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 de l'équipe LINKS
Lille
2018-02-16

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

However, some data have intrinsically the structure of graphs:

- Semantic web
- Social Networks
- Bioinformatic networks

Graph databases are becoming more widespread

More and more engines

- Neo4j
- JanusGraph
- Sparksee

More and more query-language

- Cypher
- Gremlin
- SparQL

Used in many domain

- Fraud detection
- Bioinformatics
- Investigative journalism

Cypher

- Language for querying and updating
- Data is expected to be structured as *property graphs*
- The result is a table (as in SQL)

Cypher

- Language for querying and updating
- Data is expected to be structured as *property graphs*
- The result is a table (as in SQL)

- Implemented in multiple datagraph engines: SAP HANA Graph, Redis Graph, Agens Graph, Neo4j.
- Commercial success for its inventors (Neo Technology).

Cypher

- Language for querying and updating
- Data is expected to be structured as *property graphs*
- The result is a table (as in SQL)

- Implemented in multiple datagraph engines: SAP HANA Graph, Redis Graph, Agens Graph, Neo4j.
- Commercial success for its inventors (Neo Technology).

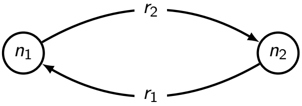
The openCypher project

- Since 2015
- Seeks to make of Cypher a standard:
  - Community-led additions and evolution
  - Complete specification

Our goal

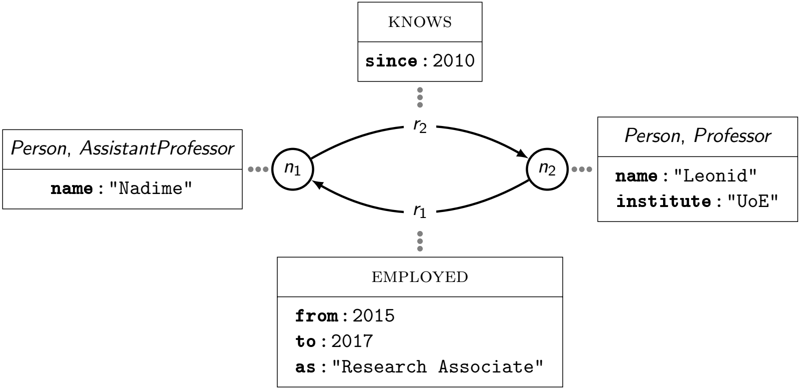
Full denotational semantics for the language Cypher.

Outline



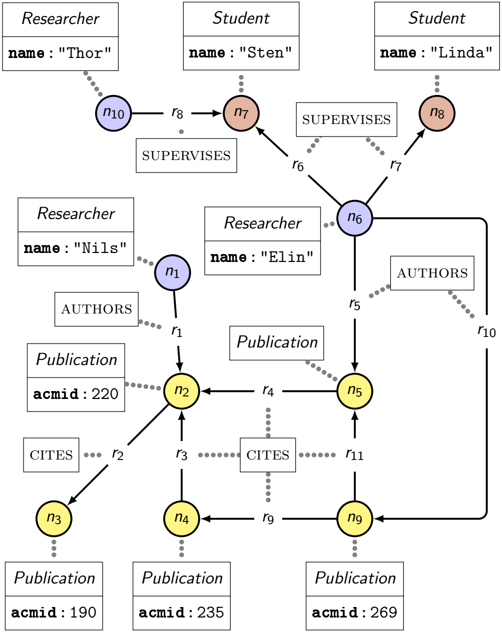
- **Nodes** (e.g. n_1)
- **Relationships** (e.g. r_1)

First property graph

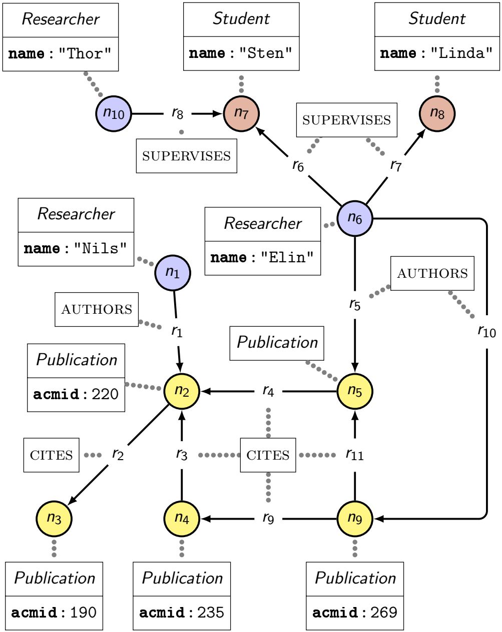


- **Nodes** (e.g. n_1)
- **Relationships** (e.g. r_1)
- Nodes may bear **Labels** (e.g. *Person*)
- Each relationship must bear a **Type** (e.g. KNOWS)
- **Properties**, ie pair key/values, may be worn by Nodes and Relationships (e.g. **name** : "Leonid" or **since** : 2010)

Our running example



Matching nodes (1)

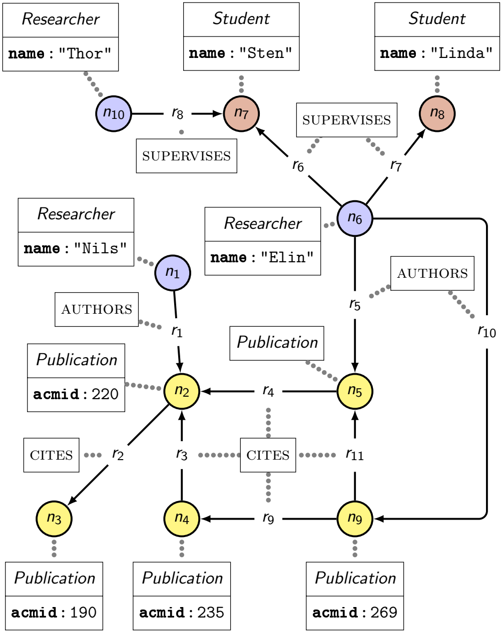



```
MATCH (x)
```

Match any node

x
n_1
n_2
$\vdots$
n_{10}

Matching nodes (2)

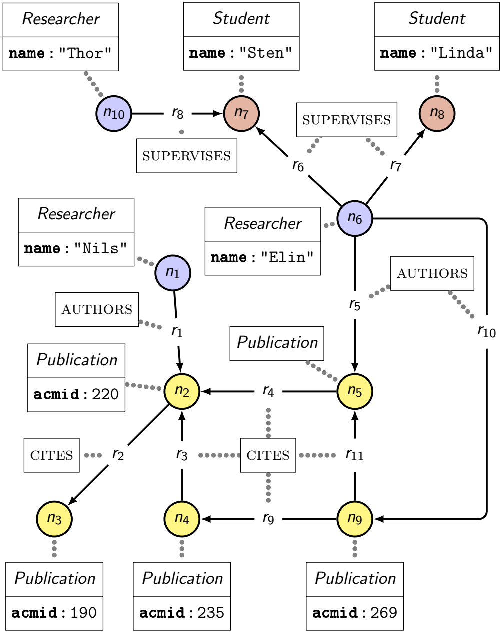



```
MATCH (x:Student)
```

Match nodes that bear the label *Student*

x
n_7
n_8

Matching nodes (3)

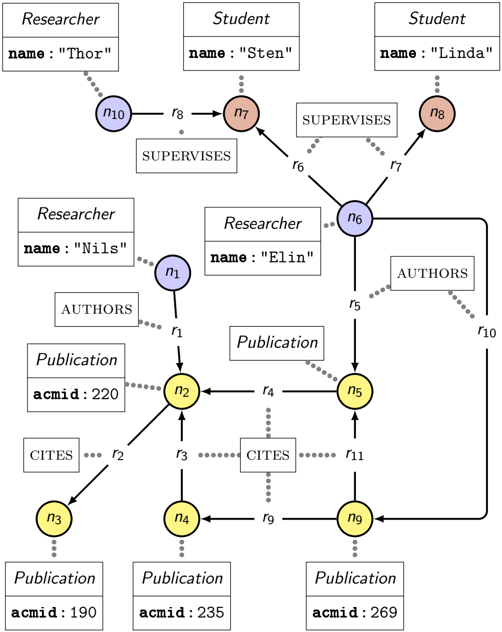



```
MATCH (x:Student
         :Researcher)
```

Match nodes that bear both the labels *Student* and *Researcher*

x

Matching nodes (4)

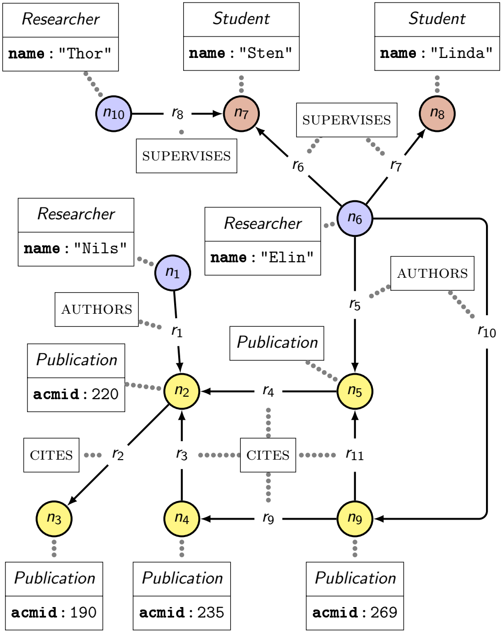



```
MATCH (x {name:"Elin"})
```

Match nodes with the property **name** set to "Elin"

x
n_6

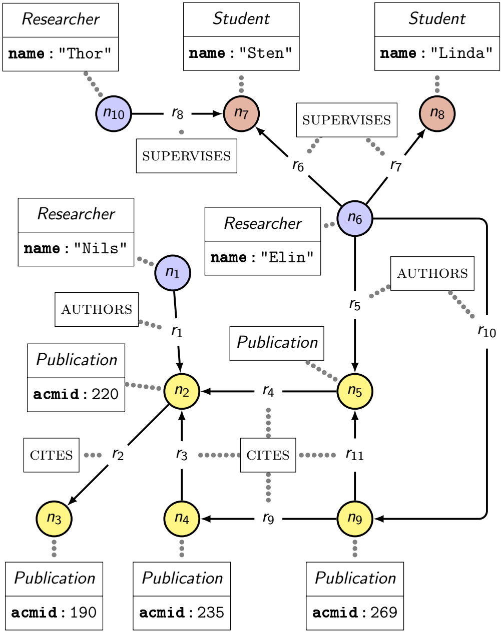
Matching relationships (1)




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

Match any relationship

x
r_1
r_2
⋮
r_{11}

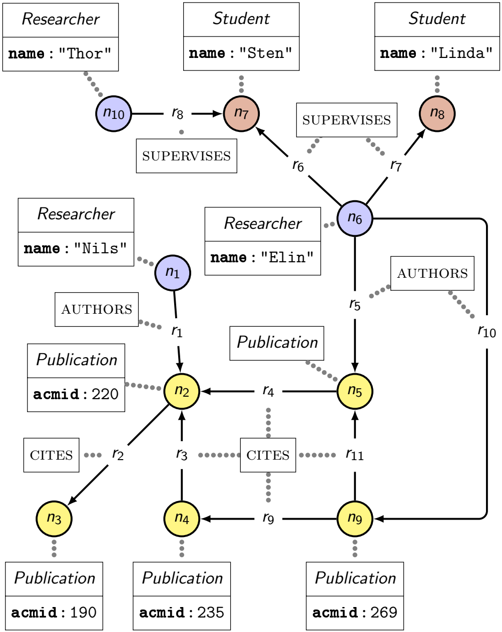



```
MATCH (o)-[x]->()
```

Match any relationship while keeping track of origin

o	x
n_1	r_1
n_2	r_2
$\vdots$	$\vdots$
n_9	r_{11}

Matching relationships (3)

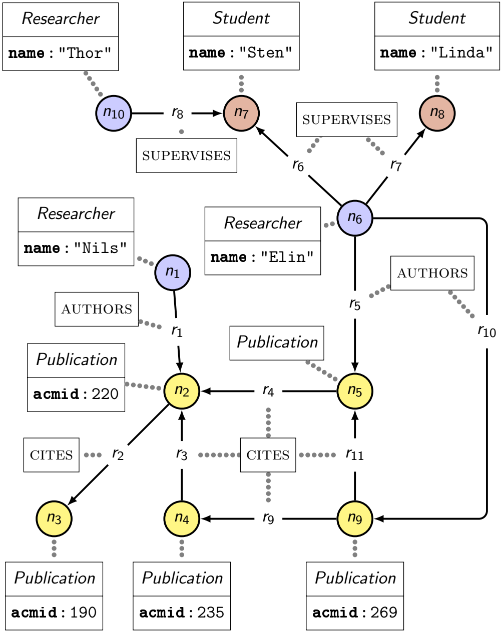



```
MATCH ()-[:SUPERVISES]->(x)
```

Match relationships of type SUPERVISES and keep tracks only of the endpoint

x
n_7
n_7
n_8

Matching paths (1)

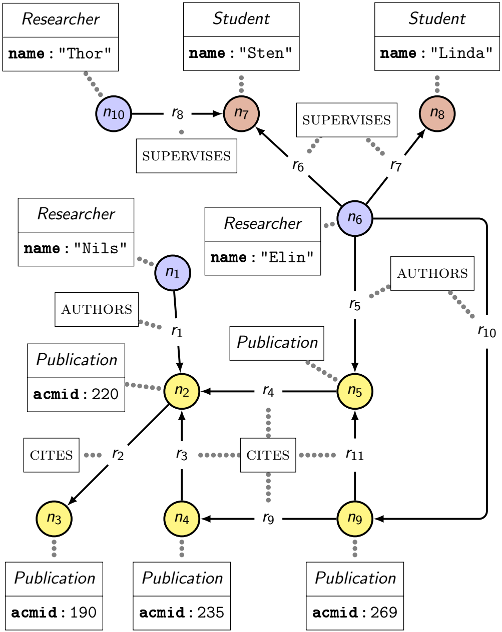



```
MATCH (x)-[]->(y)-[]->(z)
```

Match length-two forward paths

x	y	z
n_1	n_2	n_3
n_4	n_2	n_3
n_5	n_2	n_3
n_6	n_5	n_2
n_6	n_9	n_5
n_9	n_4	n_2
n_9	n_5	n_2

Matching paths (2)

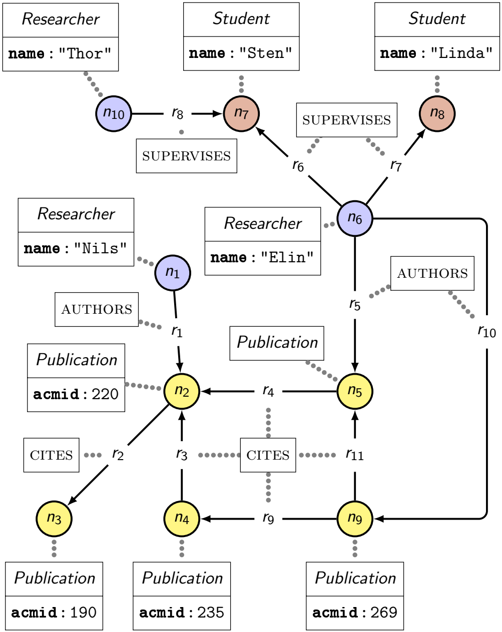



```
MATCH (x)-[:CITES]->()
          <-[:CITES]-(y)
```

Match any two nodes that CITES the same node

x	y
n_4	n_5
n_5	n_4

Matching paths (2)



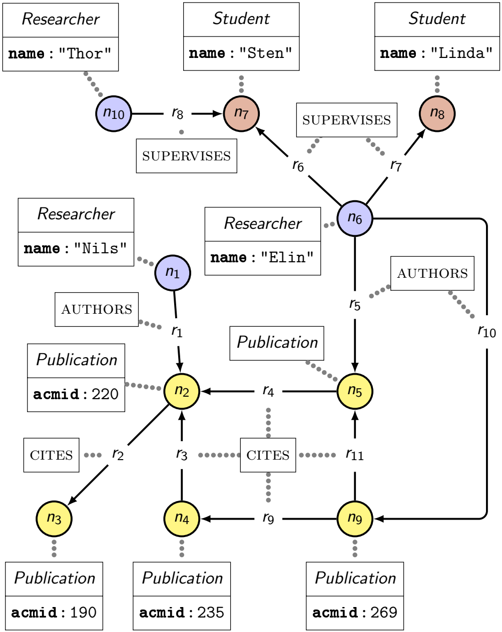

```
MATCH (x)-[:CITES]->()
          <-[:CITES]-(y)
```

Match any two nodes that CITES the same node

x	y
n_4	n_5
n_5	n_4

(Cypher-morphism $\implies$ no row $x = y = n_2$)

Matching paths (3)

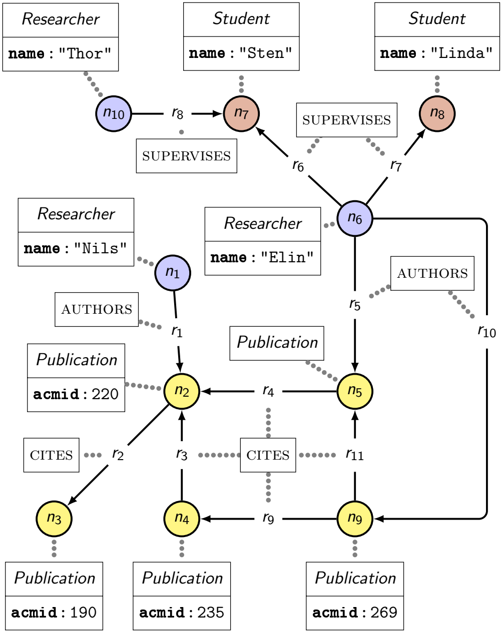



```
MATCH (x)-[r:CITES*]->(y)
```

Match any two nodes linked by a path of CITES relationships

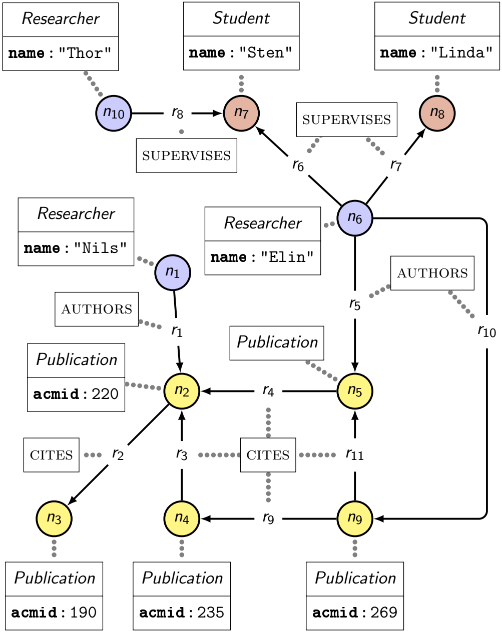
x	r	y
n_2	$[r_2]$	n_3
⋮	⋮	⋮
n_9	$[r_9, r_3]$	n_2
n_9	$[r_{11}, r_4]$	n_2
n_9	$[r_9, r_3, r_2]$	n_3
⋮	⋮	⋮

Chaining clauses




```
MATCH (x {name:"Elin"})
MATCH (x)-[]->(y)
MATCH (x)-[]->(z)<-[]-(y)
```

Chaining clauses

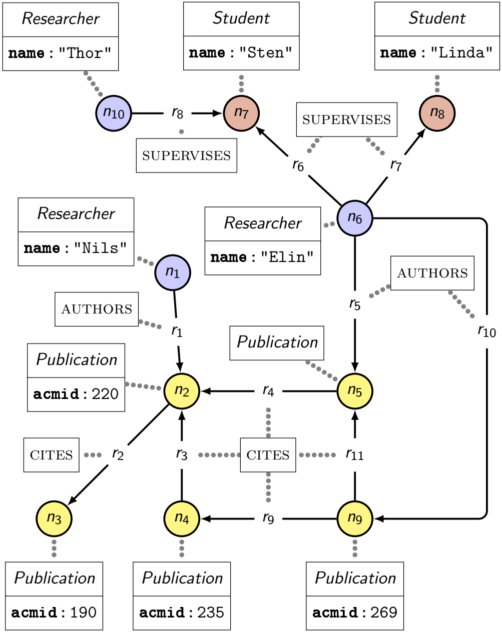



```
MATCH (x {name:"Elin"})
```

After 1 clause

x
n_6

Chaining clauses

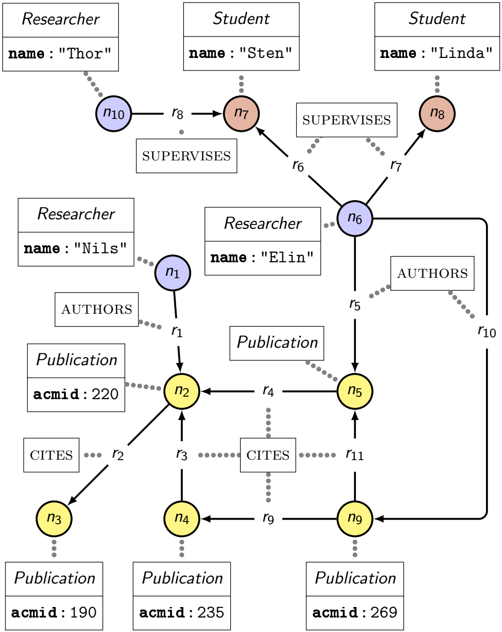



```
MATCH (x {name:"Elin"})
MATCH (x)-[]->(y)
```

After 2 clauses

x	y
n_6	n_5
n_6	n_7
n_6	n_8
n_6	n_9

Chaining clauses

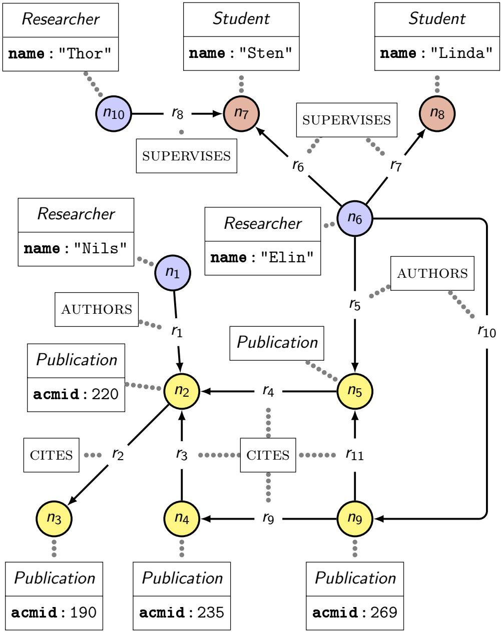



```
MATCH (x {name:"Elin"})
MATCH (x)-[]->(y)
MATCH (x)-[]->(z)<-[]-(y)
```

Computing 3rd clause

x	z	y
n_6	·	n_5
n_6	·	n_7
n_6	·	n_8
n_6	·	n_9

Chaining clauses

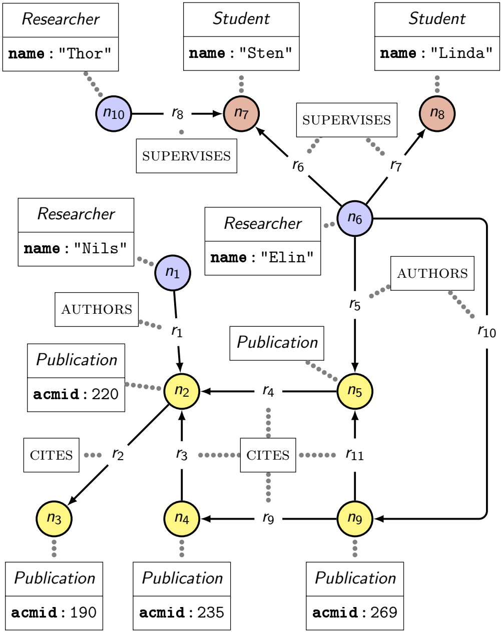



```
MATCH (x {name:"Elin"})
MATCH (x)-[]->(y)
MATCH (x)-[]->(z)<-[]-(y)
```

Computing 3rd clause

x	z	y
~~n_6~~	~~·~~	~~n_5~~
n_6	·	n_7
n_6	·	n_8
n_6	·	n_9

Chaining clauses

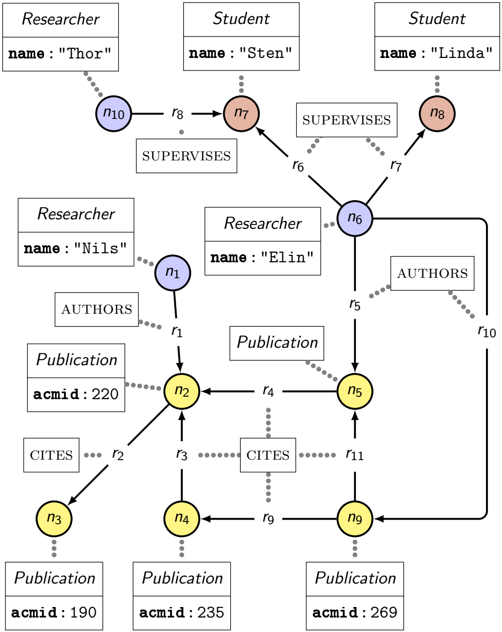



```
MATCH (x {name:"Elin"})
MATCH (x)-[]->(y)
MATCH (x)-[]->(z)<-[]-(y)
```

Computing 3rd clause

x	z	y
~~n_6~~	~~·~~	~~n_5~~
~~n_6~~	~~·~~	~~n_7~~
~~n_6~~	~~·~~	~~n_8~~
n_6	·	n_9

Chaining clauses

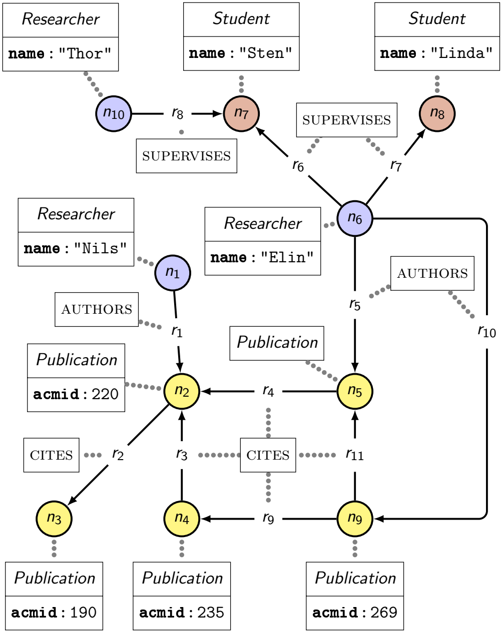



```
MATCH (x {name:"Elin"})
MATCH (x)-[]->(y)
MATCH (x)-[]->(z)<-[]-(y)
```

Computing 3rd clause

x	z	y
~~n_6~~	~~·~~	~~n_5~~
~~n_6~~	~~·~~	~~n_7~~
~~n_6~~	~~·~~	~~n_8~~
n_6	n_5	n_9

Chaining clauses

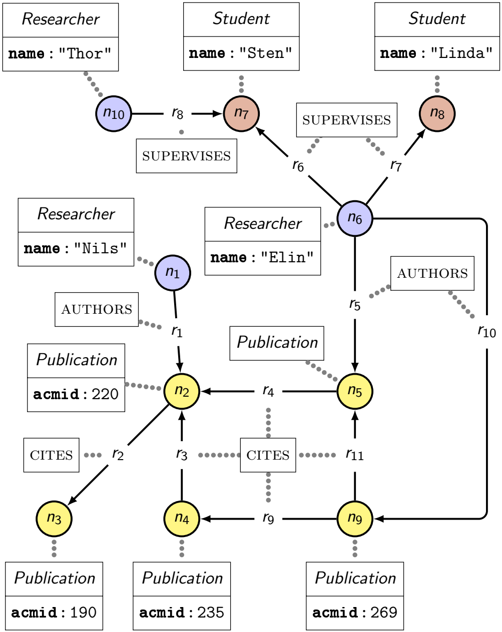



```
MATCH (x {name:"Elin"})
MATCH (x)-[]->(y)
MATCH (x)-[]->(z)<-[]-(y)
```

Final result

x	z	y
n_6	n_5	n_9

Filtering rows

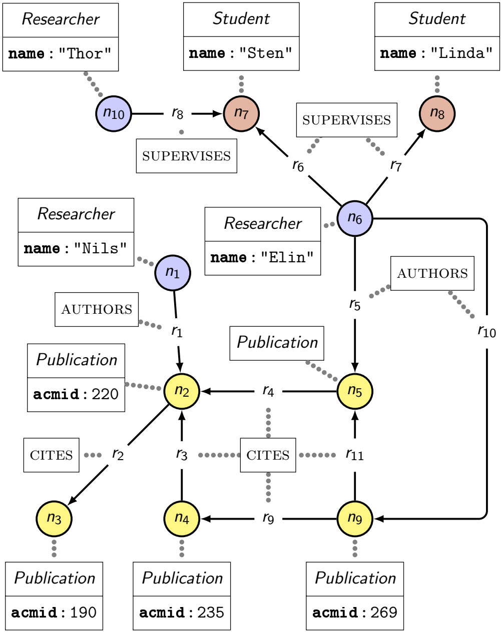



```
MATCH (x)-[r:CITES*]->(y)
WHERE x.acmid > y.acmid+50
```

Same as previously, but only if the **acmid** of the endpoint is 50 less the origin

x	r	y
n_9	$[r_9, r_3, r_2]$	n_3
n_9	$[r_{11}, r_4, r_2]$	n_3

Modifying columns (1)

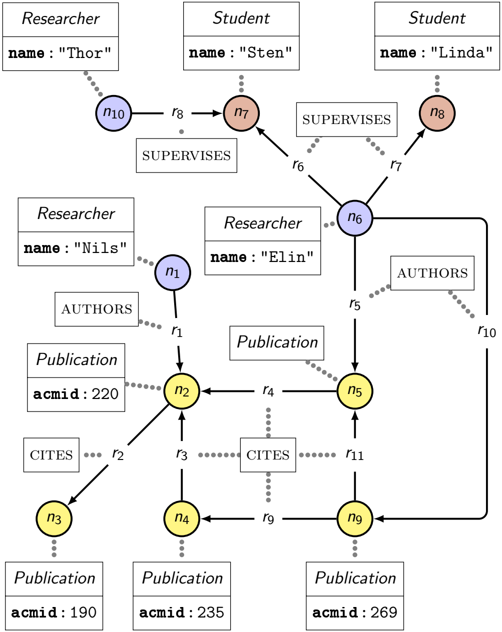



```
MATCH (x :Publication)
WITH *, x.acmid AS y
```

Match nodes ... and add a column y with the acmid.

x	y
n_2	220
n_3	190
n_4	235
n_5	null
n_9	269

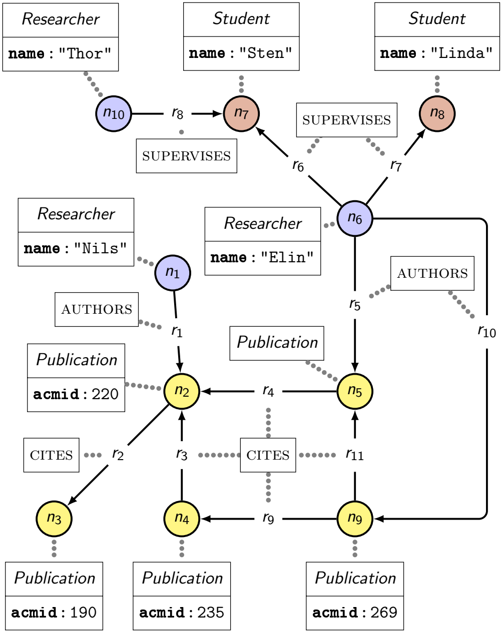
Modifying columns (2)




```
MATCH (x:Researcher)-[y]
                     ->(z)
WITH x.name AS n, y,
     z:Student AS s
```

n	y	s
"Thor"	r_8	true
"Elin"	r_6	true
"Elin"	r_7	true
"Elin"	r_5	false
"Elin"	r_{10}	false
"Nils"	r_1	false

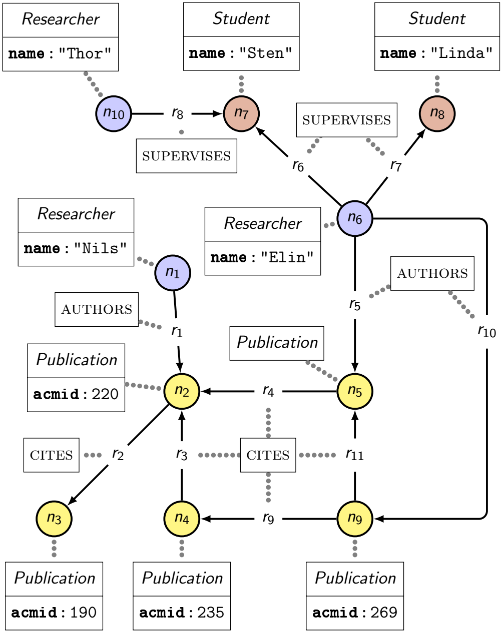
A last example




```
MATCH (a {name:"Elin"})
MATCH (b {acmid:220})
MATCH (a)-[*]->(b)<-[*]-(a)
```

Question: What does this query computes ?

A last example




```
MATCH (a {name:"Elin"})
MATCH (b {acmid:220})
MATCH (a)-[*]->(b)<-[*]-(a)
```

Question: What does this query computes ?

Answer: the disjoint paths between the nodes n_6 and n_2.

Outline

Modelling Tables

Record (or "table row")

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

Example: $(x \mapsto$ `"Nadime"` ; $y \mapsto 2012)$

Modelling Tables

Record (or "table row")

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

Example: $(x \mapsto$ `"Nadime"` $; \ y \mapsto 2012)$

Table

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

Example:

x	y
"Nadime"	2012
"Victor"	2016
"Nadime"	2012

Modelling Tables

Record (or "table row")

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

Example: (x ↦`"Nadime"` ; y ↦`2012`)

Table

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

Example:

x	y
"Nadime"	2012
"Victor"	2016
"Nadime"	2012

=

x	y
2016	"Victor"
2012	"Nadime"
2012	"Nadime"

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: query $\longmapsto$ Table

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 $[\![\text{Clause}_1]\!]_G$, $[\![\text{Clause}_2]\!]_G$, ... , $[\![\text{Clause}_n]\!]_G$

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 $[\![\text{Clause}_1]\!]_G$, $[\![\text{Clause}_2]\!]_G$, ... , $[\![\text{Clause}_n]\!]_G$
- Let $[\![Q]\!]_G = [\![\text{Clause}_n]\!]_G \circ \cdots \circ [\![\text{Clause}_2]\!]_G \circ [\![\text{Clause}_1]\!]_G$

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 $[\![\text{Clause}_1]\!]_G$, $[\![\text{Clause}_2]\!]_G$, ... , $[\![\text{Clause}_n]\!]_G$
- Let $[\![Q]\!]_G = [\![\text{Clause}_n]\!]_G \circ \cdots \circ [\![\text{Clause}_2]\!]_G \circ [\![\text{Clause}_1]\!]_G$
- Output of Q: result of applying $[\![Q]\!]_G$ to the 1-line 0-column table.

Outline

Node patterns

$\langle$node_pattern$\rangle$ ::= ($\langle$name$\rangle$? $\langle$labels$\rangle$? $\langle$properties$\rangle$?)

Examples

`()`	Any node
`(a)`	Any node; verify/map id to column *a*
`(:Researcher)`	Nodes of type Researcher
`({name:"Elin"})`	Nodes with the property **`name`** set to `"Elin"`
`(a:Researcher{name:"Elin"})`	Nodes of type Researcher 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
`[:CITES*3..]`	Any CITES 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)<-[]->()-[{acmid:220}]->()
(a)<-[*]-(b)-[*]->(a)
({name:a.name})-[*]->(a)
({name:"Elin"})-[:AUTHORS]->()-[:CITES*]->()
```

Rigid path-patterns

Definition: Rigid path-patterns

Path patterns is *Rigid* if its length is fixed.
(*i.e.* all ⟨iter⟩ are absent or derive to $*i..i$ for some i)

Examples

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

Rigid path-patterns

Definition: Rigid path-patterns

Path patterns is *Rigid* if its length is fixed.
(*i.e.* all ⟨iter⟩ are absent or derive to $*i..i$ for some i)

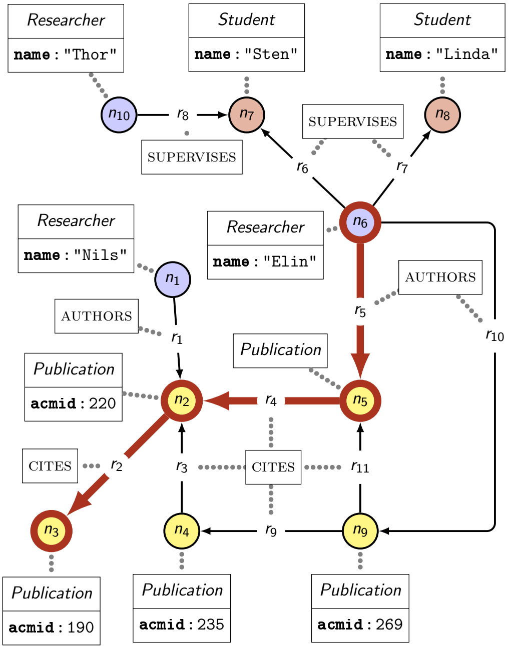
Examples

`(a)<-[]->()-[{acmid:220}]->()`	Rigid
`()<-[]->()-[*42..42]->(:Researcher)`	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 (1)



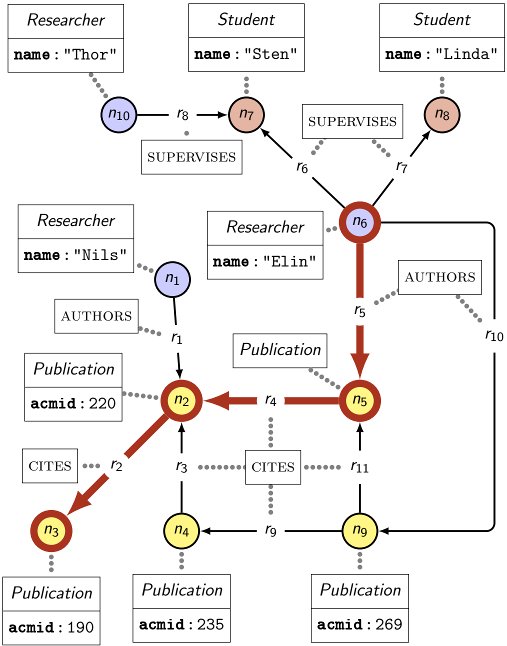

```
MATCH (src {name:"Elin"})
      -[*1..1]->(mid)
          ->[*2..2]->(dst)
```

The path

$$n_6 \xrightarrow{r_5} n_5 \xrightarrow{r_4} n_2 \xrightarrow{r_2} n_3$$

satifies the pattern.

Rigid vs flexible (1)




```
MATCH (src {name:"Elin"})
      -[*1..1]->(mid)
          ->[*2..2]->(dst)
```

The path

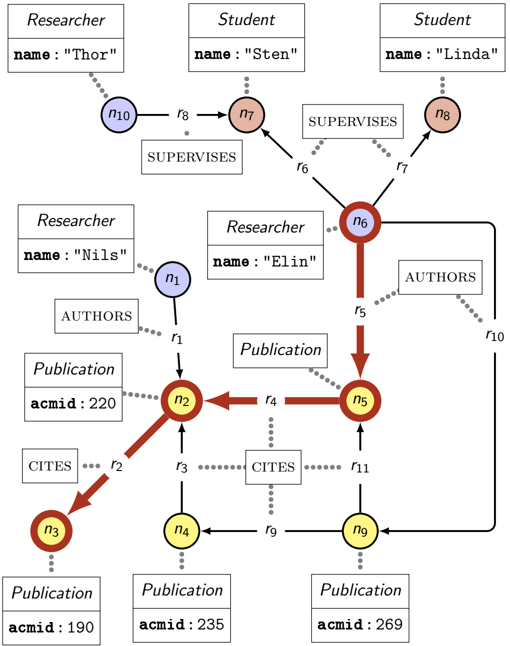
$$n_6 \xrightarrow{r_5} n_5 \xrightarrow{r_4} n_2 \xrightarrow{r_2} n_3$$

satifies the pattern.

Variables are uniquely set

src	mid	src
n_6	n_5	n_3

Rigid vs flexible (2)



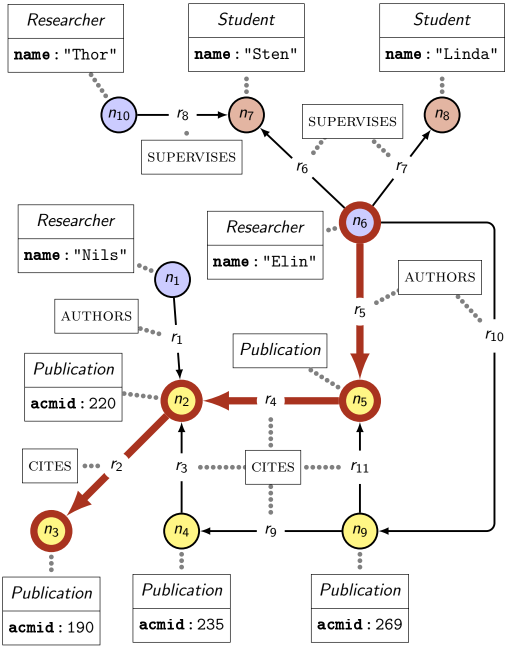

```
MATCH (src {name:"Elin"})
        -[*1..2]->(mid)
               ->[*1..2]->(dst)
```

The path

$$n_6 \xrightarrow{r_5} n_5 \xrightarrow{r_4} n_2 \xrightarrow{r_2} n_3$$

satifies this second pattern.

Rigid vs flexible (2)




```
MATCH (src {name:"Elin"})
      -[*1..2]->(mid)
           ->[*1..2]->(dst)
```

The path

$$n_6 \xrightarrow{r_5} n_5 \xrightarrow{r_4} n_2 \xrightarrow{r_2} n_3$$

satifies this second pattern.

Two possible assignments

src	*mid*	*src*
n_6	n_5	n_3
n_6	n_4	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

$$\text{rigid}(\pi) = \left\{ \text{All rigid patterns subsumed by } \pi \right\}$$

Examples

rigid :

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

Rigid path-pattern subsumed by a flexible path-pattern

$$\text{rigid}(\pi) = \left\{ \text{All rigid patterns subsumed by } \pi \right\}$$

Examples

rigid :

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

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

Rigid path-pattern subsumed by a flexible path-pattern

$$\text{rigid}(\pi) = \left\{ \text{All rigid patterns subsumed by } \pi \right\}$$

Examples

rigid :

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

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

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

$$\text{rigid}(\pi) = \left\{ \text{All rigid patterns subsumed by } \pi \right\}$$

Examples

rigid :

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

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

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

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

Table-row expansion

π: a path-pattern
G: a graph

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

$$u \longmapsto \biguplus_{\substack{\text{all paths } p \\ \text{patterns } \rho \text{ in rigid}(\pi)}} \left\{ \text{records } v \;\middle|\; \begin{array}{l} \blacksquare \text{ containing } u \\ \blacksquare \text{ such that } G, p, v \models \rho \end{array} \right\}$$

The MATCH clause

Table-row expansion

π: a path-pattern
G: a graph

$$\begin{aligned} \text{expand}_{G,\pi} : \quad & \textit{Records} \longrightarrow \textit{Tables} \\ u \quad \longmapsto \quad & \biguplus_{\substack{\text{all paths } p \\ \text{patterns } \rho \text{ in rigid}(\pi)}} \left\{ \text{records } v \;\middle|\; \begin{array}{l} \blacksquare \text{ containing } u \\ \blacksquare \text{ such that } G, p, v \models \rho \end{array} \right\} \end{aligned}$$

Semantics of MATCH

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

Outline

The core fragment

Queries are essentially sequences of clauses

$\langle$query$\rangle$:== $\langle$query$\rangle$ UNION [ALL]? $\langle$query$\rangle$
| [$\langle$clause$\rangle$]* $\langle$return$\rangle$

A clause is a main statement followed by subclauses

$\langle$clause$\rangle$:== $\langle$match$\rangle$ [$\langle$subclause$\rangle$]*
| $\langle$with$\rangle$ [$\langle$subclause$\rangle$]*
| $\langle$unwind$\rangle$ [$\langle$subclause$\rangle$]*

In the core fragment, there is only one kind of subclauses

$\langle$subclause$\rangle$:== $\langle$where$\rangle$

The WHERE subclause : Line filtering

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 WITH clause (1)

Syntax

$\langle$with$\rangle$ ::=

| WITH $\langle$expr$\rangle$ [AS $\langle$name$\rangle$]? , ⋯, $\langle$expr$\rangle$ [AS $\langle$name$\rangle$]?

| WITH * , $\langle$expr$\rangle$ [AS $\langle$name$\rangle$]? , ⋯, $\langle$expr$\rangle$ [AS $\langle$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 dependant).
- * 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

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

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

- $$\left[\!\left[\texttt{WITH } e_1 \; [\texttt{AS } a_1]?, \ldots, e_m \; [\texttt{AS } a_m]? \right]\!\right]_G (T) = \left[\!\left[\texttt{WITH } e_1 \texttt{ AS } a'_1, \ldots, e_m \texttt{ AS } a'_m \right]\!\right]_G (T)$$

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

- $$\left[\!\left[\texttt{WITH } *, \ldots \right]\!\right]_G (T) = \left[\!\left[\texttt{WITH } b_1 \texttt{ AS } b_1, \ldots, b_q \texttt{ AS } b_q, \ldots \right]\!\right]_G (T)$$

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

The UNWIND clause (1)

Syntax

⟨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"

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

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

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

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

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

Semantics

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

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 sets to single values

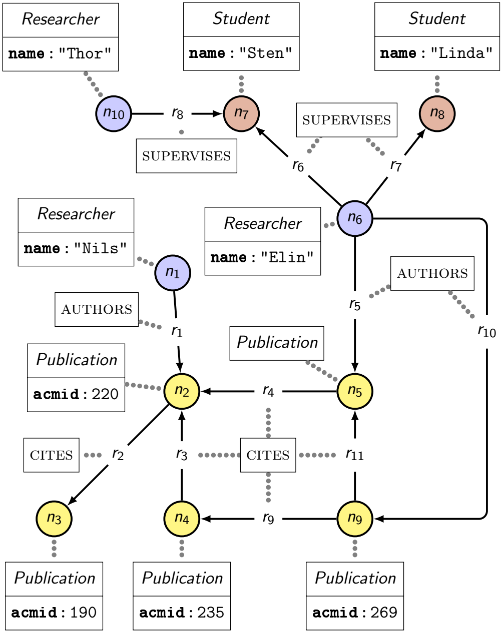
For instance, `count`, `max` $\in \mathcal{G}$

- 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

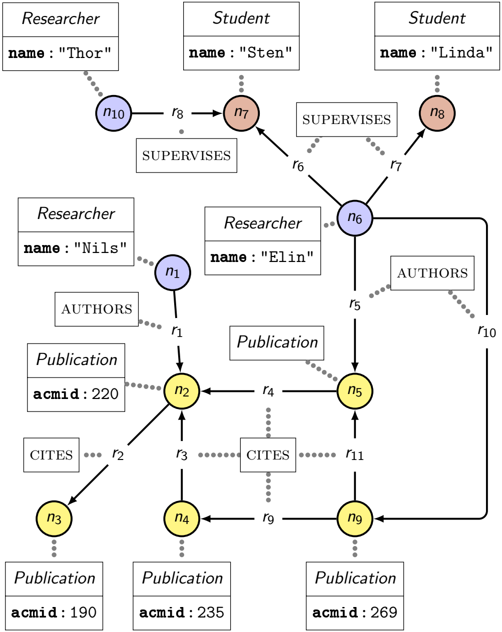



```
MATCH (a)-[r:CITES*]->(b)
WITH a, b, count(r) as c
```

How many CITES paths between each pair of nodes.

a	b	c
n_2	n_3	1
⋮	⋮	⋮
n_9	n_2	2
n_9	n_3	2
n_9	n_5	1
n_9	n_6	1

Aggregation (3) — Harder example




```
MATCH (a)-[r*]->(b)
WITH a, b, count(r) as c
WITH a, max(c) as m
```

[...]

a	m
n_2	1
n_3	2
n_5	2
n_6	1
n_9	2

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

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

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

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

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

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

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

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

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

Aggregation (4)

Semantics on an example

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

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

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

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

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

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

Line order (1)

Principle

- `ORDER BY` subclause: 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` subclause: 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` subclause: 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

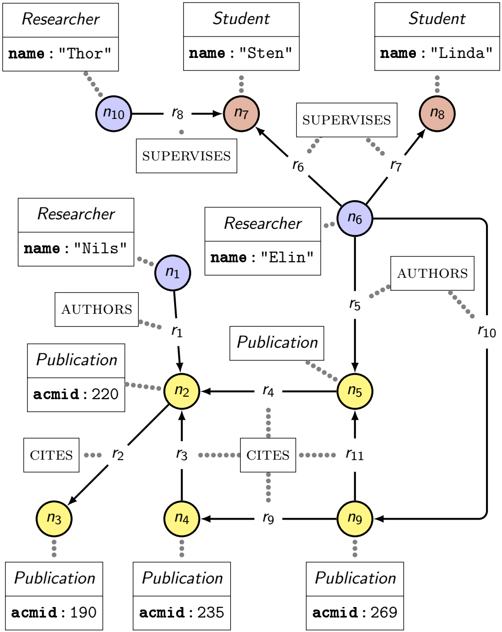
- In tables, columns and lines are not ordered
- Semantics of queries
- Semantics of clauses
- Semantics of subclauses

} are functions from tables to tables

From now on

- Queries and subclauses propagate the order.
- Clauses still do not.

Line Order (2) — Semantics through example

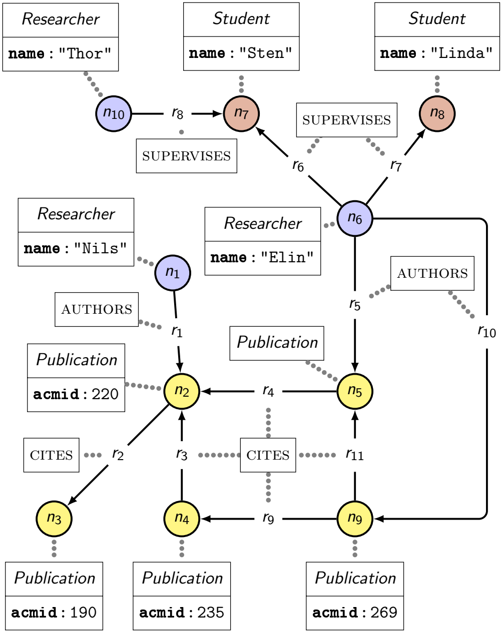


$(\texttt{"Elin"} < \texttt{"Linda"} < \texttt{"Nils"} < \texttt{"Sten"} < \texttt{"Thor"})$

```
MATCH (a)
  WHERE IS NOT NULL a.name
  ORDER BY a.name
  LIMIT 4
  SKIP 1
RETURN a.name as b
```

b
"Linda"
"Nils"
"Sten"

Line Order (2) — Semantics through example

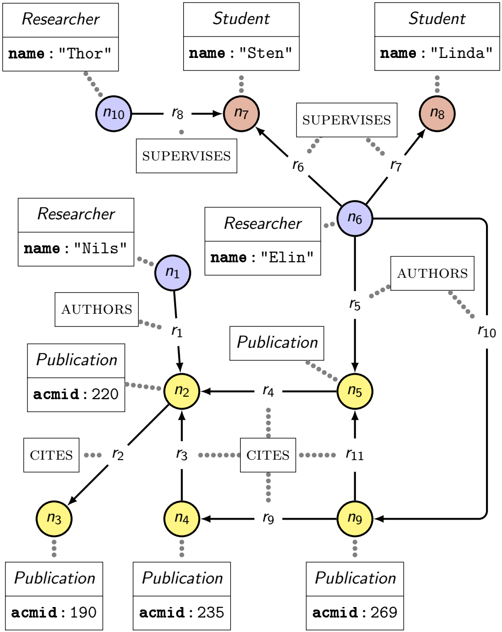


("Elin" < "Linda" < "Nils" < "Sten" < "Thor")

```
MATCH (a)
  WHERE IS NOT NULL a.name
  ORDER BY a.name
  LIMIT 4
  SKIP 1
RETURN a.name as b
```

b
"Sten"
"Linda"
"Nils"

Order (3) — Semantics through example




```
MATCH (a)
  WHERE IS NOT NULL a.name
  LIMIT 4
  SKIP 1
RETURN a.name as b
```

(Removed the ORDER BY subclause)

In this case:

- Non-determinism
- 1 column and 3 lines
- Cells could contain any three of: "Elin", "Linda", "Nils", "Sten", "Thor"

Outline

Conclusion

Cypher

- Relatively recent query-language for graph databases
- Getting 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)`
- Transactional semantics: updated graph is committed at the end of execution

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)`
- Transactional semantics: updated graph is committed at the end of execution

A few hiccups

- Inconsistent syntax accross the update clauses
- Inconsistent graph-state in the middle of transations
- Row-order dependence of MERGE and SET

  ⋮