# 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

# 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 ἴσος

**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

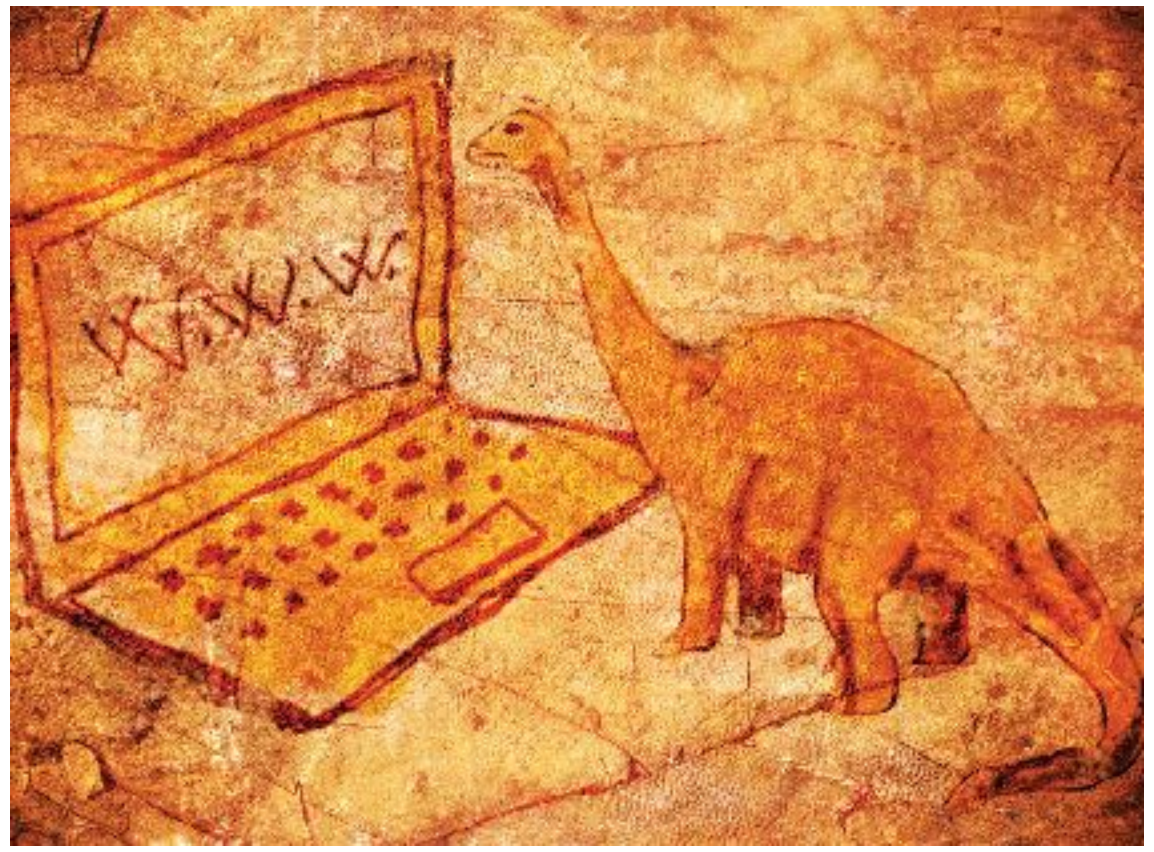
# GQL vs SQL/PGQ

- 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 —-

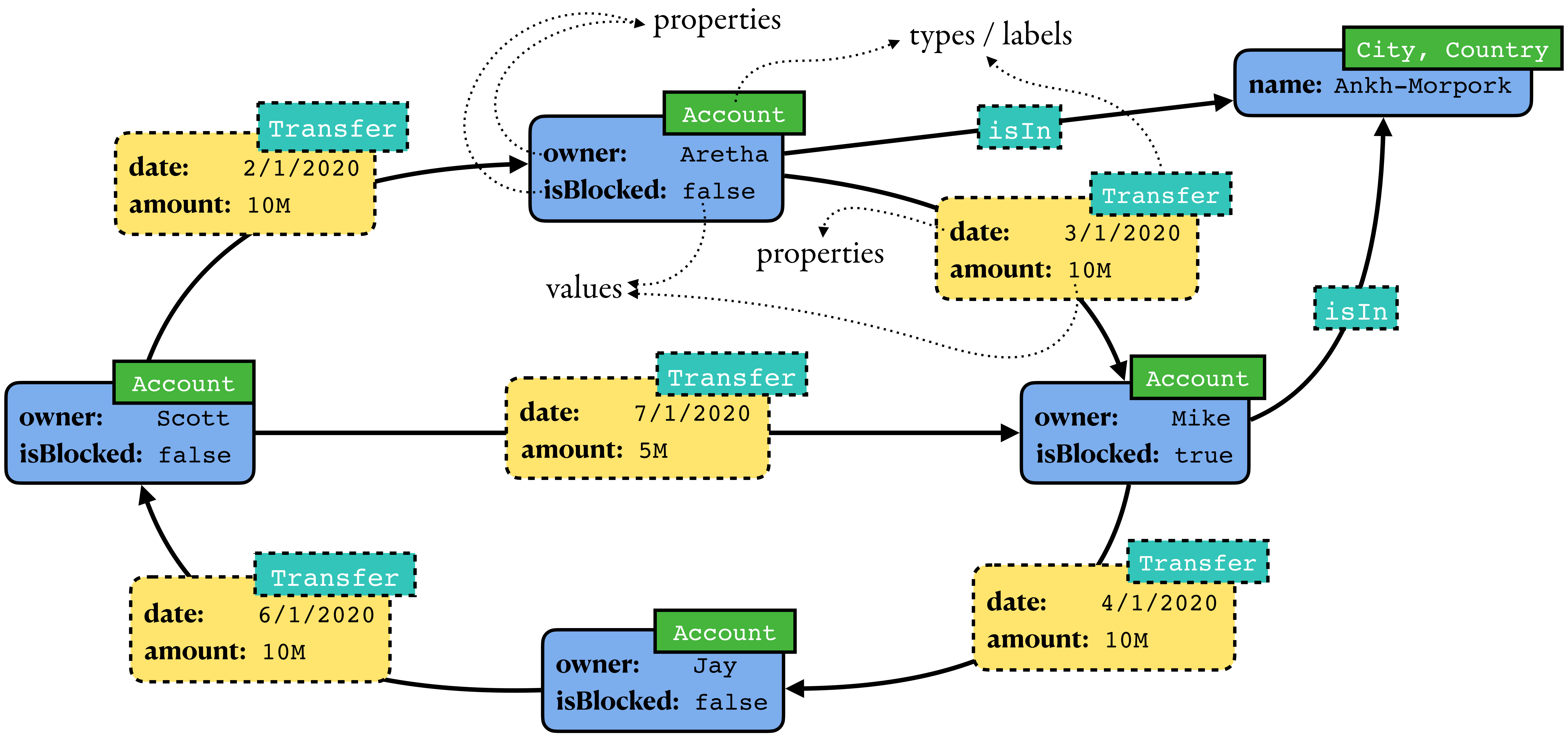
**Graph Query Languages Research**

1987: RPQs
1990: CRPQs
followed by many others

2RPQs 2UCRPQs 2CRPQs ECRPQs UCRPQs RPDQs

1959 1987 1990 2004 2011 2024

# Data Model: Property Graphs



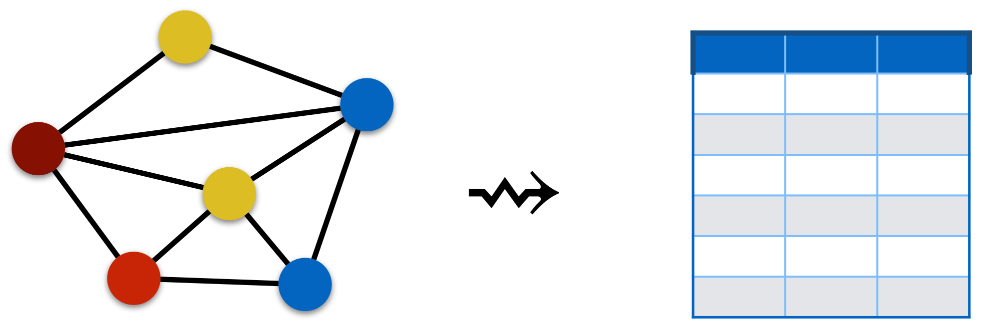

A data model based on graphs where both nodes and edges (relationships) can have

- properties (attributes)
- types (labels)

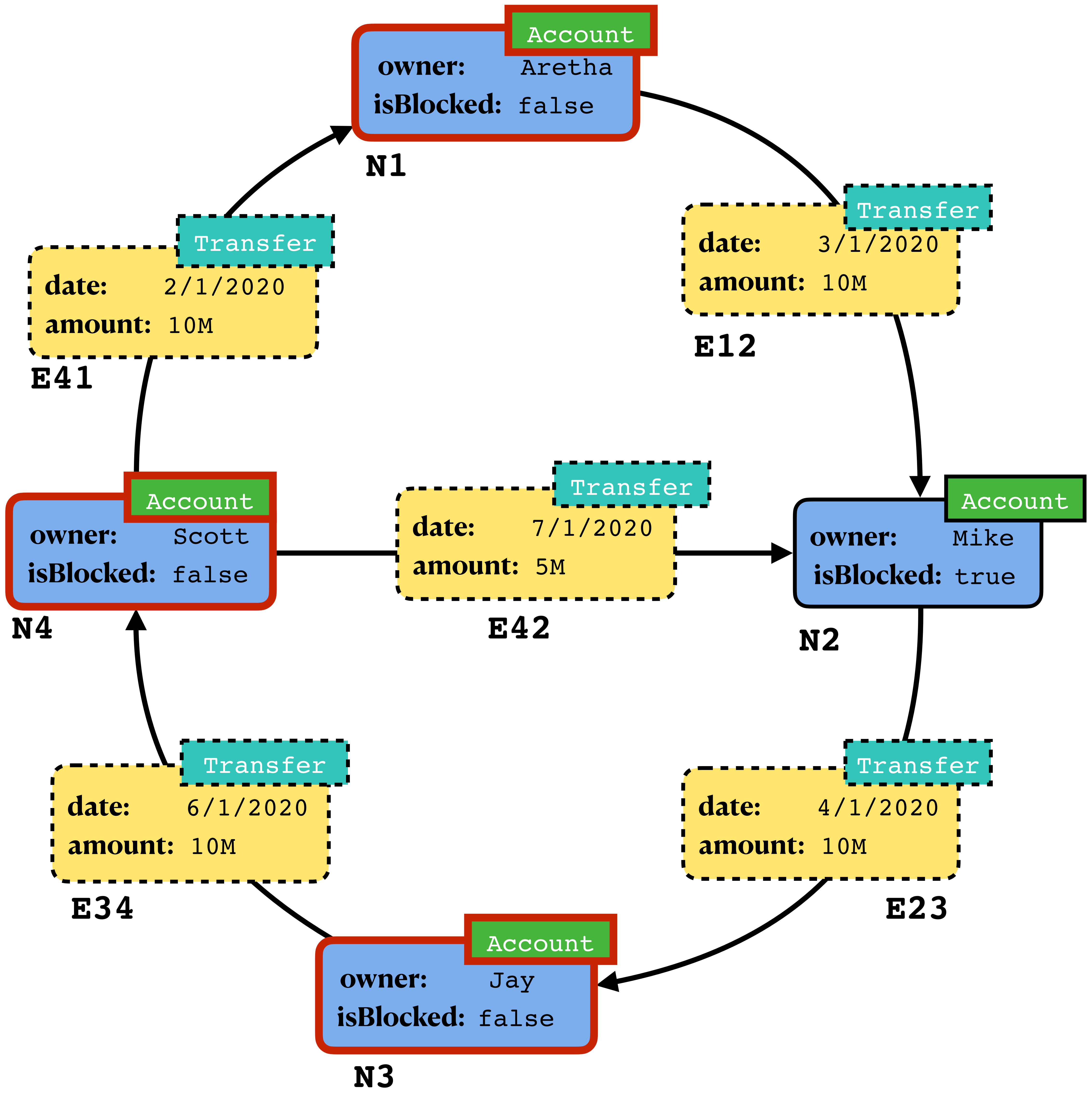
# GQL by examples

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

# The Core:
# Graph Pattern Matching



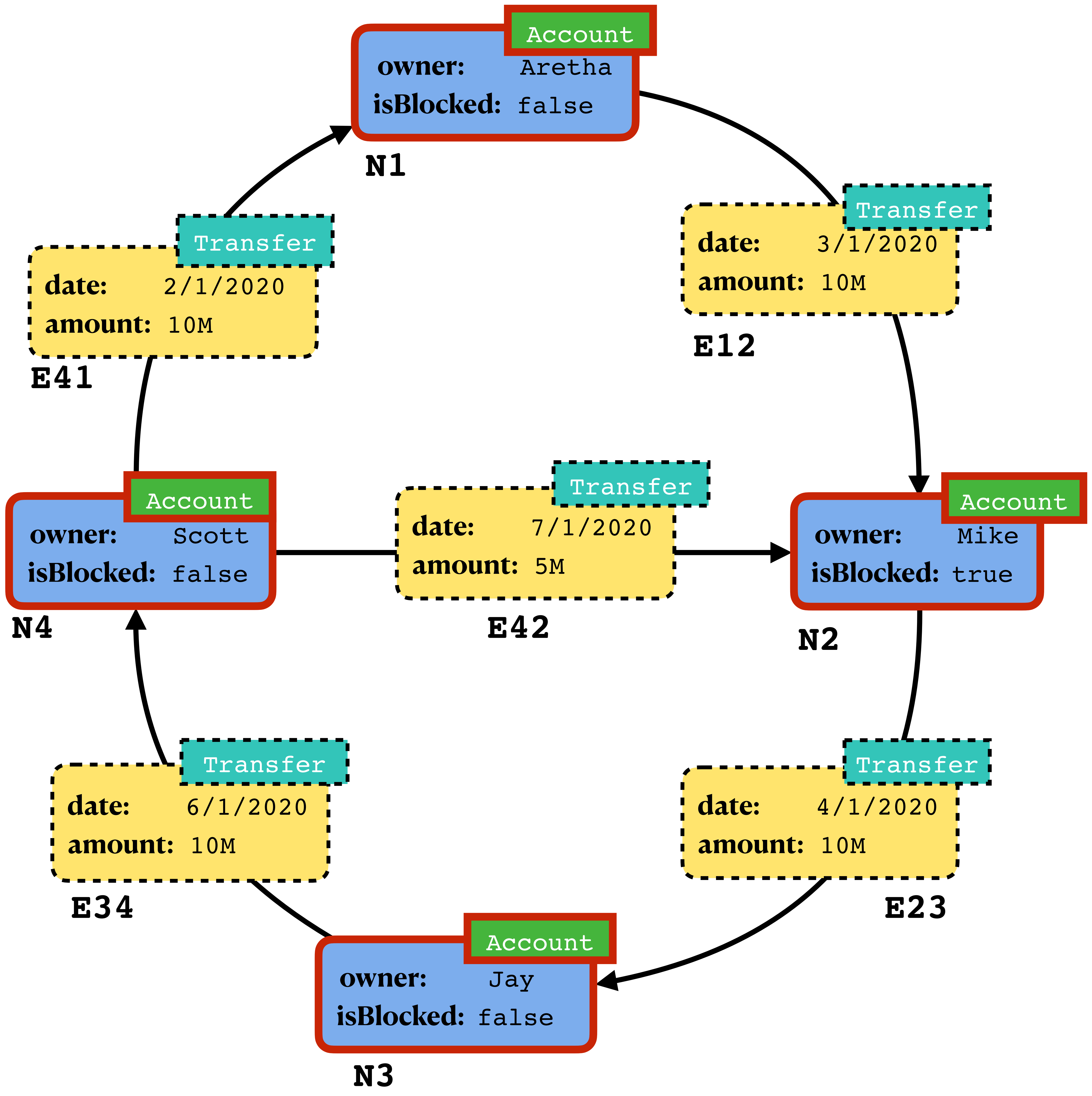
# Selecting Nodes




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

| x |
|---|
| N1 |
| N3 |
| N4 |

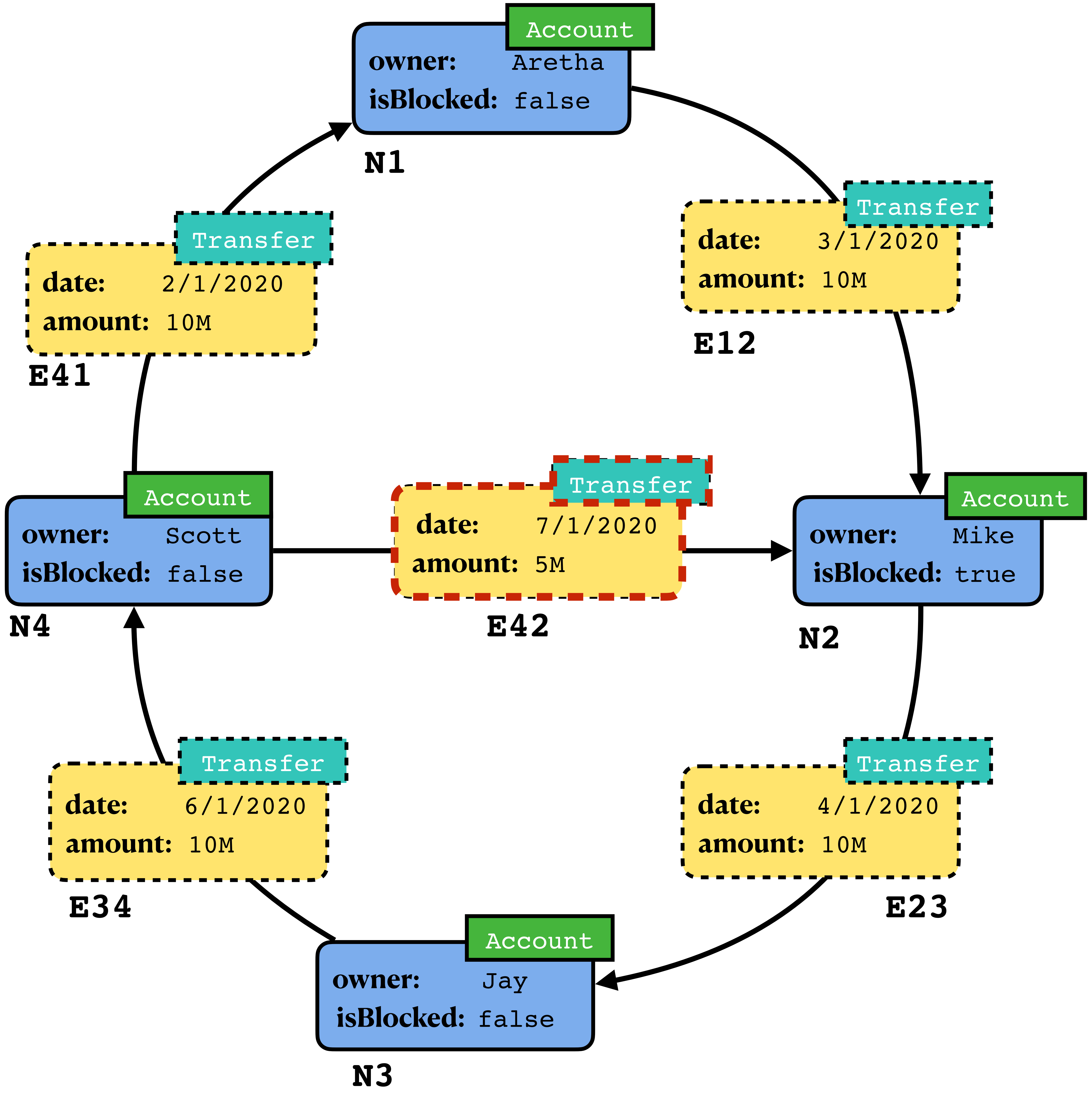
# Selecting Nodes




**MATCH** (x) ⇝ all nodes

| x |
|---|
| N1 |
| N2 |
| N3 |
| N4 |

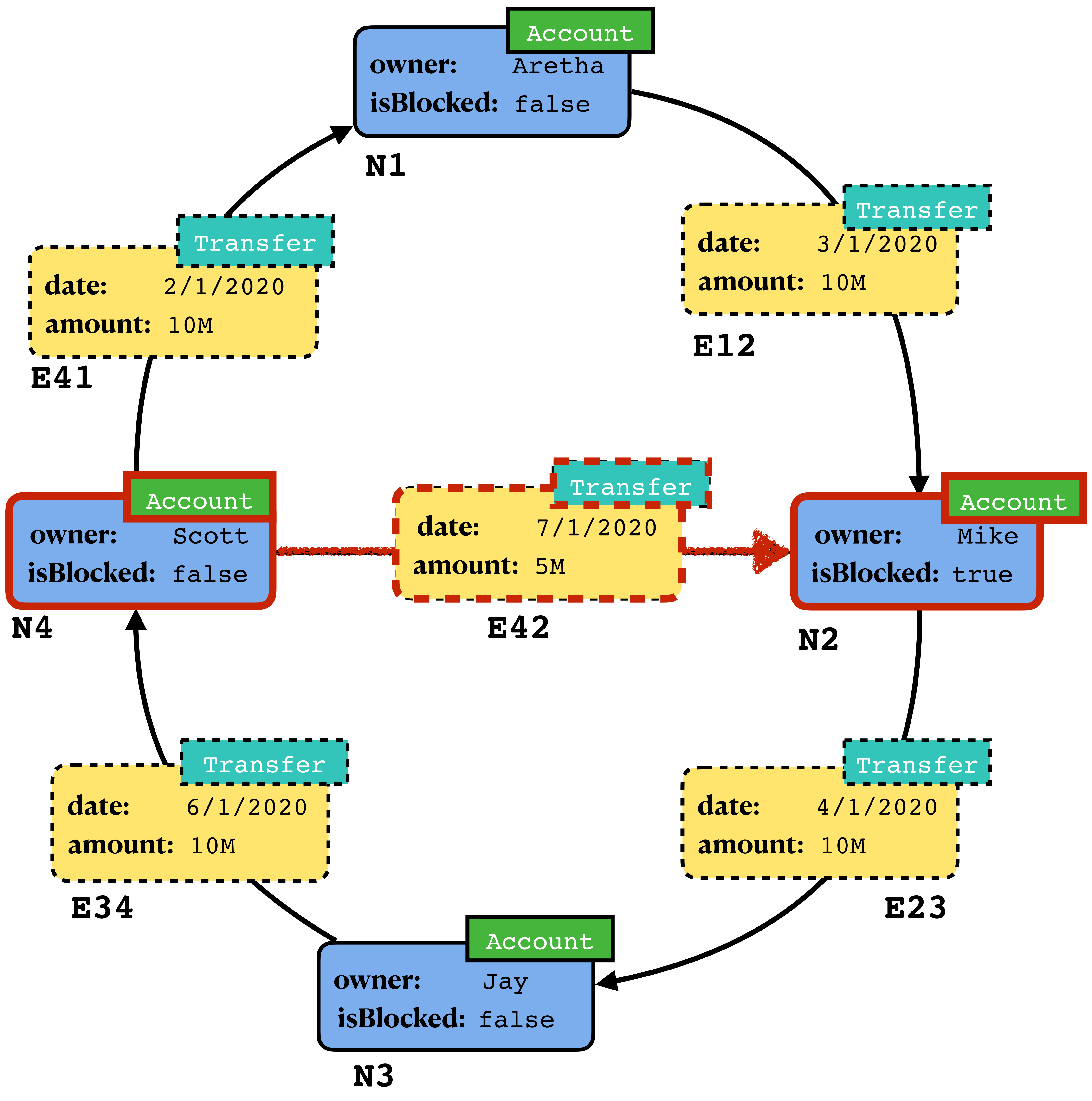
# Selecting Edges




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

| e |
|---|
| E42 |

# Combining Nodes & Edges into Paths

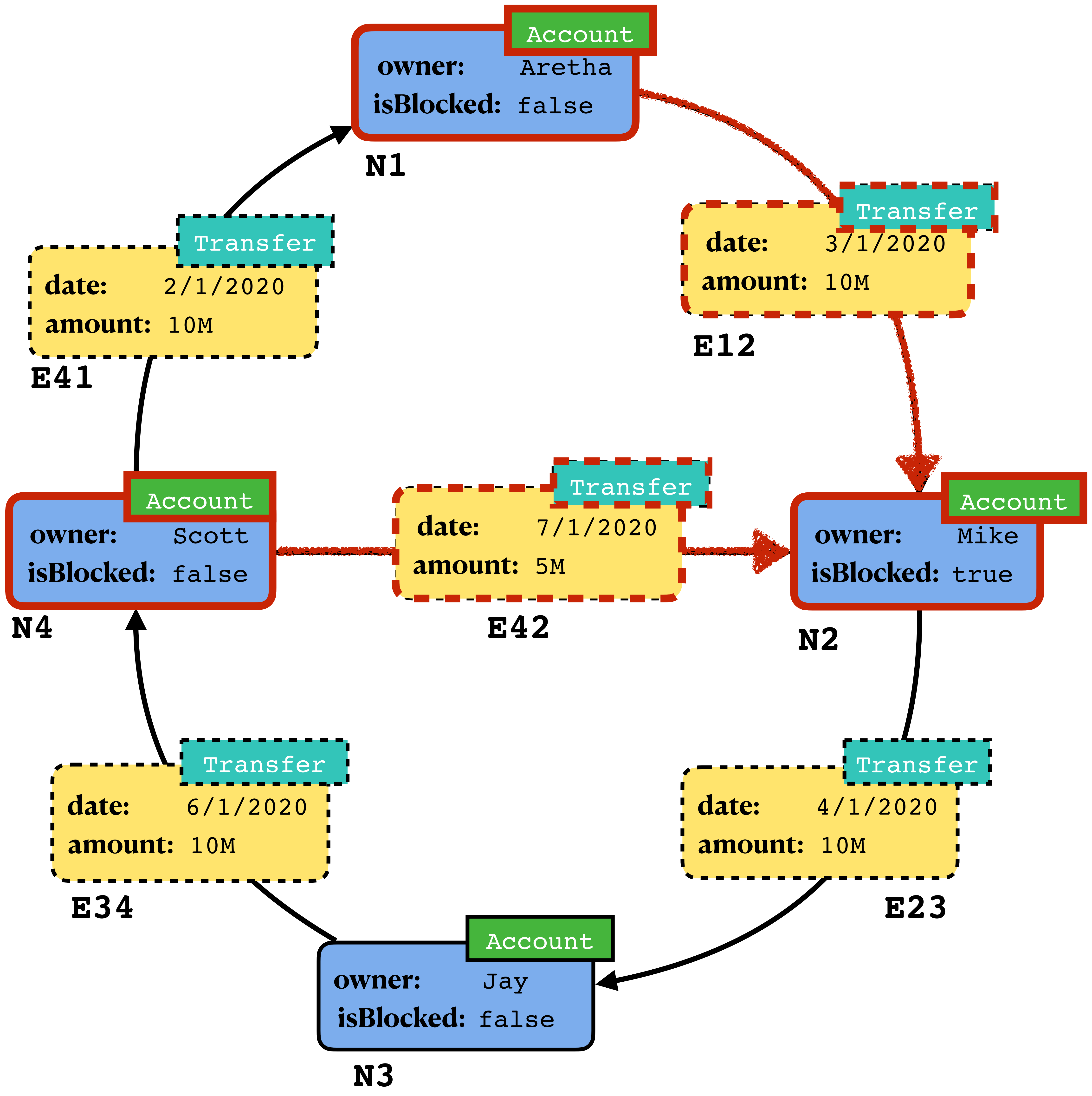



Combining nodes and edges:

```
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

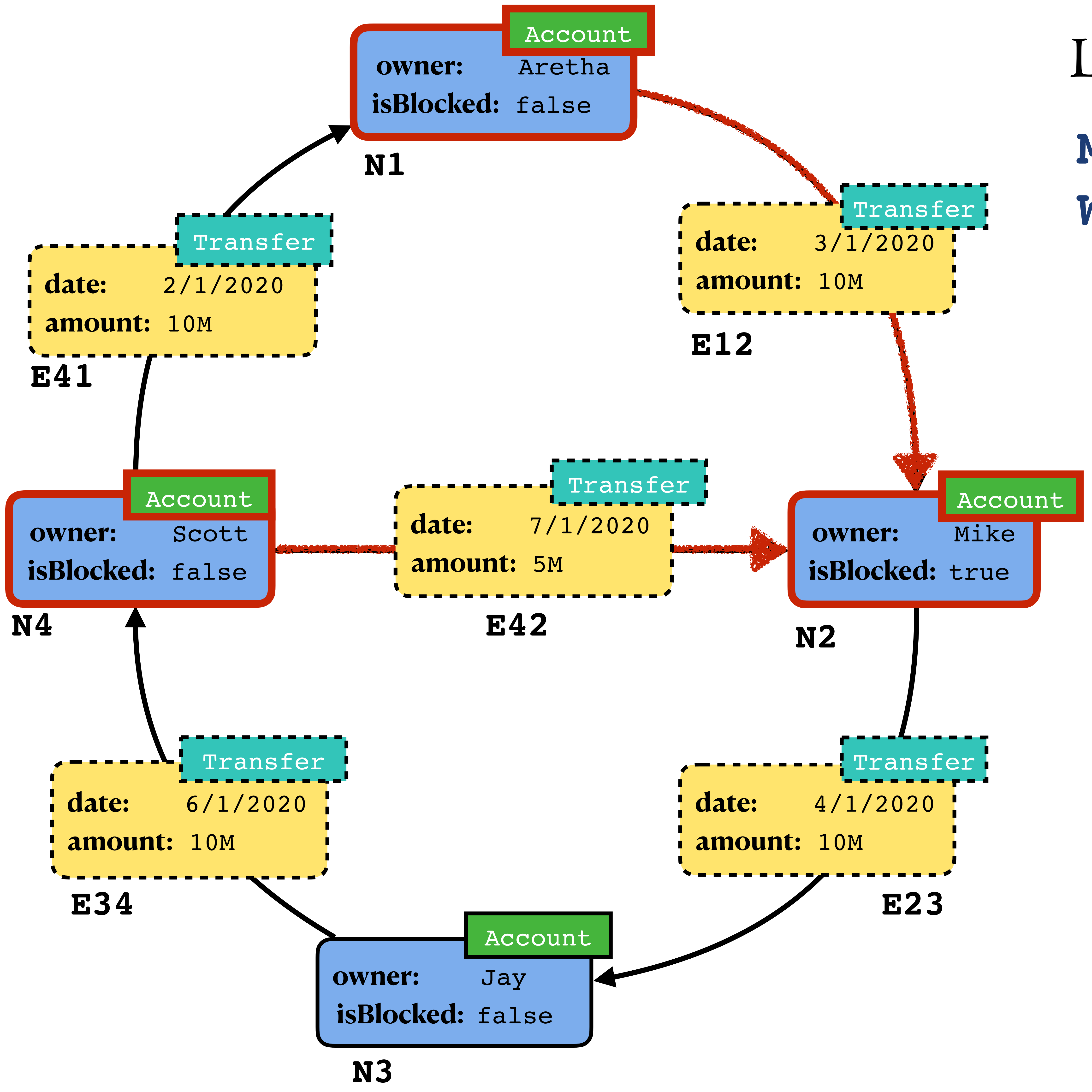



Combining nodes and edges:

```
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



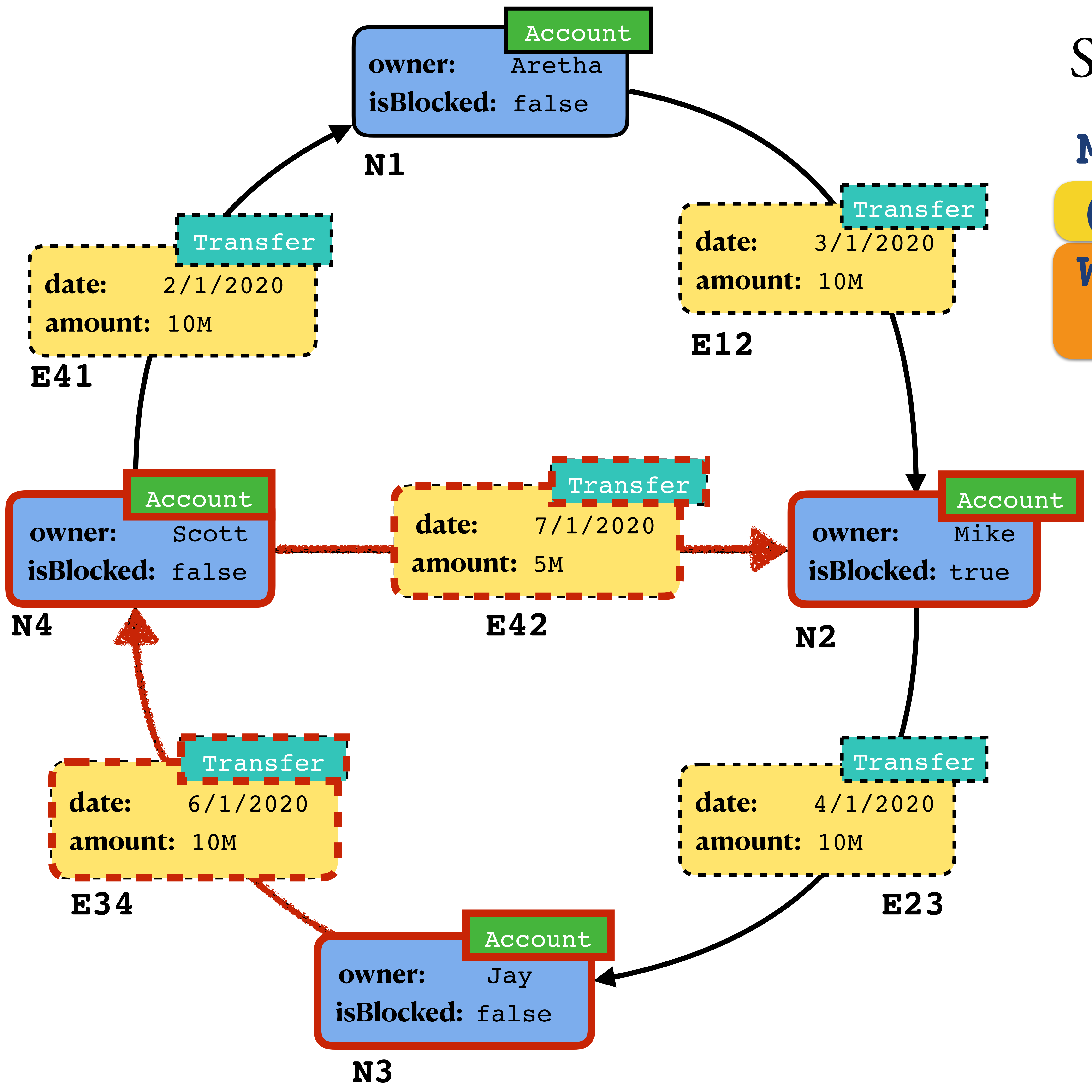

Longer paths are defined via ASCII-art :

```
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: ~, – , –>, <–

# Graph Traversal



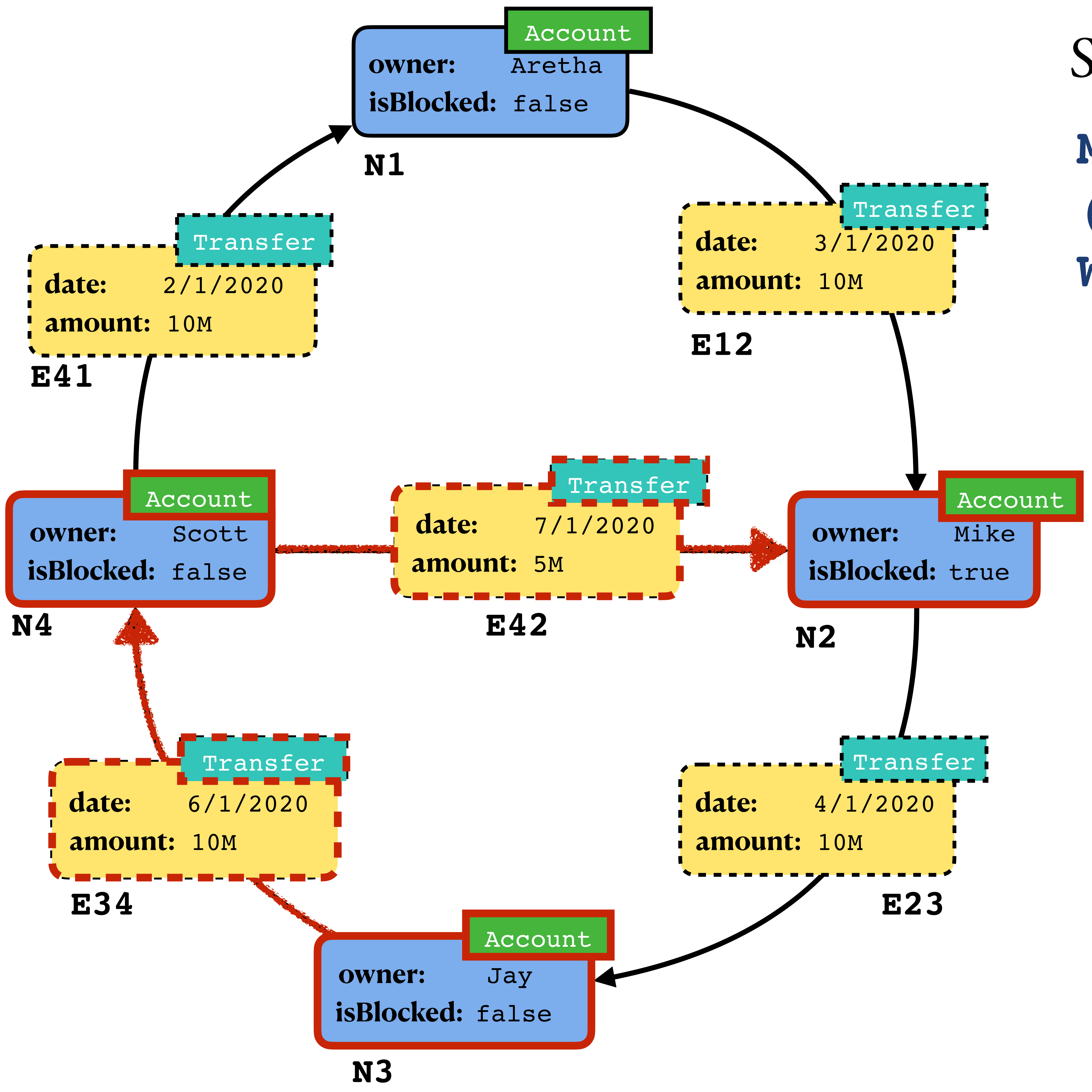

Specifying graph traversal:

```
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:

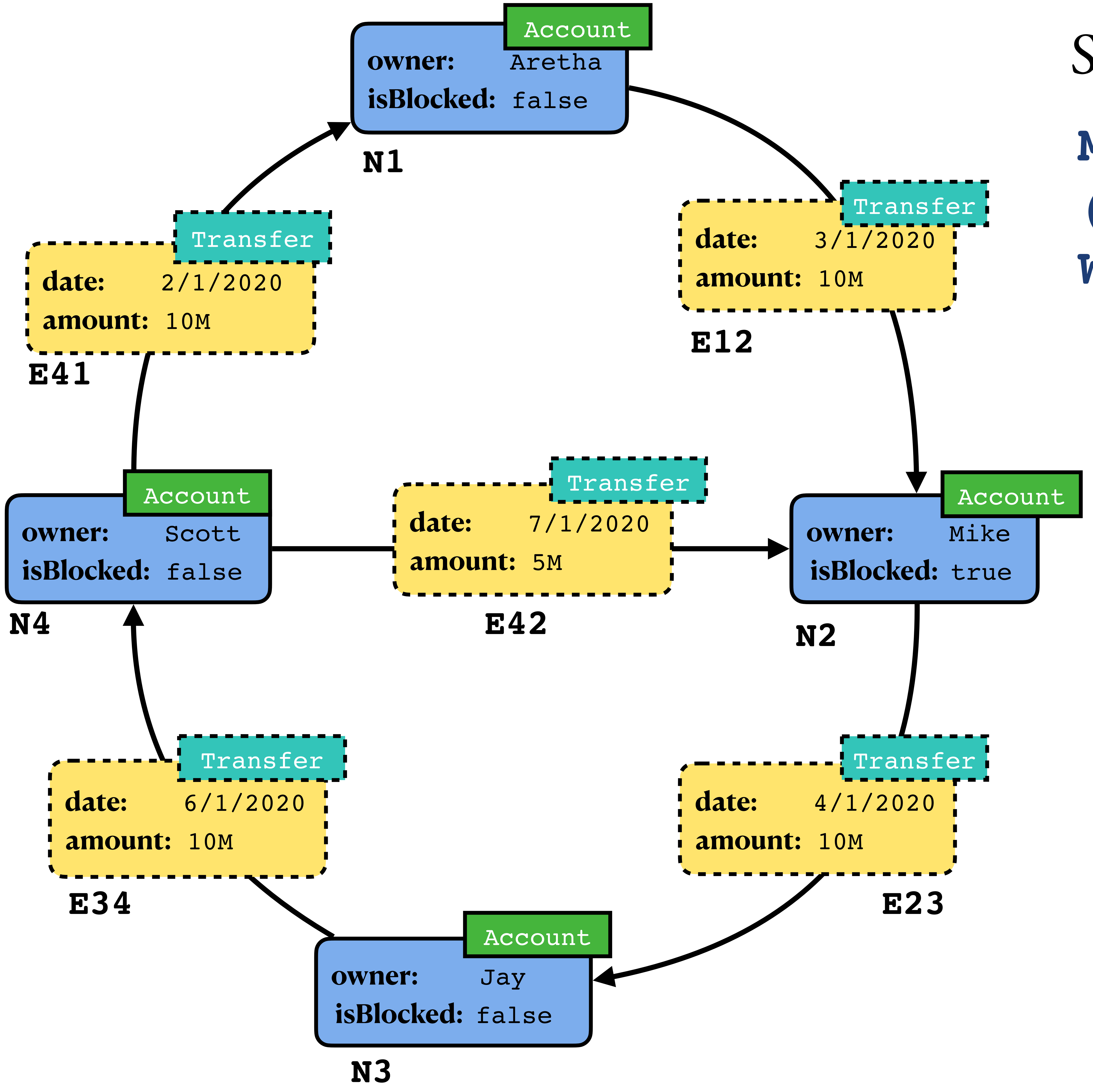
group variable

```
MATCH
(x:Account)-[t:Transfer]->{2,4}(y:Account)
WHERE x.isBlocked = 'false'
  AND y.isBlocked = 'true'
```

| 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'
```

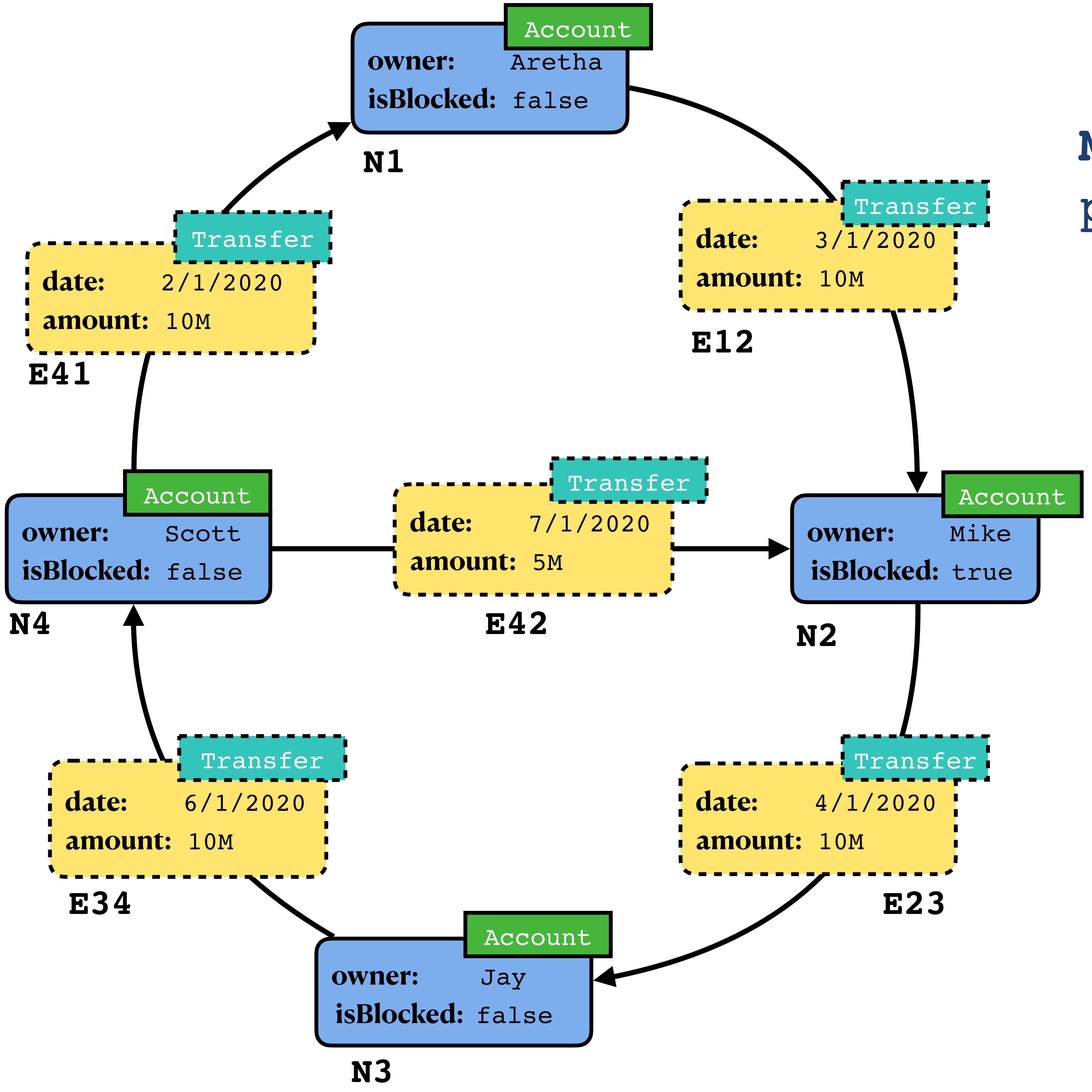
Repetitions can be `{n,m}`

```
{n,}
{,m}
*
+
```

Path conditions can be added:

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

# Path Variables

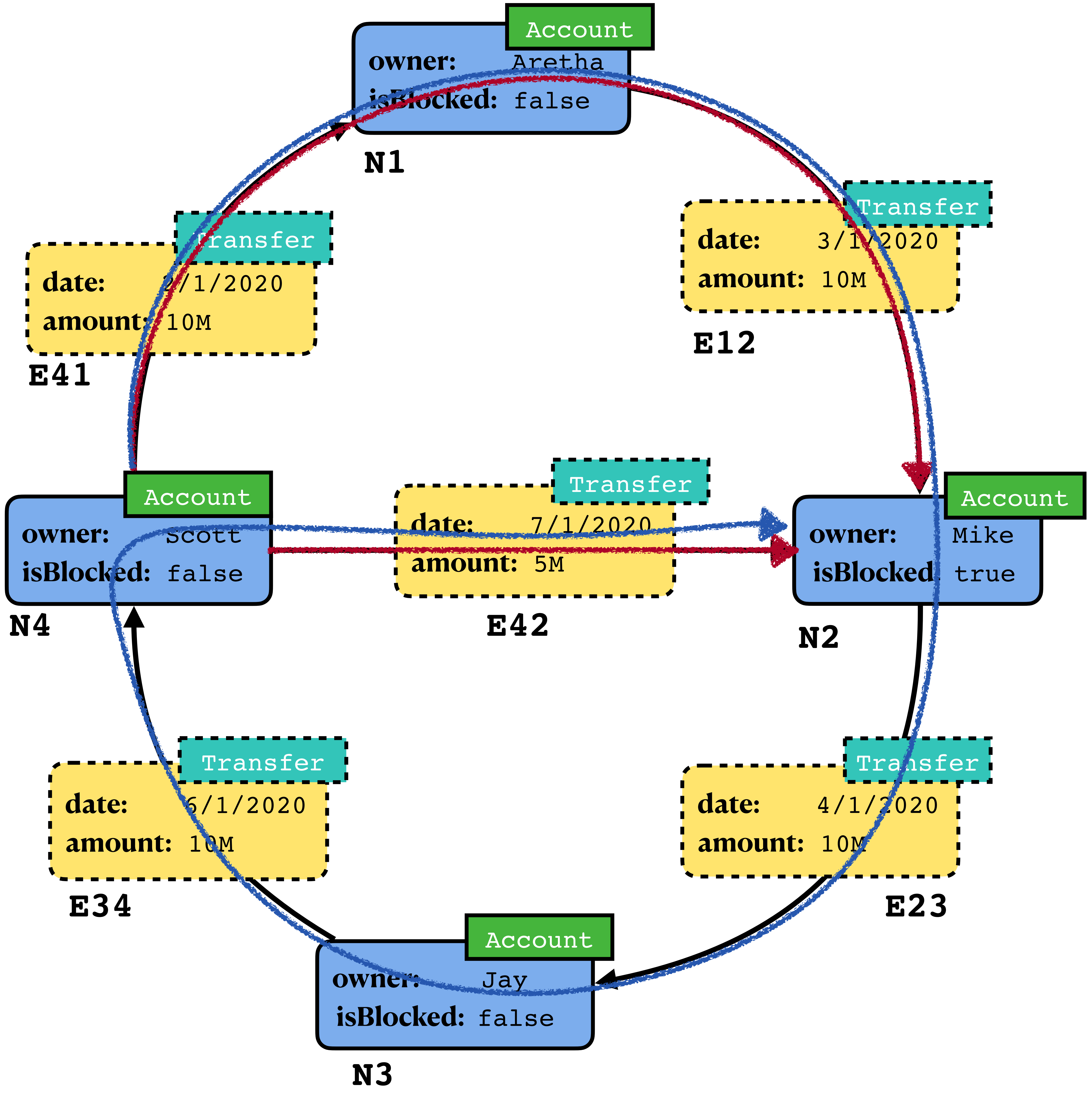



```
MATCH
p = (x WHERE x.owner = 'Scott')
              -[:Transfer]->*
        (y WHERE y.owner = 'Mike')
```

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**

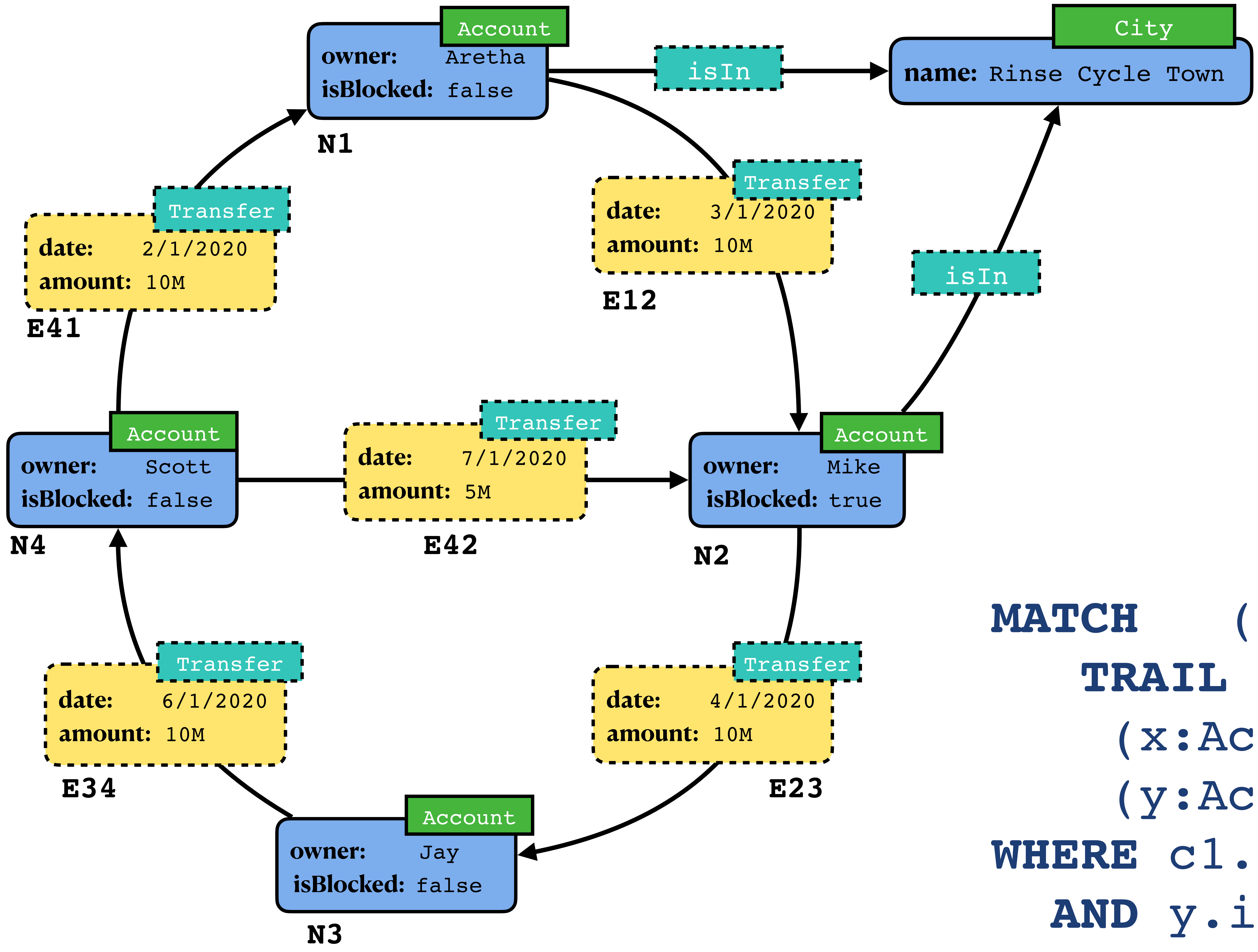
# Disjunction

As expected, there is OR |

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

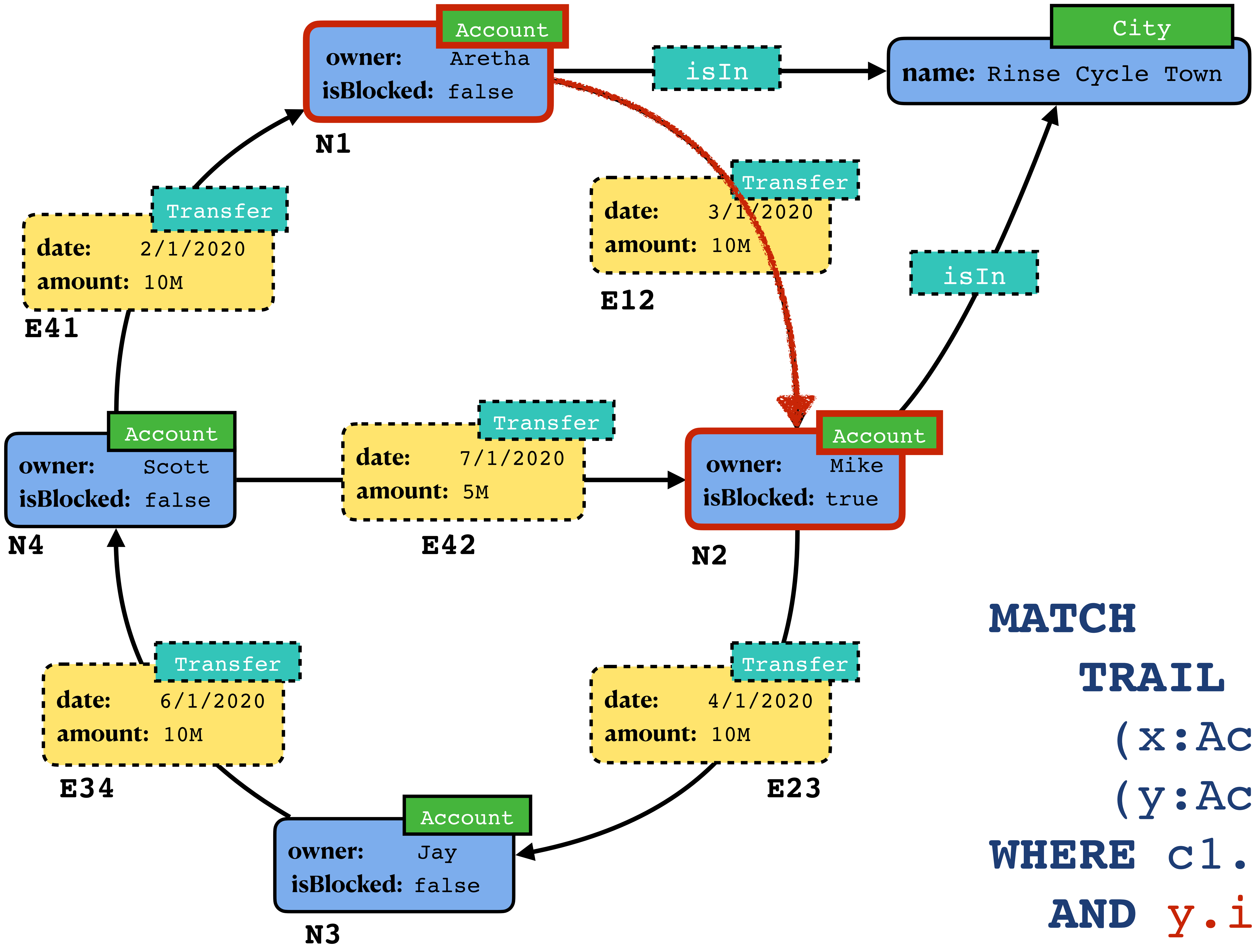
"Transfers to a blocked account and transfers initiated by Mike"

# Joins in Patterns



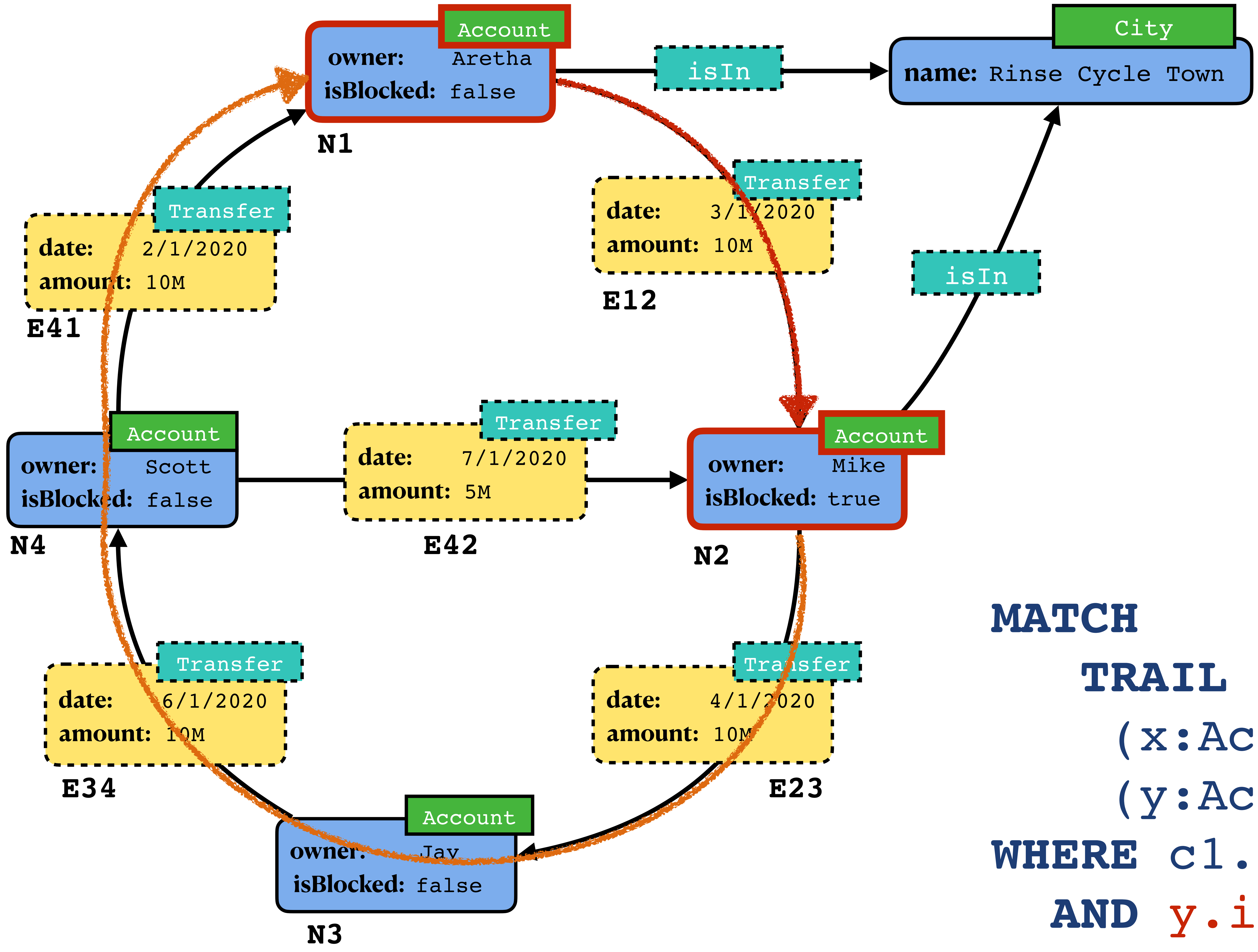

```
MATCH   (x) -[:Transfer]->  (y),
   TRAIL (y) -[:Transfer]->+ (x),
    (x:Account)-[:isIn]->(c1:City),
    (y:Account)-[:isIn]->(c2:City)
WHERE c1.name = c2.name
  AND y.isBlocked = 'true'
```

# Joins in Patterns



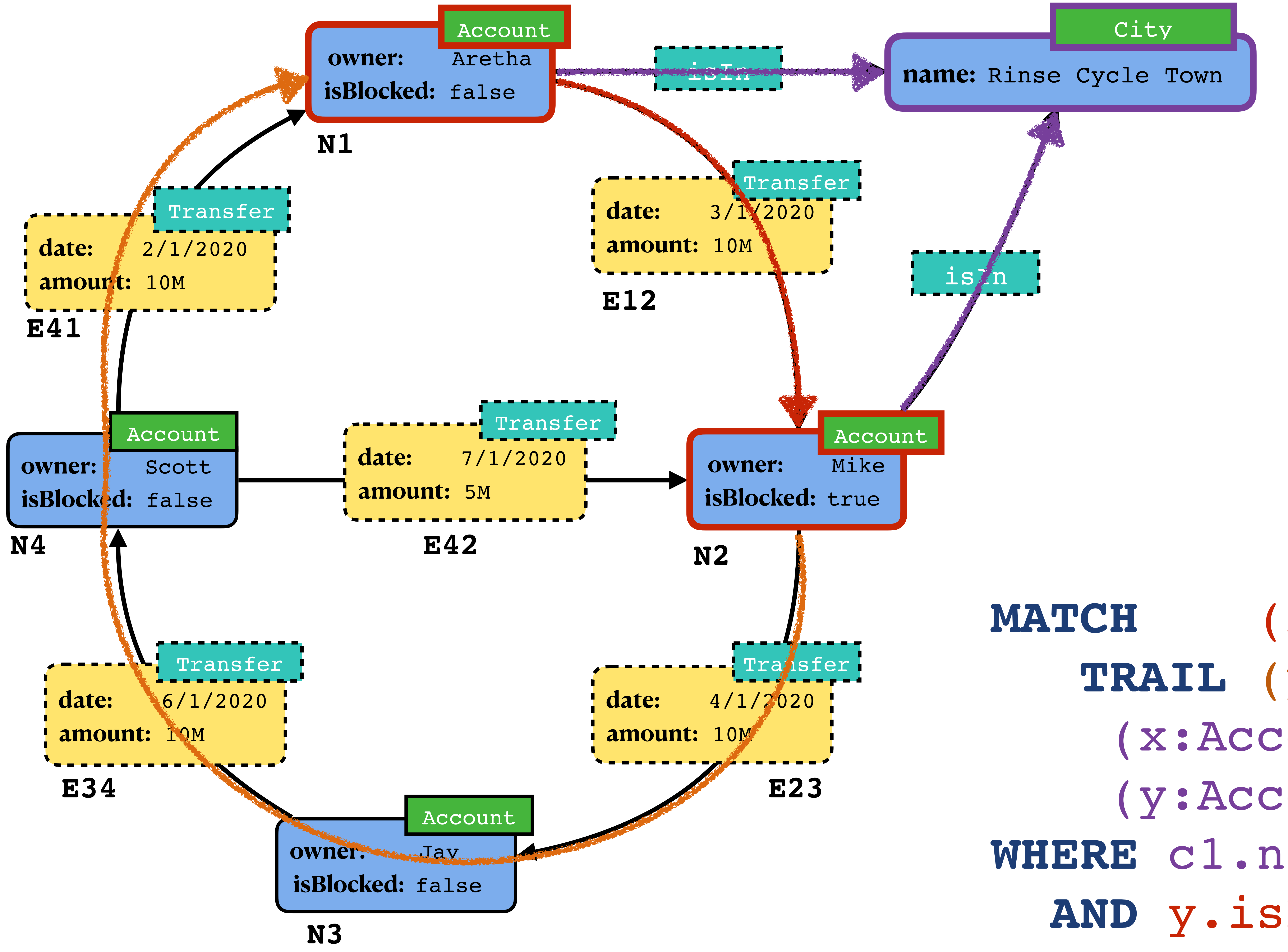

```
MATCH    (x) -[:Transfer]->  (y),
   TRAIL (y) -[:Transfer]->+ (x),
    (x:Account)-[:isIn]->(c1:City),
    (y:Account)-[:isIn]->(c2:City)
WHERE c1.name = c2.name
  AND y.isBlocked = 'true'
```

# 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'
```

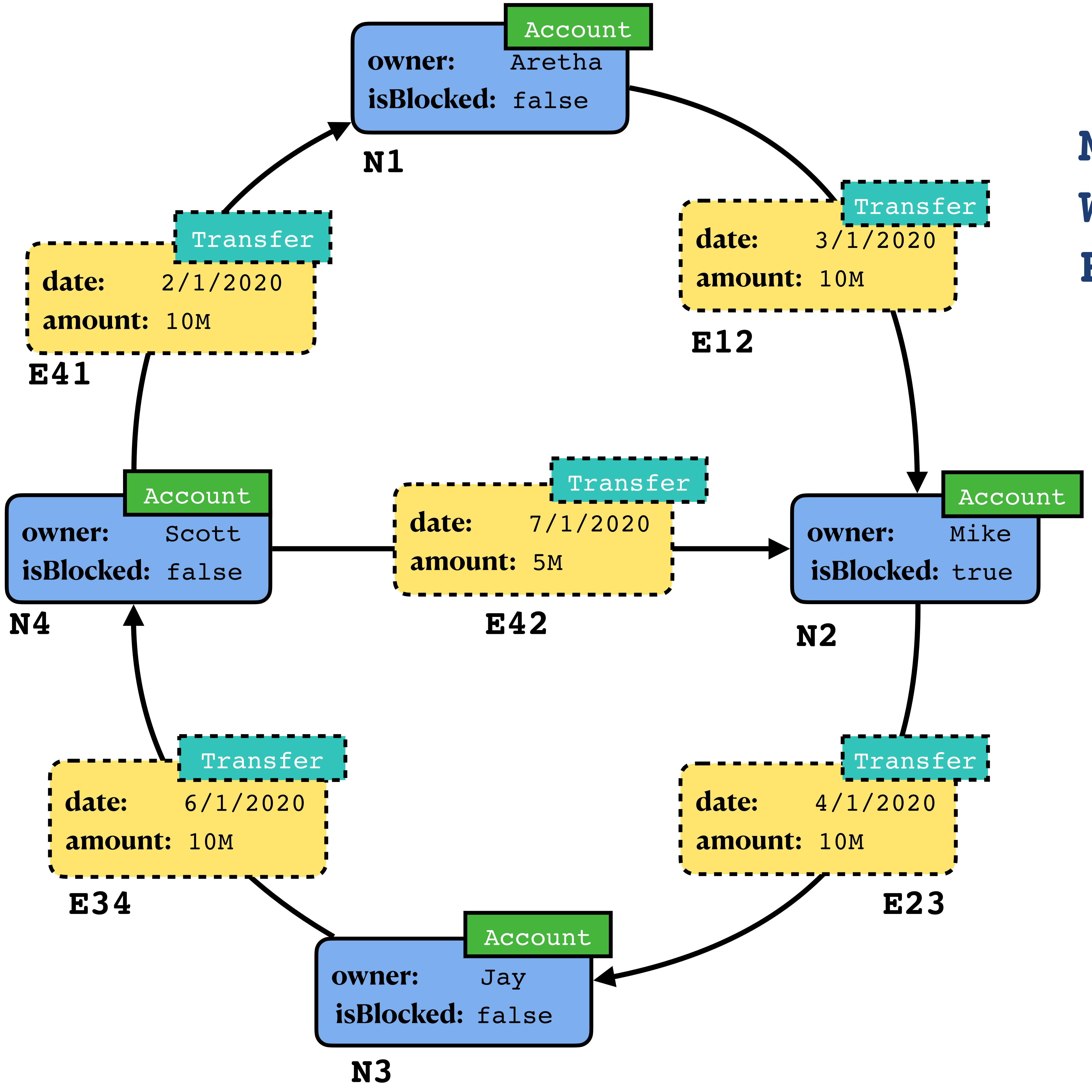
# Joins in Patterns




```
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



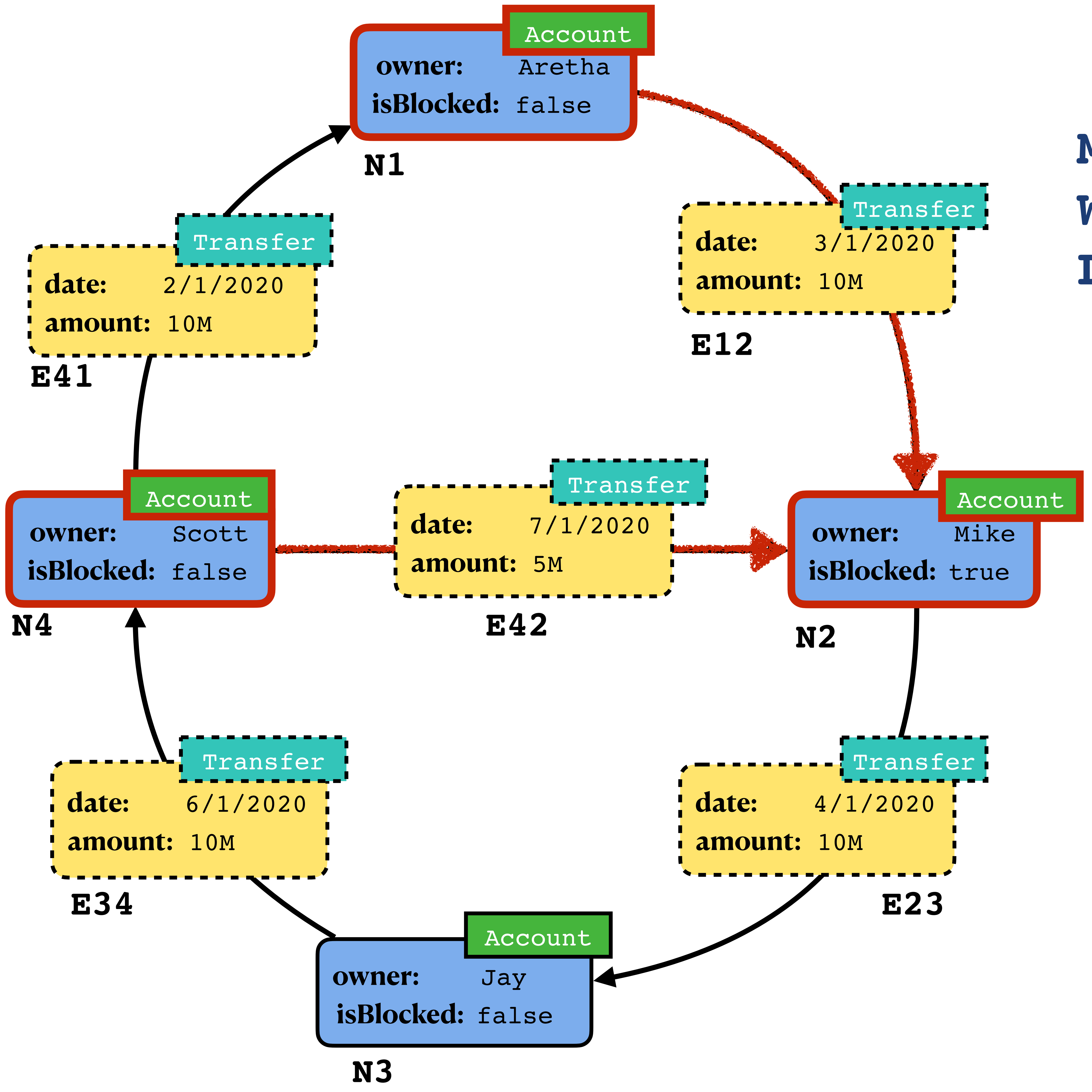

```
MATCH  (x)-[:Transfer]->(y)<-[:Transfer]-(z)
WHERE  y.isBlocked = 'true'
RETURN x.owner AS src1,
       y.owner AS tgt,
       z.owner AS src2
```

| 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 |

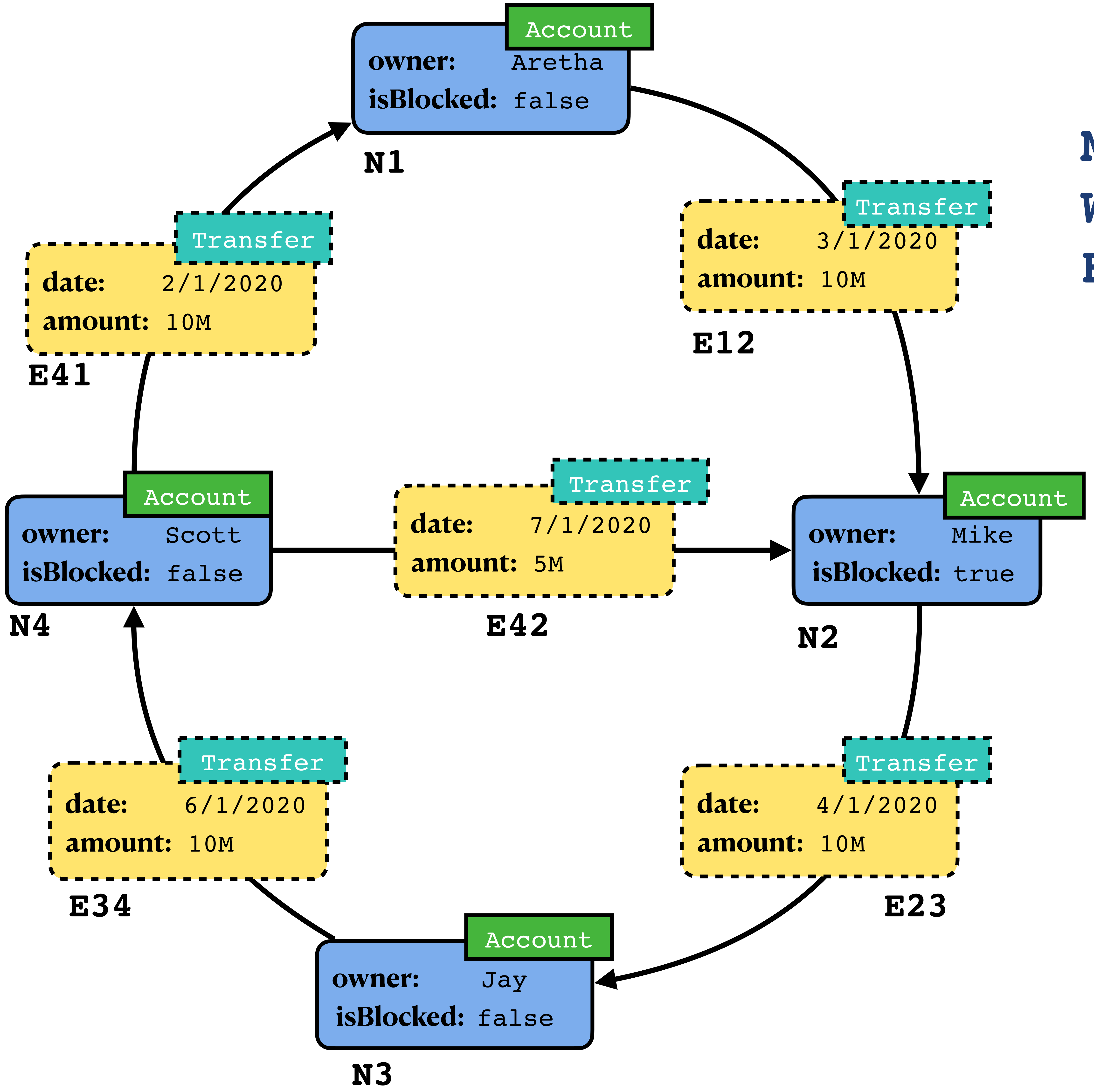
# Let




```
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



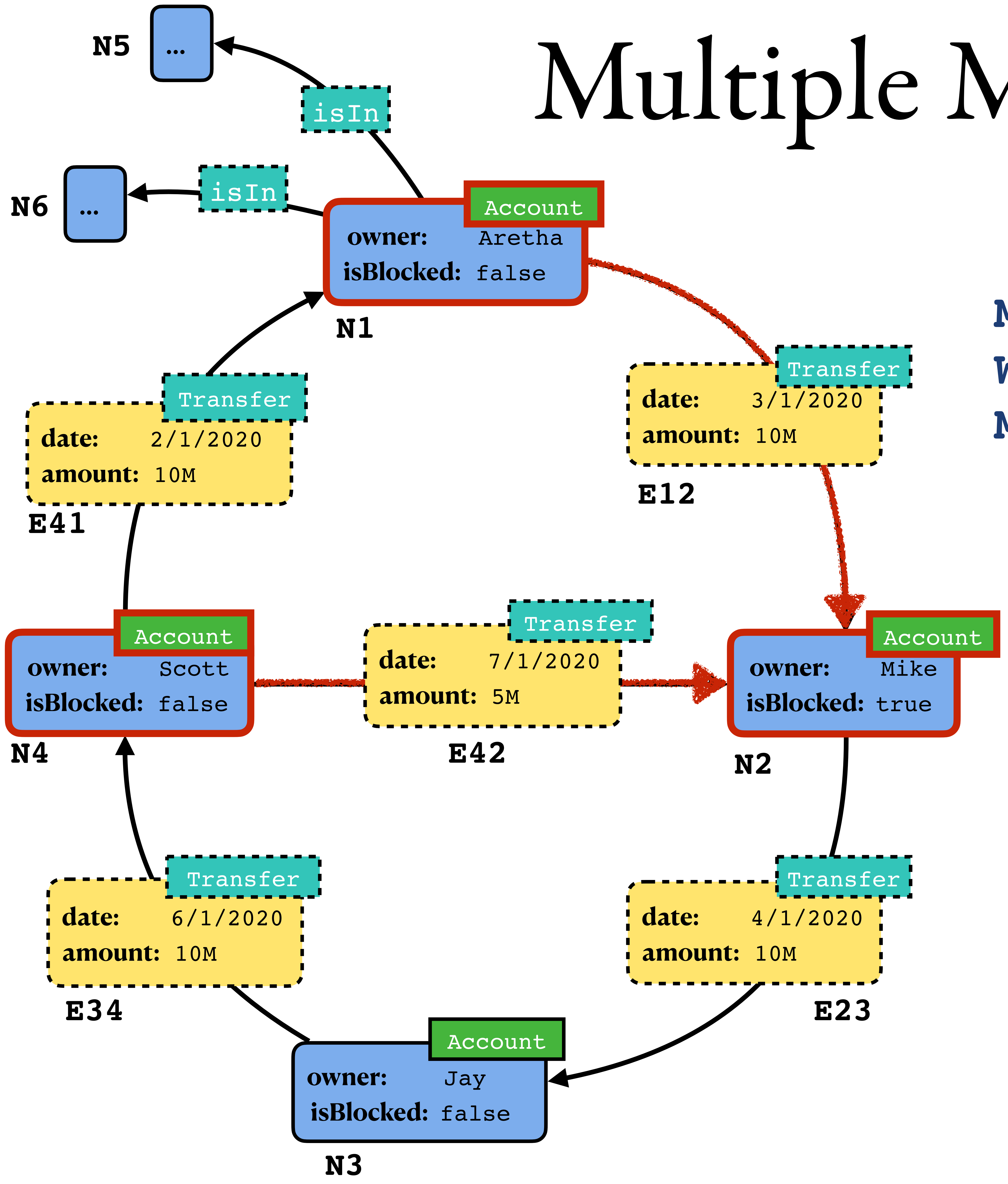

```
MATCH (x)-[:Transfer]->(y)<-[:Transfer]-(z)
WHERE  y.isBlocked = 'true'
FILTER NOT (x = y)
```

| 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 |

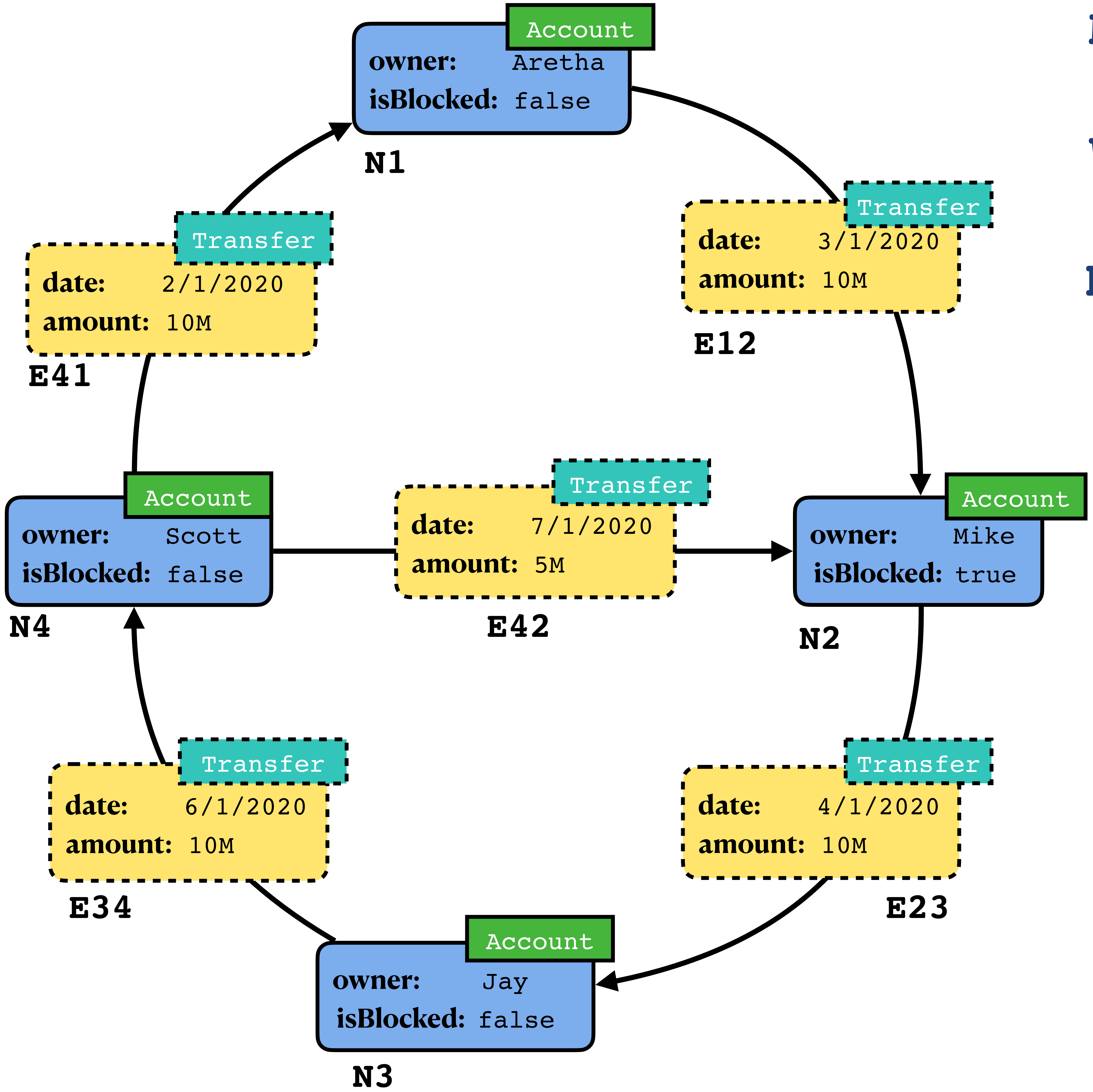
⋈

| 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:Account)-[y:Transfer]->{2,4}(v:Account)
WHERE u.isBlocked = 'false'
  AND v.isBlocked = '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**

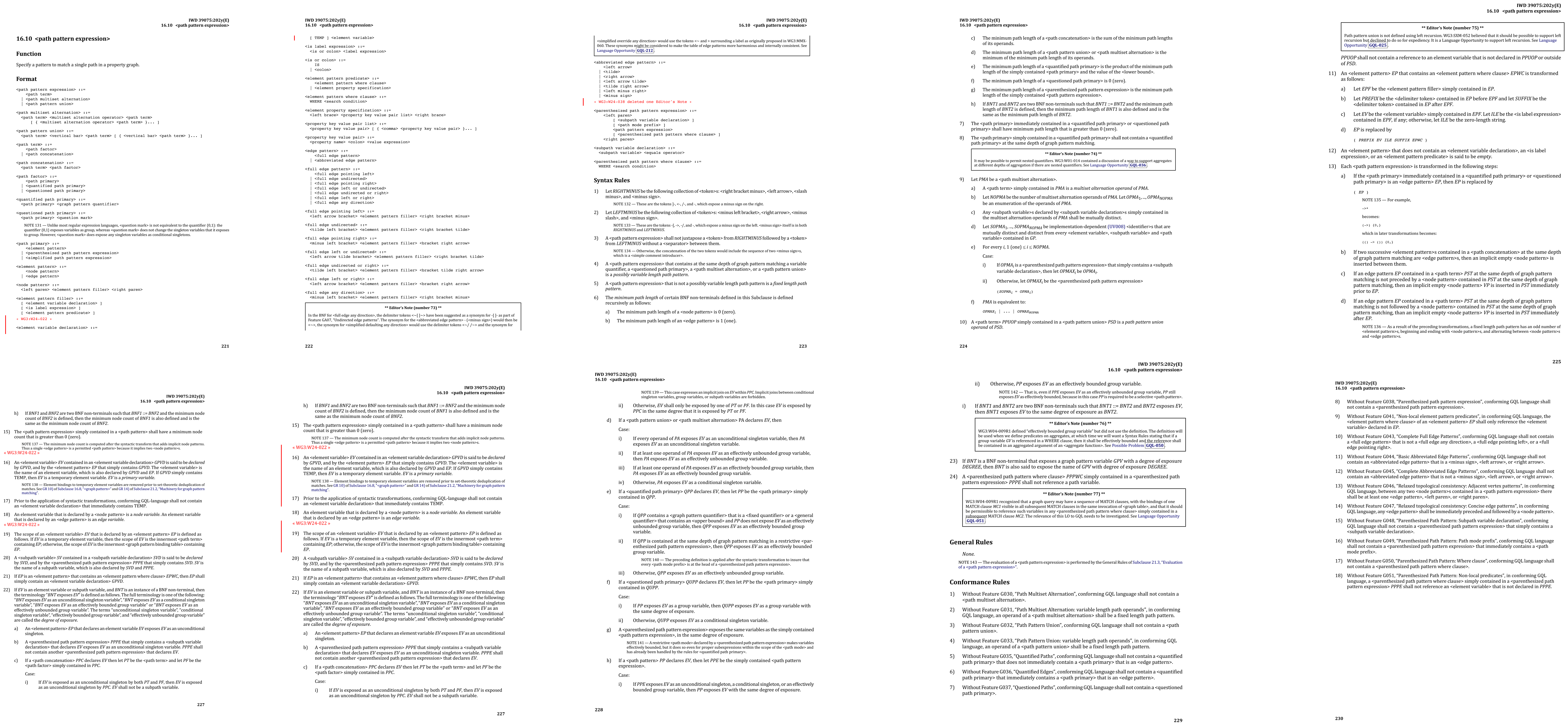
If $Q_1$ and $Q_2$ are GQL queries, then so are

- $Q_1$ `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:



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 |
| Edge pattern | $\alpha := \xrightarrow{x:\ell} \mid \xleftarrow{x:\ell} \mid \overset{x:\ell}{\text{----}}$ | $\ell$-labeled edge directed left/right/any-directed, assign to a variable |

Both $x$ and $\ell$ are optional

**Patterns**

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

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

$$\frac{}{(x) \vdash x : \text{Node}} \qquad \frac{}{(x : \ell) \vdash x : \text{Node}} \qquad \frac{}{\overset{x}{\Longleftrightarrow} \vdash x : \text{Edge}} \qquad \frac{}{\overset{x:\ell}{\Longleftrightarrow} \vdash x : \text{Edge}} \qquad \frac{x \notin \text{var}(\pi)}{x = \rho\ \pi \vdash x : \text{Path}}$$

$$\frac{\pi \vdash z : \tau}{\pi^{n..m} \vdash z : \text{Group}(\tau)} \qquad \frac{\pi \vdash z : \tau}{\rho\pi \vdash z : \tau} \qquad \frac{\pi \vdash z : \tau \quad z \neq x}{x = \rho\pi \vdash z : \tau}$$

$$\frac{\pi \vdash x : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{\pi \vdash x.a = c : \text{Bool}} \qquad \frac{\pi \vdash x : \tau \quad \pi \vdash y : \tau' \quad \tau, \tau' \in \{\text{Node}, \text{Edge}\}}{\pi \vdash x.a = y.b : \text{Bool}}$$

$$\frac{\pi \vdash \theta : \text{Bool} \quad \pi \vdash \theta' : \text{Bool}}{\pi \vdash \theta \wedge \theta' : \text{Bool}} \qquad \frac{\pi \vdash \theta : \text{Bool} \quad \pi \vdash \theta' : \text{Bool}}{\pi \vdash \theta \vee \theta' : \text{Bool}} \qquad \frac{\pi \vdash \theta : \text{Bool}}{\pi \vdash \neg\theta : \text{Bool}} \qquad \frac{\pi \vdash \theta : \text{Bool} \quad \pi \vdash z : \tau}{\pi_{\langle\theta\rangle} \vdash z : \tau}$$

$$\frac{\pi_1 \vdash z : \tau \quad \pi_2 \vdash z : \tau}{\pi_1 + \pi_2 \vdash z : \tau} \qquad \frac{\pi_1 \vdash z : \tau \quad \pi_2 \vdash z : \text{Maybe}(\tau)}{\pi_1 + \pi_2 \vdash z : \text{Maybe}(\tau)} \qquad \frac{\pi_1 \vdash z : \text{Maybe}(\tau) \quad \pi_2 \vdash z : \tau}{\pi_1 + \pi_2 \vdash z : \text{Maybe}(\tau)}$$

$$\frac{\pi_1 \vdash z : \tau \quad z \notin \text{var}(\pi_2)}{\pi_1 + \pi_2 \vdash z : \tau?} \qquad \frac{\pi_2 \vdash z : \tau \quad z \notin \text{var}(\pi_1)}{\pi_1 + \pi_2 \vdash z : \tau?}$$

$$\frac{\pi_1 \vdash z : \tau \quad \pi_2 \vdash z : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{\pi_1\ \pi_2 \vdash z : \tau} \qquad \frac{\pi_1 \vdash z : \tau \quad z \notin \text{var}(\pi_2)}{\pi_1\ \pi_2 \vdash z : \tau} \qquad \frac{\pi_2 \vdash z : \tau \quad z \notin \text{var}(\pi_1)}{\pi_1\ \pi_2 \vdash z : \tau}$$

$$\frac{Q_1 \vdash z : \tau \quad Q_2 \vdash z : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{Q_1, Q_2 \vdash z : \tau} \qquad \frac{Q_1 \vdash z : \tau \quad z \notin \text{var}(Q_2)}{Q_1, Q_2 \vdash z : \tau} \qquad \frac{Q_2 \vdash z : \tau \quad z \notin \text{var}(Q_1)}{Q_1, Q_2 \vdash z : \tau}$$

# 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

**Amélie Gheerbrant**
IRIF, Université Paris Cité, CNRS, Paris, France

**Paolo Guagliardo**
School of Informatics, University of Edinburgh, UK

**Leonid Libkin**
University of Edinburgh, UK
RelationalAI, France
ENS, PSL University, France

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

**Wim Martens**
Universität Bayreuth, Germany

**Filip Murlak**
University of Warsaw, Poland

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

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

**Domagoj Vrgoč**
University of Zagreb, Coratia
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 $x \in$ Vars, $\ell \in \mathcal{L}$, $0 \le n \le m \in \mathbb{N}$:

| | | |
|---|---|---|
| (descriptor) | $\delta$ := $x$ `:`$\ell$ `WHERE` $\theta$ | $x$, `:`$\ell$, and `WHERE` $\theta$ are optional |
| (path pattern) | $\pi$ := `(`$\delta$`)` | (node pattern) |
| | \| `-[`$\delta$`]->` \| `<-[`$\delta$`]-` \| `-[`$\delta$`]-` | (edge pattern) |
| | \| $\pi\pi$ | (concatenation) |
| | \| $\pi$`\|`$\pi$ | (union) |
| | \| $\pi$ `WHERE` $\theta$ | (conditioning) |
| | \| $\pi$`{`$n$`,`$m$`}` | (bounded repetition) |
| | \| $\pi$`{`$n$`,}` | (unbounded repetition) |

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

| | |
|---|---|
| (expression) | $\chi$ := $x$ \| $x.a$ \| $c$ |
| (condition) | $\theta$ := $\chi$ `=` $\chi$ \| $\chi$ `<` $\chi$ \| $\chi$ `IS NULL` |
| | \| $x$ : $\ell$ \| `EXISTS {`Q`}` |
| | \| $\theta$ `OR` $\theta$ \| $\theta$ `AND` $\theta$ \| `NOT` $\theta$ |

**GRAPH PATTERN** For $x \in$ Vars:

| | |
|---|---|
| (path mode) | $\mu$ := (`ALL` \| `ANY`) [`SHORTEST`] [`TRAIL` \| `ACYCLIC`] |
| (graph pattern) | $\Pi$ := $\mu$ [$x$ `=`] $\pi$ \| $\Pi$, $\Pi$ |

**CLAUSE and QUERY** For $k \ge 0$, $\ell \ge 1$, and $x, y, x_1, \ldots, x_k \in$ Vars, and $G \in \mathbb{G}$:

| | |
|---|---|
| (clause) | C := `MATCH` $\Pi$ |
| | \| `LET` $x$ `=` $\chi$ |
| | \| `FOR` $x$ `IN` $y$ |
| | \| `FILTER` $\theta$ |
| (linear query) | L := `USE` $G$ L |
| | \| C L |
| | \| `RETURN` $\chi_1$ `AS` $x_1, \ldots, \chi_k$ `AS` $x_k$ |
| (query) | Q := L |
| | \| `USE` $G$ {$Q_1$ `THEN` $Q_2$ ⋯ `THEN` $Q_\ell$} |
| | \| Q `INTERSECT` Q \| Q `UNION` Q \| Q `EXCEPT` Q |

S
Y
N
T
A
X

# Semantics



$$[\![\texttt{-[]->}]\!]_G = \{\, (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), ()) \mid e \in E_d^G \,\}$$
$$[\![\texttt{-[}x\texttt{]->}]\!]_G = \{\, (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), (x \mapsto e)) \mid e \in E_d^G \,\}$$
$$[\![\texttt{-[:}\ell\texttt{]->}]\!]_G = \left\{\, (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), ()) \;\middle|\; e \in E_d^G, \ell \in \mathsf{lab}^G(e) \,\right\}$$

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.

$$[\![\texttt{<-[]-}]\!]_G = \{\, (\mathsf{path}(\mathsf{tgt}(e), e, \mathsf{src}(e)), ()) \mid e \in E_d^G \,\}$$
$$[\![\texttt{\~{}[]\~{}}]\!]_G = \left\{ (\mathsf{path}(u_1, e, u_2), ()), (\mathsf{path}(u_2, e, u_1), ()) \;\middle|\; \begin{array}{l} e \in E_u^G \\ \{u_1, u_2\} = \mathsf{endpoints}^G(e) \end{array} \right\}$$

**Semantics of Concatenation, Union, and Conditioning**

$$[\![\pi_1\,\pi_2]\!]_G \left\{ (p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \;\middle|\; \begin{array}{l} (p_i, \mu_i) \in [\![\pi_i]\!]_G \text{ for } i = 1, 2 \\ p_1 \text{ and } p_2 \text{ concatenate} \\ \mu_1 \sim \mu_2 \end{array} \right\}$$

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) (-[:Transfer]->()-[:Transfer]->(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.

$$[\![\pi_1 \mid \pi_2]\!]_G = \{\, (p, \mu \cup \mu') \mid (p, \mu) \in [\![\pi_1]\!]_G \cup [\![\pi_2]\!]_G \,\}$$

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

$$[\![\pi \texttt{ WHERE } \theta]\!]_G = \{\, (p, \mu) \in [\![\pi]\!]_G \mid [\![\theta]\!]_G^\mu = \mathsf{true} \,\}$$

**Semantics of Repetition**

$$[\![\pi\texttt{\{}n, m\texttt{\}}]\!]_G = \bigcup_{i=n}^{m} [\![\pi]\!]_G^i$$
$$[\![\pi\texttt{\{}n,\texttt{\}}]\!]_G = \bigcup_{i=n}^{\infty} [\![\pi]\!]_G^i$$

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

$$[\![\pi]\!]_G^0 = \{\, (\mathsf{path}(u), \mu) \mid u \text{ is a node in } G \,\}$$

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

$$\forall i > 0 \quad [\![\pi]\!]_G^i = \left\{ (p_1 \cdot \ldots \cdot p_i, \mu') \;\middle|\; \begin{array}{l} (p_1, \mu_1), \ldots, (p_n, \mu_i) \in [\![\pi]\!]_G \\ p_1, \ldots, p_i \text{ concatenate} \end{array} \right\}$$

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



▶ Remark 10. Since $\pi\texttt{\{}n,\texttt{\}}$ 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 $(p, \mu) \in [\![\pi\texttt{\{}n, m\texttt{\}}]\!]_G$. 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 `(a){0,}` which is not well-formed (the minimal path length of `()` is 0). For every $i$, the set $[\![\texttt{(a)}]\!]_G^i$ contains $(\mathsf{path}(u), \mu_i)$ where $\mu_i = (a \mapsto \mathsf{list}(\underbrace{u, \ldots, u}_{i \text{ times}}))$; hence the union in the definition of $[\![\pi\texttt{\{}n,\texttt{\}}]\!]_G$ 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 (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.

$$[\![x \texttt{=} \pi]\!]_G = \{ (p, \mu \cup \{x \mapsto p\}) \mid (p, \mu) \in [\![\pi]\!]_G \}$$

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

$$[\![\texttt{TRAIL } \pi]\!]_G = \{\, (p, \mu) \in [\![\pi]\!]_G \mid \text{no edge occurs more than once in } p \,\}$$
$$[\![\texttt{ACYCLIC } \pi]\!]_G = \{\, (p, \mu) \in [\![\pi]\!]_G \mid \text{no node occurs more than once in } p \,\}$$
$$[\![\texttt{SHORTEST } \tilde{\pi}]\!]_G = \left\{ (p, \mu) \in [\![\tilde{\pi}]\!]_G \;\middle|\; \mathsf{len}(p) = \min \left\{ \mathsf{len}(p') \;\middle|\; \begin{array}{l} (p', \mu') \in [\![\tilde{\pi}]\!]_G \\ \mathsf{src}(p') = \mathsf{src}(p) \\ \mathsf{tgt}(p') = \mathsf{tgt}(p) \end{array} \right\} \right\}$$
$$[\![\texttt{ALL } \tilde{\pi}]\!]_G = [\![\tilde{\pi}]\!]_G$$
$$[\![\texttt{ANY } \tilde{\pi}]\!]_G = \bigcup_{(s,t) \in X} \{ \mathsf{any}(\{\, (p, \mu) \mid (p, \mu) \in [\![\tilde{\pi}]\!]_G, \mathsf{endpoints}(p) = (s, t) \,\}) \}$$

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

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

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

## 4.4 Semantics of Conditions and Expressions

The semantics $[\![\chi]\!]_G^\mu$ 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.

$$[\![c]\!]_G^\mu = c \qquad \text{for } c \in \mathsf{Const}$$
$$[\![x]\!]_G^\mu = \mu(x) \qquad \text{for } x \in \mathrm{Dom}(\mu)$$
$$[\![x.a]\!]_G^\mu = \begin{cases} \mathsf{prop}^G(\mu(x), a) & \text{if } (\mu(x), a) \in \mathrm{Dom}(\mathsf{prop}^G) \\ \mathsf{null} & \text{else if } \mu(x) \in (\mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u}) \end{cases} \qquad \text{for } x \in \mathrm{Dom}(\mu), a \in \mathcal{K}$$

# Semantics



▶ Remark 11. Recall that different graphs may share nodes and edges. Hence the condition $(\mu(x), a) \in \mathrm{Dom}(\mathsf{prop}^G)$, 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 $\llbracket\theta\rrbracket^\mu_G$ of a condition $\theta$ is an element in $\{\mathsf{true}, \mathsf{false}, \mathsf{null}\}$ that is evaluated with respect to a binding $\mu$ and a graph $G$, and is defined as follows:

$$\llbracket\chi_1 \texttt{ = } \chi_2\rrbracket^\mu_G = \begin{cases} \mathsf{null} & \text{if } \llbracket\chi_1\rrbracket^\mu_G = \mathsf{null} \text{ or } \llbracket\chi_2\rrbracket^\mu_G = \mathsf{null} \\ \mathsf{true} & \text{if } \llbracket\chi_1\rrbracket^\mu_G = \llbracket\chi_2\rrbracket^\mu_G \neq \mathsf{null} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\llbracket\chi_1 \texttt{ < } \chi_2\rrbracket^\mu_G = \begin{cases} \mathsf{null} & \text{if } \llbracket\chi_1\rrbracket^\mu_G = \mathsf{null} \text{ or } \llbracket\chi_2\rrbracket^\mu_G = \mathsf{null} \\ \mathsf{true} & \text{else if } \llbracket\chi_1\rrbracket^\mu_G < \llbracket\chi_2\rrbracket^\mu_G \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\llbracket\chi \texttt{ IS NULL}\rrbracket^\mu_G = \begin{cases} \mathsf{true} & \text{if } \llbracket\chi\rrbracket^\mu_G = \mathsf{null} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\llbracket\chi\texttt{:}\ell\rrbracket^\mu_G = \begin{cases} \mathsf{true} & \text{if } \llbracket\chi\rrbracket^\mu_G \in N^G \cup E^G_u \cup E^G_d \text{ and } \ell \in \mathsf{lab}^G(\llbracket\chi\rrbracket^\mu_G) \\ \mathsf{false} & \text{else if } \llbracket\chi\rrbracket^\mu_G \in \mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u} \end{cases}$$

$$\llbracket\theta_1 \texttt{ AND } \theta_2\rrbracket^\mu_G = \llbracket\theta_1\rrbracket^\mu_G \wedge \llbracket\theta_2\rrbracket^\mu_G \quad ^{(*)}$$

$$\llbracket\theta_1 \texttt{ OR } \theta_2\rrbracket^\mu_G = \llbracket\theta_1\rrbracket^\mu_G \vee \llbracket\theta_2\rrbracket^\mu_G \quad ^{(*)}$$

$$\llbracket\texttt{NOT } \theta\rrbracket^\mu_G = \neg\,\llbracket\theta\rrbracket^\mu_G \quad ^{(*)}$$

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

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

## 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* $\mathrm{Dom}(T)$.

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 $\llbracket\mathsf{C}\rrbracket_G$ of a clause $\mathsf{C}$ 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 $G$.

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

Note that if $\Pi$ uses a variable that already occurs in $\mathrm{Dom}(T)$, 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.



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

$$\llbracket\texttt{LET } x \texttt{ = } \chi\rrbracket_G(T) = \bigcup_{\mu\in T}\{\mu \bowtie (x \mapsto \llbracket\chi\rrbracket^\mu_G)\}$$

$$\llbracket\texttt{FILTER } \theta\rrbracket_G(T) = \bigcup_{\mu\in T}\{\mu \mid \llbracket\theta\rrbracket^\mu_G = \mathsf{true}\}\ .$$

If $x \notin \mathrm{Dom}(T)$ and, for every $\mu \in T$, $\mu(y)$ is a list or null,[3] then

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

### Semantics of Linear Queries

$$\llbracket\texttt{USE } G'\ \mathsf{L}\rrbracket_G(T) = \llbracket\mathsf{L}\rrbracket_{G'}(T)$$

$$\llbracket C\ \mathsf{L}\rrbracket_G(T) = \llbracket\mathsf{L}\rrbracket_G\big(\llbracket C\rrbracket_G(T)\big)$$

$$\llbracket\texttt{RETURN } \chi_1 \texttt{ AS } x_1, \ldots, \chi_\ell \texttt{ AS } x_\ell\rrbracket_G(T) = \bigcup_{\mu\in T}\Big\{(x_1 \mapsto \llbracket\chi_1\rrbracket^\mu_G, \ldots, x_\ell \mapsto \llbracket\chi_\ell\rrbracket^G_\mu)\Big\}$$

### Semantics of Queries

The *output of a query* $\mathsf{Q}$ is defined as

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

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.

$$\llbracket\texttt{USE } G' \texttt{ \{}\mathsf{Q}_1 \texttt{ THEN } \mathsf{Q}_2 \cdots \texttt{ THEN } \mathsf{Q}_k\texttt{\}}\rrbracket_G(T) = \llbracket\mathsf{Q}_k\rrbracket_{G'} \circ \cdots \circ \llbracket\mathsf{Q}_1\rrbracket_{G'}(T)$$

If $\mathrm{Dom}(\llbracket\mathsf{Q}_1\rrbracket_G(T)) = \mathrm{Dom}(\llbracket\mathsf{Q}_2\rrbracket_G(T))$, then we let

$$\llbracket\mathsf{Q}_1 \texttt{ INTERSECT } \mathsf{Q}_2\rrbracket_G(T) = \llbracket\mathsf{Q}_1\rrbracket_G(T) \cap \llbracket\mathsf{Q}_2\rrbracket_G(T)$$

$$\llbracket\mathsf{Q}_1 \texttt{ UNION } \mathsf{Q}_2\rrbracket_G(T) = \llbracket\mathsf{Q}_1\rrbracket_G(T) \cup \llbracket\mathsf{Q}_2\rrbracket_G(T)$$

$$\llbracket\mathsf{Q}_1 \texttt{ EXCEPT } \mathsf{Q}_2\rrbracket_G(T) = \llbracket\mathsf{Q}_1\rrbracket_G(T) \setminus \llbracket\mathsf{Q}_2\rrbracket_G(T)$$

## 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.

[3] Note that null is treated just as 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

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

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

$$\mathrm{FV}\big((x)\big) = \mathrm{FV}\big(\rightarrow^{x}\big) = \mathrm{FV}\big(\leftarrow^{x}\big) = \{x\}$$

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

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

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

$$\mathrm{FV}(\pi^{n..m}) = \varnothing$$

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

Pattern with output: $\pi_\Omega$

# Semantics: one simple definition, just what you expect

$$
\begin{aligned}
[\![(x)]\!]_G &:= \{(\mathsf{path}(n), \{x \mapsto n\}) \mid n \in \mathsf{N}\} \\
\left[\!\!\left[\xrightarrow{x}\right]\!\!\right]_G &:= \{(\mathsf{path}(n_1, e, n_2), \{x \mapsto e\}) \mid e \in \mathsf{E},\ \mathsf{src}(e) = n_1,\ \mathsf{tgt}(e) = n_2\} \\
\left[\!\!\left[\xleftarrow{x}\right]\!\!\right]_G &:= \{(\mathsf{path}(n_2, e, n_1), \{x \mapsto e\}) \mid e \in \mathsf{E},\ \mathsf{src}(e) = n_1,\ \mathsf{tgt}(e) = n_2\} \\
[\![\psi_1 + \psi_2]\!]_G &:= [\![\psi_1]\!]_G \cup [\![\psi_2]\!]_G \\
[\![\psi_1\, \psi_2]\!]_G &:= \left\{(p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \,\middle|\, (p_1, \mu_1) \in [\![\psi_1]\!]_G,\ (p_2, \mu_2) \in [\![\psi_2]\!]_G,\ \mu_1 \sim \mu_2,\ p_1 \odot p_2\right\} \\
[\![\psi\langle\theta\rangle]\!]_G &:= \left\{(p, \mu) \in [\![\pi]\!]_G \ \mid\ \mu \models \theta\right\} \\
[\![\psi^{n..m}]\!]_G &:= \bigcup_{i=n}^{m} [\![\psi]\!]_G^i \ \text{ where}
\end{aligned}
$$

$$
\begin{aligned}
[\![\psi]\!]_G^0 &:= \left\{(\mathsf{path}(n), \mu_\emptyset) \,\middle|\, n \in \mathsf{N}\right\} \\
[\![\psi]\!]_G^n &:= \left\{(p_1 \cdots p_n, \mu_\emptyset) \mid \exists \mu_1, \ldots, \mu_n : (p_i, \mu_i) \in [\![\psi]\!]_G \text{ and } p_i \odot p_{i+1} \text{ for all } i < n\right\},\ n > 0
\end{aligned}
$$

$$
[\![\psi_\Omega]\!]_G := \left\{\mu_\Omega \ \mid\ \exists p : (p, \mu) \in [\![\psi]\!]_G\right\}
$$

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)

$$C \;:=\; \text{db relation} \;\mid\; \pi_A \;\mid\; \sigma_\theta \;\mid\; C\,C \;\mid\; \{Q\}$$ clauses

$$Q \;:=\; C \;\mid\; Q \cup Q \;\mid\; Q \cap Q \;\mid\; Q - Q$$ queries

Semantics

$$[\![\, S \,]\!](R) = R \bowtie S$$

$$[\![\, \pi_A \,]\!](R) = \pi_A(R)$$

$$[\![\, C_1\, C_2 \,]\!](R) = [\![\, C_2 \,]\!]\big([\![\, C_1 \,]\!](R)\big)$$

$$[\![\, \{Q\} \,]\!](R) = R \bowtie [\![\, Q \,]\!](R)$$

# 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) ->* (: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

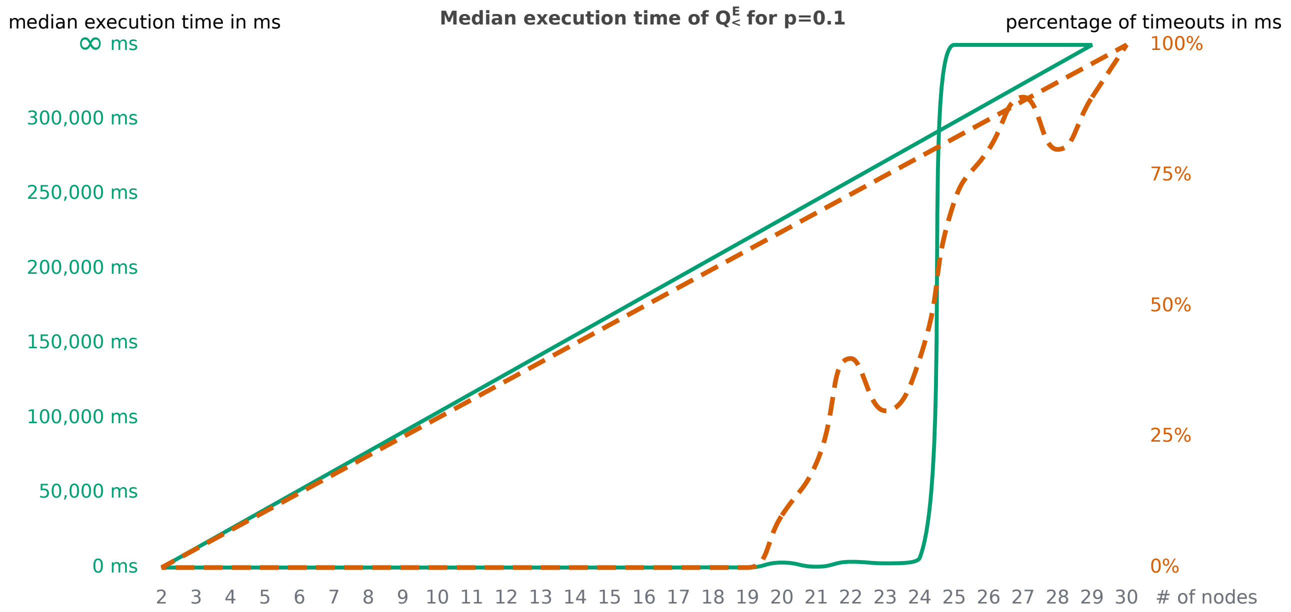
all paths

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

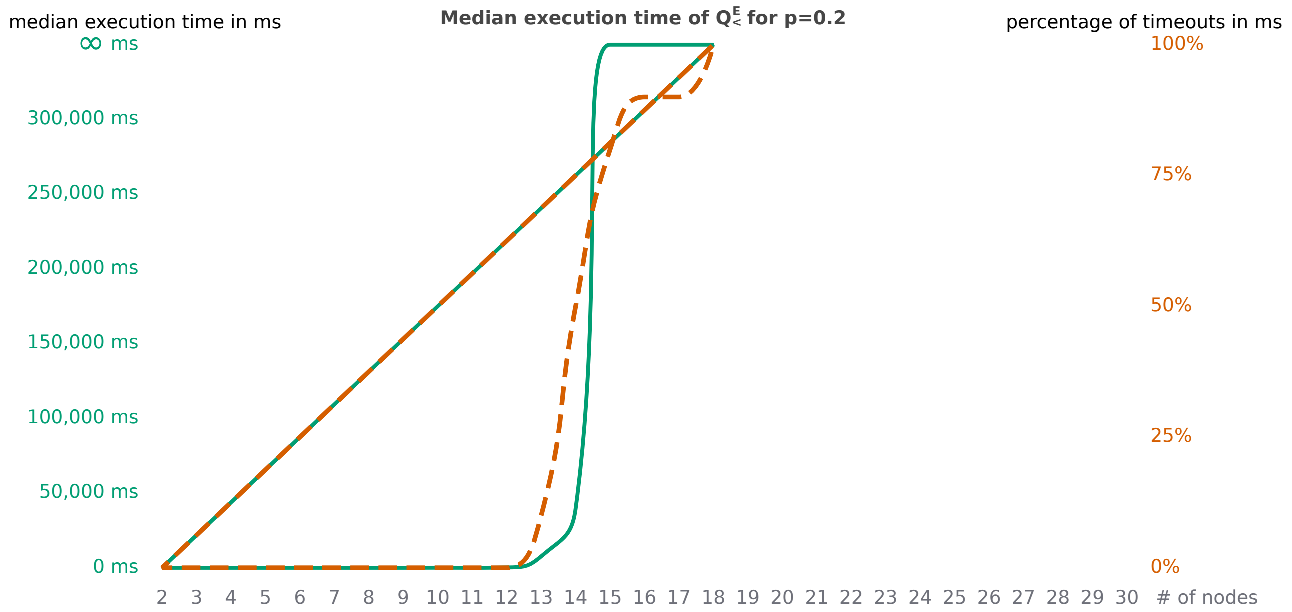
difference

bad paths

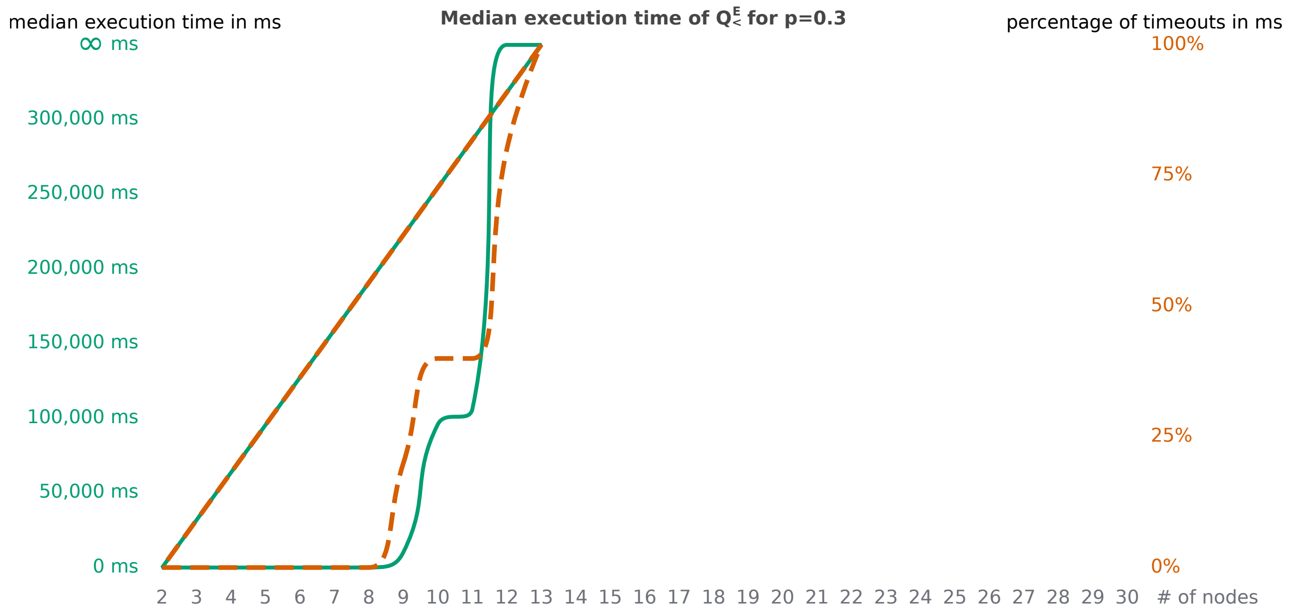
# Does it have a chance to work? No way!




(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?

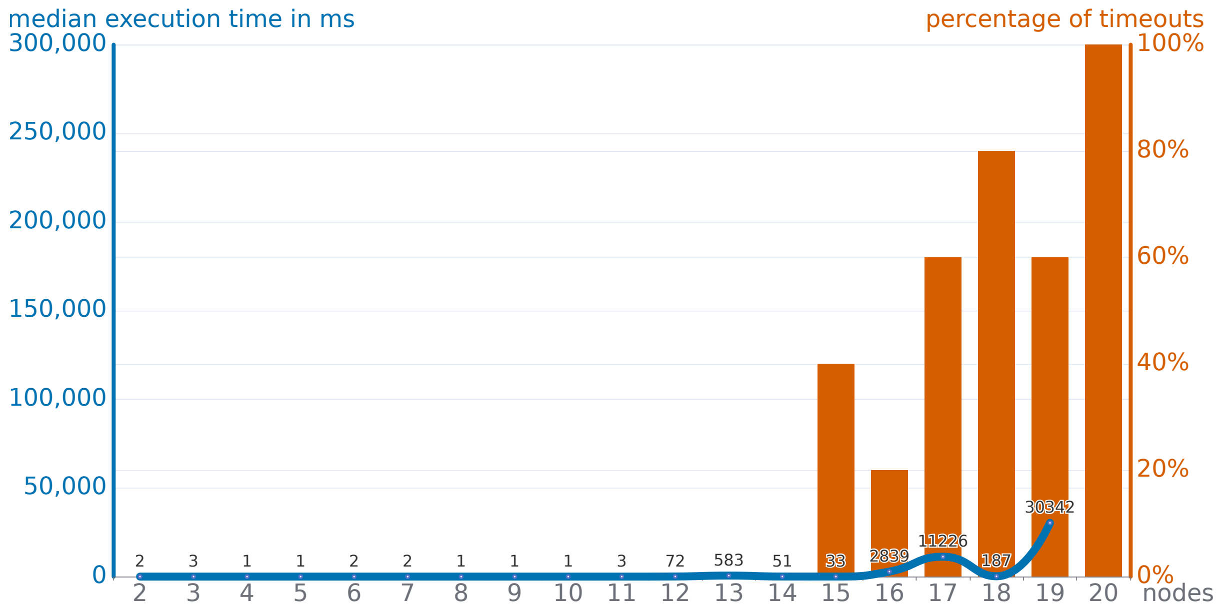
```
MATCH (n)
WITH collect(n.name) AS allNodes
MATCH path=(:Start)-[*]-()
WITH path, allNodes, [y IN nodes(path) | y.name] AS nodesInPath
WHERE all(node in allNodes WHERE node IN nodesInPath)
AND size(allNodes)=size(nodesInPath)
RETURN path LIMIT 1
```

Hamiltonian Path

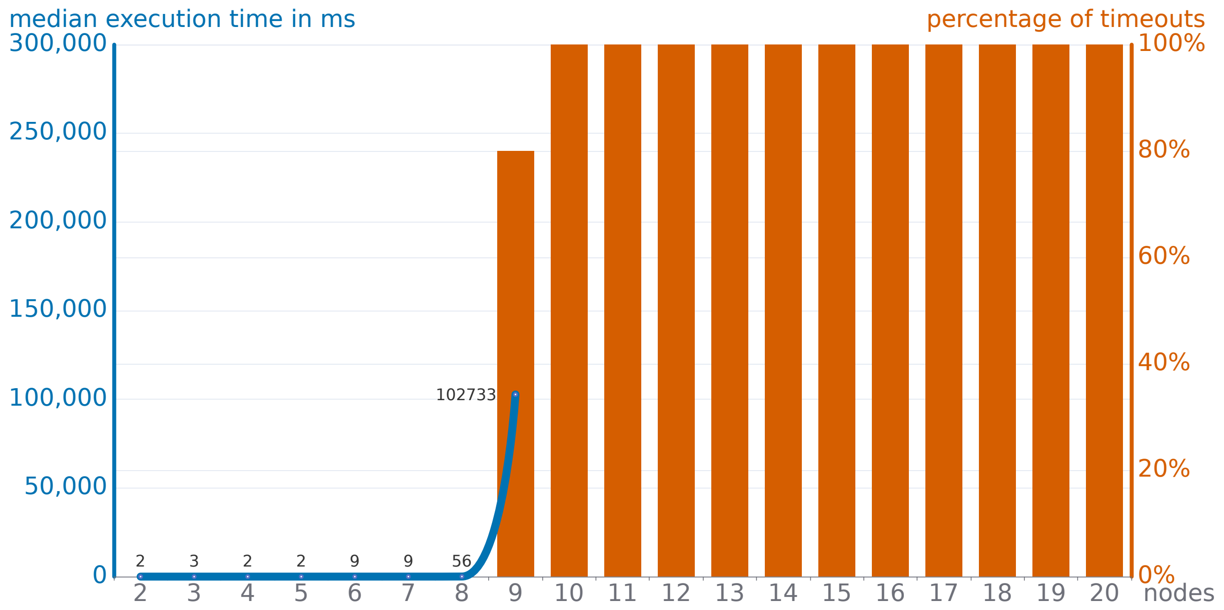
```
MATCH p = allShortestPaths((:Start)-[:Edge*]->(:Finish))
WITH [r IN relationships(p) | r.value] AS values, p
UNWIND values as valSet
WITH sum(valSet) AS sum, p
WHERE sum = $T
RETURN p
```

Subset-Sum

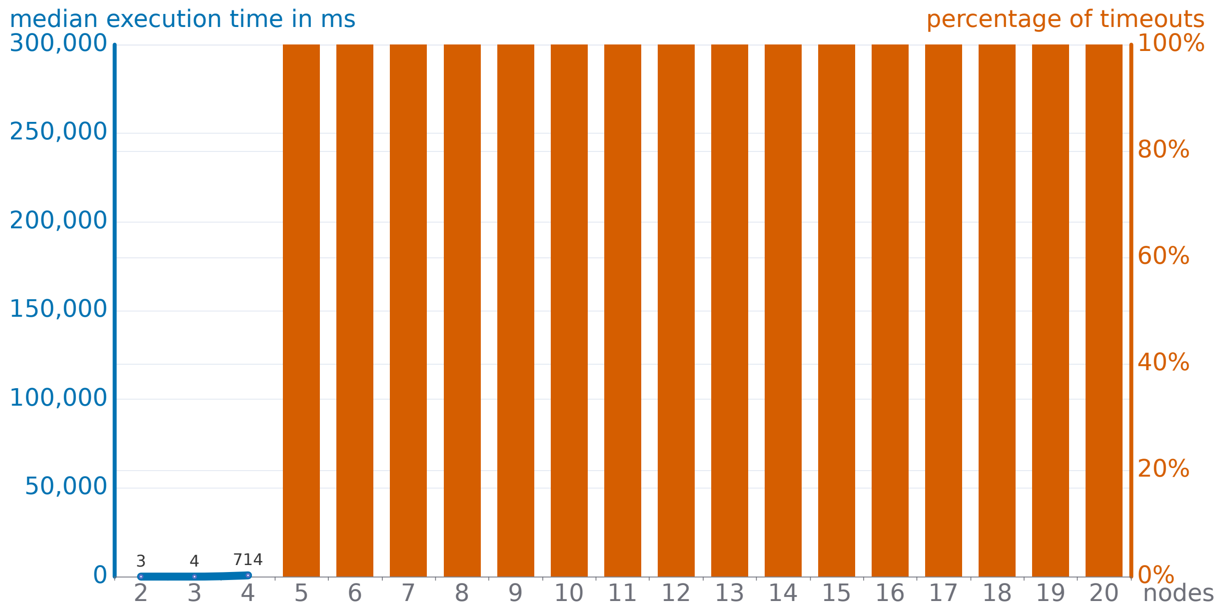
# .... and they don't work (GLR'24)



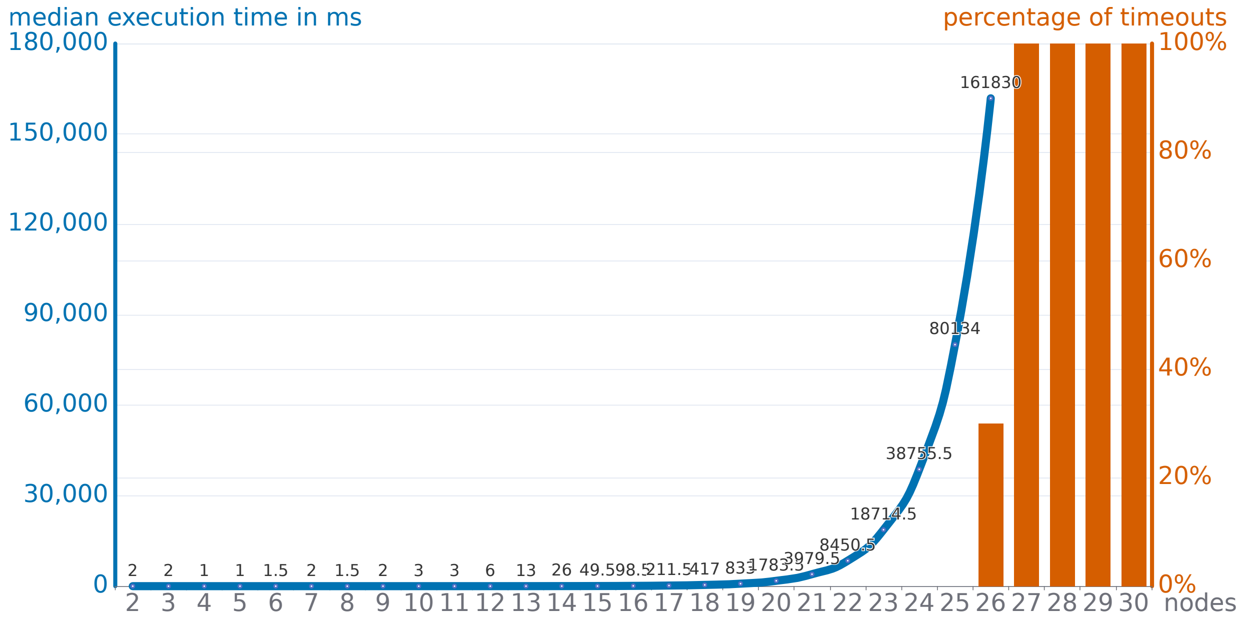

**(a) Median execution time and number of timeouts for** $p = 0.1$



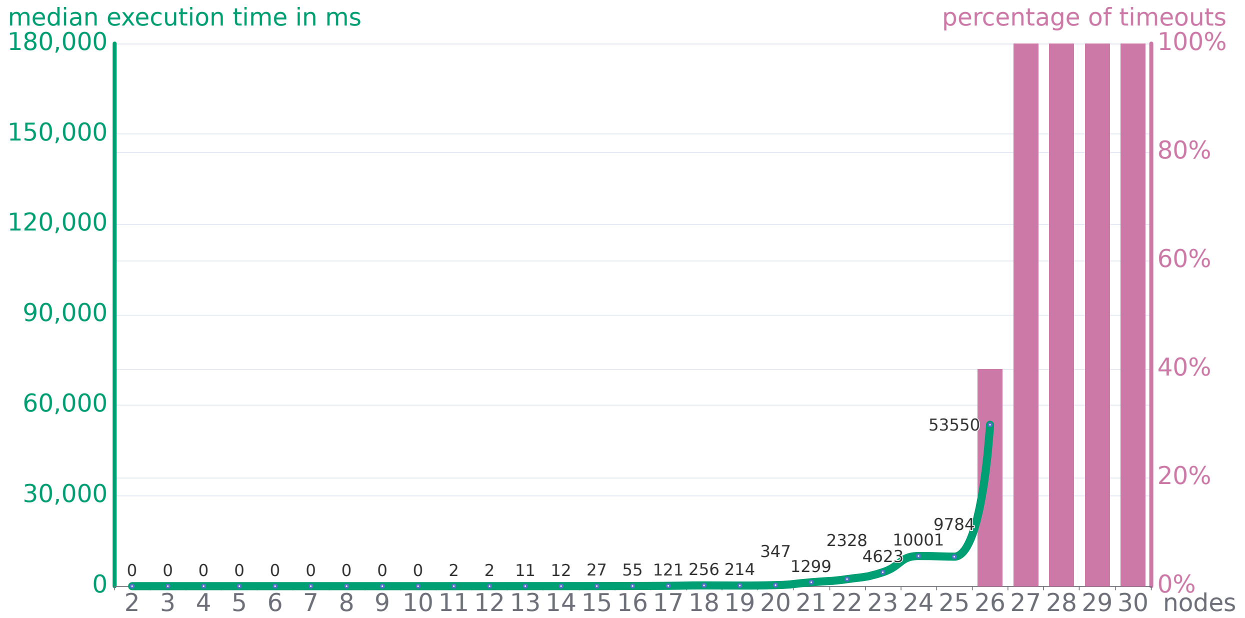

**(b) Median execution time and number of timeouts for** $p = 0.3$



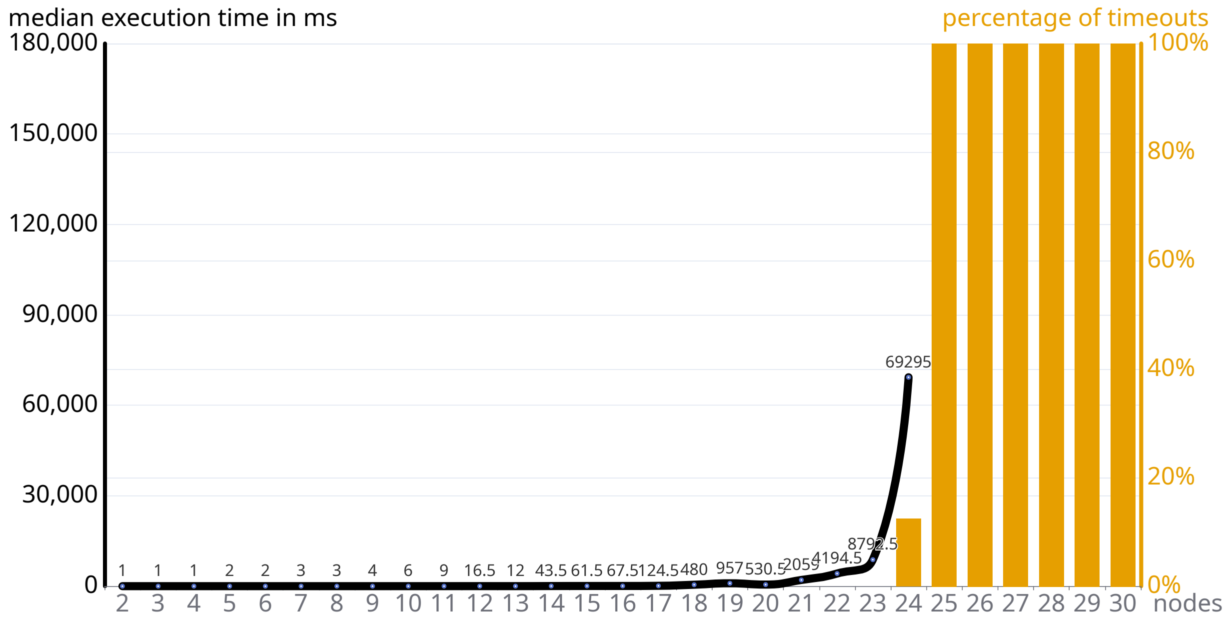

**(c) Median execution time and number of timeouts for** $p = 0.8$




**(a) Neo4j**



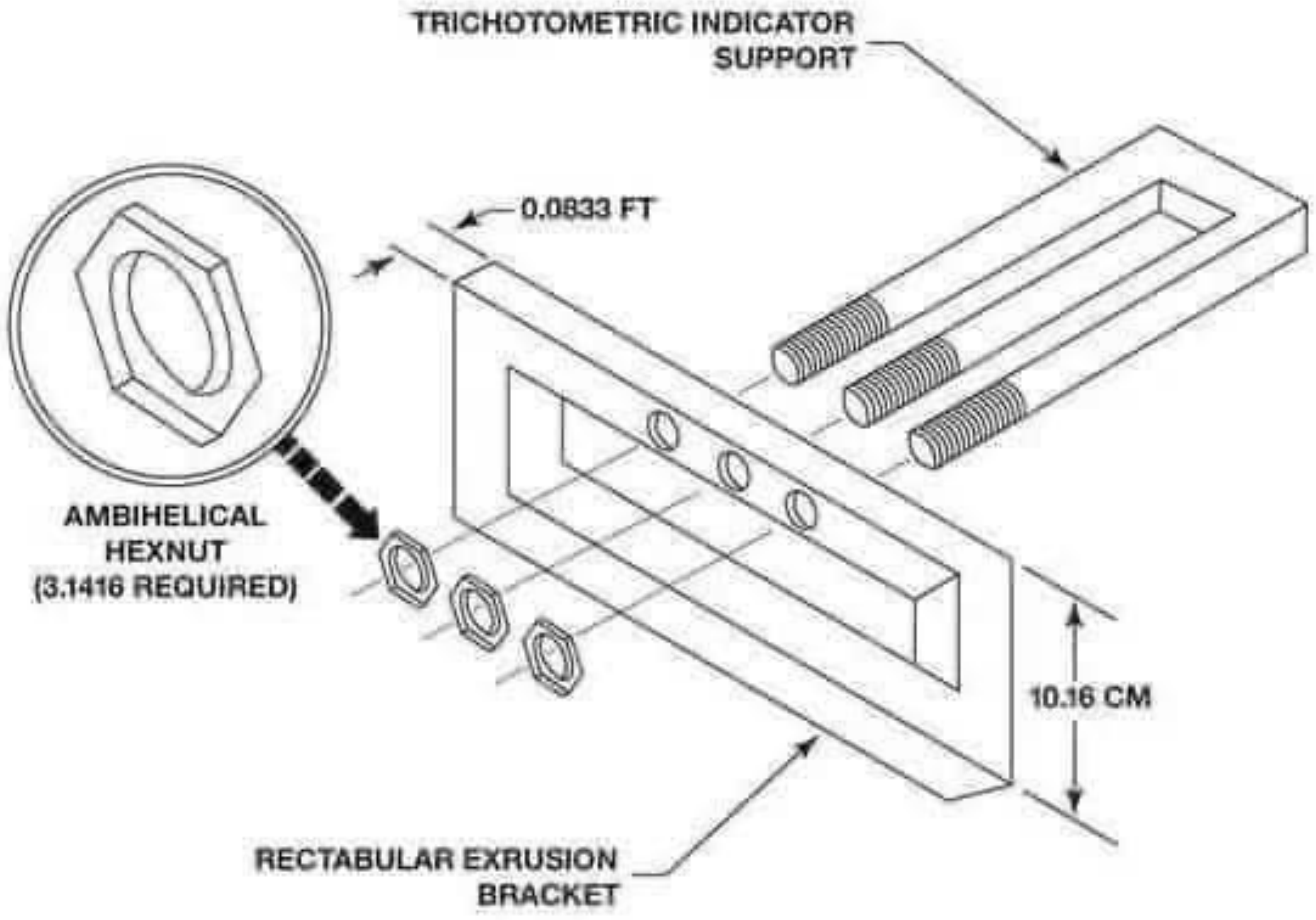

**(b) Postgres**




**(c) DuckDB**

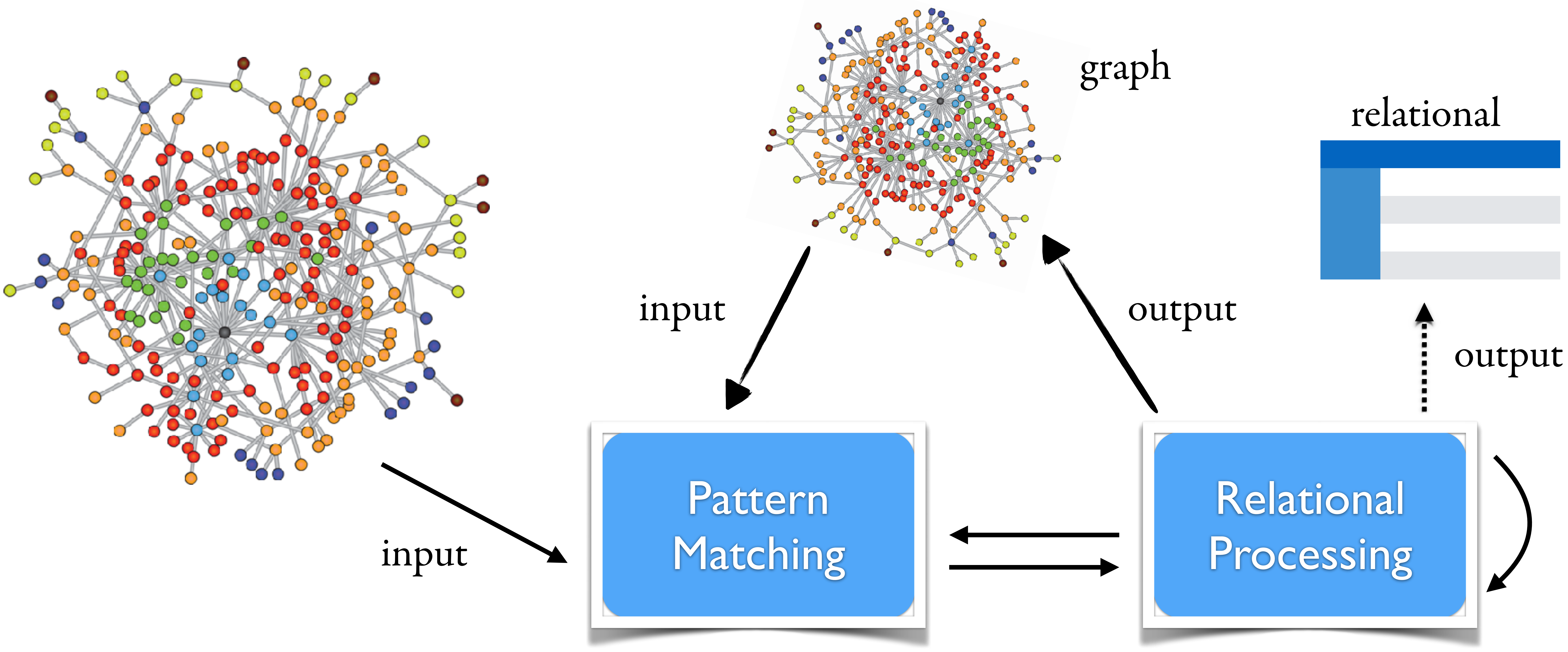
... except on tiniest graphs

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




that make sense until they don't?

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



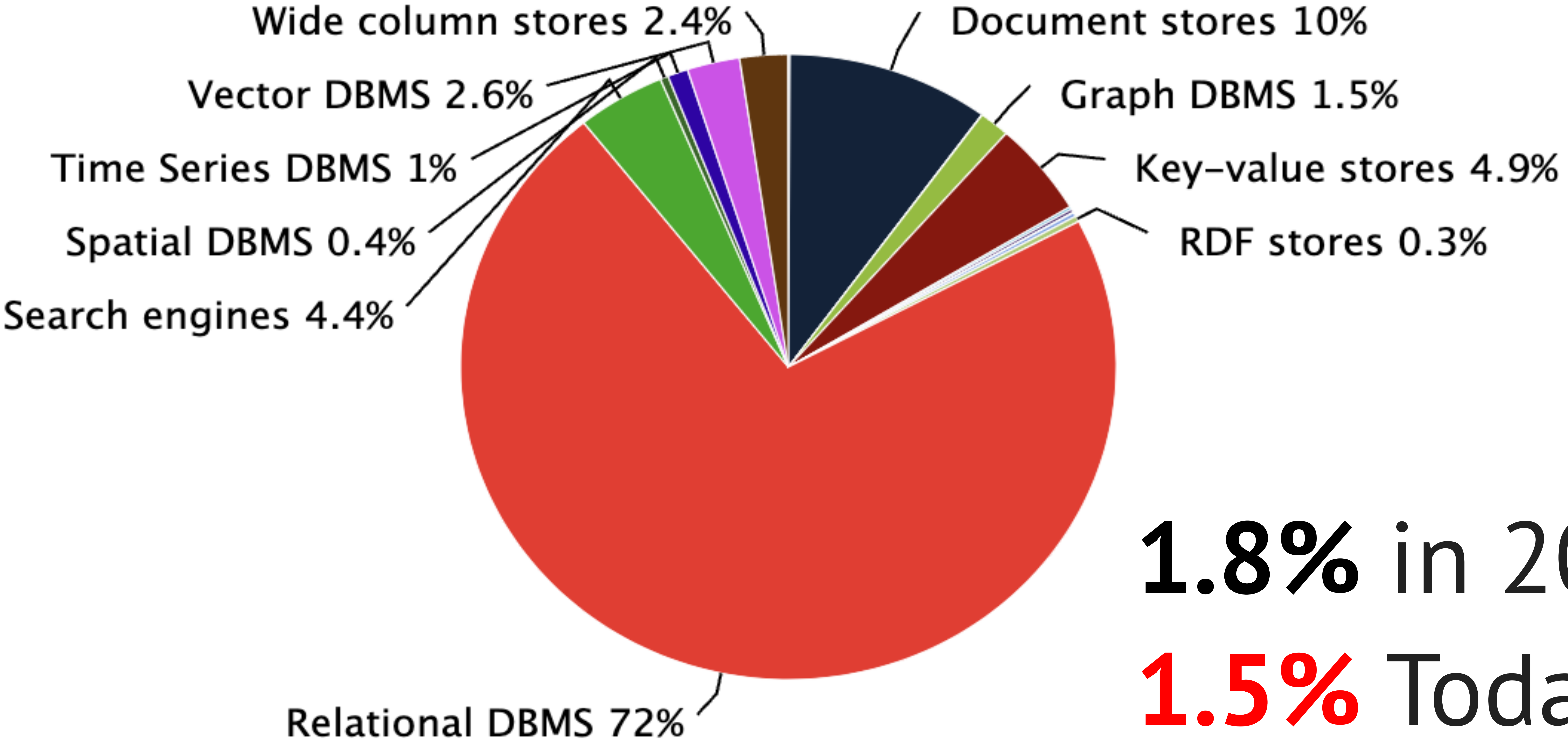

What is missing? COMPOSITIONALITY

# 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?

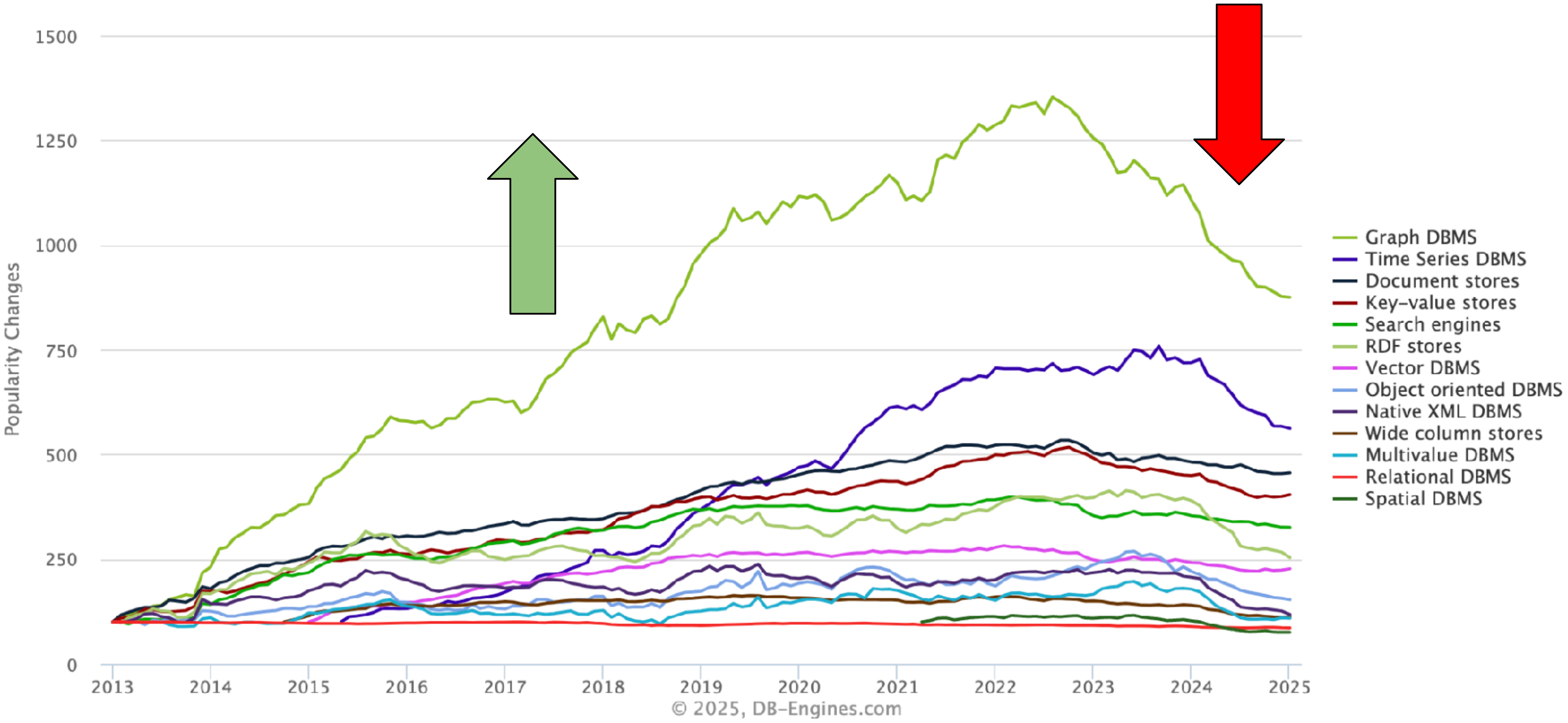
# Ranking scores per category in percent, January 2025




**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