TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

KNOWLEDGE GRAPHS

Lecture 11: Querying Property Graphs with Cypher

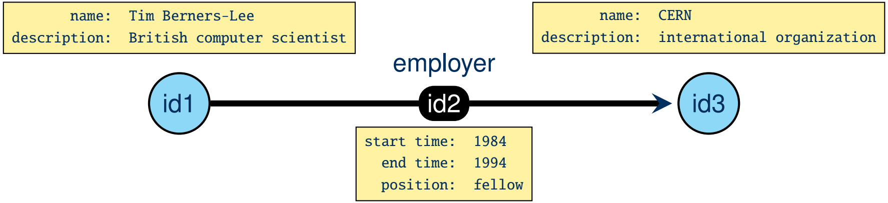
Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 21st Dec 2021

Review

Property Graph is a graph format that supports annotations as second-grade pieces of information in graphs:



Cypher is used as a query language in some property graph systems.

Finding pairs of siblings:

```
MATCH
  (parent)-[:HAS_CHILD]->(child1),
  (parent)-[:HAS_CHILD]->(child2)
WHERE id(child1) <> id(child2)
RETURN child1, child2
```

The shape of a Cypher query

Cypher queries are organised in blocks, called **clauses**:

- Match clause: `MATCH` followed by a pattern; variants of this clause are `OPTIONAL MATCH` and `MANDATORY MATCH`
- Where clause: `WHERE` followed by a filter expression; usually associated with the preceding match clause
- With clause: `WITH` followed by (possibly aliased) expressions and aggregates; ends a subquery
- Return clause: `RETURN` followed by (possibly aliased) expressions and aggregates; occurs once at the end of the query
- Modifier sub-clauses: `ORDER BY`, `LIMIT`, and `SKIP` might follow a With or Return clause
- Union clauses: keyword `UNION` can be used between two complete queries (with `RETURN` for each)

Syntactically, queries are not nested but chained. Many `MATCH-WHERE-WITH` blocks may occur. The order of clauses affects the semantics of queries, but implementations can still evaluate in modified order (as long as the meaning remains the same).

Node patterns

The most basic pattern in Cypher describes a single node.

Definition 11.1: A node pattern is denoted by a pair of parentheses `()`. It may optionally contain the following optional components:

- a variable name (a string)
- a list of labels (a list of strings, each prefixed by `:`)
- a set of properties (a comma-separated list of key:value pairs in `{...}`)

Variable names, labels and property keys are quoted with backticks ` (can be omitted if only alphanumeric characters are used). Property values that are strings are written in straight quotes '.

Example 11.2:

- `()`: an arbitrary node
- `(v { name:'Melitta Bentz', `year of birth`: 1873})`: a node with two properties (and maybe others); matching nodes are bound to variable `v`
- `(:Scientist:Composer)`: a node with two labels

Path patterns

Node patterns are a basic building block of path patterns:

Definition 11.3: A path pattern is a sequence of one or more node patterns, separated by expressions of the form `-[...]->` (forward) or `<-[...]-` (backward) or `-[...]-` (bidirectional), where the expression in `[...]` is an relationship pattern, with the following optional components:

- a variable name (a string)
- a list of relationship types (`:` followed by a `|`-separated list of strings)
- a range literal (`*`, optionally by a number range n`..`m)
- a set of properties (a comma-separated list of key:value pairs in `{...}`)

Example 11.4: The following pattern finds nodes a and b connected with a directed path that consists of between 5 and 10 relationships of types E or F, where b has an incoming relationship e with properties that include `score:0.8`:

```
(a)-[:E|F*5..10]->(b)<-[e {score:0.8}]-()
```

Features of path patterns

The various features of path patterns express the following query conditions:

- Sequences of stylised arrows express linear subgraphs (paths with edges in any direction)
- Arrow tips indicate directionality; patterns without any arrow tip match any direction
- The |-separated list of relationship types expresses a disjunction of possible types: the pattern matches if one of the types is found[1]
- The same property-map syntax as in node patterns are used to require the presence of some properties
- The range with * indicates that the specified kind of relationship has to occur multiple times (within a numeric range, where numbers can be omitted to match any finite path)

Note: * always applies to the complete relationship pattern. For example, the disjunction is nested within this iteration.

[1] Recall that relationships in this interpretation of Property Graph can only have one type.

Expressivity of path patterns

In contrast to SPARQL, Cypher does not support arbitrary regular expressions, but it can still capture certain regular languages:

Example 11.5: The SPARQL property path pattern `?a (p|^p|q|^p)* ?b` can be expressed by the Cypher path pattern `(a)-[:p|q*]-(b)`.

Example 11.6: The SPARQL property path pattern `?a (p/q)* ?b` cannot be expressed in Cypher.

Example 11.7: The SPARQL property path pattern `?a (p|^q)* ?b` cannot be expressed with Cypher path patterns, but it can be expressed in Cypher by a combination of other features.

Graph patterns

Path patterns can be combined conjunctively.

Definition 11.8: A graph pattern is a comma-separated list of path patterns.

Cypher graph patterns are similar to SPARQL basic graph patterns (with property path patterns):

- SPARQL bnodes correspond to Cypher node patterns without variable
- Path patterns are based on similar but slightly distinct features
- Individual path pattern results can be combined with join-like operations to compute overall results

Graph patterns

Path patterns can be combined conjunctively.

Definition 11.8: A graph pattern is a comma-separated list of path patterns.

Cypher graph patterns are similar to SPARQL basic graph patterns (with property path patterns):

- SPARQL bnodes correspond to Cypher node patterns without variable
- Path patterns are based on similar but slightly distinct features
- Individual path pattern results can be combined with join-like operations to compute overall results

But do their semantics agree?

Cypher matching semantics

Graph pattern matches are based on mapping parts of the pattern to parts of the graph:

- Every node pattern is mapped to a node
- Every path pattern is mapped to an alternating sequence of nodes and relationships
  - Such a sequence is called a path
  - Different matches of the same pattern can include paths of different lengths
  - Each distinct path is a distinct match
- Even the parts that have no variables associated must be matched (similar to bnodes in SPARQL)

Cypher matching semantics

Graph pattern matches are based on mapping parts of the pattern to parts of the graph:

- Every node pattern is mapped to a node
- Every path pattern is mapped to an alternating sequence of nodes and relationships
  - Such a sequence is called a path
  - Different matches of the same pattern can include paths of different lengths
  - Each distinct path is a distinct match
- Even the parts that have no variables associated must be matched (similar to bnodes in SPARQL)

Two special aspects of Cypher:

- Unique edges: In any match of a single graph pattern, each relationship must be used in only one place (but each node can be used many times).
- Path counting: When paths are matched, each path (without repeated relationships) counts as one result.

Homomorphism semantics in Cypher

The restriction that each relationship can be used only once in a pattern match is only enforced within single patterns

Homomorphism semantics in Cypher

The restriction that each relationship can be used only once in a pattern match is only enforced within single patterns

A query may use several clauses to allow for the same edge to be used several times:

Example 11.9: Find all persons who have a daughter and who has a relation with a computer scientist (possibly the same person):

```
MATCH (p:Person)-[:HAS_DAUGHTER]->()
MATCH (p:Person)-[r]-({occupation:"computer scientist"})
Return p.name, type(r)
```

Homomorphism semantics in Cypher

The restriction that each relationship can be used only once in a pattern match is only enforced within single patterns

A query may use several clauses to allow for the same edge to be used several times:

Example 11.9: Find all persons who have a daughter and who has a relation with a computer scientist (possibly the same person):

```
MATCH (p:Person)-[:HAS_DAUGHTER]->()
MATCH (p:Person)-[r]-({occupation:"computer scientist"})
Return p.name, type(r)
```

Using similar ideas as before, we immediately get:

Theorem 11.10: Query matching in Cypher is hard for NP.

Proof sketch: Reduce from graph colouring, using one **MATCH** clause per edge. □

Working with nodes and relationships

Matched nodes and relationships are represented as special types of maps. Their content can be accessed in various ways:

- Syntax "`varname.propertyKey`" is used to access the value of a property (keys using non-alphanumeric symbols must be written in backtics)
- Alternatively, syntax "`varname[expression]`" is used to access property values for the property key returned by evaluating the expression
- The function `id` returns the (numeric) ID of a nodes or relationships
- The function `keys` returns the list of all property keys of a map-like object
- The function `labels` returns the list of labels of a node; the function `type` returns the type of a relationship

Working with paths

Cypher can return not only matched nodes and relationships, but also matched paths.

Example 11.11: Find all ways in which two nodes are connected, and return the list of relationship types in each case:

```
MATCH p= ({name=’node1’})-[*]->({name=’node2’})
RETURN relationships(p)
```

- Paths are represented by lists of nodes and relationships (alternating; starting and ending with a node).
- There are several functions to access the content of paths (and other lists), e.g., `relationships` to extract a sublist of relationships only, or `length` to return its number of `relationships`.
- The syntax “`varname=`” in front of a path pattern indicates that matched paths should be assigned to the variable.

Shortest paths

Cypher provides functions that find shortest paths

Example 11.12: Find all shortest ways in which two nodes are connected, and return the list of relationship types in each case:

```
MATCH p= allShortestPaths(({name='node1'})-[*]->({name='node2'}))
RETURN relationships(p)
```

One can use range delimiters to specify an upper bound for the length of the shortest path:

Example 11.13: Find all shortest ways of length at most 10 in which two nodes are connected, and return the list of relationship types in each case:

```
MATCH p= allShortestPaths(({name='node1'})-[*..10]->({name='node2'}))
RETURN relationships(p)
```

Filter conditions

MATCH clauses can be complemented by **WHERE** clauses that express filters.

Example 11.14: Find nodes with more than one label:

```
MATCH (n)
WHERE size(labels(n)) > 1
```

As in SPARQL, filters declaratively specify part of the query condition:

- they must be placed after the relevant **MATCH** clause
- but they can be evaluated in any order by a database

Filter conditions

MATCH clauses can be complemented by **WHERE** clauses that express filters.

Example 11.14: Find nodes with more than one label:

```
MATCH (n)
WHERE size(labels(n)) > 1
```

As in SPARQL, filters declaratively specify part of the query condition:

- they must be placed after the relevant **MATCH** clause
- but they can be evaluated in any order by a database

According to openCypher v9: “If there is a WHERE clause following the match of a shortestPath, relevant predicates will be included in shortestPath.”

Example 11.15: It is not always clear how to evaluate this efficiently:

```
MATCH p=allShortestPaths((a)-[*]-(b))
WHERE a.someKey + b.someKey < length(p) RETURN p
```

Expressions

`WHERE` clauses can use any expression, which can be constructed using many different functions and operators, such as:

- Any constant datatype literal value or variable
- Values obtained by accessing content of map objects
- Boolean operations `AND`, `OR`, `XOR`, and `NOT`
- Arithmetic operators, such as + and `*`, and other mathematical functions, such as `sin` or `ceil`
- String functions, such as `substring` and `toLower`
- List and map accessing functions, such as `keys` and `size`
- Path patterns (return true or false)

Expressions

WHERE clauses can use any expression, which can be constructed using many different functions and operators, such as:

- Any constant datatype literal value or variable
- Values obtained by accessing content of map objects
- Boolean operations `AND`, `OR`, `XOR`, and `NOT`
- Arithmetic operators, such as + and `*`, and other mathematical functions, such as `sin` or `ceil`
- String functions, such as `substring` and `toLower`
- List and map accessing functions, such as `keys` and `size`
- Path patterns (return true or false)

Example 11.16: Find all nodes that have value `"scientist"` as one of their occupations (recall that property values can be lists), and that do not have a spouse:

```
MATCH (n)
WHERE ("scientist" IN n.occupation) AND NOT ( (n)-[:SPOUSE]-() )
RETURN n
```

Further Cypher Features

Groups and Aggregates

Cypher supports aggregate functions, such as `min`, `avg`, `count`, and also more complex ones like `stDev` (standard deviation, for some statistical assumptions).

Unlike SPARQL and SQL, grouping is implicit: if aggregated and non-aggregated results are defined, the non-aggregated ones will be the keys to group by.

Example 11.17: Show married professors with their number of children:

```
MATCH (prof {occupation: "Professor"})-[:SPOUSE]-()
MATCH (prof)-[:HAS_CHILD]->(child)
RETURN prof, count(child)
```

`DISTINCT` can be used to reduce the collection of expression results to contain no duplicates before aggregation, e.g., `count(DISTINCT child)`

Exercise: is DISTINCT needed in the above example or not?

Subqueries

Subqueries in Cypher are chained rather than nested, using `WITH` clauses as separators that control the passing of bindings.

Example 11.18: Find all universities located in one of the ten largest cities:

```
MATCH (c:City)
WITH c
ORDER BY c.population DESC LIMIT 10
MATCH (u:University)-[:LOCATION]->(c)
RETURN DISTINCT u.name
```

- `WITH` can be followed by sub-clauses for `ORDER BY`, `LIMIT`, and `SKIP`
- `WITH` can use aggregations and `DISTINCT`
- Expressions that are not projected are not accessible after a `WITH` clause

Union

The results of two Cypher queries can be combined on the outermost level.

Example 11.19: Find parent-child pairs in one of two possible encodings:

```
MATCH (parent) -[:HAS_CHILD]-> (child)
RETURN parent, child
UNION
MATCH (parent) <-[:HAS_PARENT]- (child)
RETURN parent, child
```

“The number and the names of the fields must be identical in all queries combined by using UNION.” (like SQL, unlike SPARQL)

`UNION` automatically removes duplicates. For keeping them, `UNION ALL` can be used instead.

Optional matches

Similar to SPARQL's `OPTIONAL`, Cypher supports `OPTIONAL MATCH` for clauses that may add additional information, if available.

Example 11.20: Find parent-child pairs and, optionally, a spouse for the parent:

```
MATCH (parent) -[:HAS_CHILD]-> (child)
OPTIONAL MATCH (parent) -[:SPOUSE]- (spouse)
RETURN parent, child, spouse
```

- If a match cannot be found, then variables will be mapped to null
- Special functions can be used to test for null (e.g., "`WHERE v IS NULL`")
- A graph pattern must match completely, i.e., partial matches will not lead to variable bindings

Error handling in Cypher

Recall: SPARQL used a dedicated “error” value to signal and propagate errors. The value effectively evaluates to false in Boolean expressions.

Cypher uses the special value null to indicate errors

- propagation rules are similar to SPARQL for Boolean operations
- other functions may lead to query errors when given nulls
- but the value cannot be distinguished from “unbound”

Error handling in Cypher

Recall: SPARQL used a dedicated “error” value to signal and propagate errors. The value effectively evaluates to false in Boolean expressions.

Cypher uses the special value null to indicate errors

- propagation rules are similar to SPARQL for Boolean operations
- other functions may lead to query errors when given nulls
- but the value cannot be distinguished from “unbound”

Many other error situations cause the whole Cypher query to fail

- For example, if queries are ordered by values of incomparable types
- Function calls with unsuitable parameters might cause errors
- This can also happen for a perfectly valid query due to unexpected database contents

Error handling in Cypher

Recall: SPARQL used a dedicated “error” value to signal and propagate errors. The value effectively evaluates to false in Boolean expressions.

Cypher uses the special value null to indicate errors

- propagation rules are similar to SPARQL for Boolean operations
- other functions may lead to query errors when given nulls
- but the value cannot be distinguished from “unbound”

Many other error situations cause the whole Cypher query to fail

- For example, if queries are ordered by values of incomparable types
- Function calls with unsuitable parameters might cause errors
- This can also happen for a perfectly valid query due to unexpected database contents

Note: The use of null in Cypher is ambiguous. It is used to express “no result” or “unknown value”, but it is also used to express “error” in function calls. Ordering and comparisons of null values in Cypher are not consistently defined yet.

Summary

Cypher is a query language for property graphs

It is based on clauses (like SQL) and graph patterns (like SPARQL)

Not all regular expressions are supported, but powerful path manipulation features are available.

Graph pattern matching is not based on homomorphisms like in other query languages (SQL, SPARQL, etc.) but on a notion of isomorphim on edges

What’s next?

- Quality assurance in knowledge graphs
- Graph analysis