

# Things we recently learned about new graph languages (GQL and SQL/PGQ)

Leonid Libkin

RelationalAI and Univ of Edinburgh

Pablo's workshop, 29 January 2025

# What it's about

- A bit of history: from RPQs in the 1980s to SQL/PGQ in 2023 and GQL in 2024
- Why GQL development is SQL development backwards?
- How can we study GQL? What's missing?
- Models of PGQ and GQL
- Early expressivity results: starting FMT from scratch
- What's done in real life and why it's horrible
- Existential questions: are graph DBMSs there to stay?

# Property Graphs in Industry



redis



**MEM**  
GRAPH



# They Must Be Queried

Systems have their own languages

- **Cypher** of Neo4j (and Amazon Neptune, SAP HANA, Memgraph, etc.)
- **PGQL** of Oracle
- **GSQL** of Tigergraph etc ...

"If only there were a standard"

International Organization for Standardization  
Organisation Internationale de Normalisation  
Международная Организация по Стандартизации

Hence  $\iota\sigma\varsigma$

Developed by ISO: 2019-2024

- **GQL** — Graph Query Language
- Developed in the same committee as SQL
- First query language to become an ISO standard in 35 years

GQL is not the only language!

- **SQL/PGQ**: of property graph querying in SQL
- Developed 2018-2023
- Part 16 of the SQL Standard

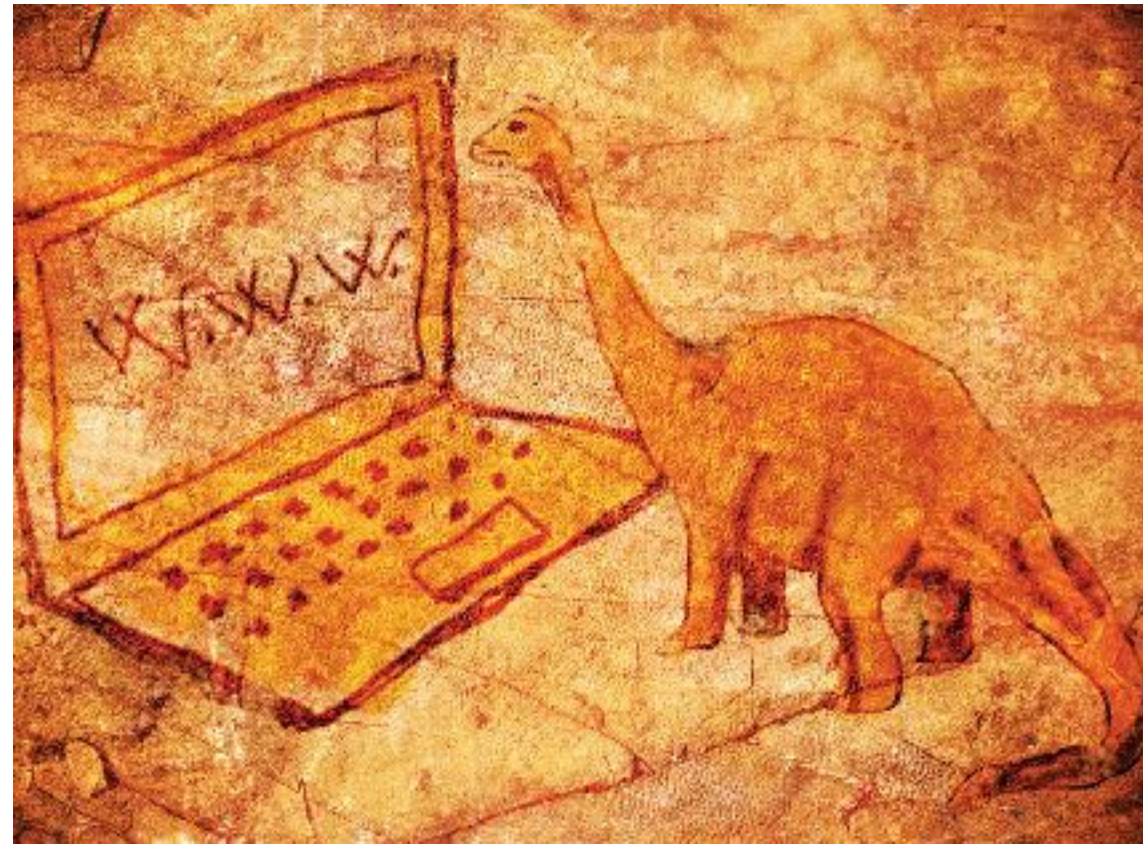
# GQL vs SQL/PGQ

- Pattern matching is **identical**
- Turns graphs into tables



- In **SQL/PGQ**:
  - works on a graph given as a relational view
  - results in a table defined in **FROM**
  - then continue with a SQL query
- In **GQL**:
  - works on a property graph
  - still produces a table
  - then additional operators modify this table
  - these can include additional pattern matching

# Timeline on Graph Query Languages



Ancient graph databases: Network model  
 CODASYL/NDL: 1959—1987  
*The first requisite of immortality is death*



Semantic Web  
 SPARQL 2004—



Modern graph querying  
 Neo4j/Cypher 2011  
 SQL/PgQ 2023  
 GQL 2024 —

Graph Query Languages Research		
1987: RPQs	2RPQs	2UCRPQs
1990: CRPQs	2CRPQs	ECRPQs
followed by many others	UCRPQs	RPDQs

1959

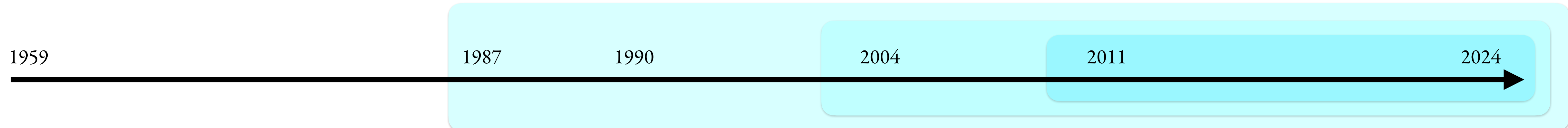
1987

1990

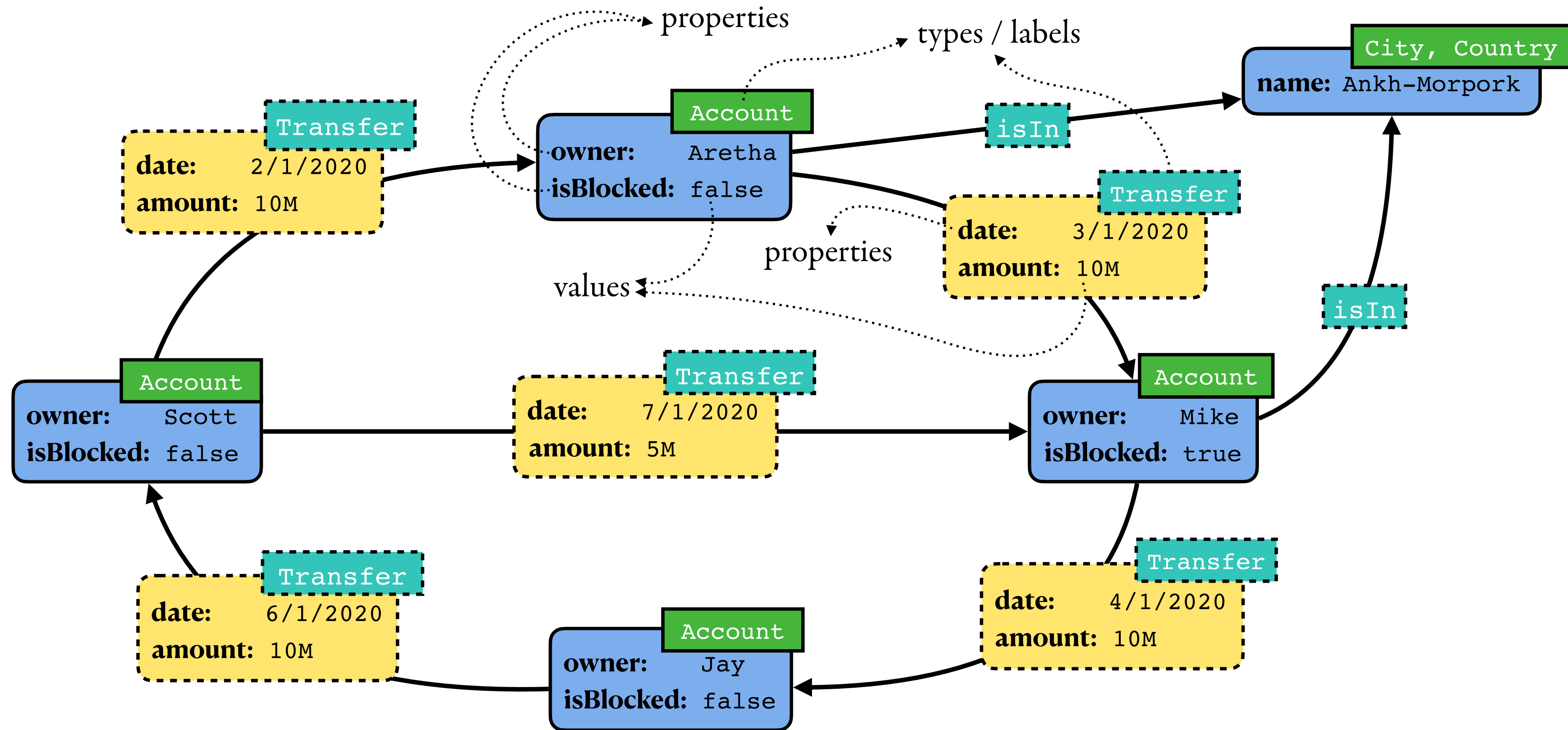
2004

2011

2024



# Data Model: Property Graphs



A data model based on graphs where both nodes and edges (relationships) can have

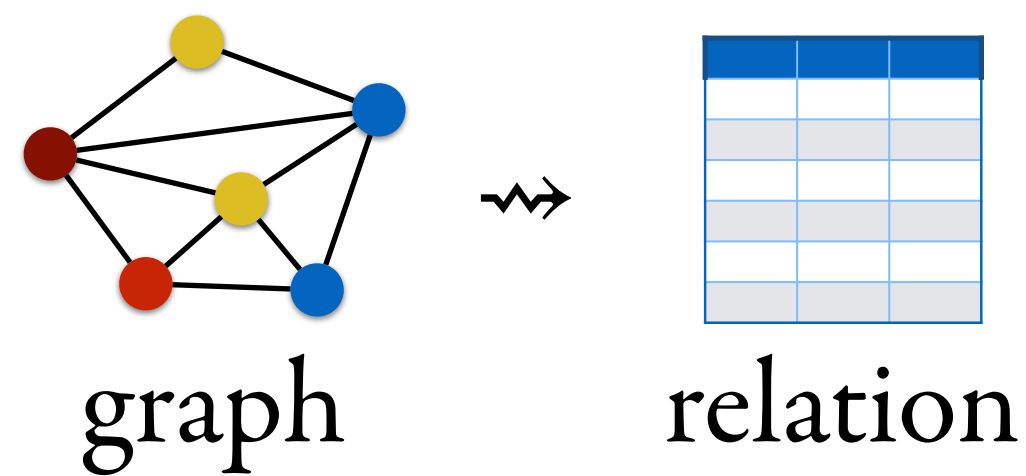
- **properties** (attributes)
- **types** (labels)

# GQL by examples

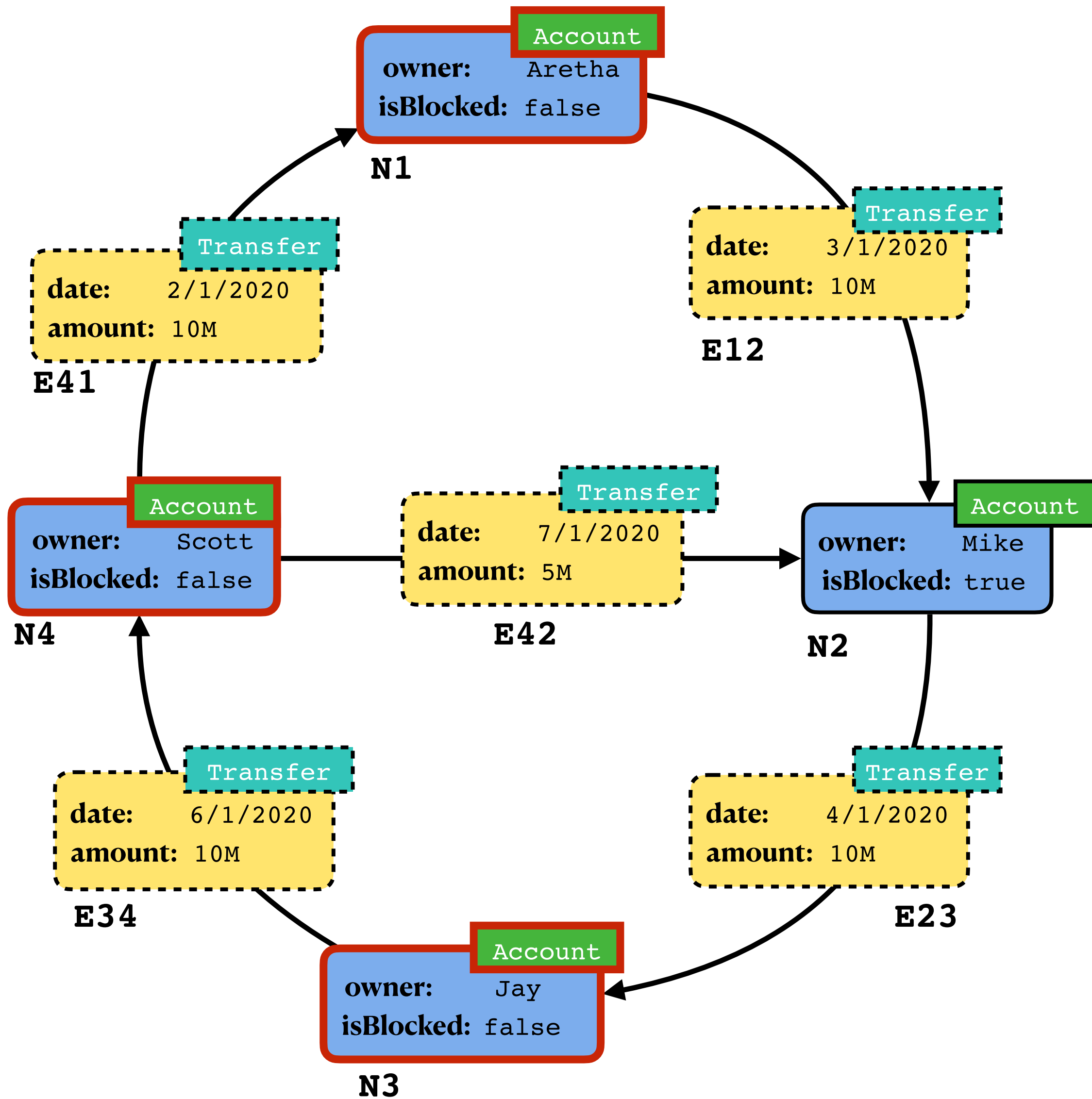
*Always expect to be disappointed and then you won't*



# The Core: Graph Pattern Matching



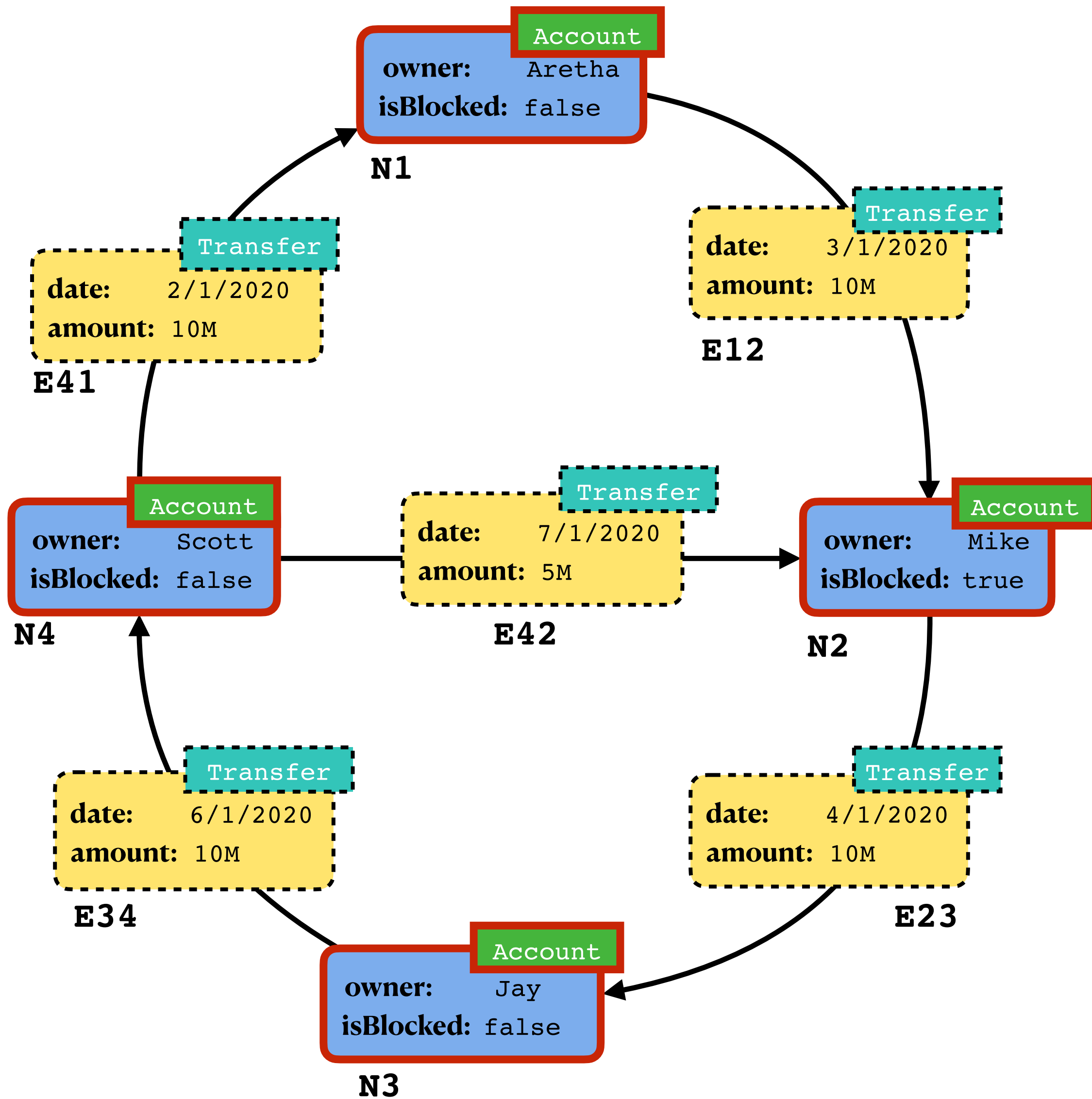
# Selecting Nodes



```
MATCH (x:Account)  
WHERE x.isBlocked = 'false'
```

x
N1
N3
N4

# Selecting Nodes

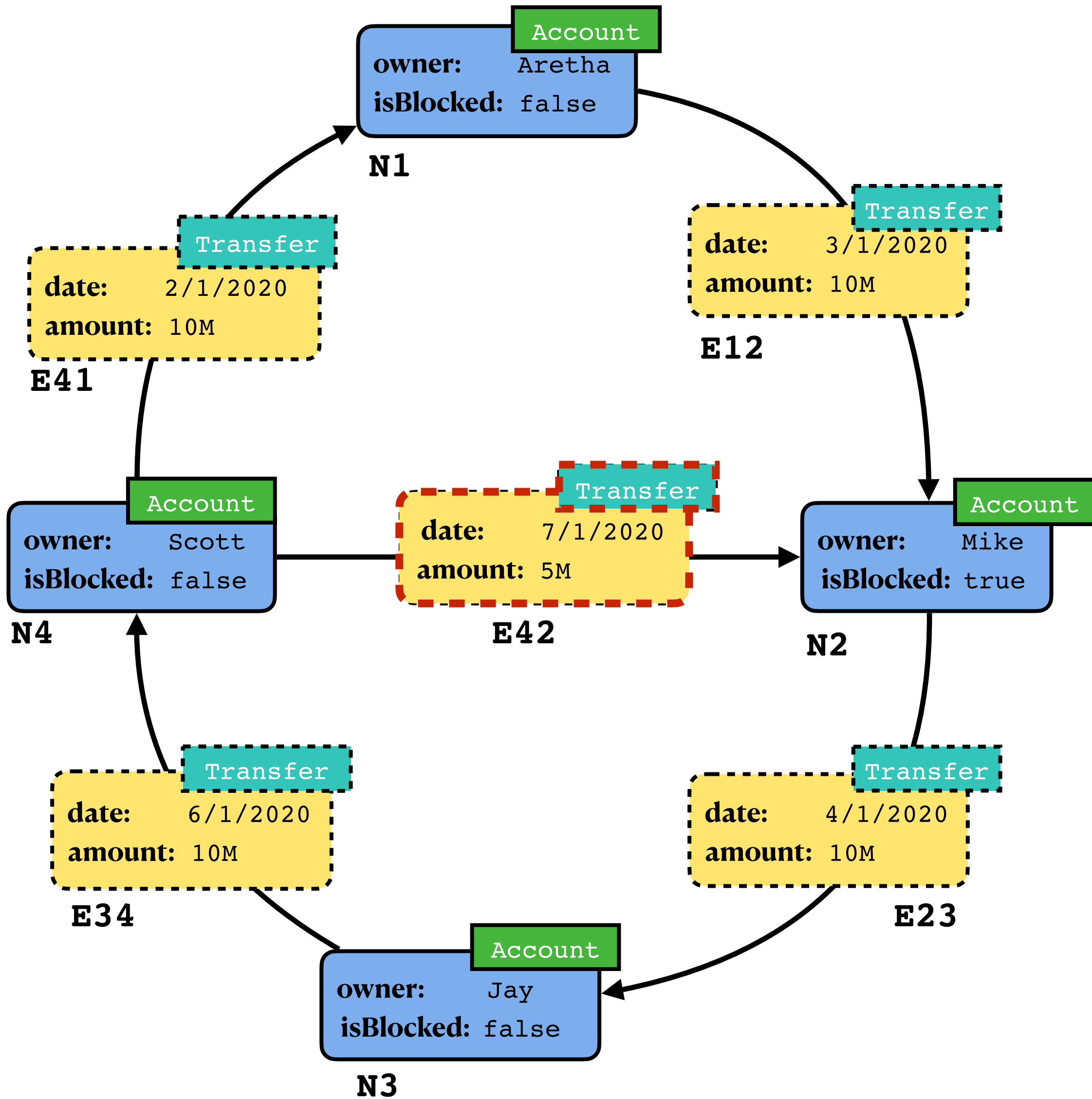


**MATCH (x)**

↔ all nodes

x
N1
N2
N3
N4

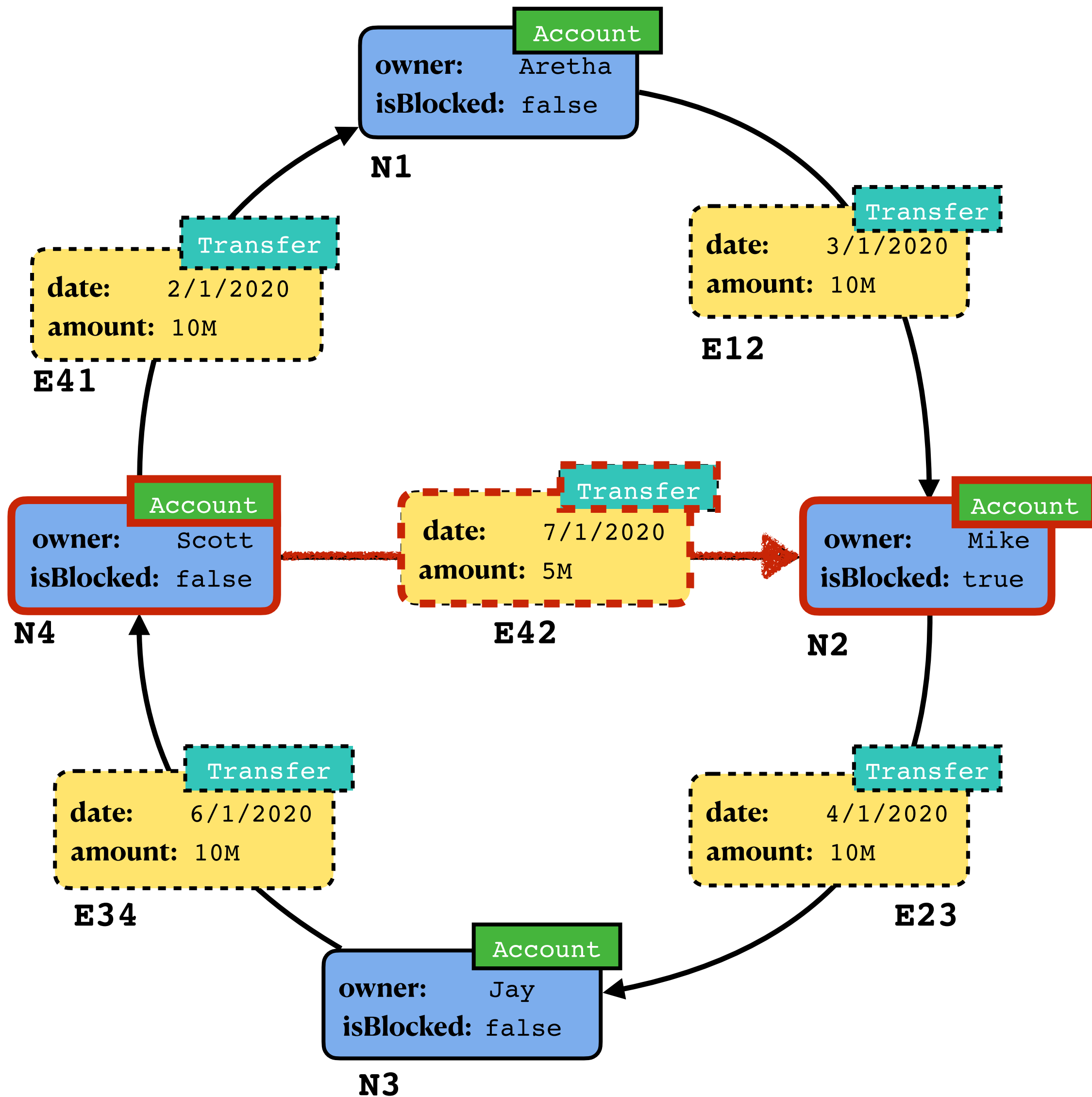
# Selecting Edges



```
MATCH [e:Transfer]  
WHERE e.amount < 10M
```



# Combining Nodes & Edges into Paths

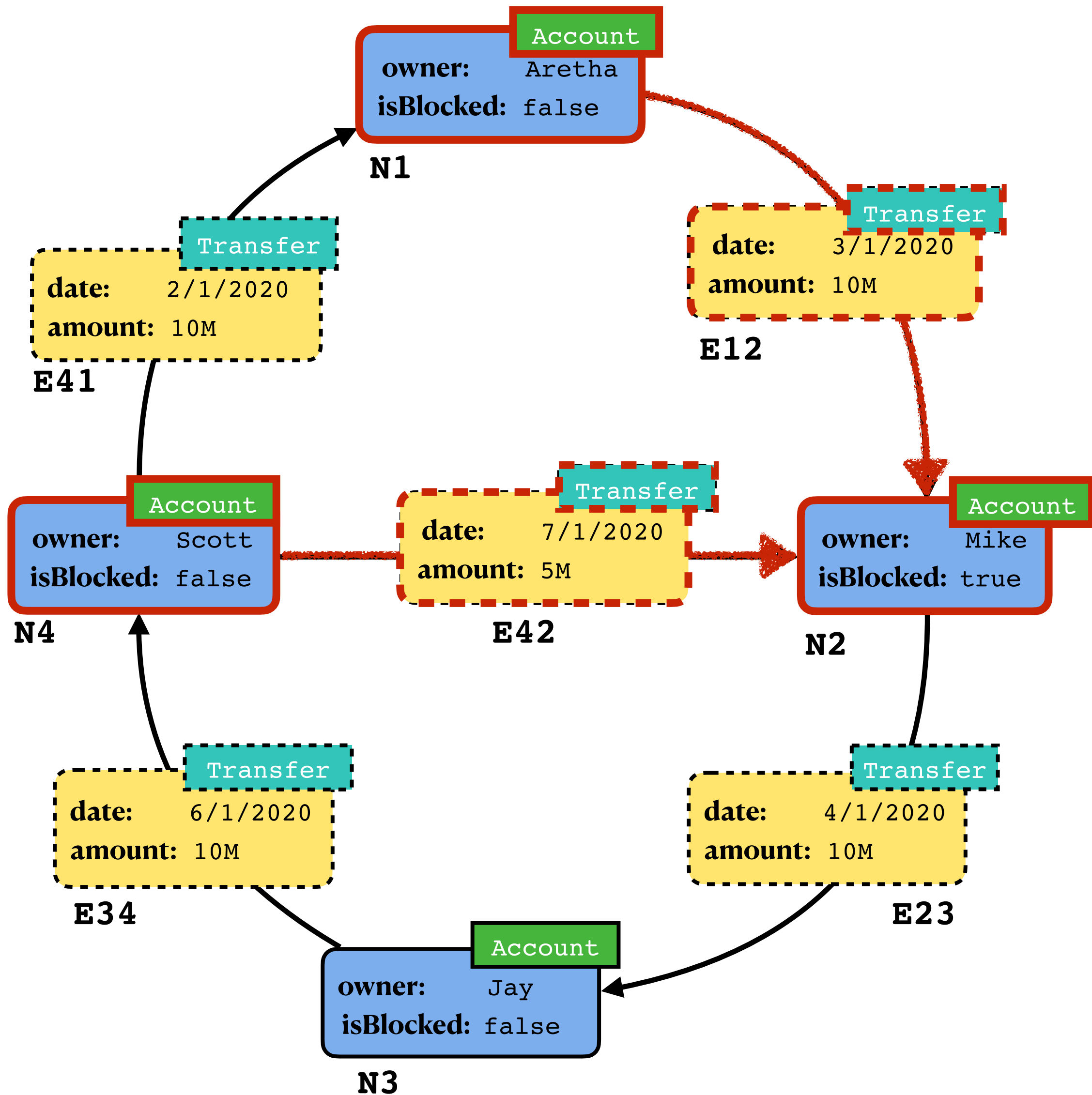


Combining nodes and edges:

```
MATCH (x)-[e:Transfer]->(y)
WHERE x.isBlocked = 'false'
AND y.isBlocked = 'true'
AND e.amount <= 5M
```

x	e	y
N4	E42	N2

# Combining Nodes & Edges into Paths

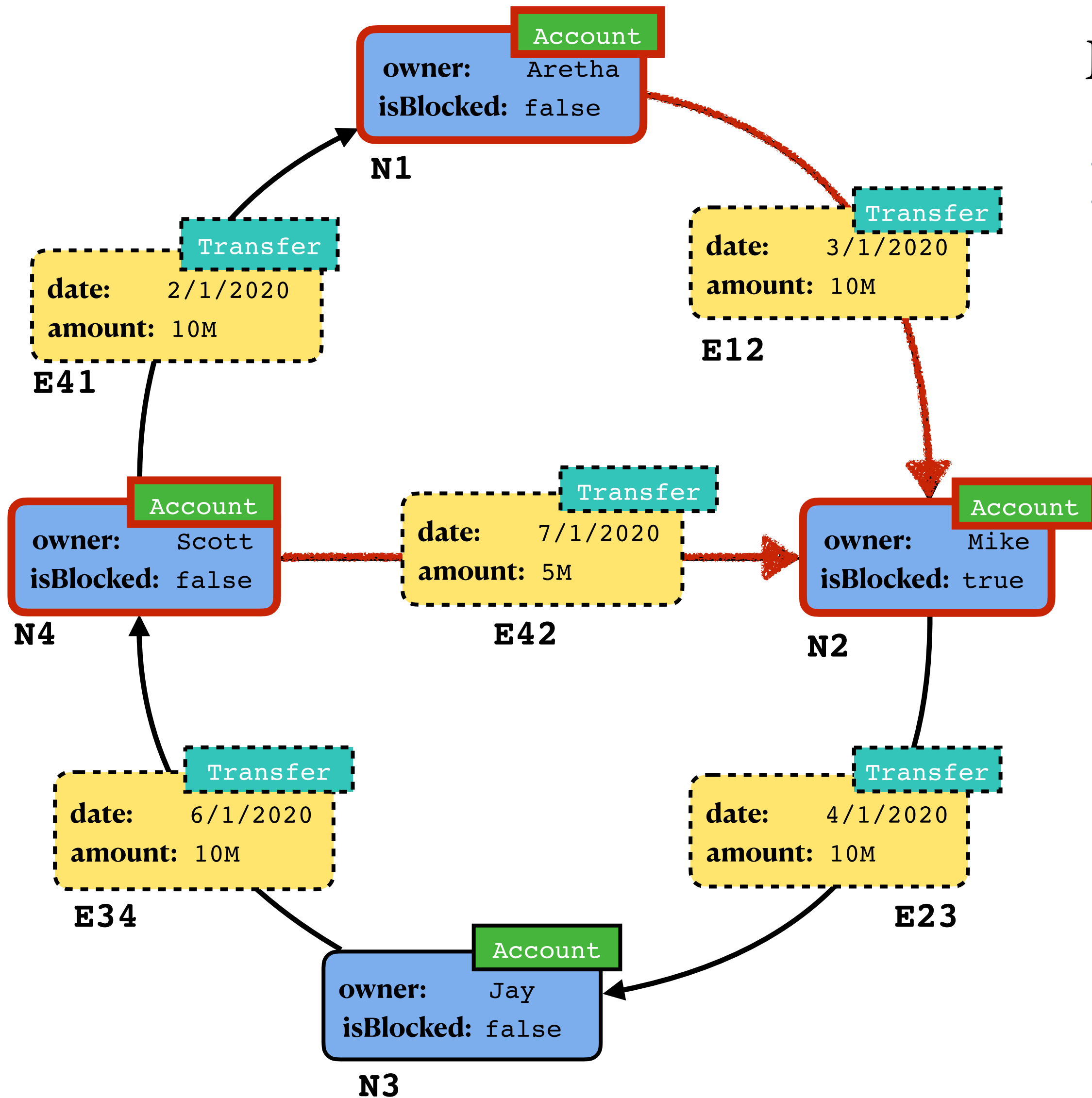


Combining nodes and edges:

```
MATCH (x)-[e:Transfer]->(y)  
WHERE x.isBlocked = 'false'  
AND y.isBlocked = 'true'
```

x	e	y
N4	E42	N2
N1	E12	N2

# Combining Nodes & Edges into Paths



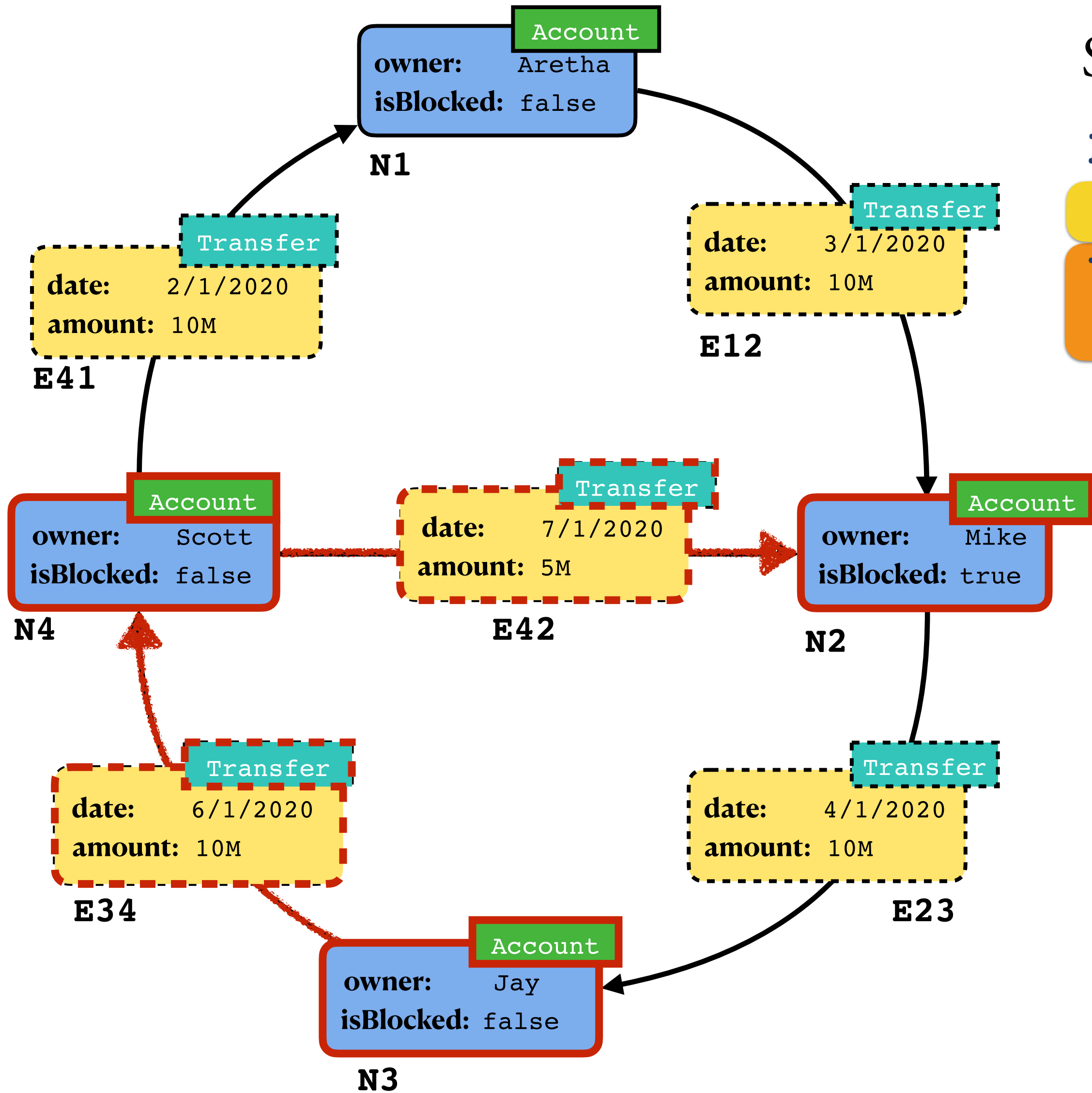
Longer paths are defined via ASCII-art :

```
MATCH (x)-[:Transfer]->(y)<-[:Transfer]-(z)
WHERE y.isBlocked = 'true'
```

x	y	z
N4	N2	N1
N1	N2	N4
N1	N2	N1
N4	N2	N4

Multiple edge options:  $\sim$ ,  $-$ ,  $->$ ,  $<-$

# Graph Traversal



Specifying graph traversal:

**MATCH**

```
(x:Account)-[t:Transfer]->{2,4}(y:Account)
```

```
WHERE x.isBlocked = 'false'
```

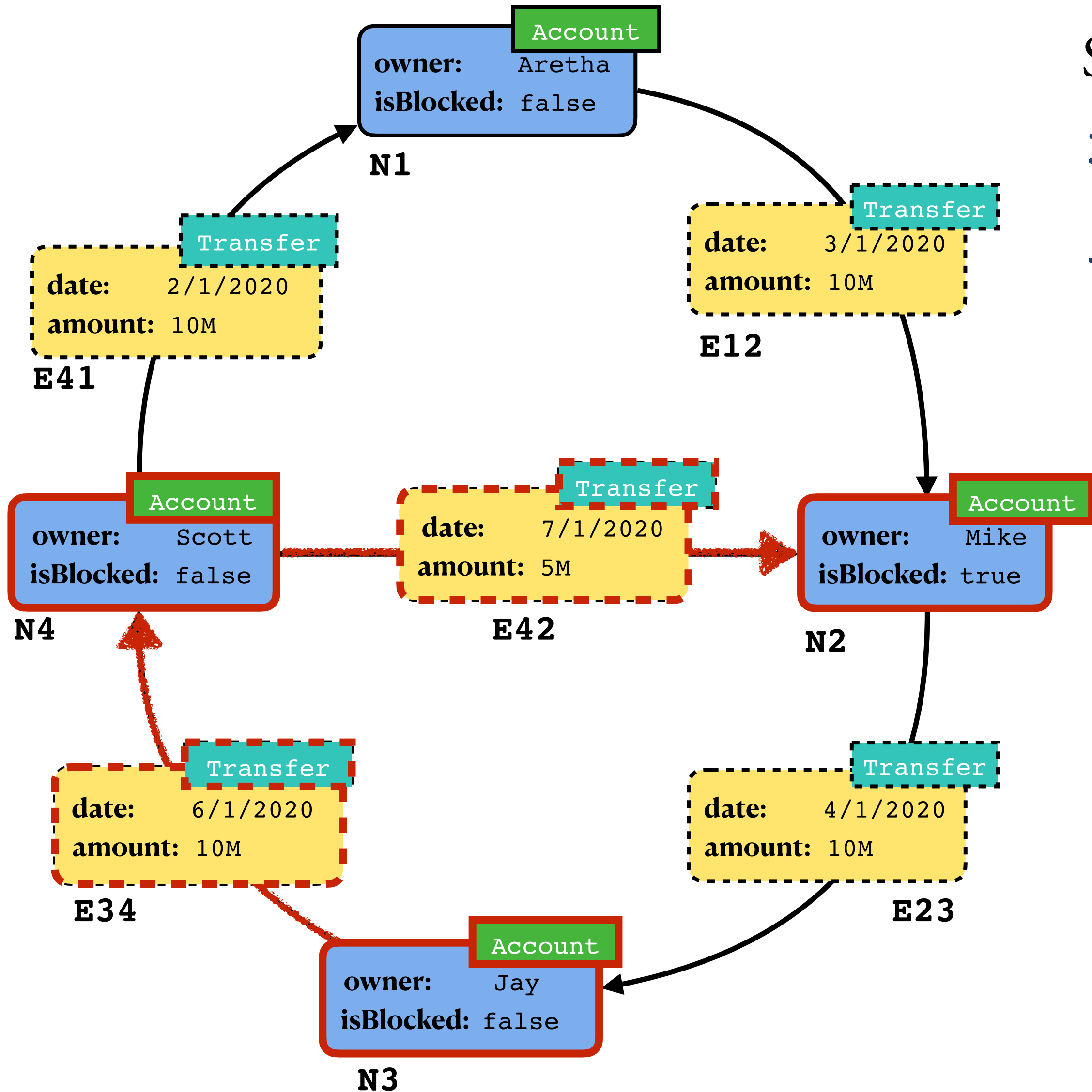
```
AND y.isBlocked = 'true'
```

Graph XPath

RPQs



# Graph Traversal



Specifying graph traversal:

**MATCH**

`(x:Account) -[t:Transfer]->{2,4}(y:Account)`

**WHERE** `x.isBlocked = 'false'`

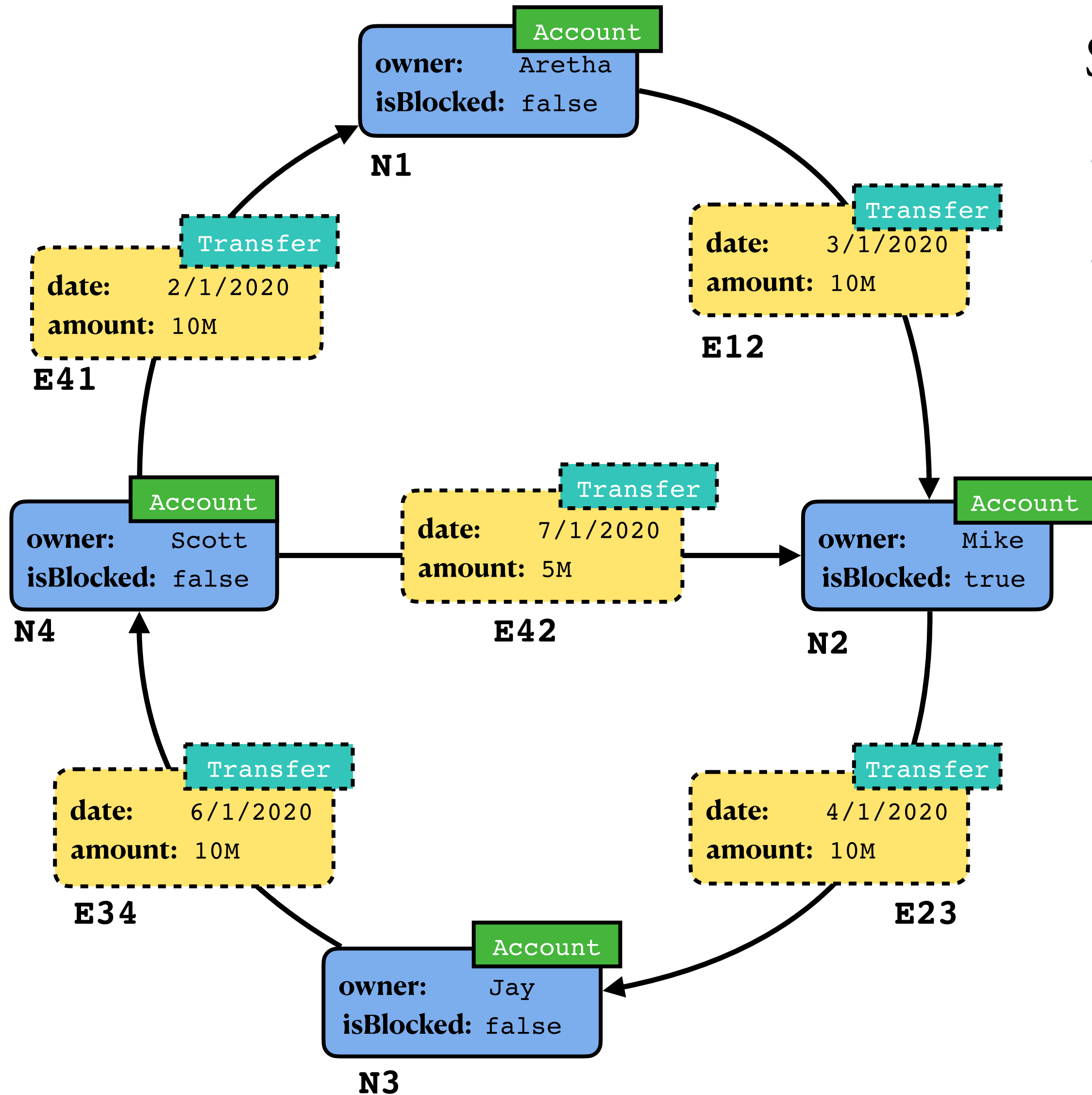
**AND** `y.isBlocked = 'true'`

group variable

x	t	y
N3	E34, E42	N2
N2	E23, E34, E42	N2
N1	E12, E23, E34, E42	N2
N4	E42, E23, E34, E42	N2

Group variables bind to lists of entities

# Graph Traversal



Specifying graph traversal:

**MATCH**

```
(x:Account)-[t:Transfer]->{2,4}(y:Account)
```

```
WHERE x.isBlocked = 'false'
```

```
AND y.isBlocked = 'true'
```

Repetitions can be

{n,m}

{n,}

{,m}

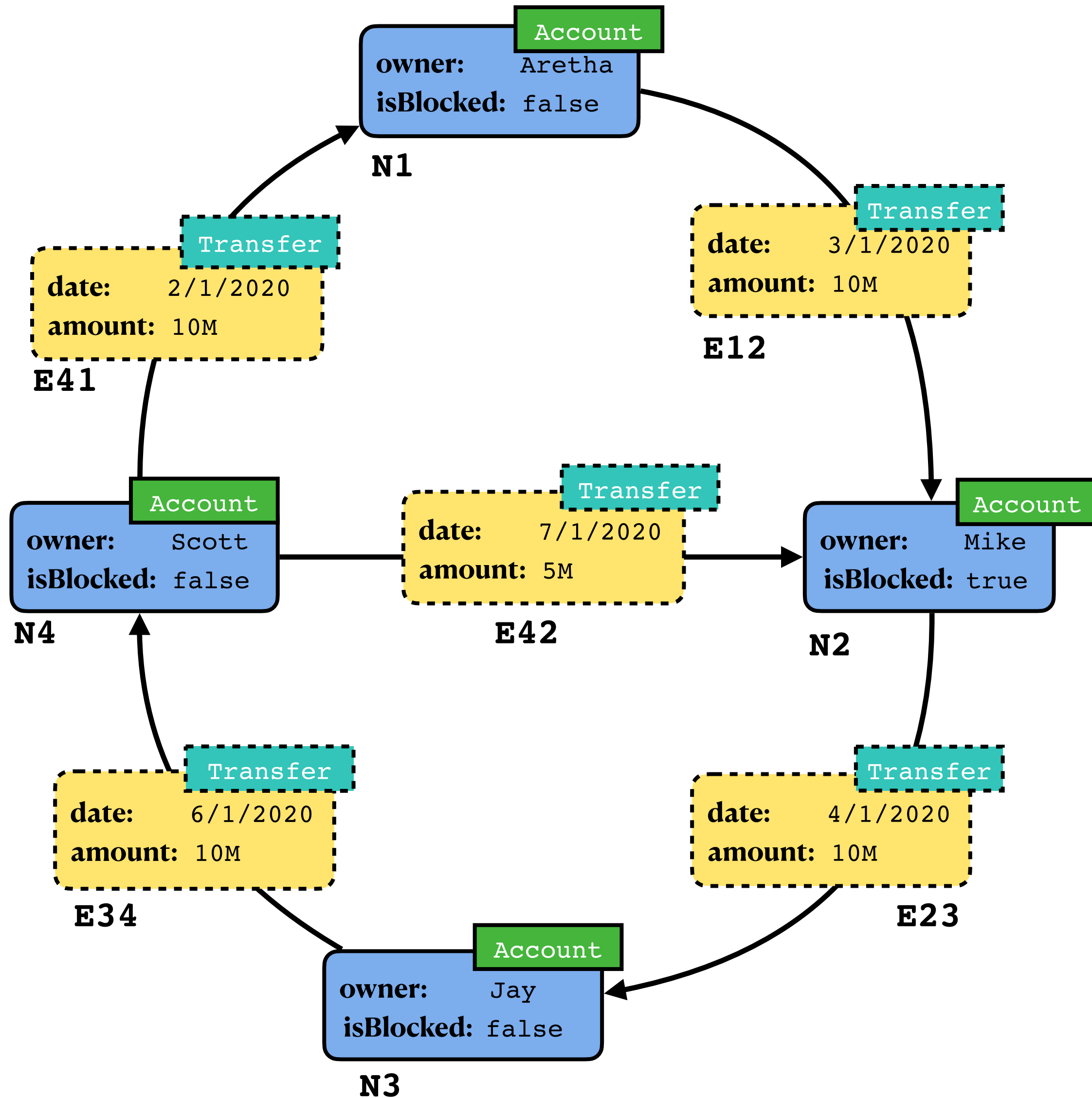
\*

+

Path conditions can be added:

```
-[t:Transfer WHERE t.amount > 7M]->{2,4}
```

# Path Variables



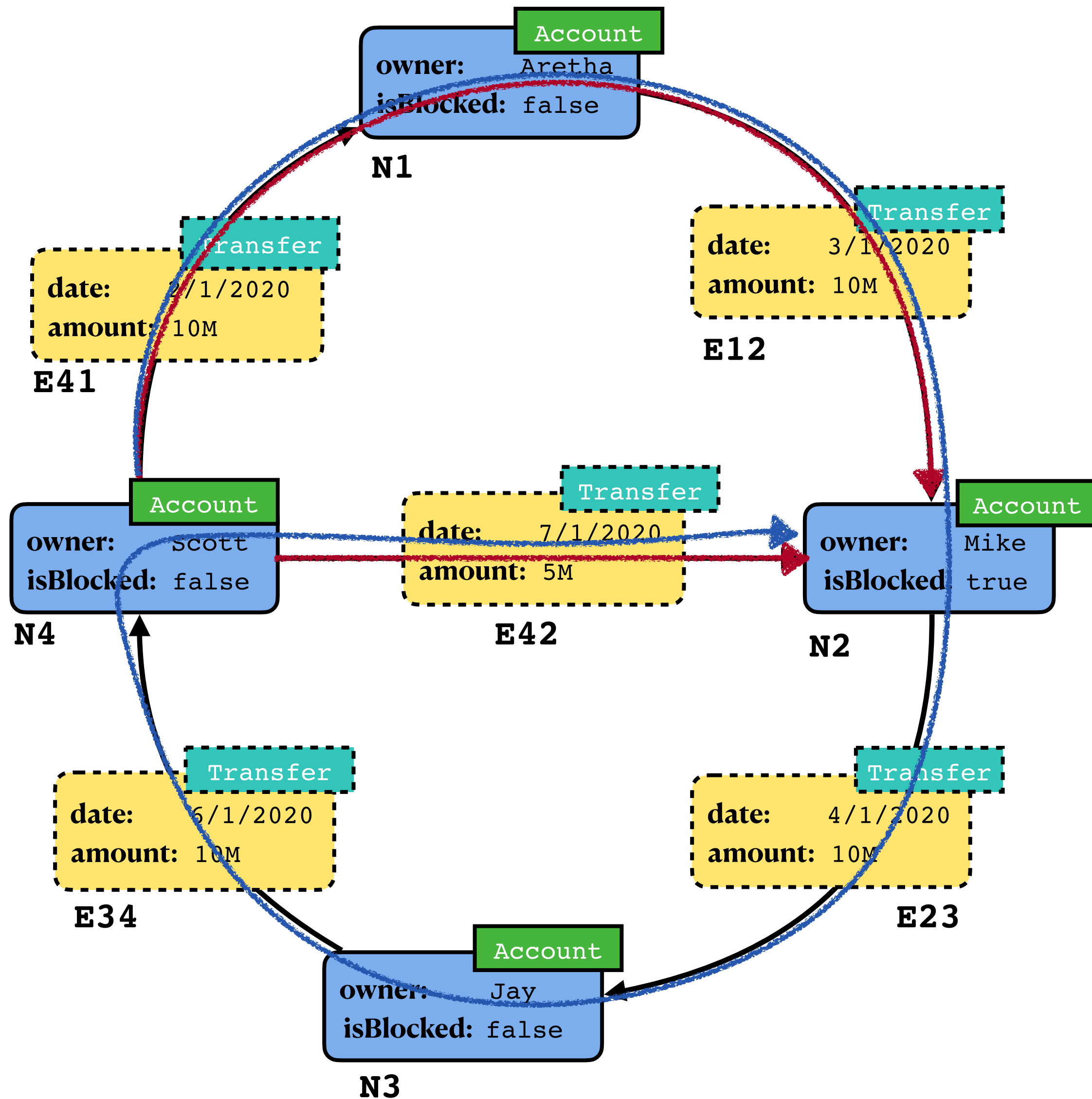
## MATCH

```
p = (x WHERE x.owner = 'Scott')  
    -[:Transfer]->*  
    (y WHERE y.owner = 'Mike')
```

But how can we return all such p ?  
(There are infinitely many...)

⇒ GQL uses **SIMPLE, TRAIL, SHORTEST**  
to ensure that only finitely many paths match

# Path Variables



## MATCH SIMPLE

```
p = (x WHERE x.owner = 'Scott')  
    -[:Transfer]->*  
    (y WHERE y.owner = 'Mike')
```

## MATCH TRAIL

```
p = (x WHERE x.owner = 'Scott')  
    -[:Transfer]->*  
    (y WHERE y.owner = 'Mike')
```

Also possible: **SHORTEST**, **ACYCLIC**

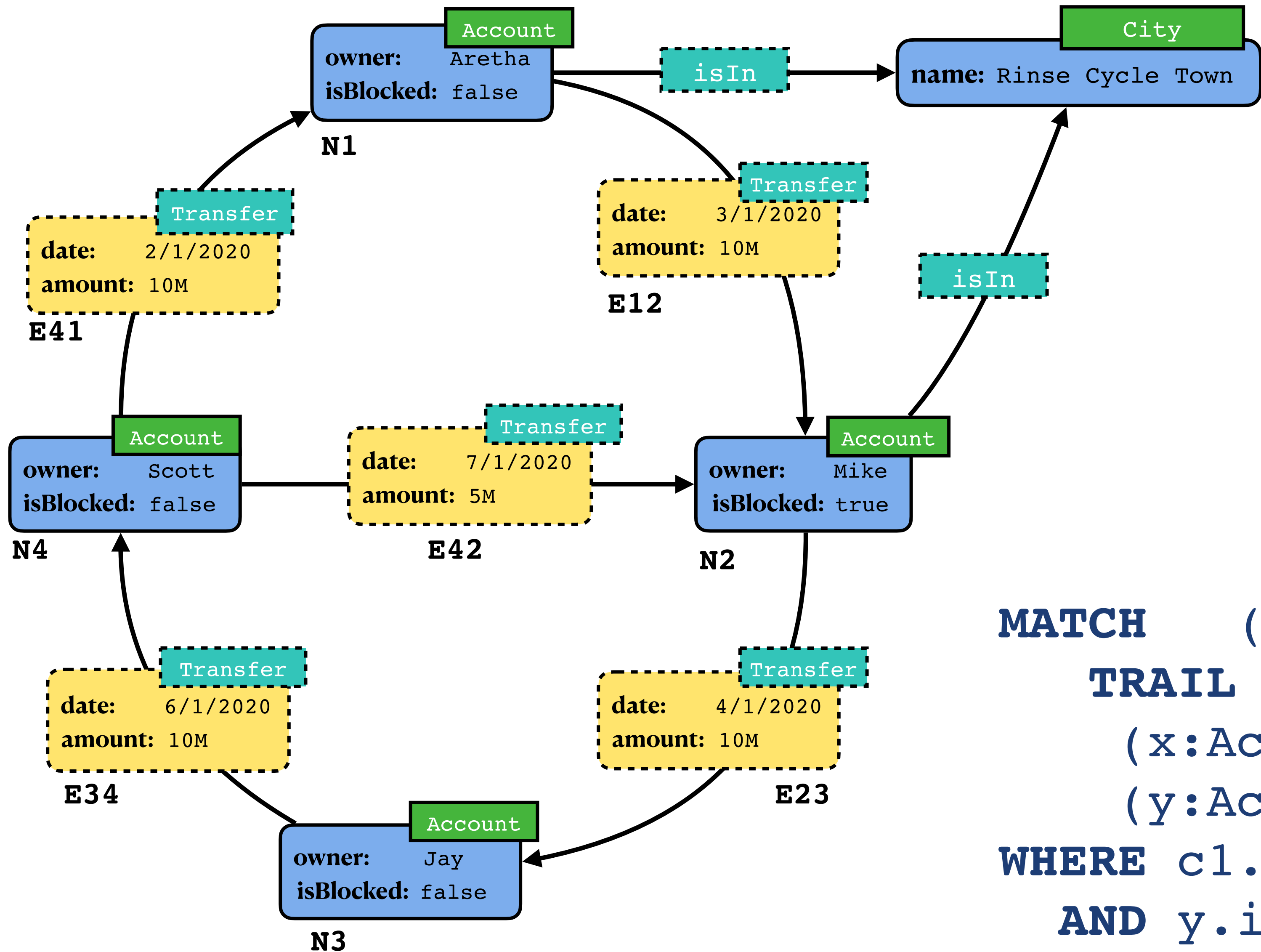
# Disjunction

*As expected, there is OR |*

```
MATCH (x)-[:Transfer]->(y) WHERE y.isBlocked = 'true' |  
        (x)-[:Transfer]->(y) WHERE x.owner = 'Mike'
```

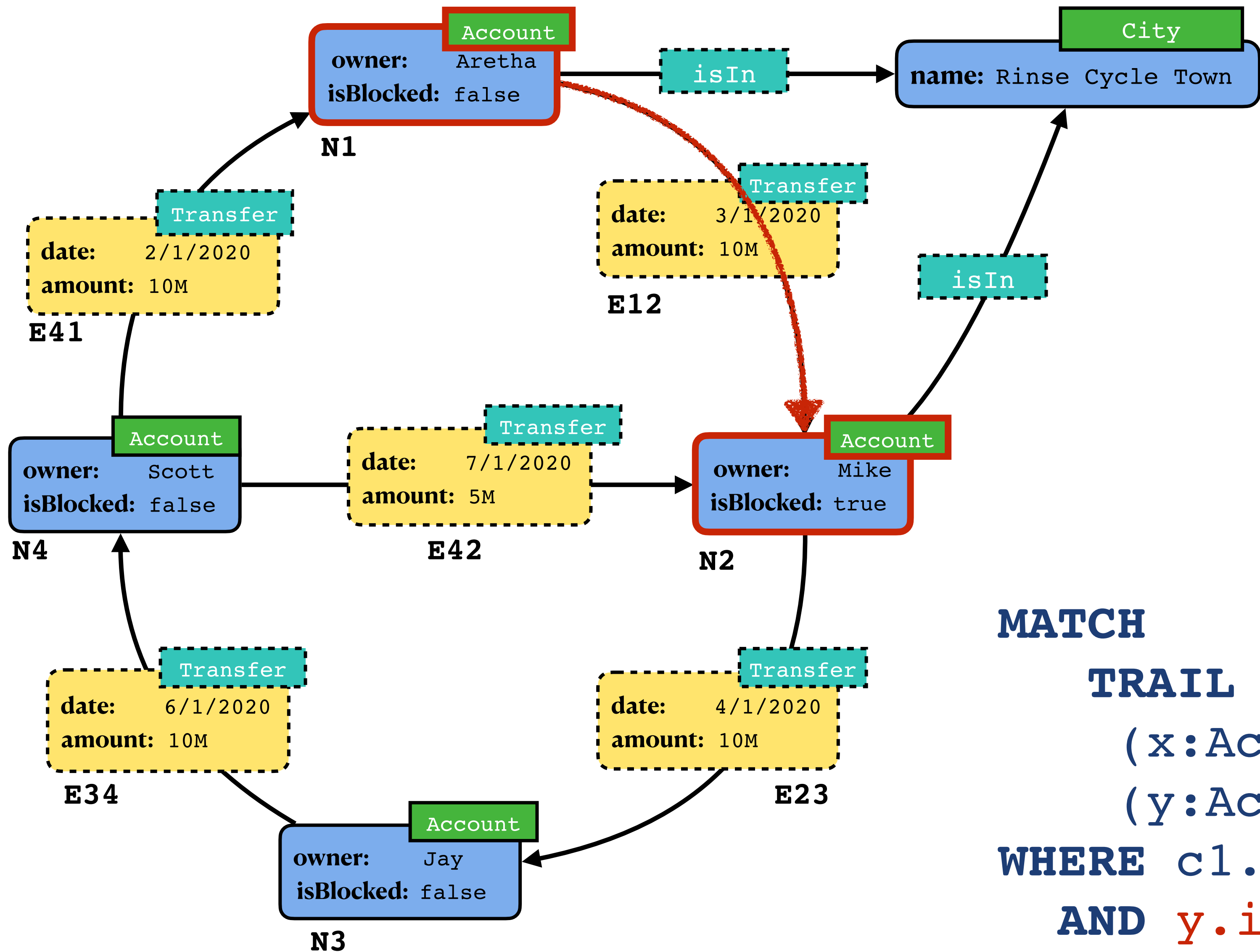
"Transfers to a blocked account and transfers initiated by Mike"

# Joins in Patterns



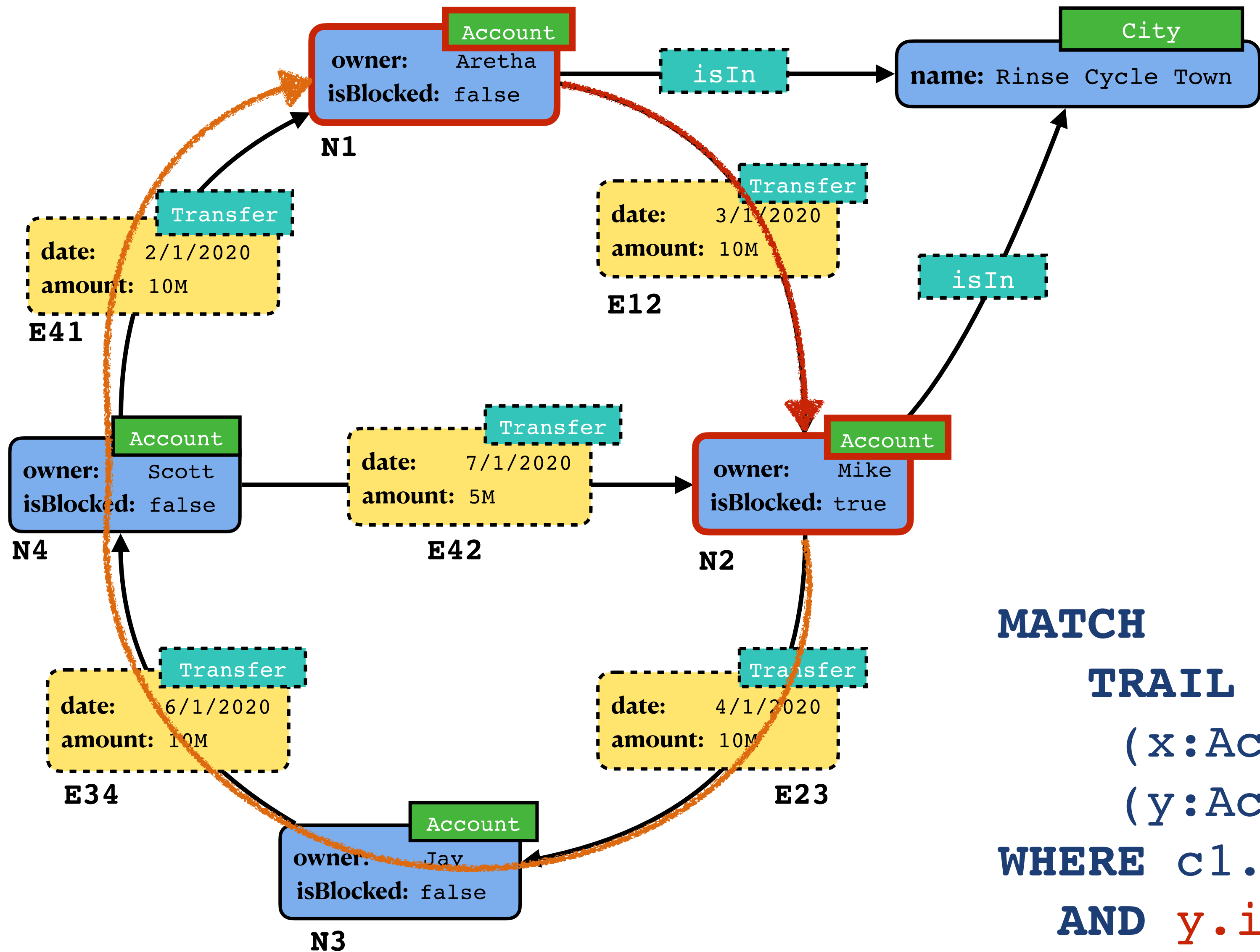
```
MATCH (x) -[:Transfer]-> (y),  
TRAIL (y) -[:Transfer]->+ (x),  
(x:Account)-[:isIn]->(c1:City),  
(y:Account)-[:isIn]->(c2:City)  
WHERE c1.name = c2.name  
AND y.isBlocked = 'true'
```

# Joins in Patterns



```
MATCH (x) -[:Transfer]-> (y),  
TRAIL (y) -[:Transfer]->+ (x),  
(x:Account)-[:isIn]->(c1:City),  
(y:Account)-[:isIn]->(c2:City)  
WHERE c1.name = c2.name  
AND y.isBlocked = 'true'
```

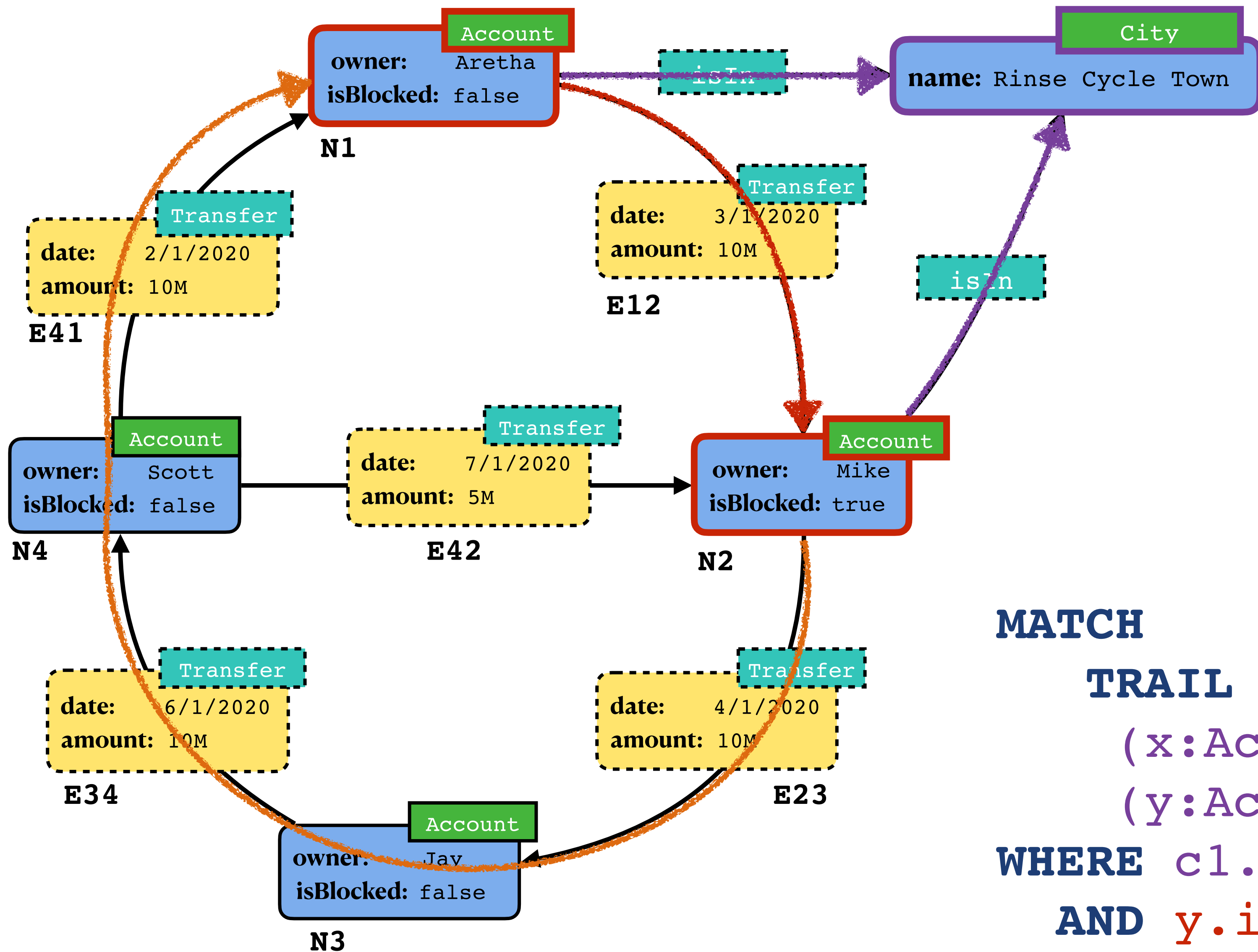
# Joins in Patterns



```
MATCH      (x) -[:Transfer]-> (y),  
TRAIL      (y) -[:Transfer]->+ (x),  
              (x:Account)-[:isIn]->(c1:City),  
              (y:Account)-[:isIn]->(c2:City)  
WHERE c1.name = c2.name  
AND y.isBlocked = 'true'
```



# Joins in Patterns

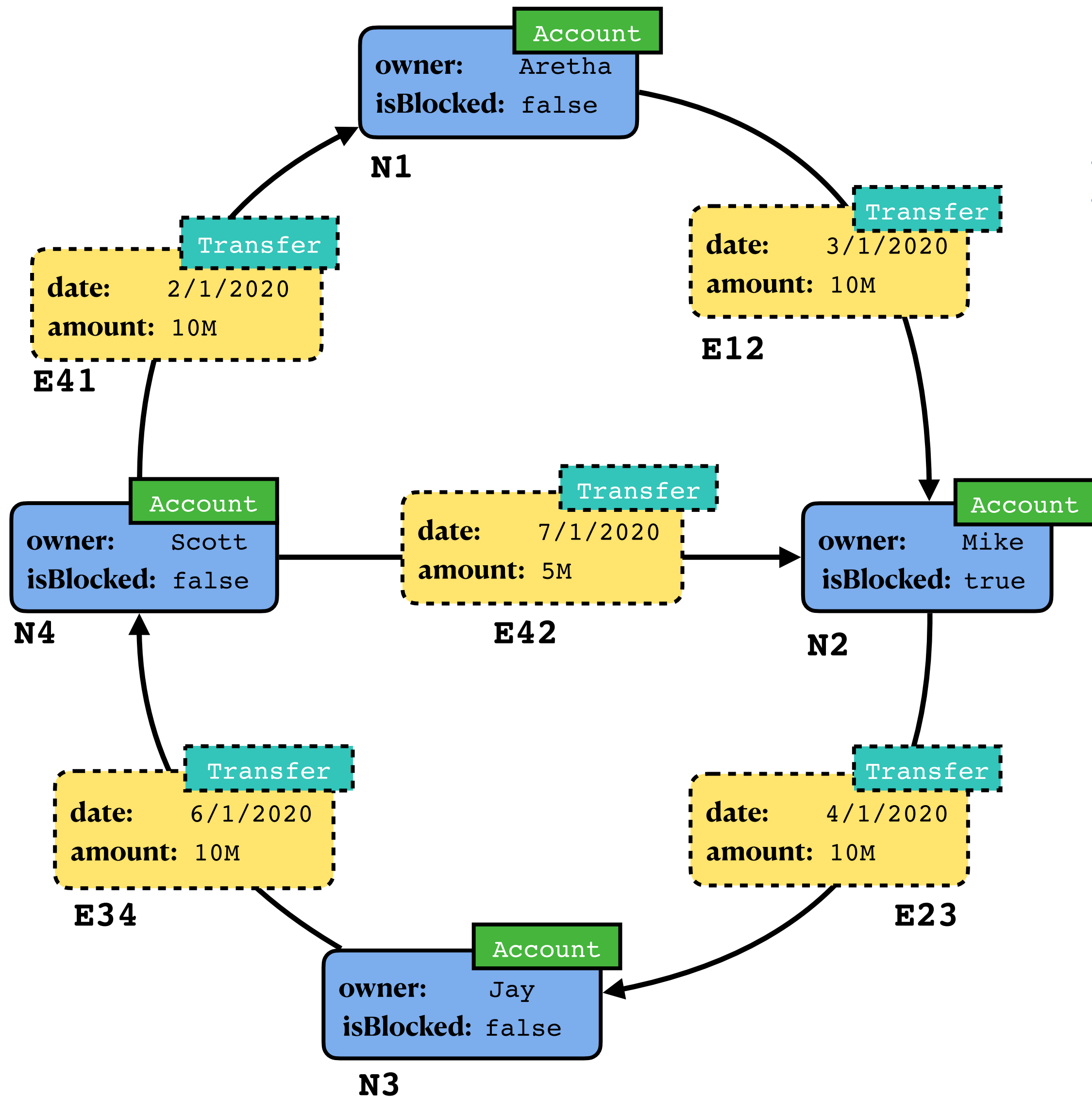


```

MATCH      (x) -[:Transfer]-> (y),
TRAIL      (y) -[:Transfer]->+ (x),
           (x:Account)-[:isIn]->(c1:City),
           (y:Account)-[:isIn]->(c2:City)
WHERE      c1.name = c2.name
AND        y.isBlocked = 'true'
    
```

# Manipulating Tables

# Return: a generalized projection



```

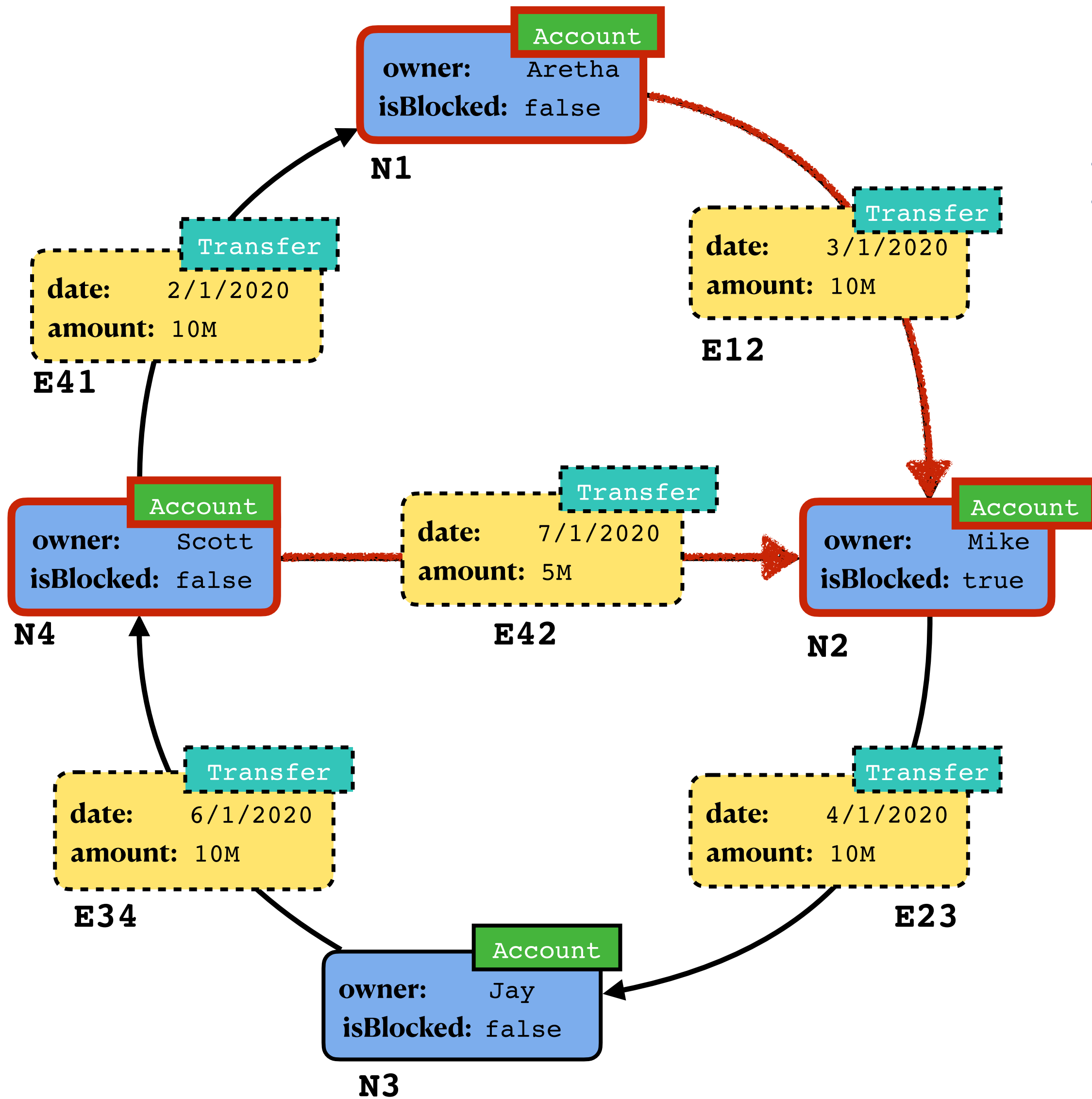
MATCH (x)-[:Transfer]->(y)<-[:Transfer]-(z)
WHERE y.isBlocked = 'true'
RETURN x.owner AS src1,
          y.owner AS tgt,
          z.owner AS src2
  
```

x	y	z
N4	N2	N1
N1	N2	N4
N1	N2	N1
N4	N2	N4

⇒

src1	tgt	src2
Scott	Mike	Aretha
Aretha	Mike	Scott
Aretha	Mike	Aretha
Scott	Mike	Scott

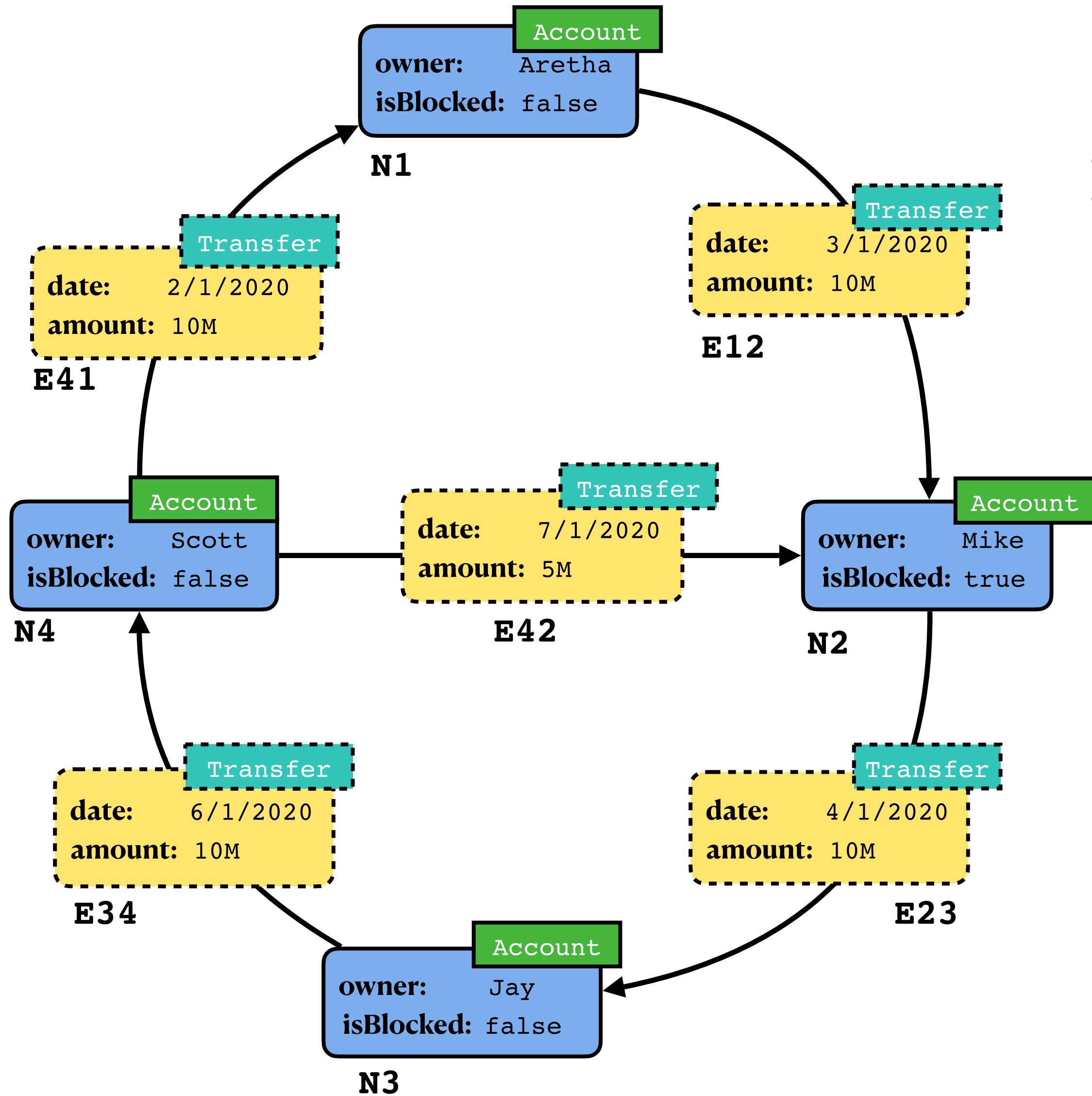
# Let



```
MATCH (x)-[:Transfer]->(y)<-[:Transfer]-(z)
WHERE y.isBlocked = 'true'
LET w = x.owner
```

x	y	z	w
N4	N2	N1	Scott
N1	N2	N4	Aretha
N1	N2	N1	Aretha
N4	N2	N4	Scott

# Filter



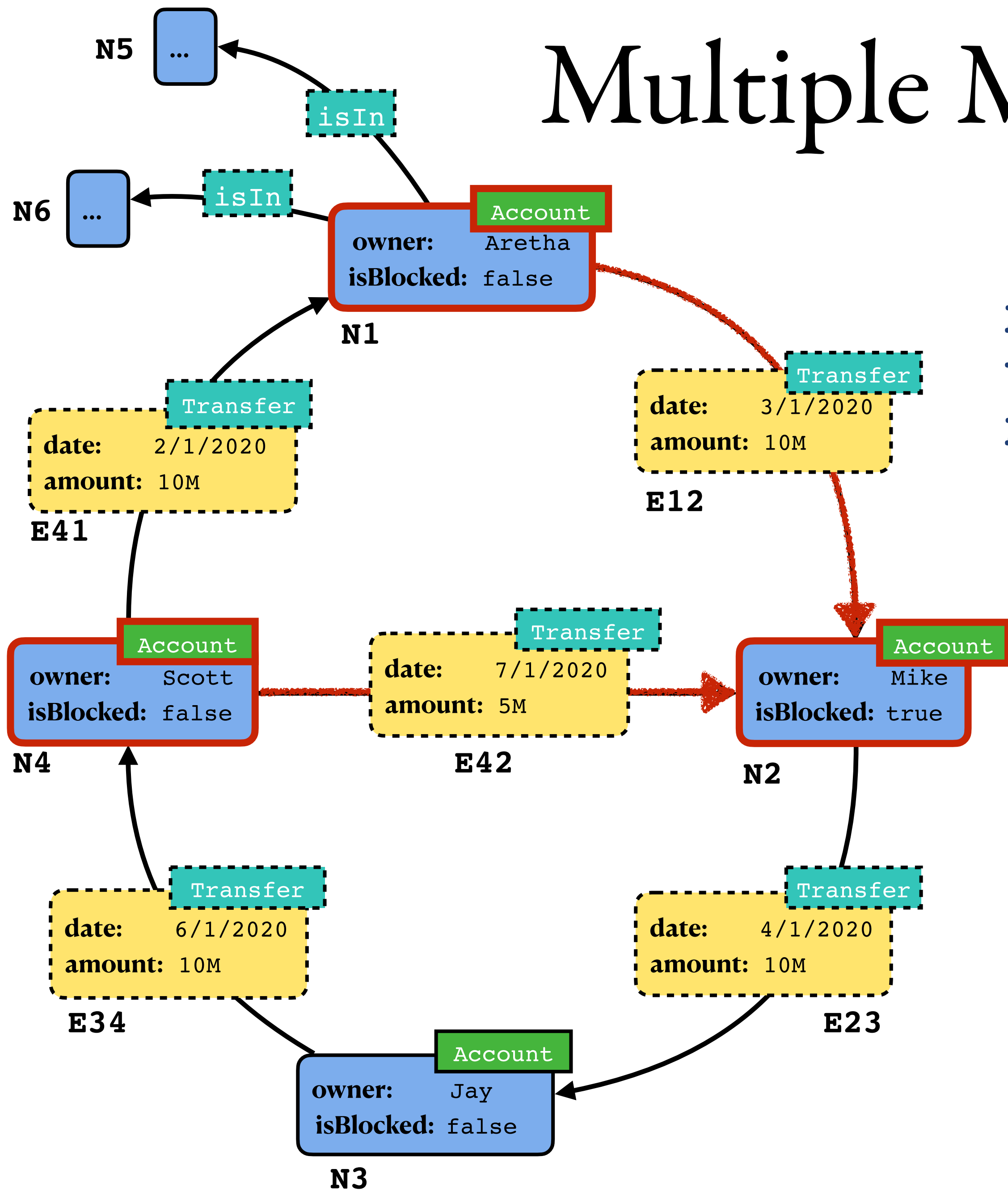
**MATCH** (x)-[:Transfer]->(y)<-[:Transfer]-(z)  
**WHERE** y.isBlocked = 'true'  
**FILTER NOT** (x = y)

x	y	z
N4	N2	N1
N1	N2	N4
N1	N2	N1
N4	N2	N4

⇒

x	y	z
N4	N2	N1
N1	N2	N4

# Multiple Match-Statements



**MATCH** (x)-[:Transfer]->(y)<-[:Transfer]-(z)  
**WHERE** y.isBlocked = 'true'  
**MATCH** (w)<-[:isIn]-(z)

x	y	z
N4	N2	N1
N1	N2	N4
N1	N2	N1
N4	N2	N4

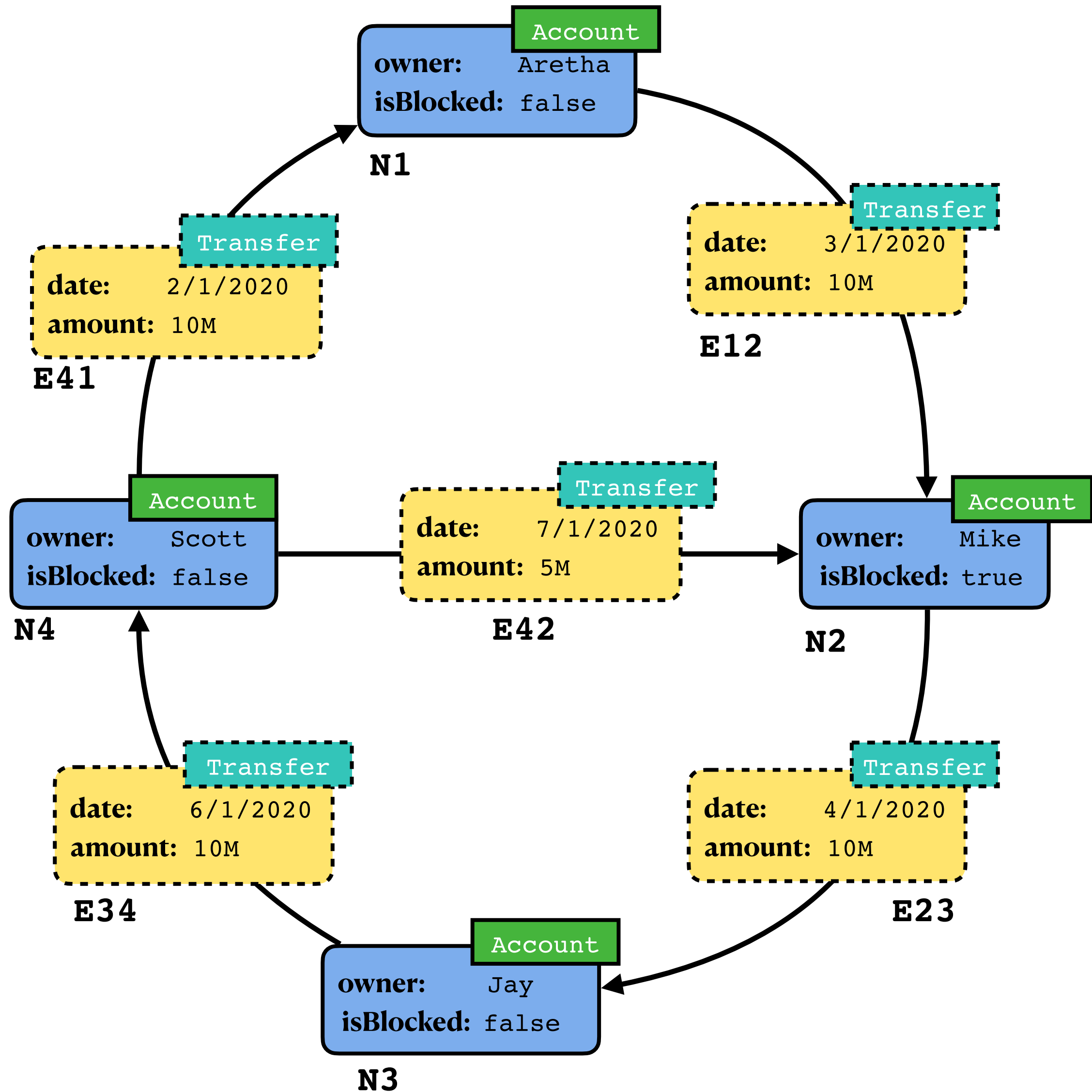
⋈

z	w
N1	N5
N1	N6

Final result:

x	y	z	w
N4	N2	N1	N5
N4	N2	N1	N6
N1	N2	N1	N5
N1	N2	N1	N6

# For x in y



## MATCH

$(u:\text{Account}) - [y:\text{Transfer}] \rightarrow \{2, 4\} (v:\text{Account})$

WHERE  $u.\text{isBlocked} = \text{'false'}$

AND  $v.\text{isBlocked} = \text{'true'}$

FOR x IN y

u	y	v
N3	E34, E42	N2
N2	E23, E34, E42	N2
N1	E12, E23, E34, E42	N2
N4	E42, E23, E34, E42	N2

first row

u	y	v	x
N3	E34, E42	N2	E34
N3	E34, E42	N2	E42

$4 + 4 + 3 = 11$  additional rows

# Set Operations

## Union, Intersection, Difference

If  $Q_1$  and  $Q_2$  are GQL queries, then so are

- $Q_1$  **UNION**  $Q_2$
- $Q_1$  **INTERSECT**  $Q_2$
- $Q_1$  **EXCEPT**  $Q_2$

Since both  $Q_1$  and  $Q_2$  produce tables, these operations work as one would expect in relational DBs



How to do research on  
GQL and PGQ?

# GQL looks like 500+ pages of this:

	IWD 39075:202y(E)
	16.10 <path pattern expression>
<b>16.10 &lt;path pattern expression&gt;</b>	
<b>Function</b>	
Specify a pattern to match a single path in a property graph.	
<b>Format</b>	
<path pattern expression> ::= <path term>   <path multiset alternation>   <path pattern union> <path multiset alternation> ::= <path term> <multiset alternation operator> <path term>   { <multiset alternation operator> <path term> [...] } <path pattern union> ::= <path term> <vertical bar> <path term> [ { <vertical bar> <path term> [...] } ] <path term> ::= <path factor>   <path concatenation> <path concatenation> ::= <path term> <path factor> <path factor> ::= <path primary>   <quantified path primary>   <questioned path primary> <quantified path primary> ::= <path primary> <graph pattern quantifier> <questioned path primary> ::= <path primary> <question mark> NOTE 131 — Unlike most regular expression languages, <question mark> is not equivalent to the quantifier {0,1}; the quantifier {0,1} exposes variables as group, whereas <question mark> does not change the singleton variables that it exposes to group. However, <question mark> does expose any singleton variables as conditional singletons. <path primary> ::= <element pattern>   <parenthesized path pattern expression>   <simplified path pattern expression> <element pattern> ::= <node pattern>   <edge pattern> <node pattern> ::= <left paren> <element pattern filler> <right paren> <element pattern filler> ::= [ <element variable declaration> ]   <is label expression> ]   <element pattern predicate> ] + WG3:W24-022 + <element variable declaration> ::=	

	IWD 39075:202y(E)
	16.10 <path pattern expression>
[ TEMP ] <element variable>	
<is label expression> ::=	
<is or color> <label expression>	
<is or color> ::=	
is	
<element pattern predicate> ::=	
<left brace> <property key value pair list> <right brace>	
<element pattern where clause> ::=	
WHERE <search condition>	
<element property specification> ::=	
<left brace> <property key value pair list> <right brace>	
<property key value pair list> ::=	
<property key value pair> [ { <comma> <property key value pair> [...] } ]	
<property key value pair> ::=	
<property name> <colon> <value expression>	
<edge pattern> ::=	
<full edge pattern>	
<abbreviated edge pattern> ::=	
<left arrow>	
<right arrow>	
<left arrow tilde>	
<right arrow tilde>	
<left minus right>	
<minus sign>	
+ WG3:W24-038 deleted one Editor's Note +	
<parenthesized path pattern expression> ::=	
[ <path mode prefix> ]	
<path pattern expression>	
[ <parenthesized path pattern where clause> ]	
<right paren>	
<subpath variable declaration> ::=	
<subpath variable> <equals operator>	
<parenthesized path pattern where clause> ::=	
WHERE <search condition>	
<b>Syntax Rules</b>	
1) Let <i>RIGHTMINUS</i> be the following collection of <tokens>: <right bracket minus>, <left arrow>, <slash minus>, and <minus sign>.	
NOTE 132 — These are the tokens ], <./, and -, which expose a minus sign on the right.	
2) Let <i>LEFTMINUS</i> be the following collection of <tokens>: <minus left bracket>, <right arrow>, <minus slash>, and <minus sign>.	
NOTE 133 — These are the tokens [,<./, and -, which expose a minus sign on the left. <minus sign> itself is both <i>RIGHTMINUS</i> and <i>LEFTMINUS</i> .	
3) A <path pattern expression> shall not juxtapose a <tokens> from <i>RIGHTMINUS</i> followed by a <tokens> from <i>LEFTMINUS</i> without a <separator> between them.	
NOTE 134 — Otherwise, the concatenation of the two tokens would include the sequence of two <minus signs>, which is a <simple comment introducer>.	
4) A <path pattern expression> that contains at the same depth of graph pattern matching a variable quantifier, a <questioned path primary>, a <path multiset alternation>, or a <path pattern union> is a <possibly variable length path pattern>.	
5) A <path pattern expression> that is not a <possibly variable length path pattern> is a <fixed length path pattern>.	
6) The <minimum path length of certain BNF non-terminals defined in this Subclause is defined recursively as follows: a) The minimum path length of a <node pattern> is 0 (zero). b) The minimum path length of an <edge pattern> is 1 (one).	
** Editor's Note (number 73) **	
In the BNF for <full edge any direction>, the delimiter tokens <[ ]> have been suggested as a synonym for [ ] as part of Feature G047, "Indirect and Inverted". The synonym for the <abbreviated edge pattern> (<minus sign>) would then be <-->, the synonym for <simplified defaulting any direction> would use the delimiter tokens <~/~/> and the synonym for	

	IWD 39075:202y(E)
	16.10 <path pattern expression>
<simplified override any direction> would use the tokens <--> surrounding a label as originally proposed in WG3:MMX-060. These synonyms might be considered to make the table of edge patterns more harmonious and internally consistent. See Language Opportunity [GQL-212]	
<abbreviated edge pattern> ::=	
<left arrow>	
<right arrow>	
<left arrow tilde>	
<right arrow tilde>	
<left minus right>	
<minus sign>	
+ WG3:W24-038 deleted one Editor's Note +	
<parenthesized path pattern expression> ::=	
[ <path mode prefix> ]	
<path pattern expression>	
[ <parenthesized path pattern where clause> ]	
<right paren>	
<subpath variable declaration> ::=	
<subpath variable> <equals operator>	
<parenthesized path pattern where clause> ::=	
WHERE <search condition>	
<b>Syntax Rules</b>	
1) Let <i>RIGHTMINUS</i> be the following collection of <tokens>: <right bracket minus>, <left arrow>, <slash minus>, and <minus sign>.	
NOTE 132 — These are the tokens ], <./, and -, which expose a minus sign on the right.	
2) Let <i>LEFTMINUS</i> be the following collection of <tokens>: <minus left bracket>, <right arrow>, <minus slash>, and <minus sign>.	
NOTE 133 — These are the tokens [,<./, and -, which expose a minus sign on the left. <minus sign> itself is both <i>RIGHTMINUS</i> and <i>LEFTMINUS</i> .	
3) A <path pattern expression> shall not juxtapose a <tokens> from <i>RIGHTMINUS</i> followed by a <tokens> from <i>LEFTMINUS</i> without a <separator> between them.	
NOTE 134 — Otherwise, the concatenation of the two tokens would include the sequence of two <minus signs>, which is a <simple comment introducer>.	
4) A <path pattern expression> that contains at the same depth of graph pattern matching a variable quantifier, a <questioned path primary>, a <path multiset alternation>, or a <path pattern union> is a <possibly variable length path pattern>.	
5) A <path pattern expression> that is not a <possibly variable length path pattern> is a <fixed length path pattern>.	
6) The <minimum path length of certain BNF non-terminals defined in this Subclause is defined recursively as follows: a) The minimum path length of a <node pattern> is 0 (zero). b) The minimum path length of an <edge pattern> is 1 (one).	
** Editor's Note (number 73) **	
In the BNF for <full edge any direction>, the delimiter tokens <[ ]> have been suggested as a synonym for [ ] as part of Feature G047, "Indirect and Inverted". The synonym for the <abbreviated edge pattern> (<minus sign>) would then be <-->, the synonym for <simplified defaulting any direction> would use the delimiter tokens <~/~/> and the synonym for	

	IWD 39075:202y(E)
	16.10 <path pattern expression>
<minimum path length of a <path concatenation> is the sum of the minimum path lengths of its operands.	
d) The minimum path length of a <path pattern union> or <path multiset alternation> is the minimum of the minimum path length of its operands.	
e) The minimum path length of a <quantified path primary> is the product of the minimum path length of the simply contained <path primary> and the value of the <lower bound>.	
f) The minimum path length of a <questioned path primary> is 0 (zero).	
g) The minimum path length of a <parenthesized path pattern expression> is the minimum path length of the simply contained <path pattern expression>.	
h) If <i>BNT1</i> and <i>BNT2</i> are two BNF non-terminals such that <i>BNT1</i> ::= <i>BNT2</i> and the minimum path length of <i>BNT2</i> is defined, then the minimum path length of <i>BNT1</i> is also defined and is the same as the minimum path length of <i>BNT2</i> .	
7) The <path primary> immediately contained in a <quantified path primary> or <questioned path primary> shall have minimum path length that is greater than 0 (zero).	
8) The <path primary> simply contained in a <quantified path primary> shall not contain a <quantified path primary> at the same depth of graph pattern matching.	
** Editor's Note (number 74) **	
It may be possible to permit nested quantifiers. WG3:W01-014 contained a discussion of a WGL to support aggregates at different depths of aggregation if there are nested quantifiers. See Language Opportunity [GQL-236].	
9) Let <i>PMA</i> be a <path multiset alternation>.	
a) A <path term> simply contained in <i>PMA</i> is a <multiset alternation operand of <i>PMA</i> >.	
b) Let <i>NOPMA</i> be the number of multiset alternation operands of <i>PMA</i> . Let <i>OPMA</i> <sub>1</sub> , ..., <i>OPMA</i> <sub><i>NOPMA</i></sub> be an enumeration of the operands of <i>PMA</i> .	
c) Any <subpath variable> declared by <subpath variable declaration> <simply contained in the multiset alternation operands of <i>PMA</i> > shall be mutually distinct.	
d) Let <i>SOPMA</i> <sub>1</sub> , ..., <i>SOPMA</i> <sub><i>NOPMA</i></sub> be implementation-dependent (UV000) <identifier>'s that are mutually distinct and distinct from every <element variable>, <subpath variable> and <path variable> contained in <i>GP</i> .	
e) For every <i>i</i> , 1 (one) ≤ <i>i</i> ≤ <i>NOPMA</i> . Case: i) If <i>OPMA</i> <sub><i>i</i></sub> is a <parenthesized path pattern expression> that simply contains a <subpath variable declaration>, then let <i>OPMA</i> <sub><i>i</i></sub> be <i>OPMA</i> <sub><i>i</i></sub> . ii) Otherwise, let <i>OPMA</i> <sub><i>i</i></sub> be the <parenthesized path pattern expression>. ( <i>SOPMA</i> <sub><i>i</i></sub> = <i>OPMA</i> <sub><i>i</i></sub> .)	
f) <i>PMA</i> is equivalent to: <i>OPMA</i> <sub>1</sub>   ...   <i>OPMA</i> <sub><i>NOPMA</i></sub>	
10) A <path term> <PPUOP> simply contained in a <path pattern union> <PSD> is a <path pattern union operand of <i>PSD</i> >.	

## IWD 39075:202y(E) 16.10 <path pattern expression>

h) Without Feature G038, "Parenthesized path pattern expression", conforming GQL language shall not contain a <path pattern expression>.	
NOTE 142 — That is, even if <i>PPE</i> exposes <i>EV</i> as an effectively unbounded group variable, <i>PP</i> still exposes <i>EV</i> as effectively bounded, because in this case <i>PP</i> is required to be a selective <path pattern>.	
i) If <i>BNT1</i> and <i>BNT2</i> are two BNF non-terminals such that <i>BNT1</i> ::= <i>BNT2</i> and <i>BNT2</i> exposes <i>EV</i> , then <i>BNT1</i> exposes <i>EV</i> to the same degree of exposure as <i>BNT2</i> .	
** Editor's Note (number 76) **	
WG3:W04-009R1 defined "effectively bounded group variable" but did not use the definition. The definition will be used when we define predicates on aggregates, at which time we will want a Syntax Rule stating that if a group variable <i>GV</i> is referenced in a <i>WHERE</i> clause, then it shall be effectively bounded and the reference shall be contained in an aggregated argument of an <aggregate function>. See Possible Problem [GQL-050].	
23) If <i>BNT</i> is a BNF non-terminal that exposes a graph pattern variable <i>GPV</i> with a degree of exposure <i>DEGREE</i> , then <i>BNT</i> is also said to expose the name of <i>GPV</i> with degree of exposure <i>DEGREE</i> .	
24) A <parenthesized path pattern where clause> <PPWC> simply contained in a <parenthesized path pattern expression> <PPPE> shall not reference a path variable.	
** Editor's Note (number 77) **	
WG3:W04-009R1 recognized that a graph query may have a sequence of <i>MATCH</i> clauses, with the bindings of one <i>MATCH</i> clause <i>MC2</i> visible in all subsequent <i>MATCH</i> clauses in the same invocation of graph tables, and that it should be permissible to reference such variables in any <parenthesized path pattern where clause> simply contained in a subsequent <i>MATCH</i> clause <i>MC2</i> . The relevance of this LD to GQL needs to be investigated. See Language Opportunity [GQL-051].	

## General Rules

Note.  
NOTE 143 — The evaluation of a <path pattern expression> is performed by the General Rules of Subclause 21.3, "Evaluation of a <path pattern expression>".

## Conformance Rules

- Without Feature G030, "Path Multiset Alternation", conforming GQL language shall not contain a <path multiset alternation>.
- Without Feature G031, "Path Multiset Alternation: variable length path operands", in conforming GQL language, an operand of a <path multiset alternation> shall be a fixed length path pattern.
- Without Feature G032, "Path Pattern Union", conforming GQL language shall not contain a <path pattern union>.
- Without Feature G033, "Path Pattern Union: variable length path operands", in conforming GQL language, an operand of a <path pattern union> shall be a fixed length path pattern.
- Without Feature G035, "Quantified Paths", conforming GQL language shall not contain a <quantified path primary> that does not immediately contain a <path primary> that is an <edge pattern>.
- Without Feature G036, "Quantified Edges", conforming GQL language shall not contain a <quantified path primary> that immediately contains a <path primary> that is an <edge pattern>.
- Without Feature G037, "Questioned Paths", conforming GQL language shall not contain a <questioned path primary>.

## IWD 39075:202y(E) 16.10 <path pattern expression>

** Editor's Note (number 75) **	
Path pattern union is not defined using left recursion. WG3:SM-052 believed that it should be possible to support left recursion but declined to do so for expediency. It is a Language Opportunity to support left recursion. See Language Opportunity [GQL-025].	
<i>PPUOP</i> shall not contain a reference to an element variable that is not declared in <i>PPUOP</i> or outside of <i>PSD</i> .	
11) An <element pattern> <EP> that contains an <element pattern where clause> <EPWC> is transformed as follows: a) Let <i>EPF</i> be the <element pattern filler> simply contained in <i>EP</i> . b) Let <i>PREFIX</i> be the <delimiter token> contained in <i>EP</i> before <i>EPF</i> and let <i>SUFFIX</i> be the <delimiter token> contained in <i>EP</i> after <i>EPF</i> . c) Let <i>EV</i> be the <element variable> simply contained in <i>EPF</i> . Let <i>ILE</i> be the <is label expression> contained in <i>EPF</i> , if any; otherwise, let <i>ILE</i> be the zero-length string. d) <i>EP</i> is replaced by ( <i>PREFIX EV ILE SUFFIX EPWC</i> )	
12) An <element pattern> that does not contain an <element variable declaration>, an <is label expression>, or an <element pattern predicate> is said to be empty.	
13) Each <path pattern expression> is transformed in the following steps: a) If the <path primary> immediately contained in a <quantified path primary> or <questioned path primary> is an <edge pattern> <EP>, then <i>EP</i> is replaced by ( <i>EP</i> ) NOTE 135 — For example, -> becomes: (-> ) which in later transformations becomes: ( ( ) -> ( ) ( ) )	
b) If two successive <element pattern>'s contained in a <path concatenation> at the same depth of graph pattern matching are <edge pattern>'s, then an implicit empty <node pattern> is inserted between them.	
c) If an edge pattern <EP> contained in a <path term> <PST> at the same depth of graph pattern matching is not preceded by a <node pattern> contained in <PST> at the same depth of graph pattern matching, then an implicit empty <node pattern> <VP> is inserted in <PST> immediately prior to <EP>.	
d) If an edge pattern <EP> contained in a <path term> <PST> at the same depth of graph pattern matching is not followed by a <node pattern> contained in <PST> at the same depth of graph pattern matching, then an implicit empty <node pattern> <VP> is inserted in <PST> immediately after <EP>.	
NOTE 136 — As a result of the preceding transformations, a fixed length path pattern has an odd number of <element pattern>'s, beginning and ending with <node pattern>'s, and alternating between <node pattern>'s and <edge pattern>'s.	

	IWD 39075:202y(E)
	16.10 <path pattern expression>
h) If <i>BNT1</i> and <i>BNT2</i> are two BNF non-terminals such that <i>BNT1</i> ::= <i>BNT2</i> and the minimum node count of <i>BNT2</i> is defined, then the minimum node count of <i>BNT1</i> is also defined and is the same as the minimum node count of <i>BNT2</i> .	
15) The <path pattern expression> simply contained in a <path pattern> shall have a minimum node count that is greater than 0 (zero). NOTE 137 — The minimum node count is computed after the syntactic transform that adds implicit node patterns. Thus a single <edge pattern> is a permitted <path pattern> because it implies two <node pattern>'s. + WG3:W24-022 +	
16) An <element variable> <EV> contained in an <element variable declaration> <EPVD> is said to be declared by <EPVD>, and by the <element pattern> <EP> that simply contains <EPVD>. The <element variable> is the name of an element variable, which is also declared by <EPVD> and <EP>. If <EPVD> simply contains <TEMP>, then <EV> is a temporary element variable. <EV> is a <primary variable>. NOTE 138 — Element bindings to temporary element variables are removed prior to set-theoretic duplication of matches. See GR 110 of Subclause 16.8, "graph patterns" and GR 141 of Subclause 21.2, "Machinery for graph pattern matching".	
17) Prior to the application of syntactic transformations, conforming GQL language shall not contain an <element variable declaration> that immediately contains <TEMP>.	
18) An element variable that is declared by a <node pattern> is a <node variable>. An element variable that is declared by an <edge pattern> is an <edge variable>. + WG3:W24-022 +	
19) The scope of an <element variable> <EV> that is declared by an <element pattern> <EP> is defined as follows. If <EV> is a temporary element variable, then the scope of <EV> is the innermost <path term> containing <EP>; otherwise, the scope of <EV> is the innermost <graph pattern binding table> containing <EP>.	
20) A <subpath variable> <SV> contained in a <subpath variable declaration> <SPVD> is said to be declared by <SPVD>, and by the <parenthesized path pattern expression> <PPPE> that simply contains <SPVD>. <SV> is the name of a subpath variable, which is also declared by <SPVD> and <PPPE>.	
21) If <EP> is an <element pattern> that contains an <element pattern where clause> <EPWC>, then <EP> shall simply contain an <element variable declaration> <EPVD>.	
22) If <EV> is an element variable or subpath variable, and <BNT> is an instance of a BNF non-terminal, then the terminology "BNT exposes <EV>" is defined as follows. The full terminology is one of the following: "BNT exposes <EV> as an unconditional singleton variable", "BNT exposes <EV> as a conditional singleton variable", "BNT exposes <EV> as an effectively bounded group variable", or "BNT exposes <EV> as an effectively unbounded group variable". The terms "unconditional singleton variable", "conditional singleton variable", "effectively bounded group variable", and "effectively unbounded group variable" are called the <degree of exposure>. a) An <element pattern> <EP> that declares an element variable <EV> exposes <EV> as an unconditional singleton. b) A <parenthesized path pattern expression> <PPPE> that simply contains a <subpath variable declaration> that declares <EV> exposes <EV> as an unconditional singleton variable. <PPPE> shall not contain another <parenthesized path pattern expression> that declares <EV>. c) If a <path concatenation> <PPC> declares <EV> then let <PP> be the <path term> and let <PF> be the <path factor> simply contained in <PPC>. Case: i) If <EV> is exposed as an unconditional singleton by both <PP> and <PF>, then <EV> is exposed as an unconditional singleton by <PPC>. <EV> shall not be a subpath variable.	

h) If <i>BNT1</i> and <i>BNT2</i> are two BNF non-terminals such that <i>BNT1</i> ::= <i>BNT2</i> and the minimum node count of <i>BNT2</i> is defined, then the minimum node count of <i>BNT1</i> is also defined and is the same as the minimum node count of <i>BNT2</i> .	
15) The <path pattern expression> simply contained in a <path pattern> shall have a minimum node count that is greater than 0 (zero). NOTE 137 — The minimum node count is computed after the syntactic transform that adds implicit node patterns. Thus a single <edge pattern> is a permitted <path pattern> because it implies two <node pattern>'s. + WG3:W24-022 +	
16) An <element variable> <EV> contained in an <element variable declaration> <EPVD> is said to be declared by <EPVD>, and by the <element pattern> <EP> that simply contains <EPVD>. The <element variable> is the name of an element variable, which is also declared by <EPVD> and <EP>. If <EPVD> simply contains <TEMP>, then <EV> is a temporary element variable. <EV> is a <primary variable>. NOTE 138 — Element bindings to temporary element variables are removed prior to set-theoretic duplication of matches. See GR 110 of Subclause 16.8, "graph patterns" and GR 141 of Subclause 21.2, "Machinery for graph pattern matching".	
17) Prior to the application of syntactic transformations, conforming GQL language shall not contain an <element variable declaration> that immediately contains <TEMP>.	
18) An element variable that is declared by a <node pattern> is a <node variable>. An element variable that is declared by an <edge pattern> is an <edge variable>. + WG3:W24-022 +	
19) The scope of an <element variable> <EV> that is declared by an <element pattern> <EP> is defined as follows. If <EV> is a temporary element variable, then the scope of <EV> is the innermost <path term> containing <EP>; otherwise, the scope of <EV> is the innermost <graph pattern binding table> containing <EP>.	
20) A <subpath variable> <SV> contained in a <subpath variable declaration> <SPVD> is said to be declared by <SPVD>, and by the <parenthesized path pattern expression> <PPPE> that simply contains <SPVD>. <SV> is the name of a subpath variable, which is also declared by <SPVD> and <PPPE>.	
21) If <EP> is an <element pattern> that contains an <element pattern where clause> <EPWC>, then <EP> shall simply contain an <element variable declaration> <EPVD>.	
22) If <EV> is an element variable or subpath variable, and <BNT> is an instance of a BNF non-terminal, then the terminology "BNT exposes <EV>" is defined as follows. The full terminology is one of the following: "BNT exposes <EV> as an unconditional singleton variable", "BNT exposes <EV> as a conditional singleton variable", "BNT exposes <EV> as an effectively bounded group variable", or "BNT exposes <EV> as an effectively unbounded group variable". The terms "unconditional singleton variable", "conditional singleton variable", "effectively bounded group variable", and "effectively unbounded group variable" are called the <degree of exposure>. a) An <element pattern> <EP> that declares an element variable <EV> exposes <EV> as an unconditional singleton. b) A <parenthesized path pattern expression> <PPPE> that simply contains a <subpath variable declaration> that declares <EV> exposes <EV> as an unconditional singleton variable. <PPPE> shall not contain another <parenthesized path pattern expression> that declares <EV>. c) If a <path concatenation> <PPC> declares <EV> then let <PP> be the <path term> and let <PF> be the <path factor> simply contained in <PPC>. Case: i) If <EV> is exposed as an unconditional singleton by both <PP> and <PF>, then <EV> is exposed as an unconditional singleton by <PPC>. <EV> shall not be a subpath variable.	

	IWD 39075:202y(E)
	16.10 <path pattern expression>
NOTE 139 — This case expresses an implicit join on <i>EV</i> within <i>PPC</i> . Implicit joins between conditional singleton variables, group variables, or subpath variables are forbidden.	
ii) Otherwise, <EV> shall only be exposed by one of <PT> or <PF>. In this case <EV> is exposed by <PPC> in the same degree that it is exposed by <PT> or <PF>.	
d) If a <path pattern union> or <path multiset alternation> <PA> declares <EV>, then: Case: i) If every operand of <PA> exposes <EV> as an unconditional singleton variable, then <PA> exposes <EV> as an unconditional singleton variable. ii) If at least one operand of <PA> exposes <EV> as an effectively unbounded group variable, then <PA> exposes <EV> as an effectively unbounded group variable. iii) If at least one operand of <PA> exposes <EV> as an effectively bounded group variable, then <PA> exposes <EV> as an effectively bounded group variable. iv) Otherwise, <PA> exposes <EV> as a conditional singleton variable.	
e) If a <quantified path primary> <QPP> declares <EV>, then let <PP> be the <path primary> simply contained in <QPP>. Case: i) If <QPP> contains a <graph pattern quantifier> that is a <fixed quantifier> or a <general quantifier> that contains an <upper bound> and <PP> does not expose <EV> as an effectively unbounded group variable, then <QPP> exposes <EV> as an effectively bounded group variable. ii) If <QPP> is contained at the same depth of graph pattern matching in a restrictive <parenthesized path pattern expression>, then <QPP> exposes <EV> as an effectively bounded group variable. NOTE 140 — The preceding definition is applied after the syntactic transformation to insure that every <path mode prefix> is at the head of a <parenthesized path pattern expression>.	
iii) Otherwise, <QPP> exposes <EV> as an effectively unbounded group variable.	
f) If a <questioned path primary> <QPPQ> declares <EV>, then let <PP> be the <path primary> simply contained in <QPPQ>. Case: i) If <PPQ> exposes <EV> as a group variable, then <QPPQ> exposes <EV> as a group variable with the same degree of exposure. ii) Otherwise, <QPPQ> exposes <EV> as a conditional singleton variable.	
g) A <parenthesized path pattern expression> <PPPE> exposes the same variables as the simply contained <path pattern expression>, in the same degree of exposure. NOTE 141 — A restrictive <path mode> declared by a <parenthesized path pattern expression> makes variables effectively bounded, but it does so even for proper subexpressions within the scope of the <path mode> and has already been handled by the rules for <quantified path primary>.	
h) If a <path pattern> <PP> declares <EV>, then let <PPE> be the simply contained <path pattern expression>. Case: i) If <PPE> exposes <EV> as an unconditional singleton, a conditional singleton, or an effectively bounded group variable, then <PP> exposes <EV> with the same degree of exposure.	

Let's try to formalize.

Attempt 1: Pattern matching (PODS'23)

# Pattern calculus in a nutshell

Node pattern

$$\nu := (x : \ell)$$

match an  $\ell$ -labeled node, assign to a variable

Both  $x$  and  $\ell$  are optional

Edge pattern

$$\alpha := \xrightarrow{x:\ell} \mid \xleftarrow{x:\ell} \mid \text{---}^{x:\ell}$$

$\ell$ -labeled edge directed left/right/any-directed, assign to a variable

Patterns

$$\pi := \nu \mid \alpha \mid \pi\pi \mid \pi + \pi \mid \pi^{n..m} \mid \pi\langle\theta\rangle \quad 0 \leq n \leq m \leq \infty$$

node

edge

concatenation

union

repetition  
n-to-m times

selection with condition

Conditions

$$\theta := x.a = c \mid x.a = y.b \mid \theta \vee \theta \mid \theta \wedge \theta \mid \neg\theta$$

key-value comparisons

Boolean combinations

Queries

$$Q := \sigma \pi \mid p = \sigma \pi \mid Q, Q$$

ensure finitely  
many paths

name  
matched  
path

join

# It needs a type system

$(x) \vdash x : \text{Node}$	$(x : \ell) \vdash x : \text{Node}$	$\overset{x}{\longleftrightarrow} \vdash x : \text{Edge}$	$\overset{x:\ell}{\longleftrightarrow} \vdash x : \text{Edge}$	$x \notin \text{var}(\pi)$ $x = \rho \pi \vdash x : \text{Path}$
	$\frac{\pi \vdash z : \tau}{\pi^{n..m} \vdash z : \text{Group}(\tau)}$	$\frac{\pi \vdash z : \tau}{\rho\pi \vdash z : \tau}$	$\frac{\pi \vdash z : \tau \quad z \neq x}{x = \rho\pi \vdash z : \tau}$	
	$\frac{\pi \vdash x : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{\pi \vdash x.a = c : \text{Bool}}$	$\frac{\pi \vdash x : \tau \quad \pi \vdash y : \tau' \quad \tau, \tau' \in \{\text{Node}, \text{Edge}\}}{\pi \vdash x.a = y.b : \text{Bool}}$		
$\frac{\pi \vdash \theta : \text{Bool} \quad \pi \vdash \theta' : \text{Bool}}{\pi \vdash \theta \wedge \theta' : \text{Bool}}$	$\frac{\pi \vdash \theta : \text{Bool} \quad \pi \vdash \theta' : \text{Bool}}{\pi \vdash \theta \vee \theta' : \text{Bool}}$	$\frac{\pi \vdash \theta : \text{Bool}}{\pi \vdash \neg\theta : \text{Bool}}$	$\frac{\pi \vdash \theta : \text{Bool} \quad \pi \vdash z : \tau}{\pi_{\langle\theta\rangle} \vdash z : \tau}$	
$\frac{\pi_1 \vdash z : \tau \quad \pi_2 \vdash z : \tau}{\pi_1 + \pi_2 \vdash z : \tau}$	$\frac{\pi_1 \vdash z : \tau \quad \pi_2 \vdash z : \text{Maybe}(\tau)}{\pi_1 + \pi_2 \vdash z : \text{Maybe}(\tau)}$		$\frac{\pi_1 \vdash z : \text{Maybe}(\tau) \quad \pi_2 \vdash z : \tau}{\pi_1 + \pi_2 \vdash z : \text{Maybe}(\tau)}$	
	$\frac{\pi_1 \vdash z : \tau \quad z \notin \text{var}(\pi_2)}{\pi_1 + \pi_2 \vdash z : \tau?}$	$\frac{\pi_2 \vdash z : \tau \quad z \notin \text{var}(\pi_1)}{\pi_1 + \pi_2 \vdash z : \tau?}$		
$\frac{\pi_1 \vdash z : \tau \quad \pi_2 \vdash z : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{\pi_1 \pi_2 \vdash z : \tau}$	$\frac{\pi_1 \vdash z : \tau \quad z \notin \text{var}(\pi_2)}{\pi_1 \pi_2 \vdash z : \tau}$	$\frac{\pi_2 \vdash z : \tau \quad z \notin \text{var}(\pi_1)}{\pi_1 \pi_2 \vdash z : \tau}$		
$\frac{Q_1 \vdash z : \tau \quad Q_2 \vdash z : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{Q_1, Q_2 \vdash z : \tau}$	$\frac{Q_1 \vdash z : \tau \quad z \notin \text{var}(Q_2)}{Q_1, Q_2 \vdash z : \tau}$	$\frac{Q_2 \vdash z : \tau \quad z \notin \text{var}(Q_1)}{Q_1, Q_2 \vdash z : \tau}$		

# Problems

- Could prove a few things but not much
- A bit **too heavy for definition 1**
- Only covers pattern matching
- Next step: add relational operators


# Complete Formalization

ICDT'23 :



## A Researcher's Digest of GQL

**Nadime Francis** ✉

Laboratoire d'Informatique Gaspard Monge,  
Université Gustave Eiffel, CNRS, France

**Paolo Guagliardo** ✉ 

School of Informatics,  
University of Edinburgh, UK

**Victor Marsault** ✉  


Laboratoire d'Informatique Gaspard Monge,  
Université Gustave Eiffel, CNRS, France

**Filip Murlak** ✉ 

University of Warsaw, Poland

**Alexandra Rogova** ✉

IRIF, Université Paris Cité, CNRS, Paris, France  
Data Intelligence Institute of Paris, Inria

**Amélie Gheerbrant** ✉ 


IRIF, Université Paris Cité, CNRS,  
Paris, France

**Leonid Libkin** ✉ 


University of Edinburgh, UK  
RelationalAI, France  
ENS, PSL University, France

**Wim Martens** ✉ 

Universität Bayreuth, Germany

**Liat Peterfreund** ✉ 

Laboratoire d'Informatique Gaspard Monge,  
Université Gustave Eiffel, CNRS, France

**Domagoj Vrgoč** ✉ 

University of Zagreb, Croatia  
Pontificia Universidad Católica de Chile,  
Santiago, Chile

### Abstract

GQL (Graph Query Language) is being developed as a new ISO standard for graph query languages to play the same role for graph databases as SQL plays for relational. In parallel, an extension of SQL for querying property graphs, SQL/PGQ, is added to the SQL standard; it shares the graph pattern matching functionality with GQL. Both standards (not yet published) are hard-to-understand specifications of hundreds of pages. The goal of this paper is to present a digest of the language that is easy for the research community to understand, and thus to initiate research on these future standards for querying graphs. The paper concentrates on pattern matching features shared by GQL and SQL/PGQ, as well as querying facilities of GQL.

2012 ACM Subject Classification Theory of computation → Database theory; Theory of computation → Database query languages (principles); Information systems → Graph-based database models; Information systems → Structured Query Language

Keywords and phrases GQL, Property Graph, Query Language, Graph Database, Pattern matching, Multi-Graph

**PATH PATTERN** For  $x \in \text{Vars}$ ,  $\ell \in \mathcal{L}$ ,  $0 \leq n \leq m \in \mathbb{N}$ :

(descriptor)	$\delta := x : \ell$ <b>WHERE</b> $\theta$	$x, : \ell$ , and <b>WHERE</b> $\theta$ are optional
(path pattern)	$\pi := (\delta)$	(node pattern)
	$-\lceil \delta \rceil \rightarrow$   $\leftarrow \lceil \delta \rceil -$   $-\lceil \delta \rceil -$	(edge pattern)
	$\pi \pi$	(concatenation)
	$\pi   \pi$	(union)
	$\pi$ <b>WHERE</b> $\theta$	(conditioning)
	$\pi \{n, m\}$	(bounded repetition)
	$\pi \{n, \}$	(unbounded repetition)

**EXPRESSION and CONDITION** For  $x \in \text{Vars}$ ,  $\ell \in \mathcal{L}$ ,  $a \in \mathcal{K}$ ,  $c \in \text{Const}$ :

(expression)	$\chi := x \mid x.a \mid c$
(condition)	$\theta := \chi = \chi \mid \chi < \chi \mid \chi$ <b>IS NULL</b>   $x : \ell \mid$ <b>EXISTS</b> $\{Q\}$   $\theta$ <b>OR</b> $\theta \mid \theta$ <b>AND</b> $\theta \mid$ <b>NOT</b> $\theta$

**GRAPH PATTERN** For  $x \in \text{Vars}$ :

(path mode)	$\mu := (\text{ALL} \mid \text{ANY})$ [ <b>SHORTEST</b> ] [ <b>TRAIL</b>   <b>ACYCLIC</b> ]
(graph pattern)	$\Pi := \mu [x =] \pi \mid \Pi, \Pi$

**CLAUSE and QUERY** For  $k \geq 0$ ,  $\ell \geq 1$ , and  $x, y, x_1, \dots, x_k \in \text{Vars}$ , and  $G \in \mathcal{G}$ :

(clause)	$C :=$ <b>MATCH</b> $\Pi$   <b>LET</b> $x = \chi$   <b>FOR</b> $x$ <b>IN</b> $y$   <b>FILTER</b> $\theta$
(linear query)	$L :=$ <b>USE</b> $G$ $L$   $C$ $L$   <b>RETURN</b> $\chi_1$ <b>AS</b> $x_1, \dots, \chi_k$ <b>AS</b> $x_k$
(query)	$Q :=$ $L$   <b>USE</b> $G$ $\{Q_1$ <b>THEN</b> $Q_2$ $\dots$ <b>THEN</b> $Q_\ell\}$   $Q$ <b>INTERSECT</b> $Q \mid Q$ <b>UNION</b> $Q \mid Q$ <b>EXCEPT</b> $Q$

S  
Y  
N  
T  
A  
X

# Semantics

## 1:12 A Researcher's Digest of GQL

$$\begin{aligned} \llbracket [- \square \rightarrow] \rrbracket_G &= \{ (\text{path}(\text{src}(e), e, \text{tgt}(e)), ()) \mid e \in E_d^G \} \\ \llbracket [- [x] \rightarrow] \rrbracket_G &= \{ (\text{path}(\text{src}(e), e, \text{tgt}(e)), (x \mapsto e)) \mid e \in E_d^G \} \\ \llbracket [- [ : \ell ] \rightarrow] \rrbracket_G &= \left\{ (\text{path}(\text{src}(e), e, \text{tgt}(e)), ()) \mid e \in E_d^G, \ell \in \text{lab}^G(e) \right\} \end{aligned}$$

Other cases of the forward edge patterns are treated by moving the label and conditions outside of the edge pattern, just as for node patterns. Backward edge patterns and undirected edge patterns are treated similarly, with the base cases given below.

$$\begin{aligned} \llbracket \leftarrow [- \square -] \rrbracket_G &= \{ (\text{path}(\text{tgt}(e), e, \text{src}(e)), ()) \mid e \in E_d^G \} \\ \llbracket [- \square -] \rrbracket_G &= \left\{ (\text{path}(u_1, e, u_2), ()), (\text{path}(u_2, e, u_1), ()) \mid \begin{array}{l} e \in E_u^G \\ \{u_1, u_2\} = \text{endpoints}^G(e) \end{array} \right\} \end{aligned}$$

### Semantics of Concatenation, Union, and Conditioning

$$\llbracket \pi_1 \pi_2 \rrbracket_G = \left\{ (p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \mid \begin{array}{l} (p_i, \mu_i) \in \llbracket \pi_i \rrbracket_G \text{ for } i = 1, 2 \\ p_1 \text{ and } p_2 \text{ concatenate} \\ \mu_1 \sim \mu_2 \end{array} \right\}$$

Note that since  $\pi_1 \pi_2$  is assumed to be well-formed, all variables shared by  $\pi_1$  and  $\pi_2$  are singleton variables (Condition 2 in Section 3). In other words, implicit joins over group and optional variables are disallowed; the same remark will also apply for the semantics of joins.

► **Remark 9.** Consider the pattern

$$(x) \ (-[:\text{Transfer}] \rightarrow () \ -[:\text{Transfer}] \rightarrow (x)) \{1, \}$$

This pattern is disallowed in GQL because the leftmost  $x$  is a singleton variable, whereas the rightmost  $x$  is a group variable. In GQL philosophy, the leftmost  $x$  will be bound to a node and the rightmost  $x$  will be bound to a list of nodes, which is a type mismatch.

$$\llbracket \pi_1 \mid \pi_2 \rrbracket_G = \{ (p, \mu \cup \mu') \mid (p, \mu) \in \llbracket \pi_1 \rrbracket_G \cup \llbracket \pi_2 \rrbracket_G \}$$

where  $\mu'$  maps every variable in  $\text{var}(\pi_1 \mid \pi_2) \setminus \text{Dom}(\mu)$  to null. (Recall that  $\text{var}$  maps a pattern to the set of variables appearing in it.)

$$\llbracket \pi \text{ WHERE } \theta \rrbracket_G = \{ (p, \mu) \in \llbracket \pi \rrbracket_G \mid \llbracket \theta \rrbracket_G^\mu = \text{true} \}$$

### Semantics of Repetition

$$\begin{aligned} \llbracket \pi \{n, m\} \rrbracket_G &= \bigcup_{i=n}^m \llbracket \pi \rrbracket_G^i \\ \llbracket \pi \{n, \} \rrbracket_G &= \bigcup_{i=n}^{\infty} \llbracket \pi \rrbracket_G^i \end{aligned}$$

Above, for a pattern  $\pi$  and a natural number  $i \geq 0$ , we use  $\llbracket \pi \rrbracket_G^i$  to denote the  $i$ -th power of  $\llbracket \pi \rrbracket_G$ , which we define as

$$\llbracket \pi \rrbracket_G^0 = \{ (\text{path}(u), \mu) \mid u \text{ is a node in } G \}$$

where  $\mu$  binds each variable in  $\text{Dom}(\text{sch}(\pi))$  to  $\text{list}()$ , that is, the empty-list value; and

$$\forall i > 0 \quad \llbracket \pi \rrbracket_G^i = \left\{ (p_1 \cdot \dots \cdot p_i, \mu') \mid \begin{array}{l} (p_i, \mu_i) \in \llbracket \pi \rrbracket_G \\ p_1, \dots, p_i \text{ concatenate} \end{array} \right\}$$

where  $\mu'$  binds each variable in  $\text{Dom}(\text{sch}(\pi))$  to  $\text{list}(\mu_1(x), \dots, \mu_i(x))$ . Recall that  $\text{sch}$  is defined in Section 3.

## N. Francis et al.

## 1:13

► **Remark 10.** Since  $\pi \{n, \}$  is assumed to be well-formed, it holds  $\|\pi\|_{\min} \geq 1$ . A simple induction then yields that each  $p_i$  in the definition above has positive length. A second induction then yields that, given a path  $p$ , there are finitely many assignments  $\mu$  such that  $(p, \mu) \in \llbracket \pi \{n, m\} \rrbracket_G$ . This fact is crucial to have a finite output in the end.

For instance, consider a graph with a single node  $u$  and no edges, and the pattern  $(\mathbf{a}) \{0, \}$  which is not well-formed (the minimal path length of  $()$  is 0). For every  $i$ , the set  $\llbracket (\mathbf{a}) \rrbracket_G^i$  contains  $(\text{path}(u), \mu_i)$  where  $\mu_i = (a \mapsto \underbrace{\text{list}(u, \dots, u)}_{i \text{ times}})$ ; hence the union in the definition of

$\llbracket \pi \{n, \} \rrbracket_G$  above would not only yield an infinite number of elements, but all of them would be associated to the same path. As a result a graph pattern such as **ALL SHORTEST**  $(\mathbf{a}) \{0, \}$  would have infinitely many results.

### 4.3 Semantics of Graph Patterns

We now define the semantics of graph patterns. We first fully define atomic graph patterns and then define their joins.

$$\llbracket x = \pi \rrbracket_G = \{ (p, \mu \cup \{x \mapsto p\}) \mid (p, \mu) \in \llbracket \pi \rrbracket_G \}$$

In the following we denote by  $\tilde{\pi}$  a graph pattern that never uses the “,” operator, hence it is of the form  $\mu x = \pi$ , where  $\mu$  is a path mode,  $x$  is a variable,  $\pi$  is a path pattern, and “ $x =$ ” is optional.

$$\llbracket \text{TRAIL } \pi \rrbracket_G = \{ (p, \mu) \in \llbracket \pi \rrbracket_G \mid \text{no edge occurs more than once in } p \}$$

$$\llbracket \text{ACYCLIC } \pi \rrbracket_G = \{ (p, \mu) \in \llbracket \pi \rrbracket_G \mid \text{no node occurs more than once in } p \}$$

$$\llbracket \text{SHORTEST } \tilde{\pi} \rrbracket_G = \left\{ (p, \mu) \in \llbracket \tilde{\pi} \rrbracket_G \mid \text{len}(p) = \min \left\{ \text{len}(p') \mid \begin{array}{l} (p', \mu') \in \llbracket \tilde{\pi} \rrbracket_G \\ \text{src}(p') = \text{src}(p) \\ \text{tgt}(p') = \text{tgt}(p) \end{array} \right\} \right\}$$

$$\llbracket \text{ALL } \tilde{\pi} \rrbracket_G = \llbracket \tilde{\pi} \rrbracket_G$$

$$\llbracket \text{ANY } \tilde{\pi} \rrbracket_G = \bigcup_{(s,t) \in X} \{ \text{any}(\{ (p, \mu) \mid (p, \mu) \in \llbracket \tilde{\pi} \rrbracket_G, \text{endpoints}(p) = (s, t) \}) \}$$

where  $X = \{ (\text{src}(p), \text{tgt}(p)) \mid (p, \mu) \in \llbracket \tilde{\pi} \rrbracket_G \}$  and  $\text{any}$  is a procedure that arbitrarily returns one element from a set;  $\text{any}$  need not be deterministic.

$$\llbracket \Pi_1, \Pi_2 \rrbracket_G = \{ (\bar{p}_1 \times \bar{p}_2, \mu_1 \bowtie \mu_2) \mid (\bar{p}_i, \mu_i) \in \llbracket \Pi_i \rrbracket_G \text{ for } i = 1, 2 \text{ and } \mu_1 \sim \mu_2 \}$$

Here,  $\bar{p}_1 = (p_1^1, p_1^2, \dots, p_1^k)$  and  $\bar{p}_2 = (p_2^1, p_2^2, \dots, p_2^l)$  are tuples of paths, and  $\bar{p}_1 \times \bar{p}_2$  stands for  $(p_1^1, p_1^2, \dots, p_1^k, p_2^1, p_2^2, \dots, p_2^l)$ . Just as it is the case of concatenation, since  $\Pi_1, \Pi_2$  is well-formed, implicit joins can occur over singleton variables only.

### 4.4 Semantics of Conditions and Expressions

The semantics  $\llbracket \chi \rrbracket_G^\mu$  of an expression  $\chi$  is an element in  $\mathbb{V}$  that is computed with respect to a binding  $\mu$  and a graph  $G$ . Intuitively, variables in  $\chi$  are evaluated with  $\mu$  and we use  $G$  to access the properties of an element. It is formally defined as follows.

$$\begin{aligned} \llbracket c \rrbracket_G^\mu &= c && \text{for } c \in \text{Const} \\ \llbracket x \rrbracket_G^\mu &= \mu(x) && \text{for } x \in \text{Dom}(\mu) \\ \llbracket x.a \rrbracket_G^\mu &= \begin{cases} \text{prop}^G(\mu(x), a) & \text{if } (\mu(x), a) \in \text{Dom}(\text{prop}^G) \\ \text{null} & \text{else if } \mu(x) \in (\mathcal{N} \cup \mathcal{E}_d \cup \mathcal{E}_u) \end{cases} && \text{for } x \in \text{Dom}(\mu), a \in \mathcal{K} \end{aligned}$$



# Semantics

## 1:14 A Researcher's Digest of GQL

► **Remark 11.** Recall that different graphs may share nodes and edges. Hence the condition  $(\mu(x), a) \in \text{Dom}(\text{prop}^G)$ , above, does imply that  $\mu(x)$  is a node or an edge in  $G$ , but does **not** imply that it was matched in  $G$ .

The semantics  $\llbracket \theta \rrbracket_G^\mu$  of a condition  $\theta$  is an element in  $\{\text{true}, \text{false}, \text{null}\}$  that is evaluated with respect to a binding  $\mu$  and a graph  $G$ , and is defined as follows:

$$\begin{aligned} \llbracket \chi_1 = \chi_2 \rrbracket_G^\mu &= \begin{cases} \text{null} & \text{if } \llbracket \chi_1 \rrbracket_G^\mu = \text{null} \text{ or } \llbracket \chi_2 \rrbracket_G^\mu = \text{null} \\ \text{true} & \text{if } \llbracket \chi_1 \rrbracket_G^\mu = \llbracket \chi_2 \rrbracket_G^\mu \neq \text{null} \\ \text{false} & \text{otherwise} \end{cases} \\ \llbracket \chi_1 < \chi_2 \rrbracket_G^\mu &= \begin{cases} \text{null} & \text{if } \llbracket \chi_1 \rrbracket_G^\mu = \text{null} \text{ or } \llbracket \chi_2 \rrbracket_G^\mu = \text{null} \\ \text{true} & \text{else if } \llbracket \chi_1 \rrbracket_G^\mu < \llbracket \chi_2 \rrbracket_G^\mu \\ \text{false} & \text{otherwise} \end{cases} \\ \llbracket \chi \text{ IS NULL} \rrbracket_G^\mu &= \begin{cases} \text{true} & \text{if } \llbracket \chi \rrbracket_G^\mu = \text{null} \\ \text{false} & \text{otherwise} \end{cases} \\ \llbracket \chi : \ell \rrbracket_G^\mu &= \begin{cases} \text{true} & \text{if } \llbracket \chi \rrbracket_G^\mu \in N^G \cup E_u^G \cup E_d^G \text{ and } \ell \in \text{lab}^G(\llbracket \chi \rrbracket_G^\mu) \\ \text{false} & \text{else if } \llbracket \chi \rrbracket_G^\mu \in \mathcal{N} \cup \mathcal{E}_d \cup \mathcal{E}_u \end{cases} \\ \llbracket \theta_1 \text{ AND } \theta_2 \rrbracket_G^\mu &= \llbracket \theta_1 \rrbracket_G^\mu \wedge \llbracket \theta_2 \rrbracket_G^\mu \quad (*) \\ \llbracket \theta_1 \text{ OR } \theta_2 \rrbracket_G^\mu &= \llbracket \theta_1 \rrbracket_G^\mu \vee \llbracket \theta_2 \rrbracket_G^\mu \quad (*) \\ \llbracket \text{NOT } \theta \rrbracket_G^\mu &= \neg \llbracket \theta \rrbracket_G^\mu \quad (*) \end{aligned}$$

(\*) Operators  $\wedge$ ,  $\vee$ , and  $\neg$  are defined as in SQL three-valued logic, e.g.  $\text{null} \vee \text{true} = \text{true}$  while  $\text{null} \wedge \text{true} = \text{null}$ .

$$\llbracket \text{EXISTS } \{Q\} \rrbracket_G^\mu = \begin{cases} \text{true} & \text{if } \llbracket Q \rrbracket_G(\{\mu\}) \text{ is not empty} \\ \text{false} & \text{otherwise} \end{cases}$$

## 4.5 Semantics of Queries

Clauses and queries are interpreted as functions that operate on tables. These tables are our abstraction of GQL's working tables.

► **Definition 12.** A table  $T$  is a set of bindings that have the same domains, referred to as  $\text{Dom}(T)$ .

Note that tables do not have schemas: two different bindings in a table might associate a variable to values of incompatible types.

### Semantics of Clauses

The semantics  $\llbracket C \rrbracket_G$  of a clause  $C$  is a function that maps tables into tables, and is parametrized by a graph  $G$ . Patterns, conditions and expression in a clause are evaluated with respect to that  $G$ .

$$\llbracket \text{MATCH } \Pi \rrbracket_G(T) = \bigcup_{\mu \in T} \{\mu \bowtie \mu' \mid (p, \mu') \in \llbracket \Pi \rrbracket_G, \mu \sim \mu'\}$$

Note that if  $\Pi$  uses a variable that already occurs in  $\text{Dom}(T)$ , a join is performed. Unlike in the case of path patterns and graph patterns, this join can involve variables bound to lists or paths. While this is not problematic mathematically, it might be disallowed in future iterations of GQL.

## N. Francis et al.

1:15

If  $x \notin \text{Dom}(T)$ , then

$$\begin{aligned} \llbracket \text{LET } x = \chi \rrbracket_G(T) &= \bigcup_{\mu \in T} \{\mu \bowtie (x \mapsto \llbracket \chi \rrbracket_G^\mu)\} \\ \llbracket \text{FILTER } \theta \rrbracket_G(T) &= \bigcup_{\mu \in T} \{\mu \mid \llbracket \theta \rrbracket_G^\mu = \text{true}\}. \end{aligned}$$

If  $x \notin \text{Dom}(T)$  and, for every  $\mu \in T$ ,  $\mu(y)$  is a list or  $\text{null}$ ,<sup>3</sup> then

$$\llbracket \text{FOR } x \text{ IN } y \rrbracket_G(T) = \bigcup_{\mu \in T} \{\mu \bowtie (x \mapsto v) \mid v \in \mu(y)\}.$$

### Semantics of Linear Queries

$$\begin{aligned} \llbracket \text{USE } G' \text{ L} \rrbracket_G(T) &= \llbracket \text{L} \rrbracket_{G'}(T) \\ \llbracket C \text{ L} \rrbracket_G(T) &= \llbracket \text{L} \rrbracket_G(\llbracket C \rrbracket_G(T)) \\ \llbracket \text{RETURN } \chi_1 \text{ AS } x_1, \dots, \chi_\ell \text{ AS } x_\ell \rrbracket_G(T) &= \bigcup_{\mu \in T} \{(x_1 \mapsto \llbracket \chi_1 \rrbracket_G^\mu, \dots, x_\ell \mapsto \llbracket \chi_\ell \rrbracket_G^\mu)\} \end{aligned}$$

### Semantics of Queries

The *output of a query*  $Q$  is defined as

$$\text{Output}(Q) = \llbracket Q \rrbracket_G(\{()\}),$$

where  $\{()\}$  is the unit table that consists of the empty binding, and  $G$  is the default graph in  $D$ . We define the semantics of queries recursively as follows.

$$\llbracket \text{USE } G' \{Q_1 \text{ THEN } Q_2 \dots \text{ THEN } Q_k\} \rrbracket_G(T) = \llbracket Q_k \rrbracket_{G'} \circ \dots \circ \llbracket Q_1 \rrbracket_{G'}(T)$$

If  $\text{Dom}(\llbracket Q_1 \rrbracket_G(T)) = \text{Dom}(\llbracket Q_2 \rrbracket_G(T))$ , then we let

$$\begin{aligned} \llbracket Q_1 \text{ INTERSECT } Q_2 \rrbracket_G(T) &= \llbracket Q_1 \rrbracket_G(T) \cap \llbracket Q_2 \rrbracket_G(T) \\ \llbracket Q_1 \text{ UNION } Q_2 \rrbracket_G(T) &= \llbracket Q_1 \rrbracket_G(T) \cup \llbracket Q_2 \rrbracket_G(T) \\ \llbracket Q_1 \text{ EXCEPT } Q_2 \rrbracket_G(T) &= \llbracket Q_1 \rrbracket_G(T) \setminus \llbracket Q_2 \rrbracket_G(T) \end{aligned}$$

## 5 A Few Known Discrepancies with the GQL Standard

In pursuing the goal of introducing the key features of GQL to the research community, we inevitably had to make decisions that resulted in discrepancies between our presentation and the 500+ pages of the forthcoming Standard. In this section, we discuss a non-exhaustive list of differences between the actual GQL Standard and our digest. To start with, in all our formal development we assumed that queries are given by their syntax trees, which result from parsing them. Hence we completely omitted such parsing-related aspects as parentheses, operator precedence etc. Also we note that many GQL features, even those described here, are optional, and not every implementation is obliged to have them all.

<sup>3</sup> Note that  $\text{null}$  is treated just as  $\text{list}()$

# Pause and think

- Development of SQL:
  - basic theory: relational calculus, algebra
  - clean foundations: relations are sets of tuples
    - finite model theory: cannot define counting, recursion
  - add aggregates (right away, 1986), recursion (1999)
  - and lots of other baggage: bags, nulls, etc
- Development of GQL and PGQ
  - start with SQL basis: bags, nulls, aggregate
  - define a language as a compromise between 3 companies
  - Now need to think:
    - what are their relational algebra/calculus
    - what is not expressible? and why?
    - and how they address it?

What are relational algebra and  
calculus of GQL and PGQ?

What can we prove about them?

# Patterns

No non-1NF relations, No nulls, No bags, No typing rules, just free variables

$$\pi := (x) \mid \longrightarrow^x \mid \longleftarrow^x \mid \pi\pi \mid \pi + \pi \mid \pi\langle\theta\rangle \mid \pi^{n..m}$$

$$\theta := x.k = y.p \mid x.k < y.p \mid \ell(x) \mid \theta \vee \theta \mid \neg\theta$$

$$\text{FV}((x)) = \text{FV}(\longrightarrow^x) = \text{FV}(\longleftarrow^x) = \{x\}$$

$$\text{FV}(\pi_1\pi_2) = \text{FV}(\pi_1) \cup \text{FV}(\pi_2)$$

$$\text{FV}(\pi_1 + \pi_2) = \text{FV}(\pi_1) \quad \text{if} \quad \text{FV}(\pi_1) = \text{FV}(\pi_2)$$

$$\text{FV}(\pi\langle\theta\rangle) = \text{FV}(\pi) \quad \text{if} \quad \text{FV}(\theta) \subseteq \text{FV}(\pi)$$

$$\text{FV}(\pi^{n..m}) = \emptyset$$

Output: a subset  $\Omega$  of  $\text{FV}(\pi)$

Pattern with output:  $\pi_\Omega$

# Semantics: one simple definition, just what you expect

$$\begin{aligned}
 \llbracket (x) \rrbracket_G &:= \{(\text{path}(n), \{x \mapsto n\}) \mid n \in \mathbb{N}\} \\
 \llbracket \begin{array}{c} x \\ \rightarrow \\ \hline \end{array} \rrbracket_G &:= \{(\text{path}(n_1, e, n_2), \{x \mapsto e\}) \mid e \in E, \text{src}(e) = n_1, \text{tgt}(e) = n_2\} \\
 \llbracket \begin{array}{c} x \\ \leftarrow \\ \hline \end{array} \rrbracket_G &:= \{(\text{path}(n_2, e, n_1), \{x \mapsto e\}) \mid e \in E, \text{src}(e) = n_1, \text{tgt}(e) = n_2\} \\
 \llbracket \psi_1 + \psi_2 \rrbracket_G &:= \llbracket \psi_1 \rrbracket_G \cup \llbracket \psi_2 \rrbracket_G \\
 \llbracket \psi_1 \psi_2 \rrbracket_G &:= \{(p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \mid (p_1, \mu_1) \in \llbracket \psi_1 \rrbracket_G, (p_2, \mu_2) \in \llbracket \psi_2 \rrbracket_G, \mu_1 \sim \mu_2, p_1 \odot p_2\} \\
 \llbracket \psi \langle \theta \rangle \rrbracket_G &:= \{(p, \mu) \in \llbracket \pi \rrbracket_G \mid \mu \models \theta\} \\
 \llbracket \psi^{n..m} \rrbracket_G &:= \bigcup_{i=n}^m \llbracket \psi \rrbracket_G^i \text{ where} \\
 &\quad \llbracket \psi \rrbracket_G^0 := \{(\text{path}(n), \mu_\emptyset) \mid n \in \mathbb{N}\} \\
 &\quad \llbracket \psi \rrbracket_G^n := \{(p_1 \cdots p_n, \mu_\emptyset) \mid \exists \mu_1, \dots, \mu_n : (p_i, \mu_i) \in \llbracket \psi \rrbracket_G \text{ and } p_i \odot p_{i+1} \text{ for all } i < n\}, n > 0 \\
 \llbracket \psi_\Omega \rrbracket_G &:= \{\mu_\Omega \mid \exists p : (p, \mu) \in \llbracket \psi \rrbracket_G\}
 \end{aligned}$$

Every output is a first-normal form relation

PGQ model

RA(all  $\pi_{\Omega}$ )

Relational algebra over all pattern outputs

# What about GQL

- Relational operators applied in a **pipelined fashion**
- Usually called **linear composition**
- A sequence of clauses: each takes a relation and returns a relation
  - while looking at the database
- It is used heavily (Cypher, GQL, PRQL, to some extent Google's piped SQL) but we — the theory community — neglected it

# Pipelined relational algebra (PRA)

$C :=$  db relation  $| \pi_A | \sigma_\theta | CC | \{Q\}$  clauses  
 $Q := C | Q \cup Q | Q \cap Q | Q - Q$  queries

## Semantics

$$\llbracket S \rrbracket(R) = R \bowtie S$$

$$\llbracket \pi_A \rrbracket(R) = \pi_A(R)$$

$$\llbracket C_1 C_2 \rrbracket(R) = \llbracket C_2 \rrbracket(\llbracket C_1 \rrbracket(R))$$

$$\llbracket \{Q\} \rrbracket(R) = R \bowtie \llbracket Q \rrbracket(R)$$



# Did we invent anything new?

- No, just formulated what's going on in these pipelined languages
- An easy observation:  $RA = PRA$
- But it gives us the formal definition of GQL

GQL model

PRA(all  $\pi_{\Omega}$ )

Pipelined relational algebra over all pattern  
outputs

Observation: PGQ = GQL (expressiveness)

Let's prove a few things

# Folklore: Cypher doesn't do all RPQs

- Cypher restriction: Kleene star only applies to labels
- Easy to model:  $(\xrightarrow{x} \langle a(x) \rangle)^{n..m}$  instead of arbitrary repetitions
- Cypher = PRA over such patterns
- **Theorem:** Cypher cannot express  $(aa)^*$ 
  - (Gheerbrant, L, Peterfreund, Rogova)

# The holy grail of ISO/IEC JTC1 SC32 WG3

- It seems GQL and PGQ have expressivity holes
- **Easy**: find paths in which a property value in **nodes** increases along the path
- **Hard**: find paths in which a property value in **edges** increases along the path
- Committee solution: add more aesthetically pleasing syntax

```
MATCH (:Start)-[:a]->*(:Finish)
FOR EACH SEGMENT (-[x]->-[y]->)
REQUIRE (x.k < y.k)
```

Dangerous! A very similarly looking

```
MATCH (:Start)-[:a]->*(:Finish)
FOR EACH SEGMENT ((x)->+(y))
REQUIRE (x.k != y.k)
```

is NP-hard in data complexity

# Did GQL have to extend the language?

- We are back in our convenient database theory world
  - we have a model and can prove a thing or two
    - as in “basic SQL can’t do recursion”
- **Theorem:** GQL cannot do the “increasing value in edges query”
  - and many more .... (GLPR’24)
  - caveat: modulo one condition, no back-edges
  - mix of FMT and some formal languages, our stuff

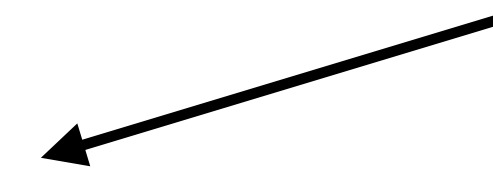
# GQL defies intuition

- REACHABILITY is complete for NLOGSPACE under FO-reductions
- GQL defines reachability:  $(:Start) \rightarrow^* (:Finish)$
- GQL expresses all FO = relational algebra
- and yet:
- Theorem: There are DLOGSPACE queries not expressible in GQL

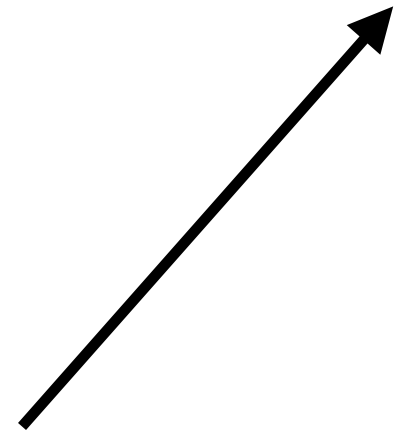
# How does GQL do the “increasing value in edges” query? It’s a real language after all

```
MATCH p = (:Start) ->* (:Finish)
EXCEPT
MATCH p = (:Start) ->*
  ( ->[x]->()->[y]-> WHERE x.k >= y.k)
  ->* (:Finish)
```

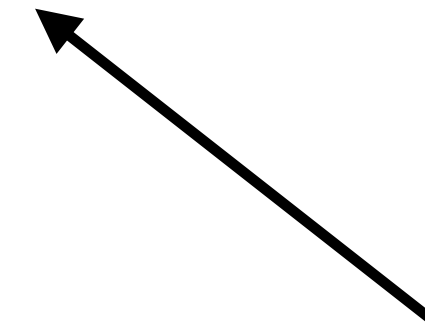
all paths



difference

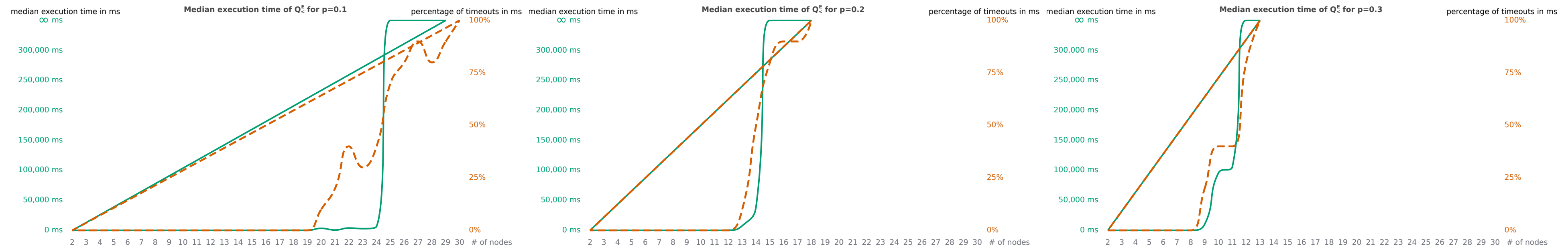


bad paths





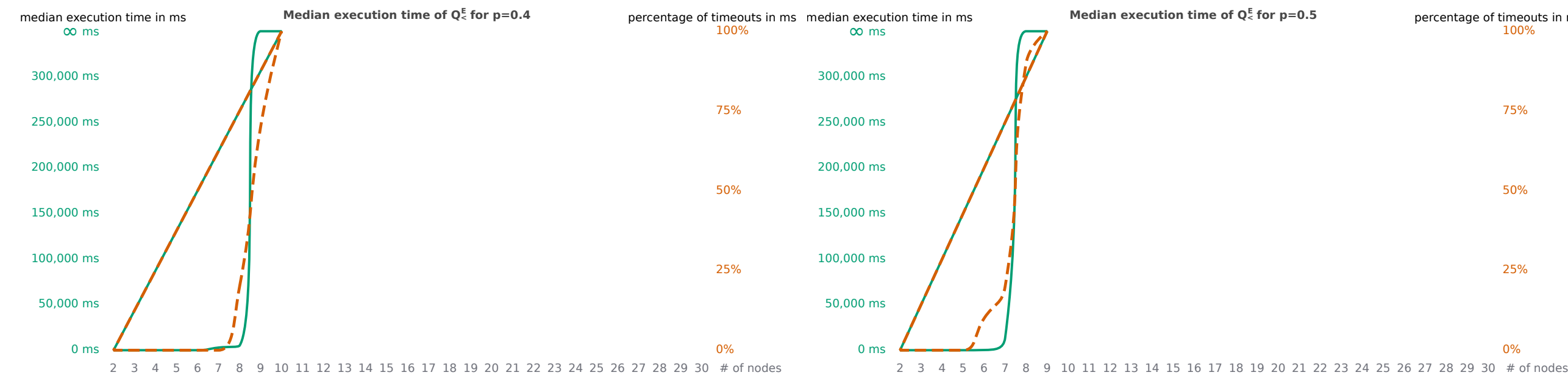
# Does it have a chance to work? No way!



(a)  $p = 0.1$

(b)  $p = 0.2$

(c)  $p = 0.3$



(d)  $p = 0.4$

(e)  $p = 0.5$

Best on sparse graphs: up to 30 nodes then 100% timeouts; dense graphs: 8 nodes

Cypher has been with us for over a decade

It must solve real life problems

What does it do?

# Cypher gives us lists

- `nodes(p)` — list of nodes of path `p`
- `relationships(p)` — list of edges of path `p`
- and `reduce` (or `fold`) over them

“Increasing positive values in edges” query

```
MATCH p=(:Start) ->* (:Finish)
WITH [r in relationships(p) | r.k] AS values, p
WITH ( reduce(res=0, v in values |
          CASE v > res
            WHEN true THEN v ELSE 0
          END ) AS result, p
WHERE result != 0
RETURN p
```

# Are lists always innocent?

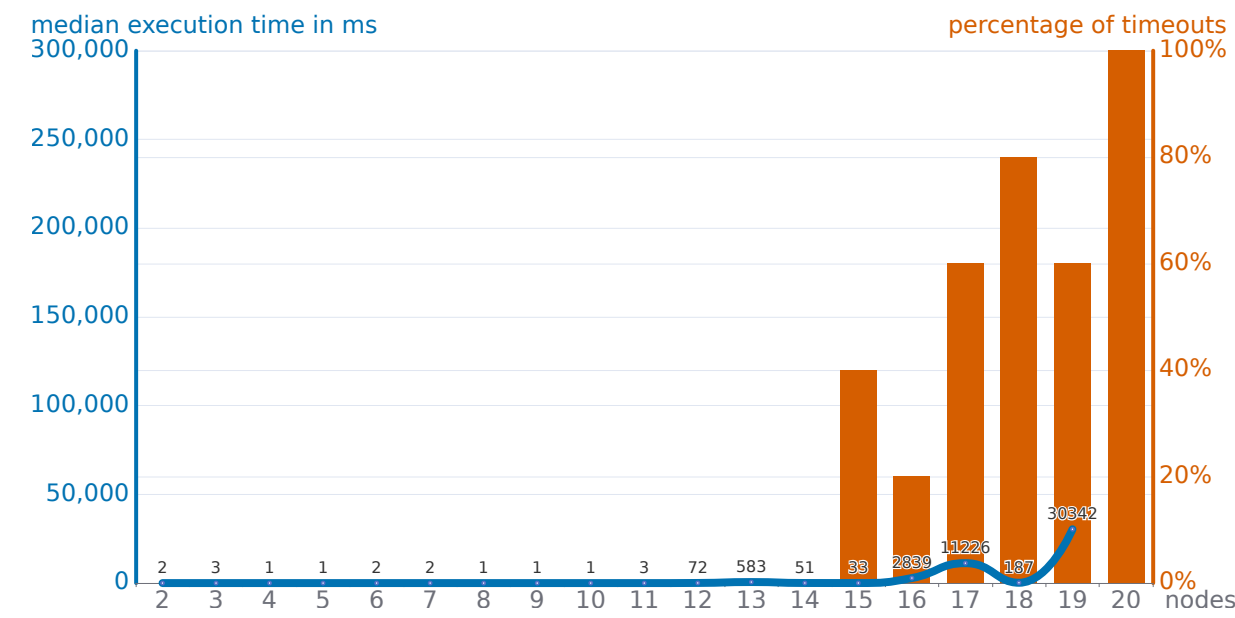
```
MATCH (n)
WITH collect(n.name) AS allNodes
MATCH path=(:Start)-[*]-()
WITH path, allNodes, [y IN nodes(path) | y.name] AS nodesInPath
WHERE all(node in allNodes WHERE node IN nodesInPath)
AND size(allNodes)=size(nodesInPath)
RETURN path LIMIT 1
```

Hamiltonian Path

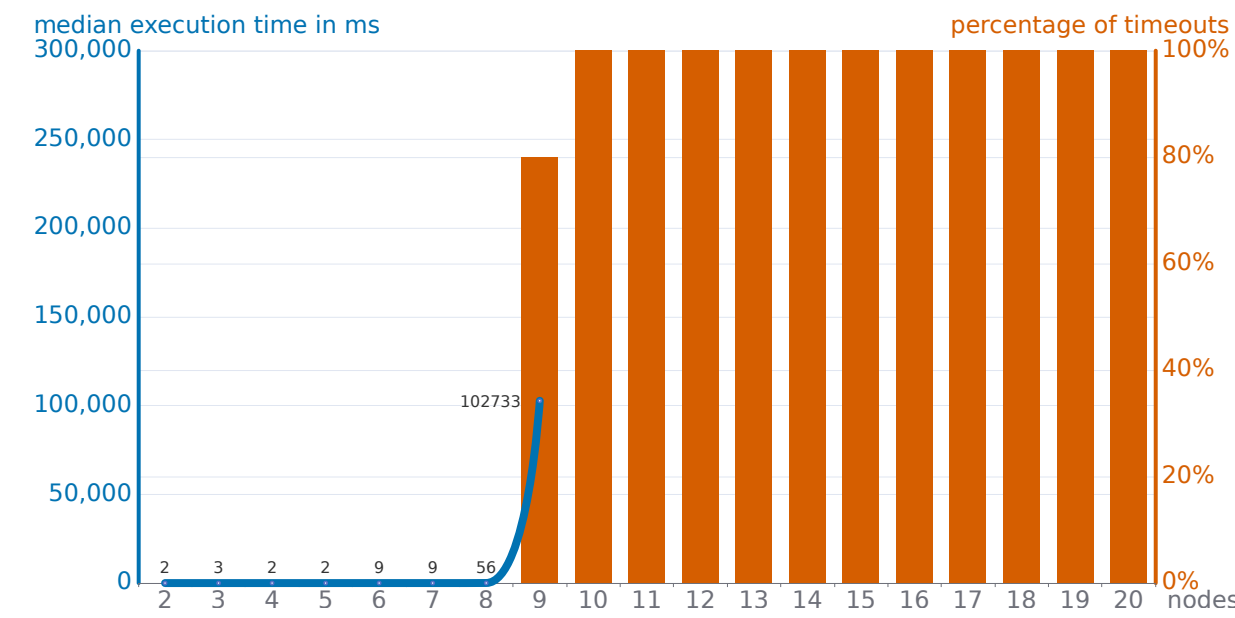
```
MATCH p = allShortestPaths((:Start)-[:Edge*]->(:Finish))
WITH [r IN relationships(p) | r.value] AS values, p
UNWIND values as valSet
WITH sum(valSet) AS sum, p
WHERE sum = $T
RETURN p
```

Subset-Sum

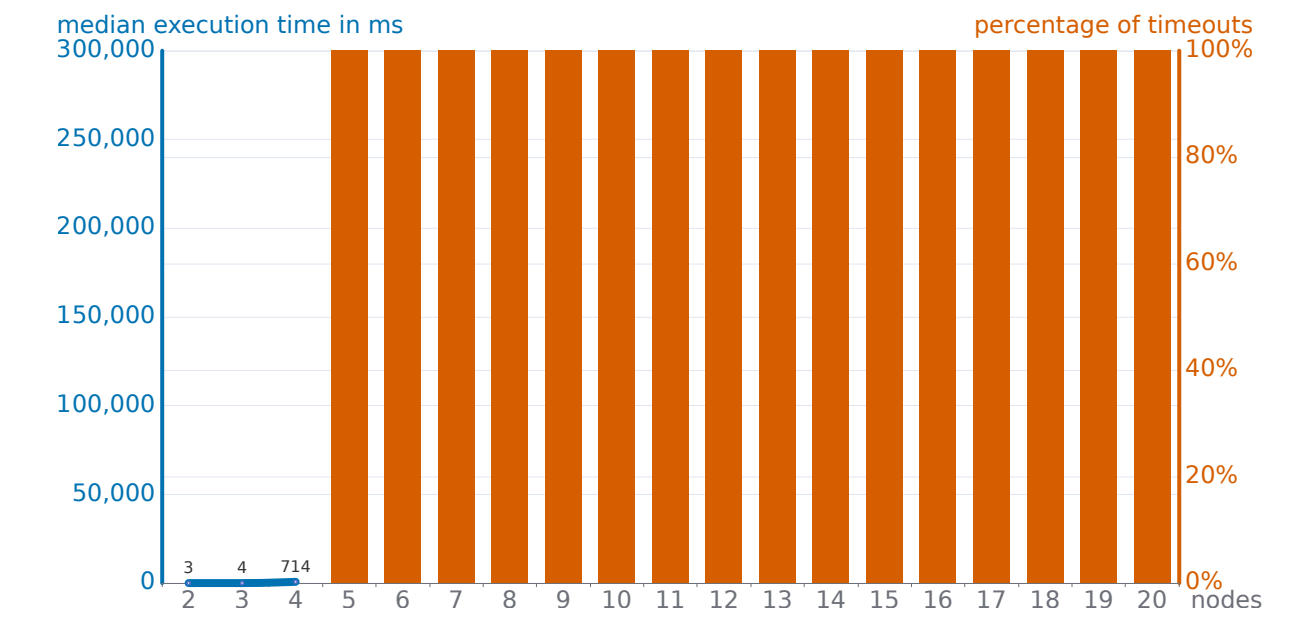
# ... and they don't work (GLR'24)



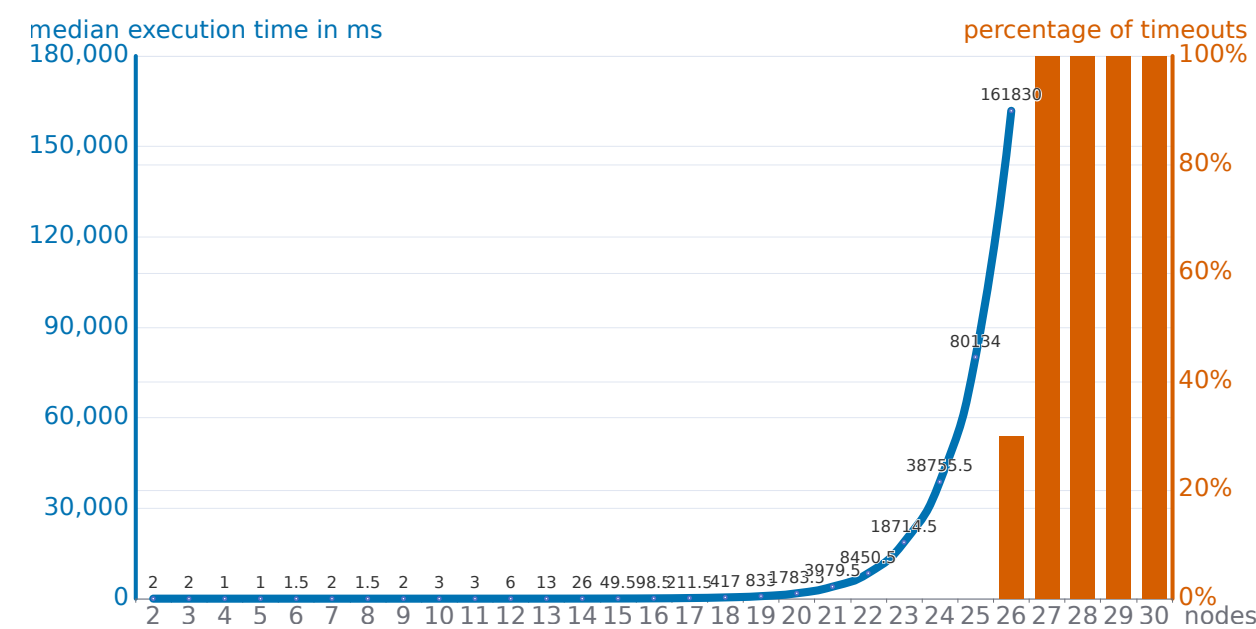
(a) Median execution time and number of time-outs for  $p = 0.1$



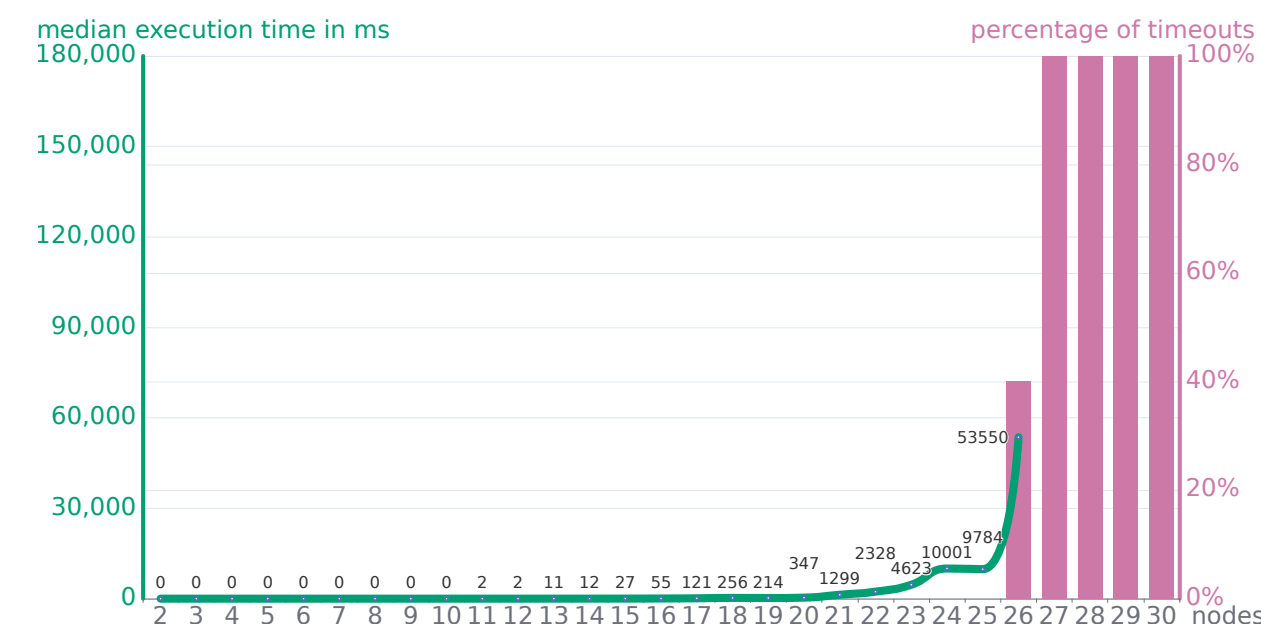
(b) Median execution time and number of time-outs for  $p = 0.3$



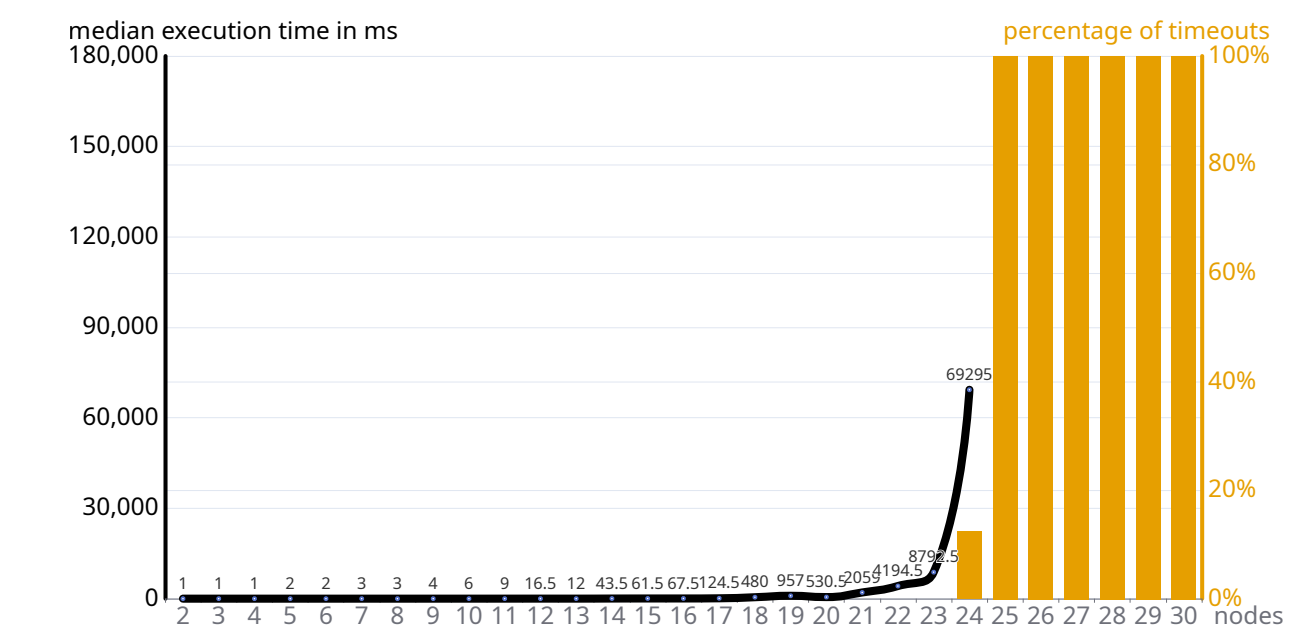
(c) Median execution time and number of time-outs for  $p = 0.8$



(a) Neo4j



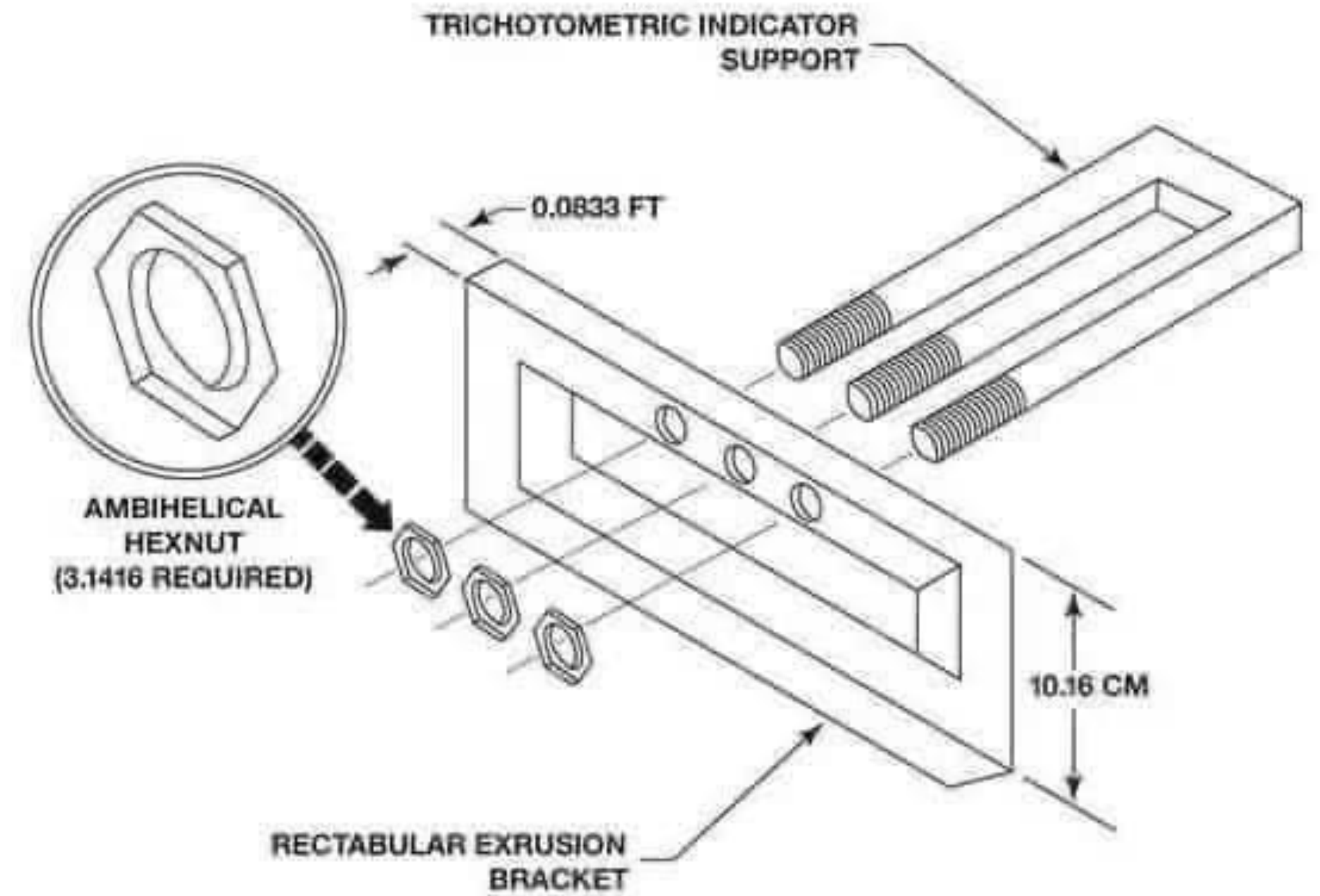
(b) Postgres



(c) DuckDB

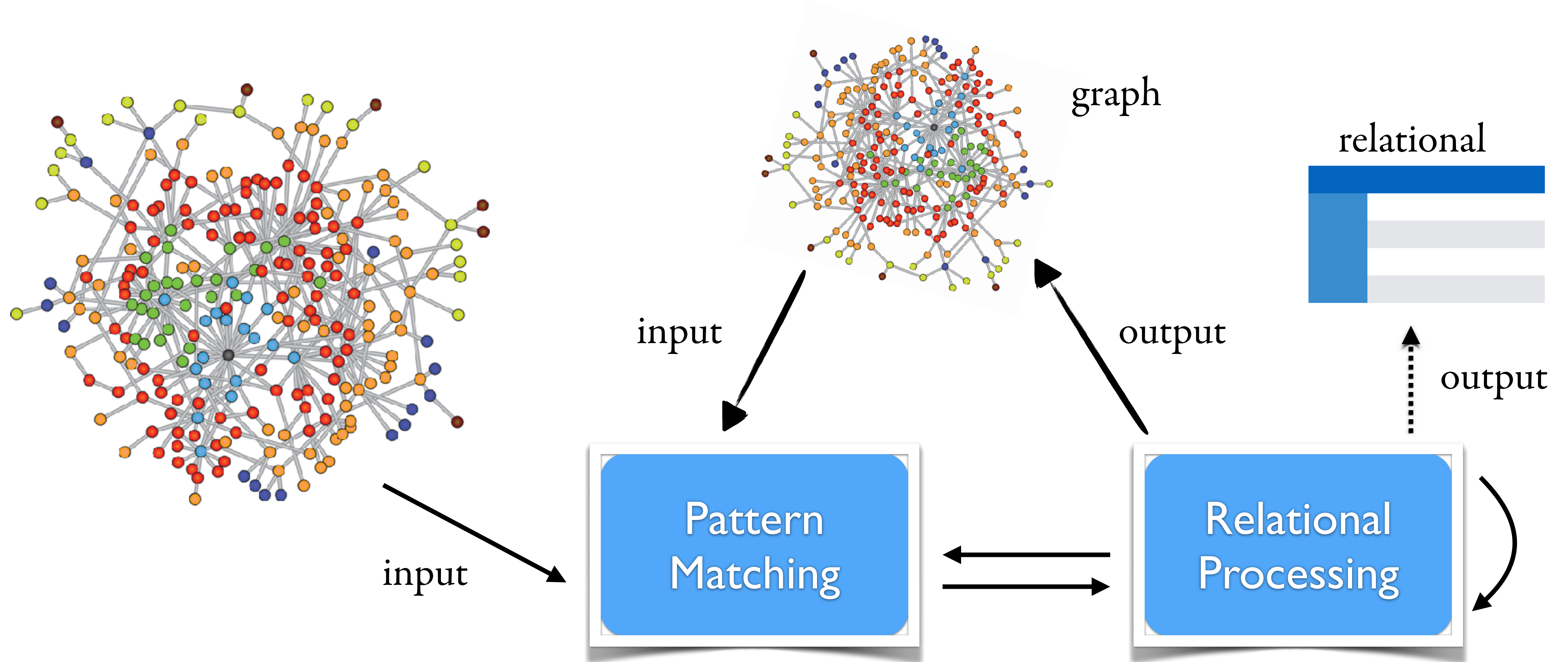
... except on tiniest graphs

# Why? Didn't we design one of these?



that make sense until they don't?

# GQL and PGQL design: bird's eye view of a single transaction



What is missing?

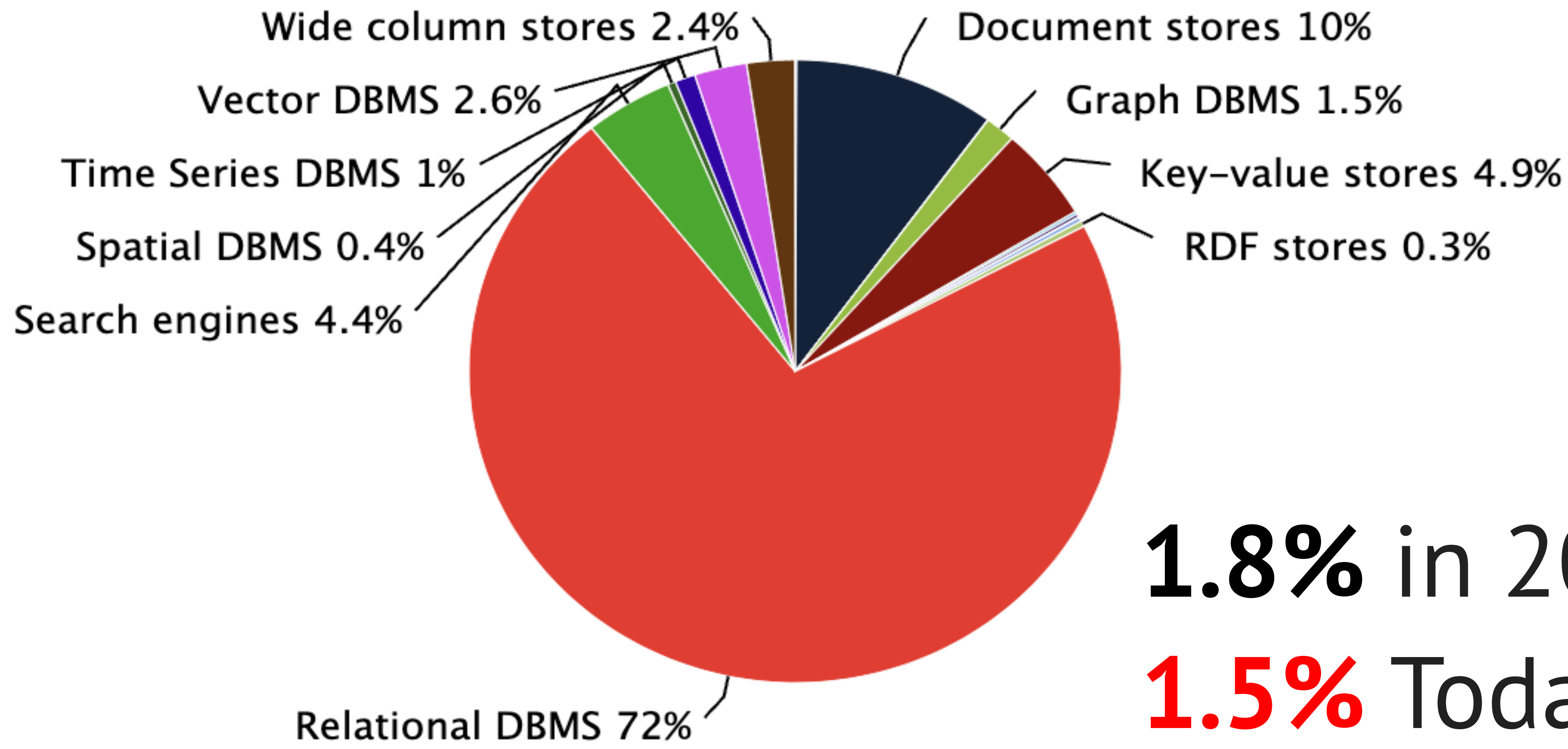
**COMPOSITIONALITY**

# The future

- Standards go ahead: SQL 2028 with updated PGQ
- GQL 2029
- Is GQL there to stay? How many remember CODASYL, NDL?
  - we had standardized graph query languages in the late 1980s!
  - Big debate (see next talk) - and they lost to **relational**
  - Relational languages are compositional:
    - give me reachability and relational algebra and you have all of **NLOGSPACE**
- Is the future graph or relational?



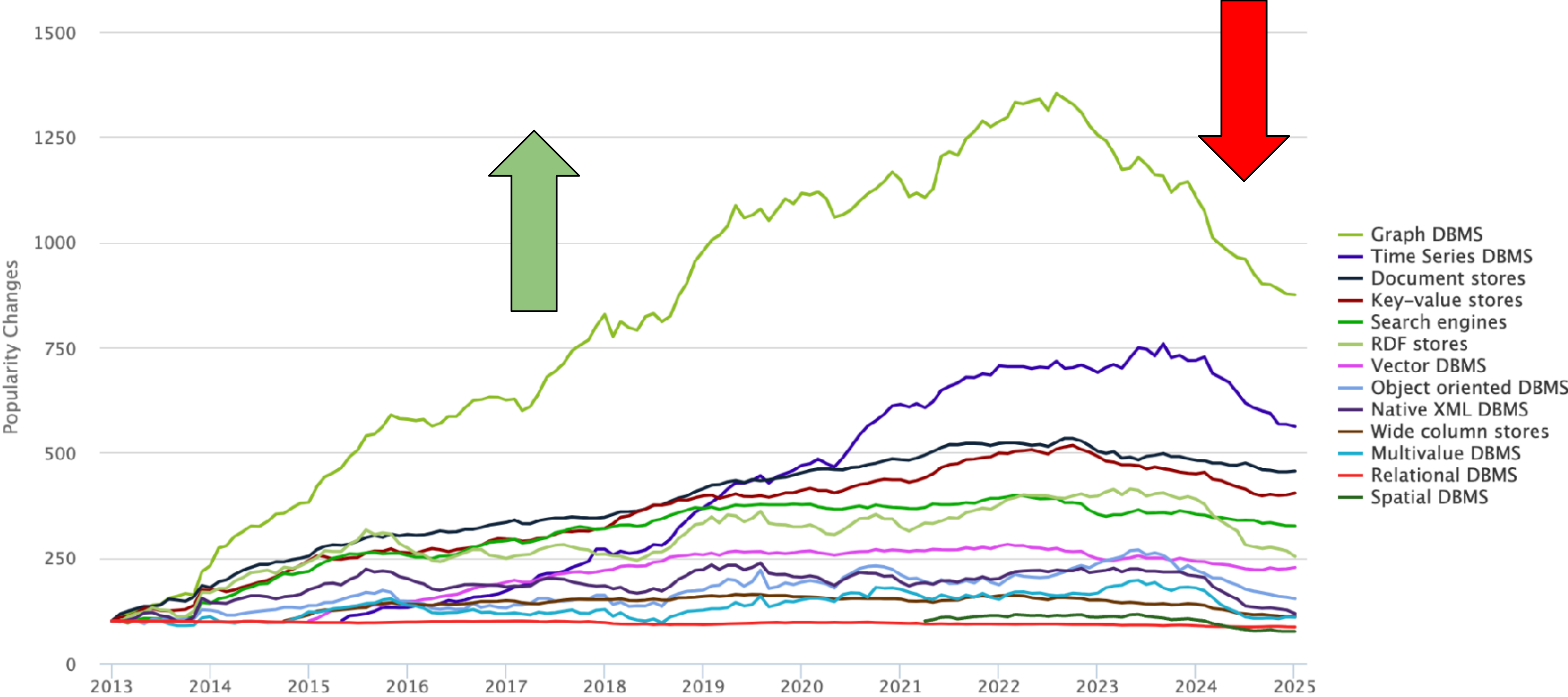
# Ranking scores per category in percent, January 2025



**1.8%** in 2023

**1.5%** Today

# Complete trend, starting with January 2013



Thanks!

And we are ready to hear about the  
bright relational future