Formal Semantics of the Language Cypher

Version 1.0 : core read-only fragment

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

[1]Université Paris-Est, France
[2]Neo4j
[3]University of Edinburgh

Abstract

Cypher is a query language for property graphs. It was originally designed and implemented as part of the Neo4j graph database, and it is currently used in a growing number of commercial systems, industrial applications and research projects. In this work, we provide denotational semantics of the core fragment of the read-only part of Cypher, which features in particular pattern matching, filtering, and most relational operations on tables.

Contents

*Affiliated with the School of Informatics at the University of Edinburgh during the time of contributing to this work.

1 Introduction

In the last decade, property graph databases [9] such as Neo4j, JanusGraph and Sparksee have become more widespread in industry and academia. They have been used in multiple domains, such as master data and knowledge management, recommendation engines, fraud detection, IT operations and network management, authorization and access control [15], bioinformatics [11], social networks [5], software system analysis [8], and in investigative journalism [2]. Using graph databases to manage graph-structured data confers many benefits such as explicit support for modeling graph data, native indexing and storage for fast graph traversal operations, built-in support for graph algorithms (e.g., Page Rank, subgraph matching and so on), and the provision of graph languages, allowing users to express complex pattern-matching operations.

This paper is about Cypher, a well-established language for querying and updating property graph databases, which began life in the Neo4j product, but has now been implemented commercially in other products such as SAP HANA Graph, Redis Graph, Agens Graph (over PostgreSQL) and Memgraph. Neo4j [15] is one of the most popular property graph databases[1] that stores graphs natively on disk and provides a framework for traversing graphs and executing graph operations. The language therefore is used in hundreds of production applications across many industry vertical domains.

The data model of Neo4j that is used by Cypher is that of *property graphs*. It is the most popular graph data model in industry, and is becoming increasingly prevalent in academia [10]. The model comprises *nodes*, representing entities (such as people, bank accounts, departments and so on), and *relationships* (synonymous with *edges*), representing the connections or relationships between the entities. In the graph model, the relationships are as important as the entities themselves. Moreover, any number of attributes (henceforth termed *properties*), in the form of key-value pairs, may be associated with the nodes and relationships. This allows for the modeling and querying of complex data.

The goal of this document is to provide denotational semantics for a core fragment of the read-only part of Cypher, which features pattern matching, filtering, and most relational operations on tables. Notable parts that are excluded from this work include all update (write) clauses, line-ordering and aggregation. Covered value types include trilean values, integers, strings, lists, maps and paths.

The need for a formal semantics stems from the fact that Cypher, in addition to being implemented in an industrial product with a significant customer base, has been picked up by others, and several implementations of it exist. Given the lack of a standard for the language (which can take many years to complete, as it did for SQL), it has become pressing to agree on the formal data model and the meaning of the main constructs. A formal semantics has other advantages too; for example, it allows one to reason about the equivalence of queries, and prove correctness of existing or discover new optimizations. The need of the formal semantics has long been accepted in the field of programming languages [13] and for several common languages their semantics has been fully worked out [1, 7, 12, 14]. Recently similar

[1] https://db-engines.com/en/ranking/graph+dbms

efforts have been made for the core SQL constructs [3, 4, 6, 16] with the goal of proving correctness of SQL optimizations and understanding the expressiveness of its features. The existence of the formal semantics of Cypher makes it possible for different implementations to agree on its core features, and paves a way to a reference implementation against which others will be compared. We also note that providing semantics for an existing real-life language like Cypher that accounts for all of its idiosyncrasies is much harder than for theoretical calculi underlying main features of languages, as has been witnessed by previous work on SQL [6] and on many programming languages.

The document is organized as follows. Section 2 is an overview of the semantics. Section 3 defines the data model that will be used throughout the document. This includes base data values that can occur in property graphs or be returned by queries, as well as property graphs themselves, and finally records and tables on which the semantics of queries are based. Section 4 defines the core mecanism of Cypher that is, *pattern matching*. It provides the syntax of patterns, defines the notion of rigid patterns and explicits how to compute the bag of the paths that satisfy a pattern. Then, section 5 provides a formal grammar that defines the syntax of the fragment of Cypher that is considered in this work. It is organized around the three main constructs of a Cypher statement: expressions, clauses and queries. Finally, Section 6 defines the semantics of Cypher over the syntax provided in Section 5. More specifically, this section defines how to evaluate an expression as a value, and to formally specify a Cypher query as a mathematical function that returns tables of values.

It is important to note that the sole purpose of this work is to formally specify the intended behaviour of Cypher. In particular, it should not be considered as a user's guide and the reader is assumed to already possess a good understanding of Cypher.

2 General principles of the semantics

This section provides an overview of the semantics. Most of the object we refer to are only briefly described here. All the proper definitions will be given later on.

The key elements of Cypher are as follows:

- data model, that includes *values*, *graphs*, and *tables*;
- query language, that includes *expressions*, *patterns*, *clauses*, and *queries*.

Values can be simple, such as strings and integers, or composite, such as lists and maps. Cypher is a language to query data from *property graphs*. As usual, such a graph consists of *nodes* that are linked by directed edges, called *relationships* but in addition, relationships bear *types*, nodes bear *labels* and both may bear properties, i.e. key-value pairs. Expressions denote values; patterns occur in **MATCH** clauses; and queries are sequences of clauses. Tables are bags of *records*, which are partial functions from (column-)*names* to values; in other words, tables are neither line-ordered nor column-ordered. Each clause denotes a function from tables to tables and each query returns a table.

Concept	Notation	Set notation
Property keys	k	$\mathcal{K}$
Node identifiers	n	$\mathcal{N}$
Relationship identifiers	r	$\mathcal{R}$
Node labels	ℓ	$\mathcal{L}$
Relationship types	t	$\mathcal{T}$
Names	a, b	$\mathcal{A}$
Base functions	f	$\mathcal{F}$
Values	v	$\mathcal{V}$
Expressions	e	–
Node patterns	χ	–
Relationship patterns	ρ	–
Path patterns	π	–

Table 1: Summary of notational conventions

To provide a formal semantics of Cypher, we will define one relation and two functions:

- The *pattern matching relation* checks if a path p in a graph G satisfies a pattern π, under an assignment u of values to the free variables of the pattern. This is written as $(p, G, u) \models \pi$.
- The *semantics of expressions* associates an expression expr, a graph G and an assignment u with a value $[\![\mathsf{expr}]\!]_{G,u}$.
- The *semantics of queries* (resp., *clauses*) associates a query Q (resp., clause C) and a graph G with a function $[\![Q]\!]_G$ (resp., $[\![C]\!]_G$) that takes a table and returns a table (perhaps with more rows or with wider rows).

Note that the semantics of a query Q is a *function*; thus it should not be confused with the *output* of Q. The evaluation of a query starts with the table containing one empty tuple, which is then progressively changed by applying functions that provide the semantics of Q's clauses. The composition of such functions, i.e., the semantics of Q, is a function again, which defines the output as

$$\mathsf{output}(Q, G) \;=\; [\![Q]\!]_G(T_{()})$$

where $T_{()}$ is the table containing the single empty tuple $()$.

With this basic understanding of the data model and the semantics of the language, we now explain it in detail. Throughout the description of the semantics, we shall use the notational conventions in Table 1 (they will be explained in the following sections; they are summarized here for a convenient reference).

3 Data Model

3.1 Values

We consider three disjoint sets $\mathcal{K}$ of *property keys*, $\mathcal{N}$ of *node identifiers* and $\mathcal{R}$ of *relationship identifiers* (ids for short). These sets are all assumed to be countably infinite (so we never run out of keys and ids). For this presentation of the model, we assume two base types: the integers $\mathbb{Z}$, and the type of finite strings over a finite alphabet Σ (this does not really affect the semantics of queries; these two types are chosen purely for illustration purposes).

The set $\mathcal{V}$ of values is inductively defined as follows:

- Identifiers (i.e., elements of $\mathcal{N}$ and $\mathcal{R}$) are values;
- Base types (elements of $\mathbb{Z}$ and Σ^*) are values;
- **`true`**, **`false`** and **`null`** are values;
- $\mathsf{list}()$ is a value (empty list), and if $v_1, \ldots, v_m$ are values, for $m > 0$, then $\mathsf{list}(v_1, \ldots, v_m)$ is a value.
- $\mathsf{map}()$ is a value (empty map), and if $k_1, \ldots, k_m$ are distinct property keys and $v_1, \ldots, v_m$ are values, for $m > 0$, then $\mathsf{map}((k_1, v_1), \ldots, (k_m, v_m))$ is a value.
- If n is a node identifier, then $\mathsf{path}(n)$ is a value. If $n_1, \ldots, n_m$ are node ids and $r_1, \ldots, r_{m-1}$ are relationship ids, for $m > 1$, then $\mathsf{path}(n_1, r_1, n_2, \ldots, n_{m-1}, r_{m-1}, n_m)$ is a value. We shall use shorthands n and $n_1 r_1 n_2 \ldots n_{m-1} r_{m-1} n_m$.

In the Cypher syntax, lists are $[v_1, \ldots, v_m]$ and maps are $\{k_1 : v_1, \ldots, k_m : v_m\}$; we use explicit notation for them to make clear the distinction between the syntax and the semantics of values.

We use the symbol "$\cdot$" to denote concatenation of paths, which is possible only if the first path ends in a node where the second starts, i.e., if $p_1 = n_1 r_1 \cdots r_{j-1} n_j$ and $p_2 = n_j r_j \cdots r_{m-1} n_m$ then $p_1 \cdot p_2$ is $n_1 r_1 n_2 \cdots n_{m-1} r_{m-1} n_m$.

Every real-life query language will have a number of functions defined on its values, e.g., concatenation of strings and arithmetic operations on numbers. To model this, we assume a finite set $\mathcal{F}$ of predefined functions that can be applied to values (and produce new values). The semantics is parameterized by this set, which can be extended whenever new types and/or basic functions are added to the language.

3.2 Property graphs

Let $\mathcal{L}$ and $\mathcal{T}$ be countable sets of node labels and relationship types, respectively. A property graph is a tuple $G = \langle N, R, \mathsf{src}, \mathsf{tgt}, \iota, \lambda, \tau \rangle$ where:

- N is a finite subset of $\mathcal{N}$, whose elements are referred to as the *nodes* of G.
- R is a finite subset of $\mathcal{R}$, whose elements are referred to as the *relationships* of G.

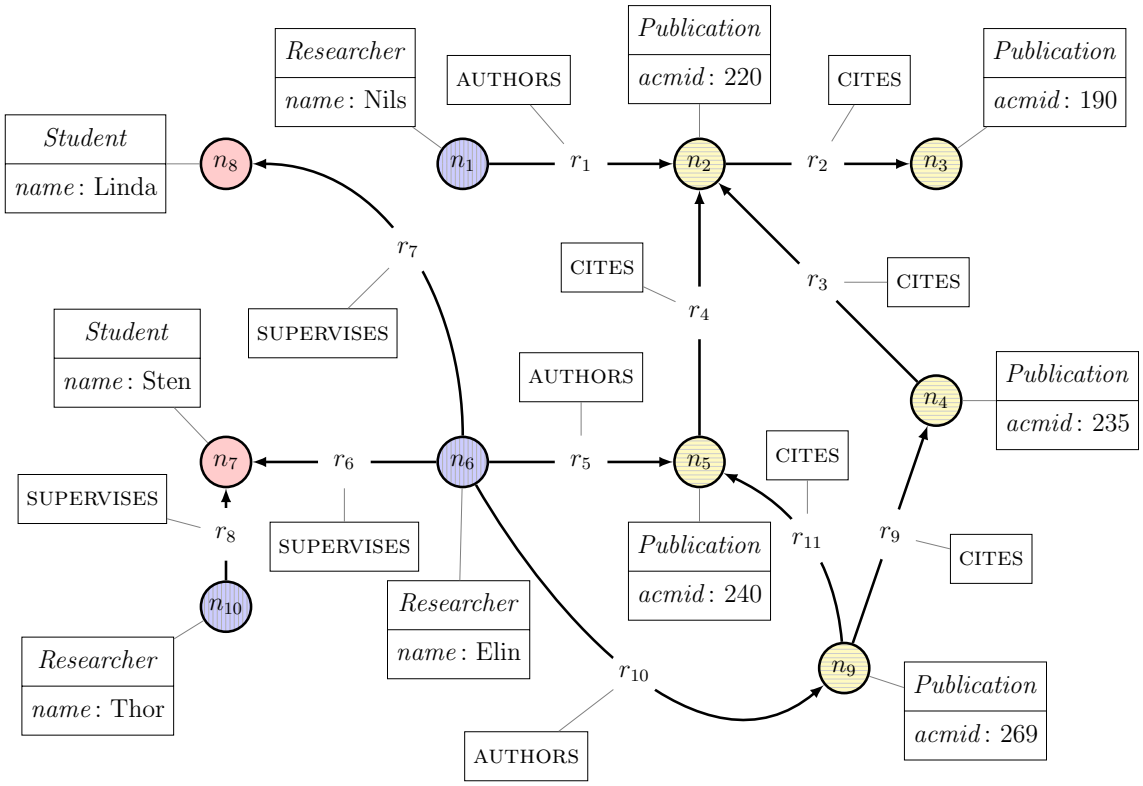

Figure 1: Example data graph showing supervision and citation data for researchers, students and publications

- $\mathsf{src}\colon R \to N$ is a function that maps each relationship to its *source* node.
- $\mathsf{tgt}\colon R \to N$ is a function that maps each relationship to its *target* node.
- $\iota\colon (N \cup R) \times \mathcal{K} \to \mathcal{V}$ is a finite partial function that maps a (node or relationship) identifier and a property key to a value.
- $\lambda\colon N \to 2^{\mathcal{L}}$ is a function that maps each node id to a finite (possibly empty) set of labels.
- $\tau\colon R \to \mathcal{T}$ is a function that maps each relationship identifier to a relationship type.

Example 1. *We now refer to the property graph in Figure 1 and show how, for a sample of its nodes and relationships, it is formally represented in this model as a graph* $G = (N, R, \mathsf{src}, \mathsf{tgt}, \iota, \lambda, \tau)$.

- $N = \{n_1, \ldots, n_{10}\}$;
- $R = \{r_1, \ldots, r_{11}\}$;
- $\mathsf{src} = \left\{ \begin{array}{llll} r_1 \mapsto n_1\,, & r_4 \mapsto n_5\,, & r_7 \mapsto n_6\,, & r_{10} \mapsto n_6 \\ r_2 \mapsto n_2\,, & r_5 \mapsto n_6\,, & r_8 \mapsto n_{10}\,, & r_{11} \mapsto n_9 \\ r_3 \mapsto n_4\,, & r_6 \mapsto n_6\,, & r_9 \mapsto n_9 & \end{array} \right\}$;

- $\mathit{tgt} = \left\{ \begin{array}{llll} r_1 \mapsto n_2\,, & r_4 \mapsto n_2\,, & r_7 \mapsto n_8\,, & r_{10} \mapsto n_9 \\ r_2 \mapsto n_3\,, & r_5 \mapsto n_5\,, & r_8 \mapsto n_7\,, & r_{11} \mapsto n_5 \\ r_3 \mapsto n_2\,, & r_6 \mapsto n_7\,, & r_9 \mapsto n_4 & \end{array} \right\};$
- $\iota(n_1, \mathit{name}) = \mathit{Nils}$, $\iota(n_2, \mathit{acmid}) = 220$, $\iota(n_3, \mathit{acmid}) = 190$, ..., $\iota(n_{10}, \mathit{name}) = \mathit{Thor}$;
- $\lambda(n_1) = \lambda(n_6) = \lambda(n_{10}) = \{\mathit{Student}\}$, $\lambda(n_2) = \lambda(n_3) = \lambda(n_4) = \lambda(n_5) = \lambda(n_9) = \{\mathit{Publication}\}$, $\lambda(n_7) = \lambda(n_8) = \{\mathit{Researcher}\}$;
- $\tau(r) = \begin{cases} \text{AUTHORS} & \text{for } r \in \{r_1, r_5, r_{10}\}\,, \\ \text{SUPERVISES} & \text{for } r \in \{r_6, r_7, r_8\}\,, \\ \text{CITES} & \text{for } r \in \{r_2, r_3, r_4, r_9, r_{11}\}\,. \end{cases}$

3.3 Tables

Let $\mathcal{A}$ be a countable set of names. A *record* is a partial function from names to values, conventionally denoted as a tuple with named fields $u = (a_1 : v_1, \ldots, a_n : v_n)$ where $a_1, \ldots, a_n$ are distinct names, and $v_1, \ldots, v_n$ are values. The order in which the fields appear is only for notation purposes. We refer to $\mathsf{dom}(u)$, i.e., the domain of u, as the set $\{a_1, \ldots, a_m\}$ of names used in u. Two records u and u' are *uniform* if $\mathsf{dom}(u) = \mathsf{dom}(u')$.

If $u = (a_1 : v_1, \ldots, a_n : v_n)$ and $u' = (a'_1 : v'_1, \ldots, a_m : v'_m)$ are two records, then (u, u') denotes the record $(a_1 : v_1, \ldots, a_n : v_n, a'_1 : v'_1, \ldots, a'_m : v'_m)$, assuming that all a_i, a'_j for $i \leq n, j \leq m$ are distinct. If $A = \{a_1, \ldots, a_n\}$ is a set of names v is a value, then $(A : v)$ denotes the record $(a_1 : v, \ldots, a_n : v)$. We use $()$ to denote the empty record, i.e., the partial function from names to values whose domain is empty.

If A is a set of names, then a *table* with fields A is a bag, or multiset, of records u such that $\mathsf{dom}(u) = A$. A table with no fields is just a bag of copies of the empty record. In most cases, the set of fields of tables will be clear from the context, and will not be explicitly stated. Given two tables T and T', we use $T \uplus T'$ to denote their *bag union*, in which the multiplicity of each record is the sum of their multiplicities in T and T'. If $B = \{b_1, \ldots, b_n\}$ is a bag, and $T_{b_1}, \ldots, T_{b_n}$ are tables, then $\biguplus_{b \in B} T_b$ stands for $T_{b_1} \uplus \ldots \uplus T_{b_n}$. Finally, we use $\varepsilon(T)$ to denote the result of duplicate elimination on T, i.e., each tuple of T is present just once in $\varepsilon(T)$.

4 Pattern matching

4.1 Syntax of patterns

It is important to remember that the Cypher grammar is defined by mutual recursion of expressions, patterns, clauses, and queries. Here, the description of patterns will make a reference to expressions, which we will cover later on; all we need to know for now is that these will denote values.

The Cypher syntax of patterns is given in Figure 2, where the highlighted symbols denote tokens of the language. Instead of the actual Cypher syntax, here we

```
      pattern ::= pattern° | a = pattern°
     pattern° ::= node_pattern | node_pattern rel_pattern pattern°
 node_pattern ::= ( a? label_list? map? )
  rel_pattern ::= -[ a? type_list? len? map? ]-> | <-[ a? type_list? len? map? ]-
                | -[ a? type_list? len? map? ]-

   label_list ::= :ℓ | :ℓ label_list
          map ::= { prop_list }
    prop_list ::= k:expr | k:expr, prop_list
    type_list ::= :t | type_list |t
          len ::= * | *d | *d1.. | *..d2 | *d1..d2      d, d1, d2 ∈ ℕ
```

Figure 2: Syntax of Cypher patterns

use an abstract mathematical notation that lends itself more naturally to a formal treatment.

A node pattern χ is a triple (a, L, P) where:

- $a \in \mathcal{N} \cup \{\mathsf{nil}\}$ is an optional name;
- $L \subset \mathcal{L}$ is a possibly empty finite set of node labels;
- P is a possibly empty finite partial map from $\mathcal{K}$ to expressions.

For example, the following node pattern in Cypher syntax:

`(x:Person:Male {name:` expr_1`, age:` expr_2`})`

is represented as $(x, \{\text{Person}, \text{Male}\}, \{\text{name} \mapsto e_1, \text{age} \mapsto e_2\})$, where e_1 and e_2 are the representations of expressions expr_1 and expr_2, respectively. The simplest node pattern `()` is represented by $(\mathsf{nil}, \varnothing, \varnothing)$.

A relationship pattern ρ is a tuple (d, a, T, P, I) where:

- $d \in \{\rightarrow, \leftarrow, \leftrightarrow\}$ specifies the *direction* of the pattern: left-to-right ($\rightarrow$), right-to-left ($\leftarrow$), or undirected ($\leftrightarrow$);
- $a \in \mathcal{N} \cup \{\mathsf{nil}\}$ is an optional name,
- $T \subset \mathcal{T}$ is a possibly empty finite set of relationship types;
- P is a possibly empty finite partial map from $\mathcal{K}$ to expressions;
- I is either nil or (m, n) with $m, n \in \mathbb{N} \cup \{\mathsf{nil}\}$.

Table 2 gives a few relationship patterns and their mathematical representations. As highlighted by these examples, I is nil if and only if the optional grammar token len does not appear in syntax of the pattern (see Figure 2); otherwise, I is equal to $(\mathsf{nil}, \mathsf{nil})$ if len derives to $*$ and I is equal to (d, d), (d_1, nil), (nil, d_2), (d_1, d_2) if other derivations rules are applied, respectively.

Pattern	Representation
`-[:KNOWS {since:1985}]-`	$(\leftrightarrow, \mathsf{nil}, \{\text{KNOWS}\}, \{\text{since} \mapsto 1985\}, nil)$
`-[:KNOWS*1 {since:1985}]-`	$(\leftrightarrow, \mathsf{nil}, \{\text{KNOWS}\}, \{\text{since} \mapsto 1985\}, (1, 1))$
`-[:KNOWS*1..1 {since:1985}]-`	$(\leftrightarrow, \mathsf{nil}, \{\text{KNOWS}\}, \{\text{since} \mapsto 1985\}, (1, 1))$
`-[:KNOWS*..1 {since:1985}]-`	$(\leftrightarrow, \mathsf{nil}, \{\text{KNOWS}\}, \{\text{since} \mapsto 1985\}, (nil, 1))$
`-[:KNOWS* {since:1985}]-`	$(\leftrightarrow, \mathsf{nil}, \{\text{KNOWS}\}, \{\text{since} \mapsto 1985\}, (nil, nil))$

Table 2: Example of relationship patterns and their representation

In general, I defines the range of the relationship pattern. The range is $[m, n]$ if $I = (m, n)$ where nil is replaced by 1 and ∞ in the place of the lower and upper bounds. The range is $[1, 1]$ if $I = \mathsf{nil}$. A relationship pattern is said *rigid* if its range [m,n] satisfies: $m = n \in \mathbb{N}$.

A path pattern is an alternating sequence of the form

$$\chi_1 \quad \rho_1 \quad \chi_2 \quad \cdots \quad \rho_{n-1} \quad \chi_n$$

where each χ_i is a node pattern and each ρ_i is a relationship pattern. A path pattern π can be optionally given a name a, written as π/a; we then refer to a *named pattern*. A path pattern is *rigid* if all relationship patterns in it are rigid, and *variable length* otherwise.

We shall now define the satisfaction relation for path patterns w.r.t. a property graph $G = (N, R, \mathsf{src}, \mathsf{tgt}, \iota, \lambda, \tau)$, a path with node ids from N and relationship ids from R, and an assignment u.

We consider rigid patterns first as a special case, because they – unlike variable length patterns – uniquely define both the length and the possible variable bindings of the paths satisfying them. The satisfaction of variable length patterns will then be defined in terms of a set of rigid patterns.

4.2 Satisfaction of rigid patterns

The definition is inductive, with the base case given by node patterns (which are trivially rigid path patterns). Let χ be a node pattern (a, L, P); then $(n, G, u) \models \chi$ if all of the following hold:

- either a is nil or $u(a) = n$;
- $L \subseteq \lambda(n)$;
- $[\![\iota(n, k) = P(k)]\!]_{G,u} = \texttt{true}$ for each k s.t. $P(k)$ is defined.

Example 2. *Consider the property graph G in Figure 3 and the node patterns $\chi_1 = (x, \{\,Teacher\}, \varnothing)$ and $\chi_2 = (y, \varnothing, \varnothing)$. Then,*

$$\begin{aligned}
&(n_1, G, u) \models \chi_1 && \text{if } u \text{ is an assignment that maps } x \text{ to } n_1\,,\\
&(n_2, G, u) \not\models \chi_1 && \text{for any assignment } u\,,\\
&(n_3, G, u) \models \chi_1 && \text{if } u \text{ is an assignment that maps } x \text{ to } n_3\,,\\
&(n_4, G, u) \models \chi_1 && \text{if } u \text{ is an assignment that maps } x \text{ to } n_4\,.
\end{aligned}$$

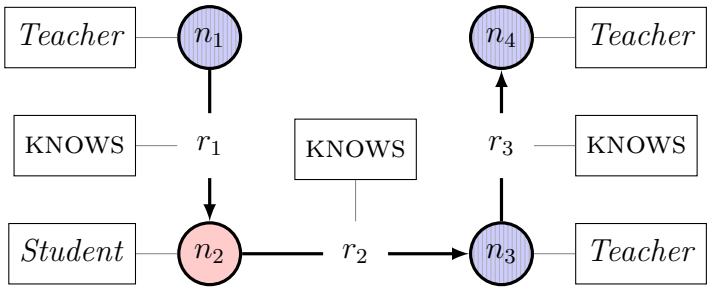

Figure 3: Property graph with students and teachers

For $i = 1, \ldots, 4$ we have that $(n_i, G, u_i) \models \chi_2$ whenever u_i is an assignment that maps y to n_i. □

For the inductive case, let χ be a node pattern, let π be a rigid path pattern, and let ρ be the relationship pattern (d, a, T, P, I). First we assume that $I \neq nil$, hence since ρ is rigid, the range defined by I is $[m, m]$ with $m \in \mathcal{N}$. For $m = 0$, we have that $(n \cdot p, G, u) \models \chi\rho\pi$ if

(a) either a is nil or $u(a) = \mathsf{list}()$; and

(b) $(n, G, u) \models \chi$ and $(p, G, u) \models \pi$.

For $m \geq 1$, we have that $(n_1 \cdots r_m n_{m+1} \cdot p, G, u) \models \chi\rho\pi$ if all of the following hold:

(a) either a is nil or $u(a) = \mathsf{list}(r_1, \ldots, r_m)$;

(b) $(n_1, G, u) \models \chi$ and $(p, G, u) \models \pi$;

(c) $r_1, \ldots, r_m$ are *distinct* relationship ids;

and, for every $i \in \{1, \ldots, m\}$, all of the following hold:

(d) $\tau(r_i) \in T$;

(e) $[\![\iota(r_i, k) = P(k)]\!]_{G,u} = \texttt{true}$ for every k s.t. $P(k)$ is defined;

(f) $(\mathsf{src}(r_i), \mathsf{tgt}(r_i)) \in \begin{cases} \{(n_i, n_{i+1}), (n_{i+1}, n_i)\} & \text{if } d \text{ is } \leftrightarrow \ , \\ \{(n_i, n_{i+1})\} & \text{if } d \text{ is } \rightarrow \ , \\ \{(n_{i+1}, n_i)\} & \text{if } d \text{ is } \leftarrow \ . \end{cases}$

Second, the case $I = \mathsf{nil}$ is treated as if $I = (1, 1)$ with the exception that item (a) is replaced by: (a) either a is nil or $u(a) = r_1$

Example 3. *Consider again the property graph G in Figure 3 and the following rigid pattern π in Cypher syntax:*

```
(x:Teacher) -[:KNOWS*2]-> (y)
```

In our mathematical representation this amounts to:

$$\underbrace{(x, \{\mathit{Teacher}\}, \varnothing)}_{\chi_1}, \underbrace{(\rightarrow, \mathsf{nil}, \{\text{KNOWS}\}, \varnothing, (2, 2))}_{\rho}, \underbrace{(y, \varnothing, \varnothing)}_{\chi_2}$$

where χ_1 and χ_2 are the node patterns we have seen in Example 2. Now, let $u = \{x \mapsto n_1, y \mapsto n_3\}$; from that example we know that $(n_1, G, u) \models \chi_1$ and $(n_3, G, u) \models$

χ_2. *Then, following the definition of satisfaction given above, one can easily see that* $(p, G, u) \models \pi$, *where* $p = n_1 r_1 n_2 r_2 n_3$ *and* $\pi = \chi_1 \rho \chi_2$.

Observe that if there is another assignment u' *s.t.* $(p, G, u') \models \pi$, *then* u' *maps* x *to* n_1 *and* y *to* n_3. *This is the intuitive reason why rigid patterns are of interest: given a path and a rigid pattern, there exists at most one possible assignment of the* free variables *(which we shall formally define shortly) of the pattern w.r.t. which the path satisfies the pattern. We will see that for variable length patterns this is no longer the case.* □

For named rigid patterns, we have that $(p, G, u) \models \pi/a$ if $u(a) = p$ and $(p, G, u) \models \pi$.

4.3 Satisfaction of variable length patterns

Informally, a variable length pattern is a compact representation for a possibly infinite set of rigid patterns; e.g., a pattern of length *at least* 1 will represent patterns of length 1, patterns of length 2, and so on.

To make this idea precise, let $\rho = (d, a, T, P, (m, n))$ be a variable length relationship pattern, and $\rho' = (d, a, T, P, (m', m'))$ be a rigid relationship pattern. We say that ρ *subsumes* ρ', and write $\rho \sqsupset \rho'$, if m' belongs to the range $[m, n]$ defined by I. If ρ is rigid, then it only subsumes itself. This subsumption relation is easily extended to path patterns. Given a variable length pattern $\pi = \chi_1 \rho_1 \chi_2 \cdots \chi_{k-1} \rho_{k-1} \chi_k$ and a rigid pattern $\pi' = \chi_1 \rho'_1 \chi_2 \cdots \chi_{k-1} \rho'_{k-1} \chi_k$, we say that π subsumes π' (written $\pi \sqsupset \pi'$) if $\rho_i \sqsupset \rho'_i$ for every $i \in \{1, \ldots, k-1\}$.

Then, we define the *rigid extension* of π as

$$\mathsf{rigid}(\pi) = \{\, \pi' \mid \pi' \text{ is rigid and } \pi \sqsupset \pi' \,\} \ ,$$

that is, the (possibly infinite) set of all rigid patterns subsumed by π. For a named pattern, $\mathsf{rigid}(\pi/a) = \{\pi'/a \mid \pi' \in \mathsf{rigid}(\pi)\}$. Finally, $(p, G, u) \models \pi$ if $(p, G, u) \models \pi'$ for some $\pi' \in \mathsf{rigid}(\pi)$, and similarly for named patterns.

Example 4. *Consider the following variable length pattern* π*:*

```
(x:Teacher) -[:KNOWS*1..2]-> (z)
            -[:KNOWS*1..2]-> (y:Teacher)
```

That is, π *is the pattern* $\chi_1 \rho \chi_2 \rho \chi_3$ *with*

$$\chi_1 = (x, \{\mathit{Teacher}\}, \varnothing)\,, \qquad \chi_3 = (y, \{\mathit{Teacher}\}, \varnothing)\,,$$
$$\chi_2 = (z, \varnothing, \varnothing)\,, \qquad \rho = (\rightarrow, \mathsf{nil}, \{\textsc{knows}\}, \varnothing, (1, 2))\,.$$

Then, $\mathsf{rigid}(\pi)$ *is the set*

$$\{\, \underbrace{\chi_1 \rho_1 \chi_2 \rho_1 \chi_3}_{\pi_1}\,,\ \underbrace{\chi_1 \rho_1 \chi_2 \rho_2 \chi_3}_{\pi_2}\,,\ \underbrace{\chi_1 \rho_2 \chi_1 \rho_1 \chi_3}_{\pi_3}\,,\ \underbrace{\chi_1 \rho_2 \chi_2 \rho_2 \chi_3}_{\pi_4} \,\}$$

where

$$\rho_1 = (\rightarrow, \mathsf{nil}, \{\textsc{knows}\}, \varnothing, (1, 1))\,,\ \rho_2 = (\rightarrow, \mathsf{nil}, \{\textsc{knows}\}, \varnothing, (2, 2))\,.$$

Consider again the property graph G in Figure 3. Let

$$p_1 = n_1 r_1 n_2 r_2 n_3 \qquad u_1 = \{\, x \mapsto n_1, y \mapsto n_3, z \mapsto n_2 \,\}$$
$$p_2 = n_1 r_1 n_2 r_2 n_3 r_3 n_4 \qquad u_2 = \{\, x \mapsto n_1, y \mapsto n_4, z \mapsto n_2 \,\}$$

Then, $(p_1, G, u_1) \models \pi_1$ and $(p_2, G, u_2) \models \pi_2$; therefore, π is satisfied in G by p_1 under u_1 and by p_2 under u_2. This shows the ability of a variable length pattern to match paths of varying length.

In addition, variable length patterns may admit several assignments even for a single given path. To see this, note that p_2 satisfies π in G also *under the assignment u_2' that agrees with u_2 on x and y but maps z to n_3, because $(p_2, G, u_2') \models \pi_3$.* □

In Cypher, we want to return the "matches" for a pattern in a graph, not simply check whether the pattern is satisfied (i.e., there exists a match). This is captured formally next.

4.4 Pattern matching

The set of *free variables* of a node pattern $\chi = (a, L, P)$, denoted by $\mathsf{free}(\chi)$, is $\{a\}$ whenever a is not nil, and empty otherwise. For a relationship pattern ρ, the set $\mathsf{free}(\rho)$ is defined analogously. Then, for a path pattern π we define $\mathsf{free}(\pi)$ to be union of all free variables of each node and relationship pattern occurring in it. For example, for the pattern π of Example 4 we have $\mathsf{free}(\pi) = \{x, y, z\}$. For named patterns, $\mathsf{free}(\pi/a) = \mathsf{free}(\pi) \cup \{a\}$. Then, for a path pattern π (optionally named), a graph G and an assignment u, we define

$$\mathsf{match}(\pi, G, u) \;=\; \biguplus_{\substack{p \text{ in } G \\ \pi' \in \mathsf{rigid}(\pi)}} \left\{ u' \;\middle|\; \begin{array}{l} \mathsf{dom}(u') = \mathsf{free}(\pi) - \mathsf{dom}(u) \\ \text{and } (p, G, u \cdot u') \models \pi' \end{array} \right\} \tag{1}$$

Note that, even though both u' and π' range over infinite sets, only a finite number of values contribute to a non-empty set in the final union. Thus $\mathsf{match}(\pi, G, u)$ is finite.

In (1), $\biguplus$ stands for bag union: whenever a new combination of π' and p is found such that $(p, G, u \cdot u') \models \pi'$, a new occurrence of u' is added to $\mathsf{match}(\pi, G, u)$. This is in line with the way Cypher combines the **`MATCH`** clause and bag semantics, which is not captured by the satisfaction relation alone.

Example 5. *Consider once again the graph G in Figure 3, and let π be the following variable length pattern:*

```
(x:Teacher) -[:KNOWS*1..2]-> ()
            -[:KNOWS*1..2]-> (y:Teacher)
```

This is similar to the pattern in Example 4, but the middle node pattern is not given any name here: $\mathsf{free}(\pi) = \{x, y\}$. Indeed, $\mathsf{rigid}(\pi)$ is the same as in the previous example, with $\chi_2 = (\mathsf{nil}, \varnothing, \varnothing)$.

Let $p = n_1 r_1 n_2 r_2 n_3 r_3 n_4$ and $u = \{x \mapsto n_1, y \mapsto n_4\}$; it is easy to see that $(p, G, u) \models \pi_3 \in \mathsf{rigid}(\pi)$. However, observe that $(p, G, u) \models \pi_2$ as well (whereas π_1 and π_4 are not satisfied by any path of G). This shows that there may be multiple ways for a single path to satisfy a variable length pattern even under the same assignment. In our example, two copies of u will be added to $\mathsf{match}(\pi, G, \varnothing)$. □

4.5 Matching tuples of path patterns

Cypher allows one to match a tuple $\bar{\pi} = (\pi_1, \ldots, \pi_n)$ of path patterns, each optionally named. We say that $\bar{\pi}$ is rigid if all its components are rigid, and $\mathsf{rigid}(\bar{\pi})$ is defined as $\mathsf{rigid}(\pi_1) \times \cdots \times \mathsf{rigid}(\pi_n)$. The set of free variables of $\bar{\pi}$ is defined as $\mathsf{free}(\bar{\pi}) = \bigcup_{\pi_i} \mathsf{free}(\pi_i)$. Let $\bar{p} = (p_1, \ldots, p_n)$ be a tuple of paths; we write $(\bar{p}, G, u) \models \bar{\pi}$ if *no relationship id occurs in more than one path in* $\bar{p}$ and $(p_i, G, u) \models \pi_i$ for each $i \in \{1, \ldots, n\}$. Then, for a tuple of patterns $\bar{\pi}$, a graph G and an assignment u, $\mathsf{match}(\bar{\pi}, G, u)$ is defined as in (1), with the difference that the bag union is now over tuples $\bar{\pi}' \in \mathsf{rigid}(\pi)$ and $\bar{p}$ of paths.

5 Complete Syntax

We now present the key components of Cypher, namely *expressions*, *clauses*, and *queries*, and define their formal semantics. Together with pattern matching defined in the previous section, they will constitute the formalization of the core of Cypher.

The syntax of Cypher patterns was given in Figure 2. *Expressions* derives from the token expr, whose derivation rules are shown in Figure 4. Similarly, *queries* derive from the token query (Figure 5) and *clauses* from the token clause (Figure 6).

expr ::=	v | a | f (expr_list?)	$v \in \mathcal{V},\ a \in \mathcal{A},\ f \in \mathcal{F}$	*values/variables*
	| expr . k | {} | { prop_list }		*maps*
	| [ ] | [expr_list] | expr IN expr		*lists*
	| expr [expr] | expr [expr . .] | expr [. . expr] | expr [expr . . expr]		
	| expr STARTS WITH expr | expr ENDS WITH expr		*strings*
	| expr CONTAINS expr		
	| expr OR expr | expr AND expr | expr XOR expr | NOT expr		*logic*
	| expr IS NULL | expr IS NOT NULL		
	| expr < expr | expr <= expr | expr >= expr | expr > expr		*comparison*
	| expr = expr | expr <> expr		
expr_list ::=	expr | expr , expr_list		*expression lists*

Figure 4: Syntax of expressions

query ::=	query° | query UNION [ALL]	*unions*
query° ::=	RETURN ret | clause query°	*clause sequences*
ret ::=	$*$ | aggexpr [AS a] | | ret , aggexpr [AS a]	*return lists*

Figure 5: Syntax of queries

clause ::=	[**OPTIONAL**] **MATCH** pattern_tuple [**WHERE** expr]	*matching clauses*
	| **WITH** ret [**WHERE** expr]	
	| **UNWIND** expr **AS** a $\quad a \in \mathcal{A}$	*relational clauses*
pattern_tuple ::=	pattern | pattern , pattern_tuple	*tuples of patterns*

Figure 6: Syntax of clauses

6 Complete Semantics

6.1 Semantics of expressions

The semantics of an expression e is a value $[\![e]\!]_{G,u}$ in $\mathcal{V}$ determined by a property graph G and an assignment u that provides bindings for the names used in e. The rules here are fairly straightforward and given in details below.

Assume that we are given a fixed property graph $G = (N, R, s, t, \iota, \lambda, \tau)$ and a fixed record $u = (a_1 : v_1, \ldots, a_n : v_n)$ that associates values $v_1, \ldots, v_n$ with names $a_1, \ldots, a_n$.

Values and variables

- $[\![v]\!]_{G,u} = v$
  where v is a value.
- $[\![a]\!]_{G,u} = u(a)$
  where a is a name that belongs to the domain of u.
- $[\![f(e_1, \ldots, e_m)]\!]_{G,u} = f([\![e_1]\!]_{G,u}, \ldots, [\![e_m]\!]_{G,u})$
  where $e_1, \ldots, e_m$ are expressions, and f is any m-ary function in $\mathcal{F}$ from values to values.

Maps

- $$[\![e.k]\!]_{G,u} = \begin{cases} \iota([\![e]\!]_{G,u}, k) & \text{if } [\![e]\!]_{G,u} \in \mathcal{N} \cup \mathcal{R} \\ w_i & \text{if } [\![e]\!]_{G,u} = \mathsf{map}((k_1, w_1), (k_2, w_2), \ldots, (k_m, w_m)) \\ & \quad \text{and } k = k_i \\ \texttt{null} & \text{if } [\![e]\!]_{G,u} = \mathsf{map}((k_1, w_1), (k_2, w_2), \ldots, (k_m, w_m)) \\ & \quad \text{and } k \notin \{k_1, \ldots, k_m\} \\ & \text{or } [\![e]\!]_{G,u} = \{\} \\ & \text{or } [\![e]\!]_{G,u} = \texttt{null} \end{cases}$$
  where k and the k_i's are property keys, and w and the w_i's are values.
- $[\![\{k_1 : e_1, \ldots, k_m : e_m\}]\!]_{G,u} = \mathrm{map}((k_1, [\![e_1]\!]_{G,u}), \ldots, (k_m, [\![e_m]\!]_{G,u}))$
  where $k_1, \ldots, k_m$ are **distinct** property keys and $e_1, \ldots, e_m$ are expressions.
- $[\![\{k_1 : e_1, \ldots, k_m : e_m\}]\!]_{G,u} = [\![\{k_{i_1} : e_{i_1}, \ldots, k_{i_\ell} : e_{i_\ell}\}]\!]_{G,u}$

where $k_1, \ldots, k_m$ are property keys, $e_1, \ldots, e_m$ are expressions, and $i_1, \ldots, i_\ell$ are distinct indices such that $\{k_{i_1}, \ldots, k_{i_\ell}\} = \{k_1, \ldots, k_m\}$ and for each p such that $i_p < m$, $k_{i_p} \notin \{k_{i_p+1}, \ldots, k_m\}$. In other words, if there are repeated keys among $k_1, \ldots, k_m$, only the last occurrence of each key is kept.

- $[\![\{\}]\!]_{G,u} = \mathsf{map}()$

Explicit Lists

- $[\![[e_1, \ldots, e_m]]\!]_{G,u} = \mathsf{list}([\![e_1]\!]_{G,u}, \ldots, [\![e_m]\!]_{G,u})$
  where $e_1, \ldots, e_m$ are expressions.
- $[\![[\ ]]\!]_{G,u} = \mathsf{list}()$

Operations on non-empty lists Assume that e is an expression such that $[\![e]\!]_{G,u} = \mathsf{list}(w_0, \ldots, w_{m-1})$ for some values $w_0, \ldots, w_{m-1}$. Then the semantics of list expressions is as follows.

- $$[\![e[e']]\!]_{G,u} = \begin{cases} w_i & \text{if } 0 \le i < m \\ w_{m+i} & \text{if } -m \le i < 0 \\ \texttt{null} & \text{if } i < -m \text{ or } i \ge m \end{cases}$$
  where $[\![e']\!]_{G,u} = i$, for some integer i.
- $$[\![e[e_1..e_2]]\!]_{G,u} = \begin{cases} \mathsf{list}(w_{\max(0,i')}, \ldots, w_{\min(m-1,j'-1)}) & \text{if } i' \le j', i' < m, j' > 0 \\ \mathsf{list}() & \text{otherwise} \end{cases}$$
  where $[\![e_1]\!]_{G,u} = i$ for some integer i, $[\![e_2]\!]_{G,u} = j$ for some integer j, $i' = i$ if $i \ge 0$ and $i' = m + i$ otherwise, $j' = j$ if $j \ge 0$ and $j' = m + j$ otherwise.
- $[\![e[e_1..]]\!]_{G,u} = [\![e[e_1..m]]\!]_{G,u}$
- $[\![e[..e_2]]\!]_{G,u} = [\![e[0..e_2]]\!]_{G,u}$
- $$[\![e' \ \texttt{IN}\ e]\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e' = w_i]\!]_{G,u} = \texttt{true}\ , \\ & \text{for some integer } i,\ 0 \le i < n \\ \texttt{null} & \text{if the previous case does not hold} \\ & \text{and } [\![e' = w_i]\!]_{G,u} = \texttt{null}\ , \\ & \text{for some integer } i,\ 0 \le i < n \\ \texttt{false} & \text{otherwise} \end{cases}$$

Operations on empty lists Assume that e is an expression such that $[\![e]\!]_{G,u} = \mathsf{list}()$. Then the semantics of list expressions is as follows.

- $[\![e\,[\,e'\,]\,]\!]_{G,u} = \texttt{null}$
  where $[\![e']\!]_{G,u} = i$ for some integer i.
- $[\![e\,[\,e_1\,..\,e_2\,]\,]\!]_{G,u} = \mathsf{list}()$
  where $[\![e_1]\!]_{G,u} = i$ for some integer i and $[\![e_2]\!]_{G,u} = j$ for some integer j.

- $[\![e\,\texttt{[}\,e_1\,\texttt{..]}\,]\!]_{G,u} = \mathsf{list}()$
  where $[\![e_1]\!]_{G,u} = i$ for some integer i.
- $[\![e\,\texttt{[..}\,e_2\,\texttt{]}\,]\!]_{G,u} = \mathsf{list}()$
  where $[\![e_2]\!]_{G,u} = j$ for some integer j.
- $[\![e'\ \texttt{IN}\ e]\!]_{G,u} = \texttt{false}$
  where $[\![e']\!]_{G,u}$ is defined.

Strings Assume that e and e' are expressions such that $[\![e]\!]_{G,u}$ and $[\![e']\!]_{G,u}$ belong to $\Sigma^* \cup \{\texttt{null}\}$.

- $$[\![e\ \texttt{STARTS WITH}\ e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } \exists s, [\![e]\!]_{G,u} = [\![e']\!]_{G,u} \cdot s \\ \texttt{null} & \text{if } [\![e]\!]_{G,u} = \texttt{null} \\ & \text{or } [\![e']\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{otherwise} \end{cases}$$
- $$[\![e\ \texttt{ENDS WITH}\ e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } \exists s, [\![e]\!]_{G,u} = s \cdot [\![e']\!]_{G,u} \\ \texttt{null} & \text{if } [\![e]\!]_{G,u} = \texttt{null} \\ & \text{or } [\![e']\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{otherwise} \end{cases}$$
- $$[\![e\ \texttt{CONTAINS}\ e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = s_1 \cdot [\![e']\!]_{G,u} \cdot s_2 \\ & \text{for some strings } s_1, s_2 \\ \texttt{null} & \text{if } [\![e]\!]_{G,u} = \texttt{null} \\ & \text{or } [\![e']\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{otherwise} \end{cases}$$

Logic Assume that e and e' are expressions such that $[\![e]\!]_{G,u}$ and $[\![e']\!]_{G,u}$ both belong to $\{\texttt{true}, \texttt{false}, \texttt{null}\}$.

- $$[\![e\ \texttt{OR}\ e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = \texttt{true} \text{ or } [\![e']\!]_{G,u} = \texttt{true} \\ \texttt{false} & \text{if } [\![e]\!]_{G,u} = [\![e']\!]_{G,u} = \texttt{false} \\ \texttt{null} & \text{otherwise} \end{cases}$$
- $$[\![e\ \texttt{AND}\ e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = [\![e']\!]_{G,u} = \texttt{true} \\ \texttt{false} & \text{if } [\![e]\!]_{G,u} = \texttt{false} \text{ or } [\![e']\!]_{G,u} = \texttt{false} \\ \texttt{null} & \text{otherwise} \end{cases}$$
- $$[\![e\ \texttt{XOR}\ e']\!]_{G,u} = \begin{cases} \texttt{null} & \text{if } [\![e]\!]_{G,u} = \texttt{null} \text{ or } [\![e']\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{if } [\![e]\!]_{G,u} = [\![e']\!]_{G,u} \text{ and } [\![e]\!]_{G,u} \neq \texttt{null} \\ \texttt{true} & \text{otherwise} \end{cases}$$
- $$[\![\texttt{NOT}\ e]\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = \texttt{false} \\ \texttt{false} & \text{if } [\![e]\!]_{G,u} = \texttt{true} \\ \texttt{null} & \text{if } [\![e]\!]_{G,u} = \texttt{null} \end{cases}$$

Value Comparisons

– **Nulls** The rules follow SQL: in an expression, if an argument is **`null`**, then the value of the expression is **`null`**. The semantics of **`IS NULL`** is also the same as for SQL.

- $[\![e \star e']\!]_{G,u}$ = **`null`** if either $[\![e]\!]_{G,u}$ = **`null`** or $[\![e']\!]_{G,u}$ = **`null`**, for $\star \in \{<, <=, >=, >, =, <>\}$.
- $[\![e \texttt{ IS NULL}]\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{if } [\![e]\!]_{G,u} \neq \texttt{null} \end{cases}$
- $[\![e \texttt{ IS NOT NULL}]\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} \neq \texttt{null} \\ \texttt{false} & \text{if } [\![e]\!]_{G,u} = \texttt{null} \end{cases}$

– **Base Types** Assume that both e and e' are expressions such that $[\![e]\!]_{G,u}$ and $[\![e']\!]_{G,u}$ are of the same base type.

- $[\![e = e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = [\![e']\!]_{G,u} \\ \texttt{false} & \text{otherwise} \end{cases}$

– **Identifiers** Assume that both e and e' are expressions such that $[\![e]\!]_{G,u}$ and $[\![e']\!]_{G,u}$ are both node identifiers or both relationship identifiers.

- $[\![e = e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = [\![e']\!]_{G,u} \\ \texttt{false} & \text{otherwise} \end{cases}$

– **Empty maps** Assume that both e and e' are expressions such that both $[\![e]\!]_{G,u}$ and $[\![e']\!]_{G,u}$ are maps, and one of them is map().

- $[\![e = e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = [\![e']\!]_{G,u} = \mathsf{map}() \\ \texttt{false} & \text{otherwise} \end{cases}$

– **Non-empty maps, same number of keys** Assume that $[\![e]\!]_{G,u} = \{k_1 : w_1, \ldots, k_m : w_m\}$ and $[\![e']\!]_{G,u} = \{k'_1 : w'_1, \ldots, k'_m : w'_m\}$, where $k_1, \ldots, k_m, k'_1, \ldots, k'_m$ are keys, and $w_1, \ldots, w_m, w'_1, \ldots, w'_m$ are values, and $m \geq 1$.

- $[\![e = e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } \{k_1, \ldots, k_m\} = \{k'_1, \ldots, k'_m\} \\ & \quad \text{and } [\![e.k_i = e'.k_i]\!]_{G,u} = \texttt{true} \text{ for all } i \leq m \\ \texttt{null} & \text{if } \{k_1, \ldots, k_m\} = \{k'_1, \ldots, k'_m\} \\ & \quad \text{and } [\![e.k_i = e'.k_i]\!]_{G,u} = \texttt{null} \text{ for some } i \leq m \\ & \quad \text{and } [\![e.k_i = e'.k_i]\!]_{G,u} \neq \texttt{false} \text{ for all } i \leq m \\ \texttt{false} & \text{otherwise} \end{cases}$

– **Non-empty maps, different number of keys** Assume that $[\![e]\!]_{G,u} = \{k_1 : w_1, \ldots, k_m : w_m\}$ and $[\![e']\!]_{G,u} = \{k'_1 : w'_1, \ldots, k'_l : w'_l\}$, where $k_1, \ldots, k_m, k'_1, \ldots, k'_l$ are keys, and $w_1, \ldots, w_m, w'_1, \ldots, w'_l$ are values, $m, l \geq 1$, and $m \neq l$. In this case, $[\![e = e']\!]_{G,u} =$ **`false`**.

– **Lists** Assume that both e and e' are expressions such that both $[\![e]\!]_{G,u}$ and $[\![e']\!]_{G,u}$ are list values.

- $[\![e = e']\!]_{G,u} = \texttt{true}$
  where $[\![e]\!]_{G,u} = [\![e']\!]_{G,u} = \mathsf{list}()$
- $[\![e = e']\!]_{G,u} = \texttt{false}$
  where $[\![e]\!]_{G,u} = \mathsf{list}(w_1, \ldots, w_n)$ and $[\![e']\!]_{G,u} = \mathsf{list}(w'_1, \ldots, w'_m)$ and $n \neq m$.
- $[\![e = e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } \forall i, [\![w_i = w'_i]\!]_{G,u} = \texttt{true} \\ \texttt{null} & \text{if the previous case does not hold} \\ & \text{and } \forall i, [\![w_i = w'_i]\!]_{G,u} \in \{\texttt{null}, \texttt{true}\} \\ \texttt{false} & \text{otherwise} \end{cases}$
  where $[\![e]\!]_{G,u} = \mathsf{list}(w_1, \ldots, w_m)$, $[\![e']\!]_{G,u} = \mathsf{list}(w'_1, \ldots, w'_m)$ and $w_1, \ldots, w_m, w'_1, \ldots, w'_m$ are values.

– **Paths** Assume that both e and e' are expressions such that both $[\![e]\!]_{G,u}$ and $[\![e']\!]_{G,u}$ are path values.

- $[\![e = e']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{G,u} = [\![e']\!]_{G,u} \\ \texttt{false} & \text{otherwise} \end{cases}$

– **Mismatched composite types** If e and e' are expressions such that $[\![e]\!]_{G,u}$ is a value of a composite type (map, list, path) and $[\![e']\!]_{G,u}$ is a non-null value of a different type, then $[\![e = e']\!]_{G,u} = \texttt{false}$. Conversely, if $[\![e']\!]_{G,u}$ is of a composite type and $[\![e]\!]_{G,u}$ is a non-null value of a different type, then $[\![e = e']\!]_{G,u} = \texttt{false}$.

– **Base types** If e and e' are expressions such that $[\![e]\!]_{G,u}$ and $[\![e']\!]_{G,u}$ are non-null values of a non-composite type, then $[\![e \star e']\!]_{G,u}$ is allowed to be implementation-dependent, for $\star \in \{<, <=, >=, >\}$. That is, for base types implementations have freedom when it comes to defining ordering. It is assumed however that for types considered here (numerical and strings), these are fixed and have their standard interpretation as ordering on numbers, and lexicographic ordering for strings.

6.2 Semantics of queries

A query is either a sequence of clauses ending with the **RETURN** statement, or a union (set of bag) of two queries. The **RETURN** statement contains the return list, which is either $*$, or a sequence of expressions, optionally followed by **AS** a, to provide their names.

To provide the semantics of queries, we assume that there exists an (implementation-dependent) injective function α that maps expressions to names. Recall that the semantics of both queries and clauses, relative to a property graph G, is a function from tables to tables, so we shall describe its value on a table T, i.e., $[\![\mathsf{query}]\!]_G(T)$.

Return We make the following assumptions. First, the fields of T are $b_1, \ldots, b_q$. Second, if we have a return list e_1 [**AS** a_1], ..., e_m [**AS** a_m] with optional **AS** for some of the expressions, then $a'_i = a_i$ if **AS** a_i is present in the list, and $a'_i = \alpha(e_i)$ otherwise, with the added requirement that all the a'_is are distinct. In some rules for the semantics, some **AS** could be optional. It is assumed that when such optional **AS** is present on the left side, then it is also present on the right hand side.

- $[\![$**RETURN** $*]\!]_G(T) = T$ if T has a least one field
- $[\![$**RETURN** $*, e_1$ [**AS** a_1], ..., e_m [**AS** a_m]$]\!]_G(T) =$
  $[\![$**RETURN** b_1 **AS** $b_1, \ldots, b_q$ **AS** b_q, e_1 [**AS** a_1], ..., e_m [**AS** a_m]$]\!]_G(T)$
- $[\![$**RETURN** e_1 [**AS** a_1], ..., e_m [**AS** a_m]$]\!]_G(T) =$
  $$\biguplus_{u \in T} \left\{ (a'_1 : [\![e_1]\!]_{G,u}, \ldots, a'_m : [\![e_m]\!]_{G,u}) \right\}$$

Union Let Q_1, Q_2 be queries.

- $[\![Q_1$ **UNION ALL** $Q_2]\!]_G(T) = [\![Q_1]\!]_G(T) \cup [\![Q_2]\!]_G(T)$
- $[\![Q_1$ **UNION** $Q_2]\!]_G(T) = \varepsilon\big([\![Q_1]\!]_G(T) \cup [\![Q_2]\!]_G(T)\big)$,
  where Recall that $\varepsilon(T)$ denotes the result of duplicate elimination on T.)

Clause list

- $[\![C\ Q]\!]_G(T) = [\![Q]\!]_G\big([\![C]\!]_G(T)\big)$
  where C is a clause and Q is a query.

6.3 Semantics of clauses

The meaning of Cypher clauses is again functions that take tables to tables. Matching clauses are essentially pattern matching statements: they are of the form **OPTIONAL MATCH** pattern_tuple **WHERE** expr. Both **OPTIONAL** and **WHERE** could be omitted. The key to their semantics is pattern matching, in particular $\mathsf{match}(\bar{\pi}, G, u)$ described in Section 4 (see Equation (1), page 12).

The **MATCH** clause extends the set of field names of T by adding to it field names that correspond to names occurring in the pattern but not in u. It also adds tuples to T, based on matches of the pattern that are found in graphs. **UNWIND** is another clause that expands the set fields, and **WITH** clauses can change the set of fields to any desired one. The **WHERE** subclause also defines a table-to-tables function that filters lines according to the evaluation of an expression; it is not a proper clause because of its interaction of with **OPTIONAL MATCH** clauses.

Matching clause The semantics of **MATCH** clauses is defined below; the semantics of **WHERE** subclause is defined afterwards.

- $[\![$**MATCH** $\bar{\pi}]\!]_G(T) = \biguplus_{u \in T} \{u \cdot u' \mid u' \in \mathsf{match}(\bar{\pi}, G, u)\}$

- $[\![\textbf{MATCH}\ \bar{\pi}\ \textbf{WHERE}\ e]\!]_G(T) = [\![\textbf{WHERE}\ e]\!]\Big([\![\textbf{MATCH}\ \bar{\pi}]\!]_G(T)\Big)$
- $$[\![\textbf{OPTIONAL MATCH}\ \bar{\pi}\ \textbf{WHERE}\ e]\!]_G(T) = \biguplus_{u\in T} \begin{cases} [\![\textbf{MATCH}\ \bar{\pi}\ \textbf{WHERE}\ e]\!]_G(\{u\}) & \text{if } [\![\textbf{MATCH}\ \bar{\pi}\ \textbf{WHERE}\ e]\!]_G(\{u\}) \neq \varnothing \\ (u, (\text{free}(u, \bar{\pi}) : \textbf{null})) & \text{otherwise} \end{cases}$$
- $[\![\textbf{OPTIONAL MATCH}\ \bar{\pi}]\!]_G(T) = [\![\textbf{OPTIONAL MATCH}\ \bar{\pi}\ \textbf{WHERE true}]\!]_G(T)$

Example 6. *Let G be the property graph defined in Figure 3. consider the clause* **MATCH** π*, where π is the pattern*

```
(x) -[:KNOWS*]-> (y)
```

Let T be the table $\{(x : n_1); (x : n_3)\}$ with a single field x. We show how to compute $[\![\textbf{MATCH}\ \pi]\!]_G(T)$.

Note that $\mathsf{rigid}(\pi)$ *is the (infinite) set of all rigid paths $\pi_m = (\rightarrow, \mathsf{nil}, \{\text{KNOWS}\}, m, m)$, for $m > 0$. These can only be satisfied by paths with exactly m distinct relationships. Since G only contains 3 relationships, only π_1, π_2 and π_3 can contribute to the result.*

Let $u = (x : n_1)$, $\pi' = \pi_1$ and $p = n_1 r_1 n_2$. Then $\mathsf{free}(\pi_1) - \mathsf{dom}(u) = \{y\}$, and thus u' must be a record over the field y. One can easily check that $(n_1 r_1 n_2, G, (x : n_1, y : n_2)) \models \pi_1$. In fact n_2 is the only suitable value for y, and thus the contribution of this specific triple u, π', p to the final result is precisely $\{(x : n_1, y : n_2)\}$.

No path p other than $n_1 r_1 n_2$ can contribute a record in the case where $u = (x : n_1)$ and $\pi' = \pi_1$. Indeed, π_1 requires p to be of length 1, and start at x, which u evaluates to be n_1. By a similar reasoning, we can compute the contribution of the following triples:

- $(x : n_1, y : n_3)$, π_2, $n_1 r_1 n_2 r_2 n_3$ *yields* $(x : n_1, y : n_3)$;
- $(x : n_1, y : n_4)$, π_3, $n_1 r_1 n_2 r_2 n_3 r_3 n_4$ *yields* $(x : n_1, y : n_4)$;
- $(x : n_3, y : n_4)$, π_1, $n_3 r_3 n_4$ *yields* $(x : n_3, y : n_4)$;

and show that the contributions of all other possible combinations of records, paths and patterns are empty. This tells us that $[\![\textbf{MATCH}\ \pi]\!]_G(T)$ *is the following table:*

x	y
n_1	n_2
n_1	n_3
n_1	n_4
n_3	n_4

Where subclause Although **WHERE** is not a clause per say, its semantics is also a table to table function.

- $[\![\textbf{WHERE}\ e]\!]_G(T) = \{u \in T \mid [\![e]\!]_{G,u} = \textbf{true}\}$

With clause Similarly to the description of the semantics of **RETURN** queries, we make the assumption that the fields of T are $b_1, \ldots, b_q$. Our convention about the names a'_i are exactly the same as for queries (see above), except that $a'_i = \alpha(e_i)$ only if e_i is a name.

- $[\![\textbf{WITH } *]\!]_G(T) = T$
  if T has a least one field
- $[\![\textbf{WITH } e_1 \ [\textbf{AS } a_1], \ldots, e_m \ [\textbf{AS } a_m]]\!]_G(T)$
  $$= \biguplus_{u \in T} \{(a'_1 : [\![e_1]\!]_{G,u}, \ldots, a'_m : [\![e_m]\!]_{G,u})\}$$
- $[\![\textbf{WITH } *, e_1 \ [\textbf{AS } a_1], \ldots, e_m \ [\textbf{AS } a_m]]\!]_G(T)$
  $$= [\![\textbf{WITH } b_1 \textbf{ AS } b_1, \ldots, b_q \textbf{ AS } b_q, e_1 \ [\textbf{AS } a_1], \ldots, e_m \ [\textbf{AS } a_m]]\!]_G(T)$$
- $[\![\textbf{WITH ret WHERE } e]\!]_G(T) = [\![\textbf{WHERE } e]\!]_G\Big([\![\textbf{WITH ret}]\!]_G(T)\Big)$

Unwind Clause

- $[\![\textbf{UNWIND } e \textbf{ AS } a]\!]_G(T) = \biguplus_{u \in T} \biguplus_{v \in E_u} \{(u, a : v)\}$,
  $$\text{where } \ E_u = \begin{cases} \biguplus_{0 \le i < m} \{v_i\} & \text{if } [\![e]\!]_{G,u} = \text{list}(v_0, \ldots, v_{m-1}) \\ \{\} & \text{if } [\![e]\!]_{G,u} = \text{list}() \\ \{[\![e]\!]_{G,u}\} & \text{otherwise} \end{cases}$$

References

[1] Harold Abelson et al. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[2] Mar Cabra. How the ICIJ used Neo4j to unravel the Panama Papers. Neo4j Blog, May 2016. `https://neo4j.com/blog/icij-neo4j-unravel-panama-papers/`.

[3] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[4] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 510–524. ACM, 2017.

[5] Georgios Drakopoulos, Andreas Kanavos, and Athanasios K. Tsakalidis. Evaluating twitter influence ranking with system theory. In *Proceedings of the 12th International Conference on Web Information Systems and Technologies, WEBIST 2016, Volume 1, Rome, Italy, April 23-25, 2016*, pages 113–120, 2016.

[6] Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1):27–39, 2017.

[7] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In *Computer Science Logic*, pages 274–308, 1992.

[8] Nathan Hawes, Ben Barham, and Cristina Cifuentes. FrappÉ: Querying the linux kernel dependency graph. In *Proceedings of the GRADES'15*, GRADES'15, pages 4:1–4:6. ACM, 2015.

[9] Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, and David Domínguez-Sal. Introduction to graph databases. In *Reasoning Web*, volume 8714 of *Lecture Notes in Computer Science*, pages 171–194. Springer, 2014.

[10] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016.

[11] Artem Lysenko, Irina A. Roznovat, Mansoor Saqi, Alexander Mazein, Christopher J. Rawlings, and Charles Auffray. Representing and querying disease networks using graph databases. *BioData Mining*, 9(1):23, Jul 2016.

[12] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1990.

[13] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[14] Nikolaos Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, NTUA, 253pp, 1998.

[15] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. O'Reilly Media, 2013.

[16] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. Qex: Symbolic SQL query explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 425–446, 2010.