TECHNISCHE
UNIVERSITÄT
DRESDEN

KNOWLEDGE GRAPHS

Lecture 11: Cypher / Knowledge Graph Quality

Markus Krötzsch
Knowledge-Based Systems

TU Dresden, 8th Jan 2019

Review: The Cypher Query Language

Example: Find pairs of siblings:

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

Cypher features

- Graph patterns that can be matched to (sets of) paths in graphs
- Querying for and counting of paths
- Various filter conditions (**WHERE**)
- Expressions for accessing aspects of the Property Graph model

Filter conditions

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

Example 11.1: 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 11.2: 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)

Example 11.3: 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, grouping is implicit: if aggregated and non-aggregated results are defined, the non-aggregated ones will be the keys to group by.

Example 11.4: 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 11.5: 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 11.6: 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." (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 11.7: 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"

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.

Knowledge Graph Quality

Motivation

The quality of KGs is an aspect of utmost importance:

- Almost any data can be turned into a graph format and called "knowledge graph"
- When switching to graphs, we need to know if it was a success or failure
- As time passes, the quality of the KG must be monitored

What does quality mean in this context?
How can it be measured?
How can it be monitored automatically?

Can quality be measured?

A project manager's dream: capture the objective quality as a single numeric value

- easy to communicate; fits into business culture
- money (cost & revenue) is also a number
- extends to valuation of employees (compare salary with quality produced)

However, the world is not that simple:

- Quality is multi-dimensional and non-linear
- Usually not metric ("We improved by 12.6%") but merely ordinal ("We improved a lot")
- Relationship to (metric) cost/revenue numbers all but clear

What is quality in KGs?

There are two prevailing dimensions of quality:

1. **Functional requirements:** the KG supports the envisioned application
   - contains necessary information (topical, accurate, complete, ...)
   - free of errors (correct, up-to-date)
   - accessible in intended ways/using intended tools, with sufficient performance
2. **Non-functional requirements:** the KG is well built
   - adheres to style guides (choice of identifiers, usage of syntax features, ...)
   - includes documentation, esp. regarding modelling approach
   - comes with useful schema information (declarations, constraints, ontologies, ...)
   - is internally consistent and non-redundant
   - based on mature technologies/standards

Functional and non-functional requirements are rarely independent

Example: TimBL's Open Data Quality proposal

Tim Berners-Lee in 2006 proposed a 5-star quality metric for open data:

⋆ make your stuff available on the Web (whatever format) under an open license
⋆ ⋆ make it available as structured data (e.g., Excel instead of image scan of a table)
⋆ ⋆ ⋆ make it available in a non-proprietary open format (e.g., CSV instead of Excel)
⋆ ⋆ ⋆ ⋆ use URIs to denote things, so that people can point at your stuff
⋆ ⋆ ⋆ ⋆ ⋆ link your data to other data to provide context

Notes:

- ⋆ is not about data quality
- All other criteria are non-functional
- The criteria are ordinal, not metric, and no means of estimating partial progress is given (esp. for ⋆ ⋆ ⋆ ⋆ ⋆)
- The term Linked Open Data refers to 5-⋆ RDF data with resolvable URIs

KGs as software?

KG as digital artefact are similar to software, so similar methods and criteria might apply.

But there are important differences:

- development process (many editors, often weakly coordinated; data imports; unclear development life cycle)
- lack of modularisation/interfaces/separation of concerns (integration vs. separation of knowledge)
- KGs have no fully self-contained function: they need to be used by some software
- KGs may be intended for multiple or yet unknown functions

And measuring software quality is already a difficult issue . . .

Checking instead of measuring

Summary: Quality is difficult to measure, and the choice of concrete quality measures is always subjective

Way forward:

- We will focus on methods of checking specific quality criteria
- The outcomes of many checks can be quantified to obtain measures
- One can aggregate measures by (subjective) weighting functions, or analyse them as multi-dimensional aspects of the KG's status

Quality checking: basic approaches

Quality criteria can be assessed in various ways:

- **Manual:** checks performed by human experts
  - Subjective: Based on expert assessment

    Example: Interview domain experts for completeness and correctness.
  - Objective: Based on clearly defined criteria

    Example: Define use cases (user stories) and check if they can been realised in the application context.
- **Automated:** checks run by computers
  - Operational: Ad hoc implementation of quality checks

    Example: Script that retrieves matching data from a third-party database and compares its values with the data in the KG.
  - Declarative: Specification of quality criteria in some formal language that can be interpreted by (standard) tools

    Example: Schema document that constrains syntactic form of KG.

Declarative quality checks for KGs

The distinction between "operational" and "declarative" is often fuzzy

- A testing script is operational: implementation-specific meaning; not portable
- A schema document in a standard language (e.g., XML Schema) is declarative: meaning standardised and understood by many tools
- Many other approaches are in between:
  - Graph database constraints: possibly proprietary language; may or may not be portable
  - Wikidata property constraints: community-developed approach for expressing schema information in data
  - SPARQL test queries: declarative queries used in some operational wrapper that validates results
  - Business rules: rule-based programs that interpreted by proprietary software
  - ...

⇝ declarativity is not an rigorously defined feature, but an ideal to strive for

Competency questions

A classical approach of knowledge model evaluation are so-called competency questions

Definition 11.8: A competency question is a (usually application-related) question towards the KG that is formalised in a query language, together with a formal specification of how an acceptable answer may look.

A competency question does not need to pre-determine the KG data in all detail.

Example 11.9: We can specify that Wikidata should "know" that humans (`Q5`) are mammals (`Q7377`) by requiring that the query `SELECT * WHERE { wd:Q5 wdt:P171* wd:Q7377 }` returns a non-empty result ("true"). This query leaves empty how the taxonomic hierarchy is modelled.

Competency questions

Competency questions focus on functional metrics:

- coverage/completeness (but cannot check all cases)
- correctness
- accessibility (using query answering software)

They can be used in several situations:

- To define the initial scope (requirements) of a new KG project
- To formalise data modelling decisions (how should knowledge be encoded to be accessible)
- For regression testing (ensure that KG does not break in the future)

However, there are also costs: modelling effort, maintenance, . . .

Unit testing

Competency questions take a content-oriented view (application- and domain-specific), but the approach can be generalised to set up unit testing:

- Define a test suite of queries + (constraints on) expected answers
- Automatically run queries to detect problems

Unit tests can also validate non-functional criteria.

Schema languages

The most formal way of defining quality criteria is by specifying structural requirements in a formal schema language

Example 11.10: XML Schema is a classical schema language to constrain the syntactic form of XML documents (DTD is another, older, approach with similar goals).

For RDF, there are mainly two schema languages today:

- SHACL, a W3C standard (since 2017)
- ShEx, a W3C member submission and community group effort

Note: RDF Schema, despite its name, is a lightweight ontology language rather than a schema language.

SHACL

SHACL is the W3C Shapes Constraint Language

Basic principles:

- Overall approach similar to query-based unit testing
- SHACL shapes specify constraints by defining two aspects:
  (1) a pattern that RDF graph a nodes may match (akin to simple queries),
  (2) and the set of target nodes that should match the pattern
- SHACL-SPARQL extension allows using SPARQL for pattern specification
- Shapes can have meta-data to define, e.g., error messages and severity levels
- Shapes and sets of shapes are encoded in RDF as shape graphs

Further reading:

- W3C SHACL Recommendation at `https://www.w3.org/TR/shacl/`
- Labra Gayo, Prud'hommeaux, Boneva, Kontokostas: Validating RDF Data (Morgan Claypool 2018); see `https://book.validatingrdf.com/`

SHACL by Example

The following RDF graph (in Turtle, without prefixes) defines a shape `ex:PersonShape`:

```
ex:PersonShape rdf:type sh:NodeShape ;
  sh:targetClass ex:Person ; # Applies to all persons
  sh:property [ # Declare a constraint on property usage
    sh:path ex:ssn ; # ... for property ex:ssn
    sh:maxCount 1 ; # ... at most one value
    sh:datatype xsd:string ; # ... having type string
    sh:pattern "^\\d{3}-\\d{2}-\\d{4}$" # ... matching this regexp
  ] ;
  sh:property [ # Declare another property constraint
    sh:path ex:worksFor ; # ... for property ex:worksFor
    sh:nodeKind sh:IRI ; # ... values are IRIs
    sh:class ex:Company # ... of rdf:type ex:Company (or a subclass)
  ] ;
  sh:closed true ; # No other properties are allowed
  sh:ignoredProperties ( rdf:type ) . # ... except for rdf:type
```

Shapes in SHACL

. . . are identified by IRIs and may optionally include:

- a specification of target nodes they apply to (`sh:targetClass`, `sh:targetSubjectsOf`, `sh:targetObjectsOf`, or `sh:targetNode`)
- a set of property shapes that define constrains on values reached through (paths of) properties
- constraints on whether the shape is closed
- non-validating constraints, e.g., `sh:description`

There is a rich vocabulary for specifying property constraints, including:

- (SPARQL-like) property paths instead of single properties
- minimal and maximal cardinalities
- resource types, datatypes, or RDF classes for values
- lists of admissible values
- ways to say that one property's values are disjoint or equal to another's
- (possibly recursive) reference to the NodeType of property values
- Boolean combinations of constraints (and, or, not)

ShEx

ShEx is the Shape Expressions Language as proposed by a W3C Community Group

Basic principles:

- Overall approach similar matching a grammar description to a graph
- ShEx shapes specify constraints by defining a pattern that refers to
  (1) required features of the RDF graph
  (2) required patterns matched by adjacent nodes (recursively)
- Validation tries to consistently map nodes in an RDF graphs to types as required (based on some initial map)
- Sets of shapes form a schema, encoded in an RDF-inspired own syntax

Further reading:

- ShEx community homepage at `http://shex.io/`
- Labra Gayo, Prud'hommeaux, Boneva, Kontokostas: Validating RDF Data (Morgan Claypool 2018); see `https://book.validatingrdf.com/`

ShEx by Example

The following defines a shape `a:PersonShape` (without prefix declarations), which is functionally exquivalent to the previous SHACL example:

```
a:PersonShape CLOSED EXTRA rdf:type {
  ex:ssn  xsd:string /^\\d{3}-\\d{2}-\\d{4}$/ ? ;
  ex:worksFor  IRI @a:CompanyShape * ;
}
a:CompanyShape [ ex:Company ] OR { rdfs:subClassOf @:CompanyShape }
```

Notes:

- **CLOSED** and **EXTRA** play the role of `sh:closed` and `sh:ingoredProperties`
- "property shapes" are compactly expressed in single lines
- taking indirect typing (instances of subclasses) into account requires the use of recursive shape definitions

Validating SHACL and ShEx

Both approaches support recursive constraints and disjunctions:

- Node types are not part of the data: their recursive use means that validation has to extend shapes to new target nodes
- Disjunction means that this assignment might be non-determinstic

⇝ worst-case NP-complete complexity in the size of the graph (data complexity!)

However:

- In SHACL, recursive assignments are a minor feature, and the specification does not specify their semantics. Selection of target nodes mostly governed by conditions on RDF.
- In ShEx, type maps are the only mechanism for selecting targets, and recursive assignments are necessary to check indirect class membership

⇝ NP-completeness seems more challenging for ShEx than for SHACL

Since SHACL is mostly deterministic, it can also provide detailed error reports in case of failing constraints (this is hard if many assignments need to be considered which may all fail, but for different reasons)

Summary

OpenCypher supports aggregates, subqueries, unions, and optional matches (like SPARQL, but differing in some design choices)

Defining and measuring knowledge graph quality is difficult; there are many criteria

Competency questions and unit tests are basic approaches for automatic quality checks

RDF constraint languages like SHACL and ShEx can declaratively specify constraints

What's next?

- Ontological modelling
- More graph analysis
- Examinations