Submitted to: BTW 2017

Formalizing openCypher Graph Queries in Relational Algebra

Gábor Szárnyas[1] József Marton[2]

Abstract: The last decade brought considerable improvements in non-relational storage and query technologies, known collectively as NoSQL systems. Most of these systems were designed to suit the needs of Big Data and real-time web applications, hence they use specialized data models: key-value, document, wide columns, graph databases, etc. While they offer high performance for their specific use cases, a major shortcoming of these systems is the lack of standardization. Consequently, migrating datasets or applications between technologies often requires a large amount of manual work or ad-hoc solutions, thus subjecting the users to the possibility of vendor lock-in.

To avoid this threat, NoSQL vendors are working on supporting existing standard languages (e.g. SQL) and introducing standardized languages. In this paper, we present formal definitions for openCypher, a high-level declarative graph query language with an on-going standardization effort.

1 Introduction

Context. Graphs are a well-known formalism, widely used for describing and analysing systems. Graphs provide an intuitive formalism for modelling real-world scenarios, as the human mind tends to interpret the world in terms of objects (*vertices*) and their respective relationships to one another (*edges*) [Ro08].

The *property graph* data model [RN11] extends graphs by adding labels and properties for both vertices and edges. This gives a rich set of features for users to model their specific domain in a natural way. Graph databases are able to store property graphs and query their contents with complex graph patterns, which, otherwise would be are cumbersome to define and/or inefficient to evaluate on traditional relational databases and query technologies.

Neo4j [Ne], a popular NoSQL property graph database, offers the Cypher [Ne16a] query language to specify graph patterns. Cypher is a high-level declarative query language which can be optimised by the query engine. The openCypher project [Ne16b] is an initiative of Neo Technology, the company behind Neo4j, to deliver an open specification of Cypher.

Problem and objectives. The openCypher project features a formal specification of the grammar of the query language (Sect. 3) and a set of acceptance tests that define the behaviour of various Cypher features. However, there is no mathematical formalisation for any of the language features. In ambiguous cases, the user is advised to consult Neo4j's

[1] Budapest University of Technology and Economics, Fault Tolerant Systems Research Group, MTA-BME Lendület Research Group on Cyber-Physical Systems, szarnyas@mit.bme.hu
[2] Budapest University of Technology and Economics, Database Laboratory, marton@db.bme.hu

Cypher documentation or to experiment with Neo4j's Cypher query engine and follow its behaviour. Our goal is to provide a formal specification for the core features of openCypher.

Contributions. In this paper, we use a formal definition of the property graph data model [HG16] and an extended version of relational algebra, operating on multisets (bags) and featuring additional operators [GUW09]. These allow us to construct a concise formal specification for the core features in the openCypher grammar, which can then serve as a basis for implementing an openCypher-compliant query engine.

2 Preliminaries

This section defines the mathematical concepts used in the paper. Our notation closely follows [HG16] and is similar to [RN11][3].

2.1 Property Graph Data Model

A *property graph* is defined as $G = (V, E, \mathsf{src\_trg}, L_v, L_e, l_v, l_e, P_v, P_e)$, where V is a set of vertices, E is a set of directed edges, $\mathsf{src\_trg} : E \to V \times V$ assigns the source and target vertices to edges. The graph is labelled (or typed):

- L_v is a set of vertex labels, $l_v : V \to 2^{L_v}$ assigns *a set of labels* to each vertex.
- L_e is a set of edge labels, $l_e : E \to L_e$ assigns *a single label* to each edge.

Furthermore, graph G has properties (*attributed graph*). Let D be a set of atomic domains.

- P_v is a set of vertex properties. A vertex property $p_i \in P_v$ is a function $p_i : V \to D_i \cup \{\varepsilon\}$, which assigns a property value from a domain $D_i \in D$ to a vertex $v \in V$, if v has property p_i, otherwise $p_i(v)$ returns ε.
- P_e is a set of edge properties. An edge property $p_j \in P_e$ is a function $p_j : E \to D_j \cup \{\varepsilon\}$, which assigns a property value from a domain $D_j \in D$ to an edge $e \in E$, if e has property p_j, otherwise $p_j(e)$ returns ε.

Running example. Fig. 1 presents an example inspired by the Movie Database dataset[4].

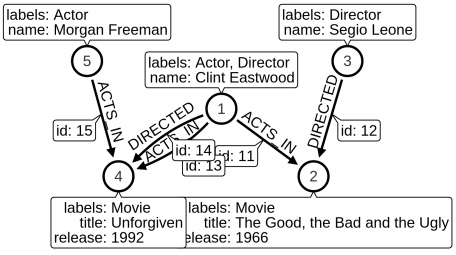

Fig. 1: Example movie graph.

$V = \{1,2,3,4,5\}; E = \{11,12,13,14,15\};$
$\mathsf{src\_trg}(11) = \langle 1,2\rangle; \mathsf{src\_trg}(12) = \langle 3,2\rangle; \ldots$
$L_v = \{\mathsf{Actor}, \mathsf{Director}, \mathsf{Movie}\};$
$L_e = \{\mathsf{ACTS\_IN}, \mathsf{DIRECTED}\};$
$l_v(1) = \{\mathsf{Actor}, \mathsf{Director}\}; l_v(2) = \{\mathsf{Movie}\}; \ldots;$
$l_e(11) = \mathsf{ACTS\_IN}; l_e(12) = \mathsf{DIRECTED}; \ldots;$
$P_v = \{\mathsf{name}, \mathsf{title}, \mathsf{release}\}; P_e = \{\};$
$\mathrm{name}(1) = \text{'Clint Eastwood'}; \mathrm{name}(2) = \varepsilon; \ldots$
$\mathrm{title}(1) = \varepsilon; \mathrm{title}(2) = \text{'The Good, the Bad and the Ugly'}; \ldots$
$\mathrm{release}(1) = \varepsilon; \mathrm{release}(2) = 1966; \ldots$

Fig. 2: The dataset represented as a property graph.

[3] The formalism presented in [RN11] lacks the notion of *vertex labels*.
[4] https://neo4j.com/developer/movie-database/

In the context of this paper, we define a *relation* as a bag (*multiset*) of tuples: a tuple can occur more than once in the relation [GUW09]. Given a property graph G, relation r is a *graph relation* if the following holds:

$$\forall A \in \text{attr}(r) : \text{dom}(A) \subseteq V \cup E \cup D,$$

where $\text{attr}(r)$ is the set of attributes of r, $\text{dom}(A)$ is the domain of attribute A. The schema of r, $\text{sch}(r)$ is a list containing the attribute names. For schema transformations, the *append* operator is denoted by $\|$, the *remove* operator is denoted by $-$.

2.2 Foundations of Relational Algebra

Basic operators of relational algebra. We give a brief summary of the operators in relational algebra. A more detailed discussion is available in database textbooks, e.g. [EN00].

Unary operators. The *projection* operator π keeps a specific set of attributes in the relation: $t = \pi_{\mathsf{A}_1,\ldots,\mathsf{A}_n}(r)$. Note that the tuples are not deduplicated by default, i.e. the results will have the same number of tuples as the input relation r. The projection operator can also rename the attributes, e.g. $\pi_{\mathsf{v1}\to\mathsf{v2}}(r)$ renames $\mathsf{v1}$ to $\mathsf{v2}$. The *selection* operator σ filters the incoming relation according to some criteria. Formally, $t = \sigma_\theta(r)$, where predicate θ is a propositional formula. The operator selects all tuples in r for which θ holds.

Binary operators. The $\cup$ operator produces the set union of two relations, while the $\uplus$ operator produces the *bag union* of two operators, e.g. $\{\langle 1,2\rangle, \langle 1,2\rangle, \langle 3,4\rangle\} \uplus \{\langle 1,2\rangle\} = \{\langle 1,2\rangle, \langle 1,2\rangle, \langle 1,2\rangle, \langle 3,4\rangle\}$. For both the *union* and *bag union* operators, the schema of the operands must have the same number of attributes. Some authors also require that they share a common schema, i.e. have the same set of attributes [GUW09].

The $\times$ operator produces the *Cartesian product* $t = r \times s$. The result of the *natural join* operator $\bowtie$ is determined by creating the Cartesian product of the relations, then filtering those tuples which are equal on the attributes that share a common name. The combined tuples are projected: from the attributes present in both of the two input relations, we only keep the ones in r and drop the ones in s. Thus, the join operator is defined as

$$r \bowtie s = \pi_{R \cup S}\left(\sigma_{\mathsf{r.A}_1=\mathsf{s.A}_1 \wedge \ldots \wedge \mathsf{r.A}_n=\mathsf{s.A}_n)}(r \times s)\right),$$

where $\{A_1, \ldots, A_n\}$ is the set of attributes that occur both in R and S, i.e. $R \cap S = \{A_1, \ldots, A_n\}$. Note that if the set of common attributes is empty, the *natural join* operator is equivalent to the Cartesian product of the relations. The join operator is both commutative and associative: $r \bowtie s = s \bowtie r$ and $(r \bowtie s) \bowtie t = r \bowtie (s \bowtie t)$, respectively. The *antijoin* operator $\triangleright$ (also known as *left anti semijoin*) collects the tuples from the left relation r which have no matching pair in the right relation s:

$$t = r \triangleright s = r \setminus \pi_R(r \bowtie s),$$

where π_R denotes a projection operator, which only keeps the attributes of the schema over relation r. The antijoin operator is not commutative and not associative. The *left outer join* $\bowtie$ pads tuples from the left relation that did not match any from the right relation with ε values and adds them to the result of the *natural join* [SKS05].

2.3 Common Extensions to Relational Algebra

Most textbooks also define *extended operators* of relational algebra [GUW09]:

- The *duplicate-elimination* operator δ eliminates duplicate tuples in a bag.
- The *grouping* operator γ groups tuples according to their value in one or more attributes and aggregates the remaining attributes.
- The *sorting* operator τ transforms a bag relation of tuples to a list of tuples by ordering them. The ordering is defined by specified attributes of the tuples with an ordering direction (ascending $\uparrow$/descending $\downarrow$) for each attribute, e.g. $\tau_{\uparrow \mathsf{v1},\downarrow \mathsf{v2}}(r)$.

The *top* operator λ_l^s (adapted from [Li05]) takes a list as its input, skips the top s tuples and returns the next l tuples.[5]

2.4 Graph-Specific extensions to Relational Algebra

We adapted graph-specific operators from [HG16][6] and propose an additional operator.

The *get-vertices* nullary operator $\bigcirc_{(\mathtt{v}\,:\,\mathsf{t}_1 \wedge \ldots \wedge \mathsf{t}_n)}$ returns a graph relation of a single attribute v that contains the ID of all vertices that have *all* of labels $t_1, \ldots, t_n$.

The *expand-both* unary operator $\updownarrow \begin{smallmatrix}(\mathsf{l}_1 \vee \ldots \vee \mathsf{l}_k * \mathtt{min} \ldots \mathtt{max}\,:\,\mathtt{v}) \\ (\mathtt{E})\end{smallmatrix} \left[\mathtt{w}\,:\,\mathsf{t}_1 \wedge \ldots \wedge \mathsf{t}_n\right](r)$ adds (1) a new attribute w to r containing the IDs of vertices having *all* labels $t_1, \ldots, t_n$ that can be reached from vertices of attribute v by traversing edges having *any* labels $l_1, \ldots, l_n$, and (2) a new attribute E for the edges of the path from v to w. The operator may use at least min and at most max hops, both defaulting to 1 if omitted. The *expand-in* operator $\downarrow$ and *expand-out* operator $\uparrow$ only consider directed paths from w to v and from v to w, respectively.

We propose the *all-different* operator to guarantee the uniqueness of edges (see the remark on *uniqueness of edges* in Sect. 3). The *all-different* operator $\not\doteq_{\mathsf{E}_1,\mathsf{E}_2,\mathsf{E}_3,\ldots}(r)$ filters r to keep tuples where the variables in $\bigcup_i E_i$ are pairwise different.[7] It can be expressed as a *selection*:

$$\not\doteq_{\mathsf{E}_1,\mathsf{E}_2,\mathsf{E}_3,\ldots}(r) = \sigma_{\bigwedge\limits_{\mathsf{e}_1,\mathsf{e}_2 \in \bigcup\limits_i \mathsf{E}_i \,\wedge\, \mathsf{e}_1 \neq \mathsf{e}_2} \mathsf{r.e}_1 \neq \mathsf{r.e}_2}(r)$$

Property access. Assuming that x is an attribute of a graph relation, we use the notation $x.a$ in (1) attribute lists for projections and (2) selection conditions to express the access to the corresponding value of property a in the property graph [HG16].

Summary. Table 1 provides an overview of the operators of relational graph algebra.

[5] SQL implementations offer the `OFFSET` and the `LIMIT`/`TOP` keywords.

[6] The GETNODES operator introduced in [HG16] and did not support labels. We extended it by allowing the specification of vertex labels and renamed it to *get-vertices* to be consistent with the rest of the definitions. We also extended the EXPANDIN and EXPANDOUT operators to allow it to return a set of edges, and introduced the *expand-both* operator to allow navigation to both directions.

[7] Should e.g. E_2 be a set of the single variable e_2, the variable name can be used as a shorthand instead, so $\not\doteq_{\mathsf{E}_1,\mathsf{e}_2,\mathsf{E}_3,\ldots}(r) \equiv \not\doteq_{\mathsf{E}_1,\{\mathsf{e}_2\},\mathsf{E}_3,\ldots}(r)$

ops	operator	name	prop.	output for			schema
				set	bag	list	
0	$\bigcirc_{(\mathsf{v})}$	*get-vertices*	set	set	set	set	$\langle \mathsf{v} \rangle$
1	$\pi_{\mathsf{v}_1,\mathsf{v}_2,\ldots}(r)$	*projection*	i	bag	bag	list	$\langle \mathsf{v}_1, \mathsf{v}_2, \ldots \rangle$
	$\sigma_{\mathsf{condition}}(r)$	*selection*	i	set	bag	list	sch(r)
	$\updownarrow \begin{smallmatrix}(\mathsf{w})\\(\mathsf{v})\end{smallmatrix}[\mathsf{e}](r)$	*expand-both*	–	set	bag	list	sch(r) $\|\|$ $\langle \mathsf{e}, \mathsf{w} \rangle$
	$\dot{\neq}_{\mathsf{variables}}(r)$	*all-different*	i	set	bag	list	sch(r)
	$\delta(r)$	*duplicate-elimination*	i	set	set	list	sch(r)
	$\tau_{\downarrow \mathsf{v}_1, \updownarrow \mathsf{v}_2, \ldots}(r)$	*sorting*	i	list	list	list	sch(r)
	$\gamma_{\mathsf{v}_1,\mathsf{v}_2,\ldots}(r)$	*grouping*	i	set	set	set	$\langle \mathsf{v}_1, \mathsf{v}_2, \ldots \rangle$
	$\lambda(r)$	*top*	–	list	list	list	sch(r)
2	$r \cup s$, $r \setminus s$	*union*, *minus*	–	set	set	set	sch(r)
	$r \uplus s$	*bag union*	c, a	bag	bag	bag	sch(r)
	$r \times s$	*Cartesian product*	c, a	set	bag	bag	sch(r) $\|\|$ sch(s)
	$r \bowtie s$	*natural join*	c, a	set	bag	bag	sch(r) $\|\|$ (sch(s) $\setminus$ sch(r))
	$r \rtimes\!\bowtie s$	*left outer join*	–	set	bag	bag	sch(r) $\|\|$ (sch(s) $\setminus$ sch(r))
	$r \rhd s$	*antijoin*	c, a	set	bag	bag	sch(r)

Tab. 1: Properties of relational graph algebra operators. A unary operator α is idempotent (i), iff $\alpha(x) = \alpha(\alpha(x))$ for all inputs. A binary operator β is commutative (c), iff $x \ \beta \ y = y \ \beta \ x$ and associative (a), iff $(x \ \beta \ y) \ \beta \ z = x \ \beta \ (y \ \beta \ z)$.

3 The openCypher Query Language

Language. As the primary query language of Neo4j [Ne], Cypher [Ne16a] was designed to read easily. It allows users to specify the graph pattern by a syntax resembling an actual graph. The goal of the openCypher project [Ne16b] is to provide a standardised specification of the Cypher language. Listing 1 shows an openCypher query, which returns all people who (1) are both actors and directors and (2) have acted in a movie together with Clint Eastwood.

```
1 MATCH (a1)-[:ACTS_IN]->(:Movie)<-[:ACTS_IN]-(a2:Actor:Director)
2 WHERE a1.name = "Clint Eastwood"
3 RETURN a2
```

List. 1: Get people who are both actors and directors and acted in a movie with Clint Eastwood.

The query returns with a bag of vertices that have both the labels Actor and Director and share a common Movie neighbor through ACTS_IN edges. Cypher guarantees that these edges are only traversed once, so the vertex of Clint Eastwood is not returned (see the section on the uniqueness of edges).

Implementation. While Neo4j uses a parsing expression grammar (PEG) [Fo04] for specifying the grammar rules of Cypher, openCypher aims to achieve an implementation-agnostic specification by only providing a context-free grammar. The parser can be implemented using any capable parser technology, e.g. ANTLR4 [Pa13] or Xtext [EB10].

Legacy grammar rules. It is not a goal of the openCypher project to fully cover the features of Neo4j's Cypher language: "Not all grammar rules of the Cypher language will be standardised in their current form, meaning that they will not be part of openCypher as-is. Therefore, the openCypher grammar will not include some well-known Cypher constructs; these are called 'legacy'."[8] The *legacy rules* include commands (`CREATE INDEX`, `CREATE UNIQUE CONSTRAINT`, etc.), pre-parser rules (`EXPLAIN`, `PROFILE`) and deprecated constructs (`START`). A detailed description is provided in the openCypher specification. In our work, we focused on the *standard core* of the language and ignored legacy rules.

Uniqueness for edges. In an openCypher query, a `MATCH` clause defines a graph pattern. A query can be composed of multiple patterns spanning multiple `MATCH` clauses. For the matches of a pattern within a single `MATCH` clause, edges are required to be unique. However, matches for multiple `MATCH` clauses can share edges. This uniqueness criterium can be expressed in a compact way with the *all-different* operator introduced in Sect. 2.4. For vertices, this restriction does not apply.

Aggregation. It indeed makes sense to calculate aggregation over graph pattern matches, though, its result will not necessarily be pattern match with vertices and edges. Based on some *grouping criteria*, matches are put into categories, and values for the grouping criteria as well as grouping functions[9] over the groups, the aggregations are evaluated in a single tuple for each and every category. In the SQL query language, grouping criteria is explicitly given by using the `GROUP BY` clause. In openCypher, however, this is done implicitly in the `RETURN` as well as in `WITH` clauses: vertices, edges and their properties that appear outside the grouping functions become the *grouping criteria*.[10]

Subqueries. One can compose an openCypher query of multiple subqueries. Subqueries, written subsequently, mostly begin by a `MATCH` clause and end at (including) a `RETURN` or `WITH` clause, the latter having an optional `WHERE` clause to follow. The `WITH` and `RETURN` clauses determine the resulting schema of the subquery by specifiying the vertices, edges, attributes and aggregates of the result. When `WITH` has the optional `WHERE` clause, it applies an other filter on the subquery result. [11] The last subquery must be ended by `RETURN`, whereas all the previous ones must be ended by `WITH`. If a query is composed by more than one subqueries, their results are joined together using *natural join* or *left outer join* operators.

4 Mapping openCypher Queries to Relational Graph Algebra

In this section, we first give the mapping algorithm of openCypher queries to relational graph algebra, then we give a more detailed listing of the compilation rules for the query language constructs in Table 2. We follow the bottom-up approach to build the relational graph

[8] https://github.com/opencypher/openCypher/tree/master/grammar

[9] For example, `count`, `avg`, `sum`, `max`, `min`, `stdDev`, `stdDevP`, `collect`. The `collect` function is an exception as it does not return a single scalar value but returns a collection (list).

[10] This approach is also used by some SQL code assistant IDEs generating the `GROUP BY` clause for a query.

[11] This is much like the `HAVING` construct of the SQL language with the major difference that it is also allowed in openCypher in case no aggregation has been done.

algebra tree based on the openCypher query. The algorithm is as follows. Join operations always use all common variables to match the two inputs (see *natural join* in Sect. 2.4).

1. A single pattern is turned left-to-right to a *get-vertices* for the first vertex and a chain of *expand-in*, *expand-out* or *expand-both* operators for inbound, outbound or undirected relationships, respectively.
2. Patterns in the same `MATCH` clause are joined by *natural join*.
3. Append an *all-different* operator for all edge variables that appear in the `MATCH` clause because of the non-repeating edges language rule.
4. Process the `WHERE` clause. Note that according to the grammar, `WHERE` is bound to a `MATCH` clause.
5. Several `MATCH` clauses are connected to a left deep tree of *natural join*. If `MATCH` has the `OPTIONAL` modifier, *left outer join* is used instead of *natural join*.
6. If there is a positive or negative pattern deferred from `WHERE` processing, append it as a *natural join* or *antijoin* operator, respectively.
7. Append *grouping*, if `RETURN` or `WITH` clause has grouping functions inside
8. Append *projection* operator based on the `RETURN` or `WITH` clause. This operator will also handle the renaming (i.e. `AS`).
9. Append *duplicate-elimination* operator, if the `RETURN` or `WITH` clause has the `DISTINCT` modifier.
10. Append a *selection* operator in case the `WITH` had the optional `WHERE` clause.
11. If this is not the first subquery, join to the relational graph algebra tree using *natural join* or *left outer join*.
12. Assemble a *union* operation from the query parts[12]. As the *union* operator is technically a binary operator, the *union* of more than two query parts are represented as a left deep tree of `UNION` operators.

Example. The example query in Listing 1 can be formalized as:

$$\pi_{a2}\left(\sigma_{a1.name='C.E.'}\left(\not\coloneqq_{\_e1,\_e2}\downarrow {}^{(a2:\ \text{Actor}\wedge\text{Director})}_{(a1)}[\_e1:\ \text{ACTS\_IN}]\uparrow {}^{(:\ \text{Movie})}_{(a1)}[\_e2:\ \text{ACTS\_IN}]\left(\bigcirc_{(a1:\ \text{Actor})}\right)\right)\right)$$

Note that the $\not\coloneqq$ guarantees the uniqueness constraint for the edges (Sect. 3), which prevents the query from returning the vertex Clint Eastwood.

Optimisations. Queries with negative conditions for patterns can also be expressed using the *antijoin* operator. For example, `MATCH «p1» WHERE NOT «p2»` can be formalized as

$$\not\coloneqq_{\text{edges of p1}}\left(\mathsf{p}_1\right) \triangleright \not\coloneqq_{\text{edges of p2}}\left(\mathsf{p}_2\right)$$

Limitations. Our mapping does not completely cover the openCypher language. As discussed in Sect. 3, some constructs are defined as legacy and thus were omitted. Also, we did not formalize expressions (e.g. conditions in selections), collections (arrays and maps), which are required for both path variables[13] and the `UNWIND` operator. The mapping does not cover parameters and data manipulation operations, e.g. `CREATE`, `DELETE`, `SET` and `MERGE`.

[12] In this context, query parts refer to those parts of the query connected by the `UNION` openCypher keyword.
[13] `MATCH p=(:Person)-[:FRIEND*1..2]->(:Person)`

Language construct	**Relational algebra expression**
Vertex, edge and path patterns	
`()`	$\bigcirc_{(\_v)}$
`(:«t1»:«t2»)`	$\bigcirc_{(\_v\,:\,t1 \wedge t2 \wedge \ldots)}$
`(«v»:«t1»:«t2»)`	$\bigcirc_{(v\,:\,t1 \wedge t2 \wedge \ldots)}$
`«p»-[«e»:«l1|...»]-(«w»:«t1»:«t2»...)`	$\updownarrow^{(w\,:\,t1 \wedge t2 \wedge \ldots)}_{(v)}[e\,:\,l1 \vee \ldots](p)$
`«p»<-[«e»:«l1|...»]->(«w»:«t1»:«t2»...)`	
`«p»-[«e»:«l1|...»]->(«w»:«t1»:«t2»...)`	$\uparrow^{(w\,:\,t1 \wedge t2 \wedge \ldots)}_{(v)}[e\,:\,l1 \vee \ldots](p)$
`«p»<-[«e»:«l1|...»]-(«w»:«t1»:«t2»...)`	$\downarrow^{(w\,:\,t1 \wedge t2 \wedge \ldots)}_{(v)}[e\,:\,l1 \vee \ldots](p)$
`«p»-[«l1|...»*«min»..«max»]->(«w»:«t2»)`	$\uparrow^{(w\,:\,t1 \wedge t2 \wedge \ldots)}_{(v)}[E\,:\,l1 \vee \ldots](p)$
Combining and filtering pattern matches	
`MATCH «p»`	$\not\equiv_{\text{edges of p}}(p)$
`MATCH «p1», «p2»`	$\not\equiv_{\text{edges of p1 and p2}}(p_1 \bowtie p_2)$
`MATCH «p1»` `MATCH «p2»`	$\not\equiv_{\text{edges of p1}}(p_1) \bowtie \not\equiv_{\text{edges of p2}}(p_2)$
`MATCH «p1»` `OPTIONAL MATCH «p2»`	$\not\equiv_{\text{edges of p1}}(p_1) \bowtie \not\equiv_{\text{edges of p2}}(p_2)$
`MATCH «p»` `WHERE «condition»`	$\sigma_{\text{condition}}(r)$, where condition may specify patterns and arithmetic constraints on existing variables
Result and sub-result operations. Rules for `RETURN` also apply to `WITH`.	
`RETURN «variables»`	$\pi_{\text{variables}}(r)$
`RETURN «v1» AS «alias1» ...`	$\pi_{\text{v1}\to\text{alias1},\ldots}(r)$
`RETURN DISTINCT «variables»`	$\delta\left(\pi_{\text{variables}}(r)\right)$
`RETURN «variables», «aggregates»`	$\gamma_{\text{variables,aggregates}}(r)$
List operations	
`ORDER BY «v1» [ASC|DESC] ...`	$\tau_{\uparrow/\downarrow v1,\ldots}(r)$
`LIMIT «l»`	$\lambda_l(r)$
`SKIP «s»`	$\lambda^s(r)$
`SKIP «s» LIMIT «l»`	$\lambda^s_l(r)$
Combining results	
`«query1» UNION «query2»`	$r_1 \cup r_2$
`«query1» UNION ALL «query2»`	$r_1 \uplus r_2$

Tab. 2: Mapping from openCypher constructs to relational algebra.

5 Related Work

The TinkerPop framework [Ap] aims to provide a standard data model for property graphs, along with Gremlin, a high-level graph-traversal language [Ro15] and the Gremlin Structure API, a low-level programming interface.

Besides property graphs, graph queries can be formalized on different graph-like data models and even relational databases.

EMF. The Eclipse Modeling Framework (EMF) is an object-oriented modelling framework widely used in model-driven engineering. Henshin [Ar10] provides a visual language for defining patterns, while Epsilon [KPP08] and VIATRA Query [Be10] provide high-level declarative (textual) query languages, Epsilon Pattern Language and VIATRA Query Language.

RDF. The Resource Description Framework (RDF) [Woa] aims to describe entities of the semantic web. RDF assumes sparse, ever-growing and incomplete data stored as triples that can be queried using the SPARQL [Wob] graph pattern language.

SQL. In general, relational databases offer limited support for graph queries: recursive queries are supported by PostgreSQL using the `WITH RECURSIVE` keyword and by the Oracle Database using the `CONNECT BY` keyword. Graph queries are supported in SAP HANA Graph Scale-Out Extension prototype [Ru13], through a SQL-based language [Kr16].

6 Conclusion and Future Work

In this paper, we presented a formal specification for a subset of the openCypher query language. This provides the theoretical foundations to use openCypher as a language for graph query engines. Using the proposed mapping, an openCypher-compliant query engine could be built on any relational database engine to (1) store property graphs as graph relations and to (2) efficiently evaluate the extended operators of relational graph algebra.

As a future work, we will give formal specification of the operators for incremental query evaluation, which requires us to define *maintenance operations* to keep their result in sync with the latest set of changes. Our long-term research objective is to design and prototype a *distributed, incremental graph query engine* [Sz14] for the property graph data model.

Acknowledgements

This work was partially supported by the MTA-BME Lendület Research Group on Cyber-Physical Systems. The authors would like to thank Gábor Bergmann and Dániel Varró for their suggestions and comments on the draft of this paper.

References

[Ap] TinkerPop3. http://tinkerpop.apache.org/.

[Ar10] Arendt, Thorsten; Biermann, Enrico; Jurack, Stefan; Krause, Christian; Taentzer, Gabriele: pp. 121–135, 2010.

[Be10] Bergmann, Gábor et al.: Incremental Evaluation of Model Queries over EMF Models. In: MODELS. pp. 76–90, 2010.

[EB10] Eysholdt, Moritz; Behrens, Heiko: Xtext: implement your language faster than the quick and dirty way. In: SIGPLAN, SPLASH/OOPSLA. pp. 307–309, 2010.

[EN00] Elmasri, Ramez; Navathe, Shamkant B.: Fundamentals of Database Systems, 3rd Edition. Addison-Wesley-Longman, 2000.

[Fo04] Ford, Bryan: Parsing expression grammars: a recognition-based syntactic foundation. In: SIGPLAN-SIGACT, POPL. ACM, pp. 111–122, 2004.

[GUW09] Garcia-Molina, Hector; Ullman, Jeffrey D.; Widom, Jennifer: Database systems – The complete book. Pearson Education, 2nd edition, 2009.

[HG16] Hölsch, Jürgen; Grossniklaus, Michael: An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In: EDBT/ICDT. 2016.

[KPP08] Kolovos, Dimitrios S.; Paige, Richard F.; Polack, Fiona: The Epsilon Transformation Language. pp. 46–60, 2008.

[Kr16] Krause, Christian et al.: An SQL-Based Query Language and Engine for Graph Pattern Matching. In: ICGT. pp. 153–169, 2016.

[Li05] Li, Chengkai; Chang, Kevin Chen-Chuan; Ilyas, Ihab F.; Song, Sumin: RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In: SIGMOD. pp. 131–142, 2005.

[Ne] Neo4j. http://neo4j.org/.

[Ne16a] Cypher query language. https://neo4j.com/docs/developer-manual/current/cypher/.

[Ne16b] openCypher Project. http://www.opencypher.org/.

[Pa13] Parr, Terence: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2nd edition, 2013.

[RN11] Rodriguez, Marko A.; Neubauer, Peter: The Graph Traversal Pattern. In: Graph Data Management: Techniques and Applications., pp. 29–46. IGI Global, 2011.

[Ro08] Rodriguez, Marko A.: A collectively generated model of the world. In: Collective intelligence: creating a prosperous world at peace. Oakton: Earth Intelligence Network, 2008.

[Ro15] Rodriguez, Marko A.: The Gremlin Graph Traversal Machine and Language (Invited Talk). DBPL 2015, ACM, New York, NY, USA, pp. 1–10, 2015.

[Ru13] Rudolf, Michael; Paradies, Marcus; Bornhövd, Christof; Lehner, Wolfgang: The Graph Story of the SAP HANA Database. In: BTW. volume 214 of LNI. GI, pp. 403–420, 2013.

[SKS05] Silberschatz, Abraham; Korth, Henry F.; Sudarshan, S.: Database System Concepts, 5th Edition. McGraw-Hill Book Company, 2005.

[Sz14] Szárnyas, Gábor et al.: IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In: MODELS. pp. 653–669, 2014.

[Woa] Resource Description Framework (RDF). http://www.w3.org/standards/techs/rdf/.

[Wob] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.