TECHNISCHE UNIVERSITÄT DRESDEN

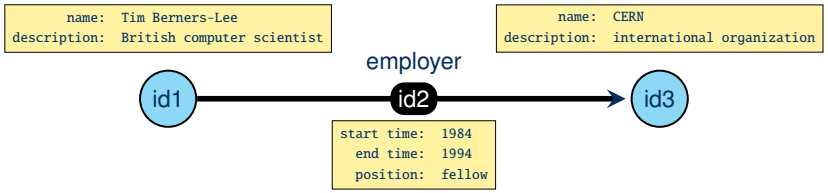
KNOWLEDGE GRAPHS

Lecture 10: Querying Property Graphs with Cypher

Markus Krötzsch
Knowledge-Based Systems

TU Dresden, 18th Dec 2018

Review



From Property Graph to RDF

As expected, every Property Graph can be expressed as an RDF graph:

- Use reification to represent edges by auxiliary nodes
- Use special RDF properties to encode source and target of edges, and the node-label and relationship-relationship type association
- Use application-based RDF properties to encode property keys
- Depending on the exact Property graph implementation, use some appropriate datatype-to-RDF mapping
  (e.g., based on RDF2RDB mappings from SQL datatypes to RDF)

Remarks:

- This scheme is implemented natively in existing DBMS (Blazegraph, Amazon Neptune), and scales despite of the increase size of the graph data that the system has to work with
- The use of reification can be avoided for relationships without properties
- One can also use both (reified and direct statements) for flexibility (similar to Wikidata's RDF encoding)

From RDF to Property Graph

As expected, every RDF Graph can be expressed as a Property Graph:

- Represent all RDF resources by nodes
- Use property keys to associate resources with IRIs and/or datatype literanl information
- Represent all RDF triples with auxiliary nodes
- Use relationships with special relationship types to associate auxiliary nodes with triple subject, predicate, and object

Remarks:

- There does not seem to be any simpler way of capturing the full power of RDF in Property Graph, due to the restrictions of the latter (no reference in relationship types or property values to nodes)
- In some cases, certain triples could be represented as properties (if their datatype is supported in Property Graph and we do not need their RDF property to be addressable in the graph)
- Many Property Graph implementations will have performance problems in handling graphs with so many edges (part of the motivation for moving data into properties is to reduce the graph size)

Cypher

Cypher overview

Cypher is a (family of) query languages for Property Graph:

- Proprietary query language of the Neo4j graph database
- Subset supported by other tools as well: openCypher
- Might be an important input to future graph query language standards

openCypher supports two main functions:

- A query language
- An update language

Current specification of openCypher 9:
`https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf`

Currently not defined in openCypher v9 and depending on implementation:

- Parts of the query language (e.g., comparison and ordering of some values)
- Result formats for sending query results
- Protocol for sending queries and receiving answers

Cypher queries

The heart of Cypher is its query language.

Example 10.1: The following Cypher query asks for a list of all nodes that are in an `EMPLOYER` relationship:

```
MATCH (person)-[:EMPLOYER]->(company)
RETURN person, company
```

This corresponds to the following SPARQL query:

```
SELECT ?person ?company
WHERE { ?person  :EMPLOYER  ?company }
```

Basic concepts:

- Cypher uses variables, marked by their context of use
- The core of a query is the query condition within `MATCH { ... }`
- Conditions can be simple patterns based on graph edges in a custom syntax
- `RETURN` specifies how results are produced from query matches

Basic Cypher by example

Example 10.2: Find up to ten people whose daughter is a professor:

```
MATCH
  (parent)-[:HAS_DAUGHTER]->(child {occupation:'Professor'})
RETURN parent
LIMIT 10
```

Example 10.3: Count all relationships in the database:

```
MATCH ()-[relationship]->()
RETURN count(relationship) AS count
```

Example 10.4: Count all relationship types in the database:

```
MATCH ()-[relationship]->()
RETURN count(DISTINCT type(relationship)) AS count
```

Basic Cypher by example (2)

Example 10.5: Find the person with most friends:

```
MATCH (person)-[:HAS_FRIEND]->(friend)
RETURN person, count(DISTINCT friend) AS friendCount
ORDER BY friendCount DESC
LIMIT 1
```

Example 10.6: Find pairs of siblings:

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


Basic Cypher by example (3): properties and labels

Queries can also access

- Labels (the additional strings used on nodes)
- Properties (of nodes and relationships)

Example 10.7: Find friends of all people with name Paul Erdős, and return their name and the start date of the friendship:

```
MATCH
(:Human {name: 'Paul Erdős'})-[rel:HAS_FRIEND]->(friend:Human)
RETURN friend.name, rel.startDate
```

Here `Human` is a label, and `name` and `startDate` are property keys.

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 10.8: 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 10.9:

- `()`: 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 10.10: 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 10.11: 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 10.12: The SPARQL property path pattern `?a (p|^p|q|^p)* ?b` can be expressed by the Cypher path pattern `(a)-[:p|q*]-(b)`.

Example 10.13: The SPARQL property path pattern `?a (p/q)* ?b` cannot be expressed in Cypher.

Example 10.14: 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 10.15: 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)

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

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

Example 10.16: 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 10.17: 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 10.18: 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 10.19: 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 10.20: 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 10.21: 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 10.22: It is not always clear how to evaluate this efficiently:

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


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?

- Holidays
- Final remarks on Cypher
- Quality assurance in knowledge graphs

Here's wishing you

a Merry Christmas, a Happy Hanukkah,

a Joyous Yalda, a Cheerful Dōngzhì,

a Great Feast of Juul,

and a Wonderful Winter Solstice,

respectively!