntax

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

Semantics

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

The WHERE subclause (2)

Combined semantics

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

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

where $T_u = \left[\!\left[\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e\right]\!\right]_G(\{u\})$

(and $\mathsf{cmpl}(u, \bar{\pi})$ is the completion of u w.r.t. variables of $\bar{\pi}$)

The WITH clause (1)

Syntax

⟨with⟩ ::=

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

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

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

The WITH clause (2)

Example

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

↓

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

↓

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

The WITH clause (3)

Semantics

- $\left[\!\left[\texttt{WITH } * \right]\!\right]_G(T) = T$
- $$\left[\!\left[\texttt{WITH } e_1 \texttt{ AS } a_1, \ldots, e_m \texttt{ AS } a_m \right]\!\right]_G(T) = \biguplus_{u \in T} \left\{ (a_1 : \left[\!\left[e_1 \right]\!\right]_{G,u}, \ldots, a_m : \left[\!\left[e_m \right]\!\right]_{G,u}) \right\}$$
- $$\left[\!\left[\texttt{WITH } e_1 \texttt{ [AS } a_1\texttt{]?}, \ldots, e_m \texttt{ [AS } a_m\texttt{]?} \right]\!\right]_G(T) = \left[\!\left[\texttt{WITH } e_1 \texttt{ AS } a'_1, \ldots, e_m \texttt{ AS } a'_m \right]\!\right]_G(T)$$

  where a'_i equals a_i if it is given, or arbitrary otherwise.
- $\left[\!\left[\texttt{WITH } *, \ldots \right]\!\right]_G(T) = \left[\!\left[\texttt{WITH } b_1 \texttt{ AS } b_1, \ldots, b_q \texttt{ AS } b_q, \ldots \right]\!\right]_G(T)$

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

The UNWIND clause (1)

Syntax

⟨unwind⟩ ::= UNWIND ⟨expr⟩ AS ⟨name⟩

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

The UNWIND clause (2)

Example

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

↓

```
UNWIND list AS x
```

↓

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

The UNWIND clause (3)

Semantics

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

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

The RETURN clause

RETURN and WITH clauses have the exact same semantics.

The RETURN clause

RETURN and WITH clauses have the exact same semantics.

RETURN clause is the "end-marker" of a clause sequence:

- it is mandatory;
- it cannot appear earlier.

UNION and UNION ALL

Syntax

⟨query⟩ ::= ⟨query⟩ UNION [ALL]? ⟨query⟩

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

UNION and UNION ALL

Syntax

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

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

Semantics

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

Outline

Aggregation (1)

Principle

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

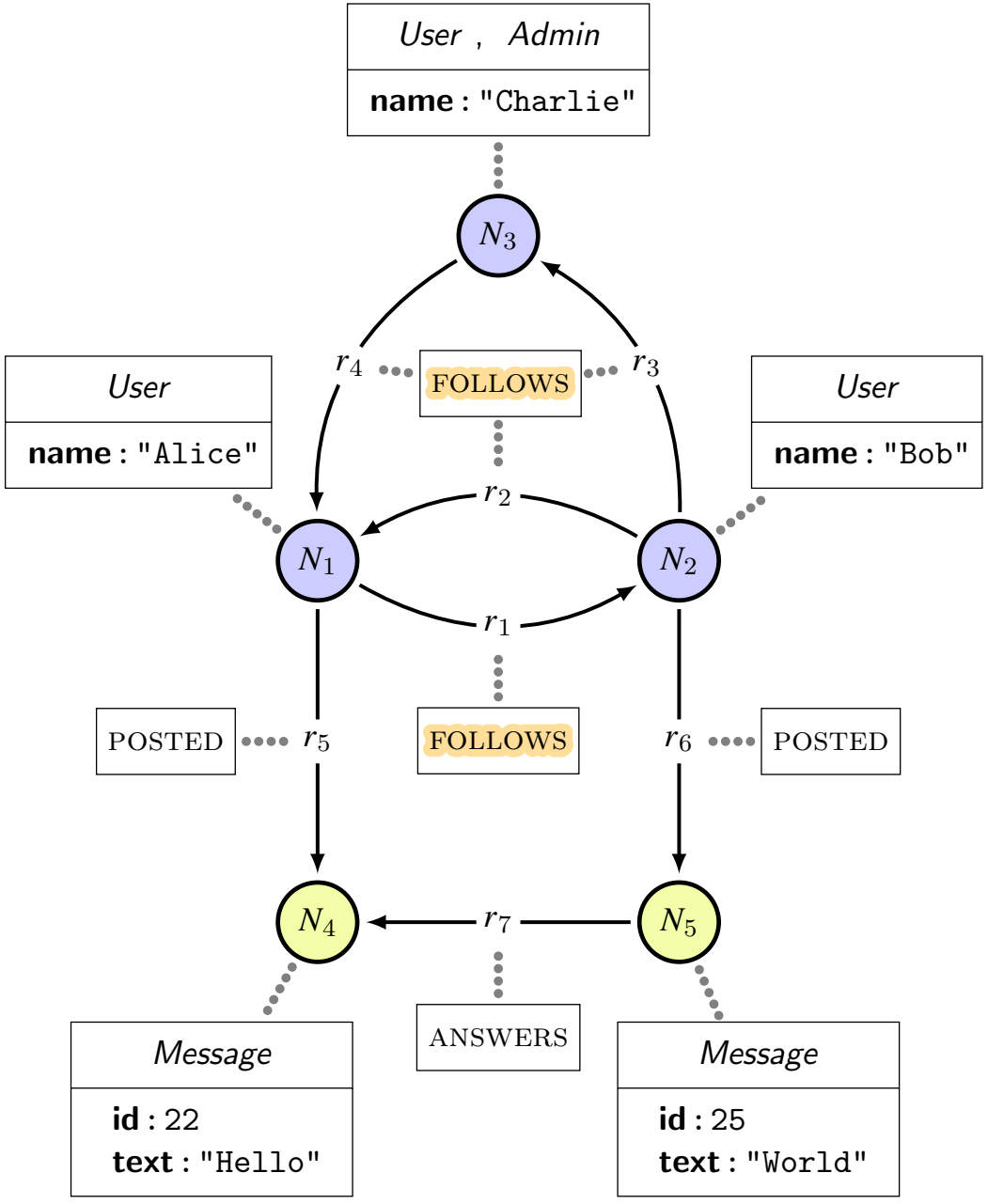
Aggregation functions

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

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

- In `WITH` clauses, we allow *aggregate expression*:
  (*WITH* ⟨aggexpr⟩ [`AS` ⟨name⟩]?)
- Aggregate expressions are of the form: $g(\langle \text{expr} \rangle)$ where $g \in \mathcal{G}$

Aggregation (2) — Example



Query :

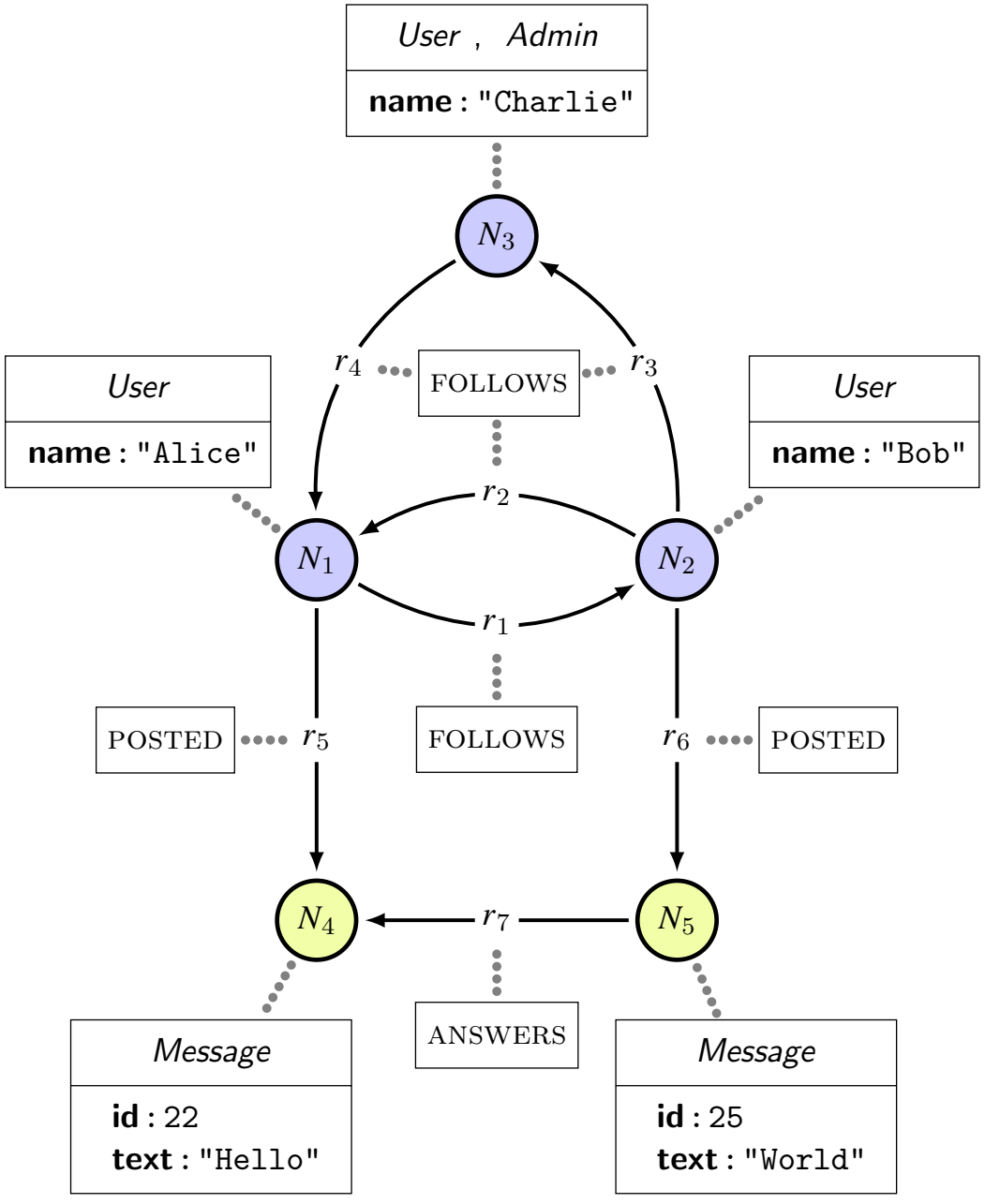
```
MATCH (a)-[r:FOLLOWS]->()
WITH a.name AS b, count(r) AS c
```

Aggregate

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

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

Aggregation (3) — Example



Query :

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

d
2

Aggregation (4)

Semantics on an example

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

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

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

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

Aggregation (4)

Semantics on an example

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

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

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

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

Aggregation (4)

Semantics on an example

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

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

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

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

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

Aggregation (4)

Semantics on an example

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

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

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

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

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

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

Aggregation (4)

Semantics on an example

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

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

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

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

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

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

Line order (1)

Principle

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

Line order (1)

Principle

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

Up to now

- In tables, columns and lines are not ordered

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

} are functions from tables to tables

Line order (1)

Principle

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

Up to now

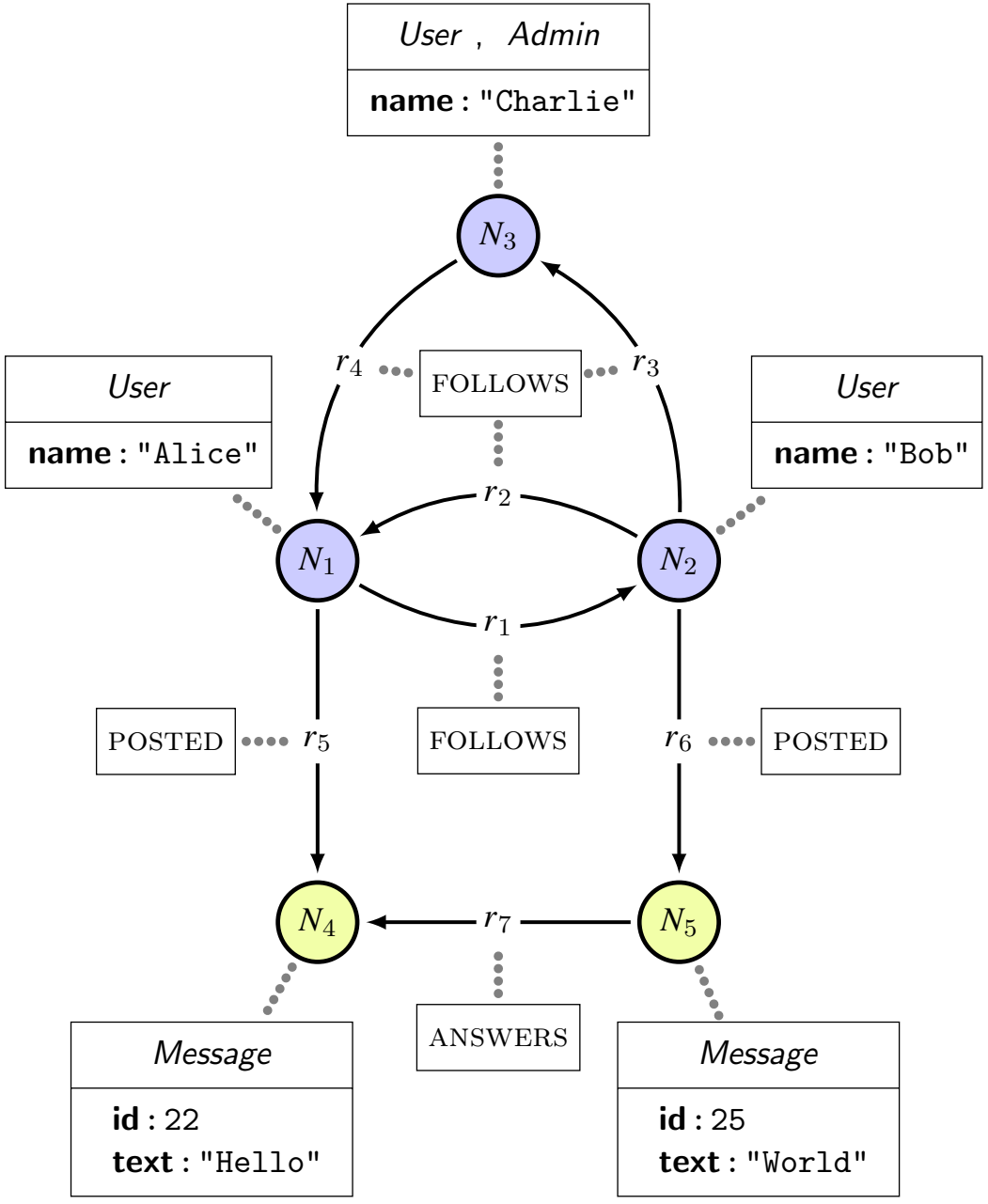
- In tables, columns and lines are not ordered
- Semantics of queries
- Semantics of clauses
- Semantics of subclauses

} are functions from tables to tables

From now on

- `ORDER BY` and its subclause propagate an order

Line order (2) — Semantics through example



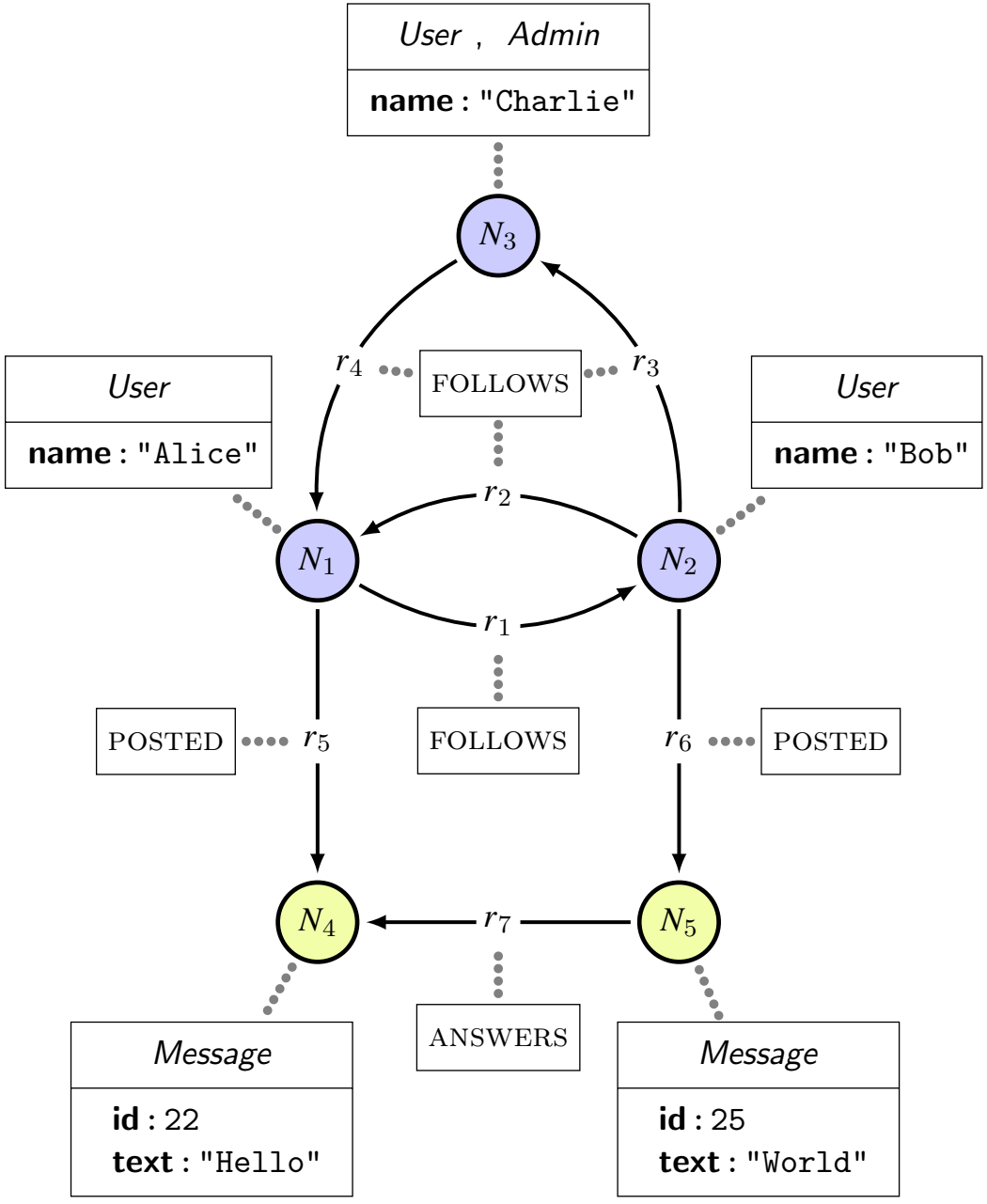
("Alice" < "Bob" < "Charlie")

Query :

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

b
"Bob"

Line order (3) — Semantics through example



(`false` < `true`)

Query :

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

In this case:

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

Outline

Conclusion

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

Future work: semantics for updates

Features

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

Future work: semantics for updates

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)

Future work: semantics for updates

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
- Inconsistent graph-state in the middle of transations
- Row-order dependence of `MERGE` and `SET`

⋮