Formalising openCypher Graph Queries in Relational Algebra

József Marton[1], Gábor Szárnyas[2,3], and Dániel Varró[2,3]

[1] Budapest University of Technology and Economics, Database Laboratory
marton@db.bme.hu
[2] Budapest University of Technology and Economics
Fault Tolerant Systems Research Group
MTA-BME Lendület Research Group on Cyber-Physical Systems
{szarnyas, varro}@mit.bme.hu
[3] McGill University
Department of Electrical and Computer Engineering

The final publication is available at Springer via
https://doi.org/10.1007/978-3-319-66917-5_13.

Abstract. Graph database systems are increasingly adapted for storing and processing heterogeneous network-like datasets. However, due to the novelty of such systems, no standard data model or query language has yet emerged. Consequently, migrating datasets or applications even between related 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, vendors are working on supporting existing standard languages (e.g. SQL) or standardising languages.

In this paper, we present a formal specification for openCypher, a high-level declarative graph query language with an ongoing standardisation effort. We introduce relational graph algebra, which extends relational operators by adapting graph-specific operators and define a mapping from core openCypher constructs to this algebra. We propose an algorithm that allows systematic compilation of openCypher queries.

1 Introduction

Context. Graphs are a well-known formalism, widely used for describing and analysing systems. Graphs provide an intuitive formalism for modelling many 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*) [15].

The *property graph* data model [17] extends graphs by adding labels/types and properties for 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 [21].

Neo4j[4], a popular NoSQL property graph database, offers the Cypher query language to specify graph patterns. Cypher is a high-level declarative query language which allows the query engine to use sophisticated optimisation techniques. Neo Technology, the company behind Neo4j initiated the openCypher project [13], which aims to deliver an open specification of Cypher.

Problem and objectives. The openCypher project provides a formal specification of the *grammar* of the query language and a set of acceptance tests that define the semantics of various language constructs. This allows other parties to develop their own openCypher-compatible query engine. However, there is no mathematical formalisation for the language. In ambiguous cases, developers are advised to consult Neo4j's 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 [7] and relational graph algebra, which operates on multisets (bags) [6] and is extended with additional graph-specific operators. Using these foundations, we construct a concise formal specification for the core features in the openCypher grammar. This specification is detailed enough to serve as a basis for an openCypher compiler [23].

2 Data Model and Running Example

Data model. A *property graph* is defined as $G = (V, E, st, L, T, \mathcal{L}, \mathcal{T}, P_v, P_e)$, where V is a set of vertices, E is a set of edges and $st : E \to V \times V$ assigns the source and target vertices to edges. Vertices are labelled and edges are typed:

- L is a set of vertex labels, $\mathcal{L} : V \to 2^L$ assigns a *set of labels* to each vertex.
- T is a set of edge types, $\mathcal{T} : E \to T$ assigns a *single type* to each edge.

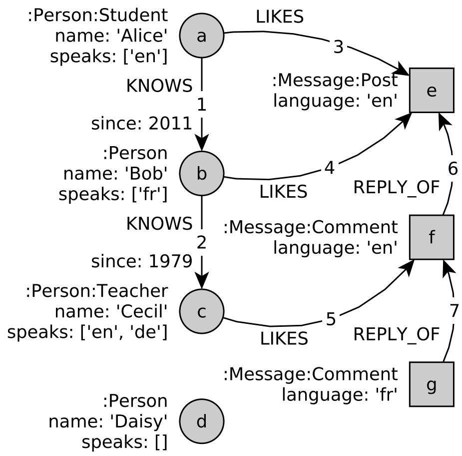
To define properties, let $D = \cup_i D_i$ be the union of atomic domains D_i and let ε represent the NULL value.

- P_v is a set of vertex properties. A vertex property $p_i \in P_v$ is a partial 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 partial 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.

In the context of this paper, we define a *relation* as a *bag* (*multiset*) of tuples: a tuple can occur more than once in a relation [6]. Given a property graph G, relation r is a *graph relation* if the following holds:

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

[4] `https://neo4j.com/`



(a) Example social network graph.

$$
\begin{aligned}
L &= \{\mathsf{Person}, \mathsf{Student}, \mathsf{Teacher}, \mathsf{Message} \ldots\} \\
T &= \{\mathsf{KNOWS}, \mathsf{LIKES}, \mathsf{REPLY\_OF}\} \\
P_v &= \{\mathsf{name}, \mathsf{speaks}, \mathsf{language}\} \\
P_e &= \{\mathsf{since}\} \\
V &= \{a, b, c, d, e, f, g\} \\
E &= \{1, 2, 3, 4, 5, 6, 7\} \\
st :&\ 1 \to \langle a, b\rangle, 2 \to \langle b, c\rangle, \ldots \\
\mathcal{L} :&\ b \to \{\mathsf{Person}\}, e \to \{\mathsf{Message}, \mathsf{Post}\}, \ldots \\
\mathcal{T} :&\ 1 \to \mathsf{KNOWS}, 3 \to \mathsf{LIKES}, \ldots \\
\mathsf{name} :&\ a \to \text{“Alice”}, b \to \text{“Bob”}, e \to \varepsilon, \ldots \\
\mathsf{since} :&\ 1 \to 2011, 2 \to 1979, 3 \to \varepsilon, \ldots
\end{aligned}
$$

(b) The dataset as a property graph.

Fig. 1: Social network example represented graphically and formally. To improve readability, we use letters for vertex identifiers and numbers for edge identifiers.

where the schema of r, $\mathrm{sch}(r)$, is a list containing the attribute names, $\mathrm{dom}(A)$ is the domain of attribute A, V is the vertices of G, and E is the edges of G.

Property access. When defining relational algebra expression on graph relations, it is often required (e.g. in projection and selection operators) to access a certain property of a vertex/edge. Following the notation of [7], if x is an attribute of a graph relation, we use $x.p$ to access the corresponding value of property p. Also, $\mathcal{L}(v)$ returns the labels of vertex v and $\mathcal{T}(e)$ returns the type of edge e.

Running example. An example graph inspired by the LDBC Social Network Benchmark [5] is shown on Fig. 1(a), while Fig. 1(b) presents the formalised graph. The graph vertices model four Persons and three Messages, with edges representing LIKES, REPLY_OF and KNOWS relations. In social networks, the KNOWS relation is symmetric, however, the property graph data model does not allow undirected edges. Hence, we use directed edges with an arbitrary direction and model the symmetric semantics of the relation in the queries.

3 The openCypher Query Language

Cypher is the a high-level declarative graph query language of the Neo4j graph database. It allows users to specify graph patterns with a syntax resembling an actual graph, which makes the queries easy to comprehend. The goal of the openCypher project [13] is to provide a standardised specification of the Cypher language. In the following, we introduce features of the language using examples.

3.1 Language Constructs

Inputs and outputs. openCypher queries take a *property graph* as their input, however the result of a query is not a graph, but a *graph relation*.

Vertex and path patterns. The basic building blocks of queries are patterns of vertices and edges. List. 3.1 shows a query that returns all vertices that model a Person. The query in List. 3.2 matches Person and Message pairs connected by a LIKES edge and returns the person's name and the message language. List. 3.3 describes person pairs that know each other directly or have a friend in common, i.e. from person p1, the other person p2 can be reached using one or two hops.

```
MATCH (p:Person)
RETURN p
```

List. 3.1: Getting vertices.

```
MATCH (p:Person)-[:LIKES]->(m:Message)
RETURN p.name, m.language
```

List. 3.2: Pattern matching.

```
MATCH
 (p1:Person)-[ks:KNOWS*1..2]-
 (p2:Person)
RETURN p1, p2
```

List. 3.3: Variable length path.

```
MATCH (p:Person) WITH p
UNWIND p.speaks AS language
RETURN language,
 count(DISTINCT p.name) as cnt
```

List. 3.4: Grouping.

Filtering. Pattern matches can be filtered in two ways as illustrated in List. 3.5 and List. 3.6. (1) Vertex and edge patterns in the `MATCH` clause might have vertex label/edge type constraints written in the pattern after a colon, and (2) the optional `WHERE` subclause of `MATCH` might hold predicates.

```
MATCH (p1:Person)-[k:KNOWS]-(p2:Person)
WHERE k.since < 2000
RETURN p1.name, p2.name
```

List. 3.5: Filtering for edge property.

```
MATCH (p:Person)
WHERE p.name = 'Bob'
RETURN p.speaks
```

List. 3.6: Filtering.

Unique and non-unique edges. A `MATCH` clause defines a graph pattern. A query can be composed of multiple patterns spanning multiple `MATCH` clauses. For 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 means that in matches returned by List. 3.7, k1 and k2 are required to be different, while in matches returned by List. 3.8, k1 and k2 are allowed to be equal. For vertices, this restriction does not apply.[5] This is illustrated in List. 3.9, which returns adjacent persons who like the same message.

[5] Requiring uniqueness of edges is called *edge isomorphic matching*. Other query languages and execution engines might use *vertex isomorphic matching* (requiring uniqueness of vertices), *isomorphic matching* (requiring uniqueness of both vertices and edges) or *homomorphic matching* (not requiring uniqueness of either) [8].

```
MATCH (p1)-[k1:KNOWS]-(p2),
      (p2)-[k2:KNOWS]-(p3)
RETURN p1, k1, p2, k2, p3
```

List. 3.7: Different edges.

```
MATCH (p1)-[k1:KNOWS]-(p2)
MATCH (p2)-[k2:KNOWS]-(p3)
RETURN p1, k1, p2, k2, p3
```

List. 3.8: Non-unique edges.

```
MATCH (m:Message)<-[:LIKES]-(p1:Person)--(p2:Person)-[:LIKES]->(m)
RETURN p1, p2, m
```

List. 3.9: Triangle.

Creating the result set. The result set[6] of a query is basically given in the `RETURN` clause, which can be de-duplicated using the `DISTINCT` modifier, sorted using the `ORDER BY` subclause. Skipping rows after sorting and limiting the result set to a certain number of records can be achieved using `SKIP` and `LIMIT` modifiers.

List. 3.10 illustrates these concepts by returning the name of the persons. The result set is restricted to the second and third names in alphabetical order.

```
MATCH (p:Person)
RETURN DISTINCT p.name
 ORDER BY p.name
  SKIP 1 LIMIT 2
```

List. 3.10: Deduplicate and sort.

```
1 MATCH
2  ()-[:LIKES]->(m:Message)<-[:LIKES]-(),
3  (m)<-[:REPLY_OF]-(r)
4 RETURN r
```

List. 3.11: Multiple patterns.

Combining patterns. Multiple patterns (in the same or in different) `MATCH` clauses are combined together based on their common variables. List. 3.11 illustrates this by showing two patterns on lines 2 and 3. The first pattern describes a message `m` that has at least two likes. The second pattern finds replies to `m`.

Aggregation. openCypher specifies aggregation operators for performing calculations on multiple tuples.[7] Unlike in SQL queries, the *grouping criteria* is determined implicitly in the `RETURN` as well as in and `WITH` clauses. Each expression of the expression list in `WITH` and `RETURN` are forced to contain either (1) no aggregate functions or (2) a single aggregate function at the outermost level. The grouping key is the tuple built from expressions of type (1).[8] The query of List. 3.4 counts the number of persons commanding each language.

```
MATCH (p:Person) WITH p
UNWIND p.speaks AS lang
RETURN p.name, lang
```

List. 3.12: Unwind.

p.name	**lang**
Alice	en
Bob	fr
Cecil	en
Cecil	de

Fig. 2: Output of the unwind query.

[6] The term *result set* refers to the *result collection*, which can be a set, a bag or a list.

[7] The `avg`, `count`, `max`, `min`, `percentileCont`, `percentileDisc`, `stdDev`, `stdDevP`, `sum` functions return a single scalar value, while `collect` returns a list.

[8] Decision on grouping semantics is due after the camera ready submission deadline. The semantics presented in this paper is one of the possible approaches.

Unwinding a list. The `UNWIND` construct takes an attribute and multiplies each tuple by appending the list elements one by one to the tuple, thus modifying the schema of the query part. By applying `UNWIND` to the speaks attribute List. 3.12 lists persons along the languages they speak. Fig. 2 shows the output of this query. As Cecil speaks two languages, he appears twice in the output. Note that "Daisy" speaks no languages, thus no tuples belong to her in the output.

3.2 Query Structure

In openCypher a query is composed as the `UNION` of one or more single queries. Each single query must have the same resulting schema, i.e. the resulting tuples must have the same arity and the same name at each position.

Single queries. A single query is composed of one or more query parts written subsequently. Query parts that form a prefix of a single query have one result set with the schema of the last query part's schema in that prefix.

Query parts. Clause sequence of a query part matches the regular expression as follows: `MATCH*((WITH UNWIND?)|UNWIND|RETURN)`. They begin with an arbitrary number of `MATCH` clauses, followed by either (1) `WITH` and an optional `UNWIND`, (2) a single `UNWIND`, or (3) a `RETURN` in case of the last query part.[9]

The `RETURN` and `WITH` clauses use similar syntax and have the same semantics, the only difference being that `RETURN` should be used in the last query part while `WITH` should only appear in the preceding ones. These clauses list expressions whose value form the tuples, thus they determine the schema of the query parts.

```
1 MATCH (m1:Message)
2 WITH m1.language AS singleLang, count(*) AS cnt
3 WHERE cnt = 1
4 MATCH (m2:Message) WHERE m2.language = singleLang
5 OPTIONAL MATCH (m2)-[:REPLY_OF]->(m3:Message)
6 RETURN m2.language as reply, m3.language as orig
```

List. 3.13: Single query with multiple query parts.

reply	**orig**
fr	en

Fig. 3: Result.

Example. An openCypher single query composed of two query parts is shown on List. 3.13 along with its result on Fig. 3. It retrieves the language of messages that were written in a language no other message uses. If that message was a reply, the language of the original message is also retrieved.

The first query part spans lines 1–3 and the second spans lines 4–6. The result of the first query part is a single tuple ⟨"fr", 1⟩ with the schema ⟨singleLang, cnt⟩.

[9] In openCypher, the filtering `WHERE` operation is a subclause of `MATCH` and `WITH`. When used in `WITH` as illustrated on line 3 of List. 3.13, `WHERE` is similar to the `HAVING` construct of SQL with the major difference that, in openCypher it is also allowed when no aggregation was specified in the query.

The second query part takes this result as an input to retrieve messages of the given languages and in case of a reply the original message in `m3`. The result of these two query parts together produces the final result whose schema is determined by the `RETURN` of the last query part (line 6).

4 Mapping openCypher to Relational Graph Algebra

In this section, we present relational graph algebra using the examples of Sec. 3.1 and provide a mapping that allows compilation from openCypher to this algebra.

#ops.	notation	name	props.	schema
0	$\bigcirc_{(\mathtt{v:L})}$	get-vertices	–	$\langle \mathtt{v} \rangle$
1	$\updownarrow^{(\mathtt{w:L})}_{(\mathtt{v})} [\mathsf{e} : \mathsf{T}]\,(r)$	expand-both	–	$\mathrm{sch}(r) \parallel \langle \mathtt{e}, \mathtt{w} \rangle$
	$\not\equiv_{\mathsf{variables}}(r)$	all-different	i	$\mathrm{sch}(r)$
	$\omega_{\mathtt{xs} \to \mathtt{x}}(r)$	unwind	–	$\mathrm{sch}(r) \setminus \langle \mathtt{xs} \rangle \parallel \langle \mathtt{x} \rangle$
	$\sigma_{\mathsf{condition}}(r)$	selection	i	$\mathrm{sch}(r)$
	$\pi_{\mathtt{x}_1, \mathtt{x}_2, \ldots}(r)$	projection	i	$\langle \mathtt{x}_1, \mathtt{x}_2, \ldots \rangle$
	$\gamma^{\mathtt{c}_1, \mathtt{c}_2, \ldots}_{\mathtt{x}_1, \mathtt{x}_2, \ldots}(r)$	grouping	i	$\langle \mathtt{x}_1, \mathtt{x}_2, \ldots \rangle$
	$\delta(r)$	duplicate-elimination	i	$\mathrm{sch}(r)$
	$\tau_{\downarrow \mathtt{x}_1, \uparrow \mathtt{x}_2, \ldots}(r)$	sorting	i	$\mathrm{sch}(r)$
	$\lambda^{\mathsf{limit}}_{\mathsf{skip}}(r)$	top	–	$\mathrm{sch}(r)$
2	$r \cup s$	union	c, a	$\mathrm{sch}(r)$
	$r \uplus s$	bag union	c, a	$\mathrm{sch}(r)$
	$r \bowtie s$	natural join	c, a	$\mathrm{sch}(r) \parallel (\mathrm{sch}(s) \setminus \mathrm{sch}(r))$
	$r ⟕ s$	left outer join	–	$\mathrm{sch}(r) \parallel (\mathrm{sch}(s) \setminus \mathrm{sch}(r))$

Table 1: Number of operands, properties and result schemas 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 $(x \ \beta \ y) \ \beta \ z = x \ \beta \ (y \ \beta \ z)$. For schema transformations, append is denoted by $\parallel$, while removal is marked by $\setminus$. L represents a (possibly empty) set of vertex labels and T represents a (possibly empty) set of edge types.

4.1 An Algebra for Formalising Graph Queries

We present both standard operators of relational algebra [4] and operators for graph relations. Tab. 1 provides an overview of the operators of relational graph algebra. We follow the openCypher query language and present a mapping from the language constructs to their algebraic equivalents[10], summarized in Tab. 2. The corresponding rows of the table (e.g. ①) are referred to in the text.

[10] Patterns in the openCypher query might contain anonymous vertices and edges. In the algebraic form, we denote this with names starting with an underscore, such as `_v1` and `_e2`.

Basic operators. The *projection* operator π keeps a specific set of attributes in the relation: $t = \pi_{\mathtt{x_1,\ldots,x_n}}(r)$. Note that the tuples are not deduplicated (i.e. filtered to sets), thus t will have the same number of tuples as r. The projection operator can also rename the attributes, e.g. $\pi_{\mathtt{x1}\rightarrow\mathtt{y1}}(r)$ renames `x1` to `y1`.

The *selection* operator σ filters the incoming relation according to some criteria. Formally, $t = \sigma_\theta(r)$, where predicate θ is a propositional formula. Relation t contains all tuples from r for which θ holds.

Vertices and patterns. ①–② The *get-vertices* [7] nullary operator $\bigcirc_{(\mathtt{v}:\mathsf{l_1}\wedge\ldots\wedge\mathsf{l_n})}$ returns a graph relation of a single attribute v that contains vertices that have *all* of labels $l_1, \ldots, l_n$. Using this operator, the query in List. 3.1 is compiled to

$$\bigcirc_{(\mathtt{p}:\mathsf{Person})}$$

③–⑥ The *expand-out* unary operator $\uparrow^{(\mathtt{w}:\mathsf{l_1}\wedge\ldots\wedge\mathsf{l_n})}_{(\mathtt{v})}[\mathtt{e}:\mathsf{t_1}\vee\ldots\vee\mathsf{t_k}](r)$ adds new attributes e and w to each tuple iff there is an edge e from v to w, where e has *any* of types $t_1, \ldots, t_k$, while w has *all* labels $l_1, \ldots, l_n$.[11] More formally, this operator appends the $\langle e, w\rangle$ to a tuple iff $st(e) = \langle v, w\rangle$, $l_1, \ldots, l_n \in \mathcal{L}(w)$ and $\mathcal{T}(e) \in \{t_1, \ldots, t_k\}$. Using this operator, the query in List. 3.2 can be formalised as

$$\pi_{\mathtt{p.name,m.language}} \uparrow^{(\mathtt{m}:\mathsf{Message})}_{(\mathtt{p})} [\_\mathtt{e1}:\mathsf{LIKES}] \bigcirc_{(\mathtt{p}:\mathsf{Person})}$$

Similarly to the expand-out operator, the *expand-in* operator $\downarrow$ appends $\langle e, w\rangle$ iff $st(e) = \langle w, v\rangle$, while the *expand-both* operator $\updownarrow$ uses edge e iff either $st(e) = \langle v, w\rangle$ or $st(e) = \langle w, v\rangle$. We also propose an extended version of this operator, $\uparrow^{(\mathtt{w})}_{(\mathtt{v})}[\mathtt{e}*^{\mathsf{max}}_{\mathsf{min}}]$, which may use between min and max hops. Using this extension, List. 3.3 is compiled to

$$\pi_{\mathtt{p1,p2}} \not\equiv_{\mathtt{ks}} \updownarrow^{(\mathtt{p2}:\mathsf{Person})}_{(\mathtt{p1})} [\mathtt{ks}:\mathsf{KNOWS}*^{2}_{1}] \bigcirc_{(\mathtt{p1}:\mathsf{Person})}$$

Combining and filtering pattern matches. ⑦–⑪ In order to express the uniqueness criterion for edges (illustrated in Sec. 3.1) in a compact way, we propose the *all-different* operator. The all-different operator $\not\equiv_{E_1,\ldots,E_n}(r)$ filters r to keep tuples where variables in $\cup_i E_i$ are pairwise different. Note that the operator is actually a shorthand for the following selection:

$$\not\equiv_{E_1,\ldots,E_n}(r) = \sigma_{\bigwedge_{\substack{e_1,e_2\in\cup_i E_i\\ e_1\neq e_2}} r.e_1\neq r.e_2}(r)$$

[11] Label and type constraints can be omitted for the get-vertices operator and the expand operators. For example, $\bigcirc_{(\mathtt{v})}$ returns all vertices, while $\uparrow^{(\mathtt{w})}_{(\mathtt{v})}[\mathtt{e}](r)$ traverses all outgoing edges e from vertices v to w, regardless of their labels/types.

Using the all-different operator, query in List. 3.7 is compiled to

$$\pi_{\texttt{p1,k1,p2,k2,p3}} \not\equiv_{\texttt{k1,k2}} \updownarrow\,^{(\texttt{p3})}_{(\texttt{p2})}\,[\texttt{k2} : \mathsf{KNOWS}] \updownarrow\,^{(\texttt{p2})}_{(\texttt{p1})}\,[\texttt{k1} : \mathsf{KNOWS}] \bigcirc_{(\texttt{p1})}$$

⑦–⑧ 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

$$r \bowtie s = \pi_{R \cup S} \left(\sigma_{r.A_1 = s.A_1 \,\wedge\, \ldots \,\wedge\, r.A_n = s.A_n} \right) (r \times s) \big) ,$$

where $\{A_1, \ldots, A_n\} = R \cap S$ is the set of attributes that occur both in R and S. In order to allow pattern matches to share the same edge, they must be included in different `MATCH` clauses as shown on List. 3.8 which is compiled to

$$\pi_{\texttt{p1,p2,p3}} \left(\left(\updownarrow\,^{(\texttt{p2})}_{(\texttt{p1})}\,[\texttt{k1} : \mathsf{KNOWS}] \bigcirc_{(\texttt{p1})} \right) \bowtie \left(\updownarrow\,^{(\texttt{p3})}_{(\texttt{p2})}\,[\texttt{k2} : \mathsf{KNOWS}] \bigcirc_{(\texttt{p2})} \right) \right)$$

The query in List. 3.11 with two patterns in one `MATCH` clause is compiled to:

$$\pi_{\texttt{r}} \not\equiv_{\texttt{\_e1,\_e2,\_e3}} \left(\left(\downarrow\,^{(\texttt{\_v2})}_{(\texttt{m})}\,[\texttt{\_e2} : \mathsf{LIKES}] \uparrow\,^{(\texttt{m}:\mathsf{Message})}_{(\texttt{\_v1})}\,[\texttt{\_e1} : \mathsf{LIKES}] \bigcirc_{(\texttt{\_v1})} \right) \right.$$
$$\left. \bowtie \left(\downarrow\,^{(\texttt{r})}_{(\texttt{m})}\,[\texttt{\_e3} : \mathsf{REPLY\_OF}] \bigcirc_{(\texttt{m}:\mathsf{Message})} \right) \right)$$

⑨–⑪ The *left outer join* operator produces $t = r \,⟕\, s$ combining matching tuples of r and s according to a given matching semantics.[12] In case there is no matching tuple in s for a particular tuple $e \in r$, e is still included in the result, with tuple attributes that exclusively belong to relation s having a value of ε.

Result and subresult operations. ⑯ The *duplicate-elimination* operator δ eliminates duplicate tuples in a bag.

⑰ The *grouping* operator γ groups tuples according to their value in one or more attributes and aggregates the remaining attributes.

We generalize the grouping operator to explicitly state the *grouping criteria* and allow for complex aggregate expressions. This is similar to the SQL query language where the grouping criteria is explicitly given in `GROUP BY`.

We use the notation $\gamma^{c_1, c_2, \ldots}_{e_1, e_2, \ldots}$, where $c_1, c_2, \ldots$ in the superscript form the *grouping criteria*, i.e. the list of expressions whose values partition the incoming tuples into groups. For each and every group this aggregation operator emits a single tuple of expressions listed in the subscript, i.e. $\langle e_1, e_2, \ldots \rangle$. Given attributes

[12] Matching semantics might use value equality of attributes that share a common name (similarly to natural join) or use an arbitrary condition (similarly to θ-join).

$\{a_1, \ldots, a_n\}$ of the input relation, c_i is an arithmetic expression built from a_j attributes using common arithmetic operators, while e_i is an expression built from a_j using common arithmetic operators and grouping functions.

We have discussed the aggregation semantics of openCypher in Sec. 3.1. The formal algorithm for determining the grouping criteria is given in Alg. 1. Building on this algorithm and the grouping operator, List. 3.4 is compiled to

$$\gamma^{\texttt{language}}_{\texttt{language},\mathsf{count\_distinct}(\texttt{p.name})\rightarrow\texttt{cnt}}\,\omega_{\texttt{p.speaks}\rightarrow\texttt{language}}\bigcirc_{(\mathsf{p:Person})}$$

```
Data: E is the list of expressions in the RETURN or WITH clause
Function DetermineGroupingCriteria(E)
    G ← {}                                          // initial set of grouping criteria
    foreach e ∈ E do
        if e has an aggregate function call at its outermost level then
                                                    // do nothing as this is an aggregation
        else if e contains aggregate function call then
                                                    // aggregation allowed only at the outermost level
            raise SemanticError(Illegal use of aggregation function)
        else
            G ← G ∪ {e}                             // append to the grouping key
        end
    end
    return G
```

Algorithm 1: Determine grouping criteria from return item list.

Unwinding and list operations. (19) The *unwind* [3] operator $\omega_{\texttt{xs}\rightarrow\texttt{x}}$ takes the list in attribute `xs` and multiplies each tuple adding the list elements one by one to an attribute `x`, as demonstrated in Fig. 2. Using this operator, the query in List. 3.12 can be formalised as

$$\pi_{\texttt{p.name},\texttt{lang}}\,\omega_{\texttt{p.speaks}\rightarrow\texttt{lang}}\,\pi_{\texttt{p}}\bigcirc_{(\mathsf{p:Person})}$$

(20) The *sorting* operator τ transforms a bag relation of tuples to a list of tuples by ordering them. The ordering is defined for selected attributes and with a certain direction for each of them (ascending $\uparrow$/descending $\downarrow$), e.g. $\tau_{\uparrow\texttt{x1},\downarrow\texttt{x2}}(r)$.

(21) The *top* operator λ^s_l (adapted from [11]) takes a list as its input, skips the first s tuples and returns the next l tuples.[13]

Using the sorting and top operators, the query of List. 3.10 is compiled to:

$$\lambda^1_2\,\tau_{\uparrow\texttt{p.name}}\,\delta\,\pi_{\texttt{p.name}}\bigcirc_{(\mathsf{p:Person})}$$

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

Combining results. The $\cup$ operator produces the *set union* of two relations, while the $\uplus$ operator produces the *bag union* of two operators, e.g. $\{\langle 1,2\rangle, \langle 3,4\rangle\} \uplus \{\langle 1,2\rangle\} = \{\langle 1,2\rangle, \langle 1,2\rangle, \langle 3,4\rangle\}$. For both the union and bag union operators, the schema of the operands must have the same attributes.

4.2 Mapping openCypher Queries to Relational Graph Algebra

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

1. Process each single query as follows and combine their result using the union operation. As the union operator is technically a binary operator, the union of more than two single queries are represented as a left-deep tree of `UNION` operators.
2. For each query part of a single query, denoted by t, the relational graph algebra tree built from the prefix of query parts up to—but not including—the current query part, process the current query part as follows.
   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. Comma-separated patterns in a single `MATCH` are connected 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` subclause of a single `MATCH` clause.
   5. Several `MATCH` clauses are connected to a left-deep tree of natural join. For `OPTIONAL MATCH`, left outer join is used instead of natural join. In case there is a `WHERE` subclause, its condition becomes part of the join condition, i.e. it will never filter on the input from previous `MATCH` clauses.
   6. If there is a positive or negative pattern deferred from `WHERE` processing, append it as a natural join or a combination of left outer join and selection operator filtering on no matches were found, respectively.
   7. If this is not the first query part, combine the curent query part with the relational graph algebra tree of the preceding query parts by appending a natural join here. Its left operand will be t and its right operand will be the relational graph algebra tree built so far from the current subquery.
   8. Append grouping, if `RETURN` or `WITH` clause has grouping functions inside.
   9. Append a projection operator based on the `RETURN` or `WITH` clause. This operator will also handle the renaming (i.e. `AS`).
   10. Append a duplicate-elimination operator, if the `RETURN` or `WITH` clause has the `DISTINCT` modifier.
   11. Append a selection operator if `WITH` had the optional `WHERE` subclause.

Language construct	**Relational algebra expression**	
Vertices and patterns. ⦇p⦈ denotes a pattern that contains a vertex «v».		
(«v»)	$\bigcirc_{(\mathtt{v})}$	①
(«v»:«l1»:⋯:«ln»)	$\bigcirc_{(\mathsf{v}:\mathsf{l1}\wedge\cdots\wedge\mathsf{ln})}$	②
⦇p⦈-[«e»:«t1»|⋯|«tk»]->(«w»)	$\uparrow^{(\mathtt{w})}_{(\mathtt{v})}[\mathsf{e}:\mathsf{t1}\vee\cdots\vee\mathsf{tk}]\,(p)$, where e is an edge	③
⦇p⦈<-[«e»:«t1»|⋯|«tk»]-(«w»)	$\downarrow^{(\mathtt{w})}_{(\mathtt{v})}[\mathsf{e}:\mathsf{t1}\vee\cdots\vee\mathsf{tk}]\,(p)$, where e is an edge	④
⦇p⦈<-[«e»:«t1»|⋯|«tk»]->(«w»)	$\updownarrow^{(\mathtt{w})}_{(\mathtt{v})}[\mathsf{e}:\mathsf{t1}\vee\cdots\vee\mathsf{tk}]\,(p)$, where e is an edge	⑤
⦇p⦈-[«e»*«min»..«max»]->(«w»)	$\uparrow^{(\mathtt{w})}_{(\mathtt{v})}[\mathsf{e}*^{\mathsf{max}}_{\mathsf{min}}]\,(p)$, where e is a list of edges	⑥
Combining and filtering pattern matches		
MATCH ⦇p1⦈, ⦇p2⦈, ⋯	$\not\equiv_{\text{edges of p1, p2, …}}(p1 \bowtie p2 \bowtie \cdots)$	⑦
MATCH ⦇p1⦈ MATCH ⦇p2⦈	$\not\equiv_{\text{edges of p1}}(p1) \bowtie \not\equiv_{\text{edges of p2}}(p2)$	⑧
OPTIONAL MATCH ⦇p⦈	$\{\langle\rangle\} ⟕ \not\equiv_{\text{edges of p}}(p)$	⑨
OPTIONAL MATCH ⦇p⦈ WHERE ⦇condition⦈	$\{\langle\rangle\} ⟕_{\text{condition}} \not\equiv_{\text{edges of p}}(p)$	⑩
⟦r⟧ OPTIONAL MATCH ⦇p⦈	$r ⟕ \not\equiv_{\text{edges of p}}(p)$	⑪
⟦r⟧ WHERE «condition»	$\sigma_{\text{condition}}(r)$	⑫
⟦r⟧ WHERE («v»:«l1»:⋯:«ln»)	$\sigma_{\mathcal{L}(v)=\mathtt{l1}\wedge\cdots\wedge\mathcal{L}(v)=\mathtt{ln}}(r)$	⑬
⟦r⟧ WHERE ⦇p⦈	$r \ltimes p$	⑭
Result and subresult operations. Rules for RETURN also apply to WITH.		
⟦r⟧ RETURN «x1» AS «y1», ⋯	$\pi_{\mathtt{x1}\to\mathtt{y1},\cdots}(r)$	⑮
⟦r⟧ RETURN DISTINCT «x1» AS «y1», ⋯	$\delta\left(\pi_{\mathtt{x1}\to\mathtt{y1},\cdots}(r)\right)$	⑯
⟦r⟧ RETURN «x1», «aggr»(«x2»)	$\gamma^{\mathtt{x1}}_{\mathtt{x1},\mathtt{aggr(x2)}}(r)$ (see Sec. 3.1)	⑰
⟦r⟧ WITH «x1» ⟦s⟧ RETURN «x2»	$\pi_{\mathtt{x2}}\Big(\big(\pi_{\mathtt{x1}}(r)\big) \bowtie s\Big)$	⑱
Unwinding and list operations		
⟦r⟧ UNWIND «xs» AS «x»	$\omega_{\mathtt{xs}\to\mathtt{x}}(r)$	⑲
⟦r⟧ ORDER BY «x1» ASC, «x2» DESC, ⋯	$\tau_{\uparrow\mathtt{x1},\downarrow\mathtt{x2},\cdots}(r)$	⑳
⟦r⟧ SKIP «s» LIMIT «l»	$\lambda^{\mathsf{s}}_{\mathsf{l}}(r)$	㉑
Combining results		
⟦r⟧ UNION ⟦s⟧	$r \cup s$	㉒
⟦r⟧ UNION ALL ⟦s⟧	$r \uplus s$	㉓

Table 2: Mapping from openCypher constructs to relational algebra. Variables, labels, types and literals are typeset as «v». The notation ⦇p⦈ represents patterns resulting in a relation p, while ⟦r⟧ denotes previous query fragment resulting in a relation r. To avoid confusion with the “..” language construct (used for ranges), we use ⋯ to denote omitted query fragments.

4.3 Summary and Limitations

In this section, we presented a mapping that allows us to express the example queries of Sec. 3.1 in graph relational algebra. We extended relational algebra by adapting operators ($\bigcirc$, $\uparrow$, τ, λ), precisely specifying grouping semantics (γ) and defining the all-different operator ($\not\equiv$). Finally, we proposed an algorithm for compiling openCypher graph queries to graph relational algebra.

Our mapping does not completely cover the openCypher language. As discussed in Sec. 3, some constructs are defined as legacy and thus were omitted. The current formalisation does not include expressions (e.g. conditions in selections) and maps. Compiling data manipulation operations (such as `CREATE`, `DELETE`, `SET`, and `MERGE`) to relational algebra is also subject of future work.

5 Related Work

Property graph data models. The TinkerPop framework aims to provide a standard data model for property graphs, along with Gremlin, a high-level imperative graph traversal language [16] and the Gremlin Structure API, a low-level programming interface.

EMF. The Eclipse Modeling Framework is an object-oriented modelling framework widely used in model-driven engineering. Henshin [1] provides a visual language for defining patterns, while Epsilon [9] and VIATRA Query [2] provide high-level declarative (textual) query languages, the Epsilon Pattern Language and the VIATRA Query Language.

SPARQL. Widely used in semantic technologies, SPARQL is a standardised declarative graph pattern language for querying RDF [24] graphs. SPARQL bears close similarity to Cypher queries, but targets a different data model and requires users to specify the query as triples instead of graph vertices/edges [14]. G-SPARQL [19] extended the SPARQL language for attributed graphs, resulting in a language with an expressive power similar to openCypher.

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 the SAP HANA prototype [18], through a SQL-based language [10].

Cypher. Due to its novelty, there are only a few research works on the formalisation of (open)Cypher. The authors of [7] defined *graph relations* and introduced the GETNODES, EXPANDIN and EXPANDOUT operators. While their work focused on optimisation transformations, this paper aims to provides a more complete and systematic mapping from openCypher to relational algebra.

In [8], graph queries were defined in a Cypher-like language and evaluated on Apache Flink. However, formalisation of the queries was not discussed in detail.

Comparison of graph query frameworks. Previously, we published the Train Benchmark, a framework for comparing graph query frameworks across different technological spaces, such as property graphs, EMF, RDF and SQL [21].

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.

As future work, we plan to provide a formalisation based on graph-specific theoretical query frameworks, such as [12]. We will also give the formal specification of the operators for incremental query evaluation, which requires the definition of *maintenance operations* that keep the result in sync with the latest set of changes [22]. Our long-term research objective is to design an openCypher-compatible *distributed, incremental graph query engine* [20].[14]

Acknowledgements

Gábor Szárnyas and Dániel Varró were supported by the MTA-BME Lendület Research Group on Cyber-Physical Systems and the NSERC RGPIN-04573-16 project. The authors would like to thank Gábor Bergmann and János Maginecz for their comments on the draft of this paper.

References

1. T. Arendt et al. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MODELS*, pages 121–135, 2010.
2. G. Bergmann et al. Incremental evaluation of model queries over EMF models. In *MODELS*, pages 76–90, 2010.
3. E. Botoeva et al. OBDA beyond relational DBs: A study for MongoDB. In *Proceedings of the 29th International Workshop on Description Logics*, 2016.
4. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley-Longman, 3rd edition, 2000.
5. O. Erling et al. The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.
6. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems – The complete book.* Pearson Education, 2nd edition, 2009.
7. J. Hölsch and M. Grossniklaus. An algebra and equivalences to transform graph patterns in Neo4j. In *GraphQ at EDBT/ICDT*, 2016.
8. M. Junghanns et al. Cypher-based graph pattern matching in Gradoop. In *GRADES at SIGMOD*, 2017.
9. D. S. Kolovos et al. The Epsilon transformation language. In *ICMT*, 2008.
10. C. Krause et al. An SQL-based query language and engine for graph pattern matching. In *ICGT*, pages 153–169, 2016.
11. C. Li, K. C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.

[14] Our prototype, *ingraph*, is available at: `http://docs.inf.mit.bme.hu/ingraph/`

12. L. Libkin et al. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, 2016.
13. Neo Technology. openCypher project. `http://www.opencypher.org/`, 2017.
14. J. Pérez et al. Semantics and complexity of SPARQL. *ACM TODS*, 34(3), 2009.
15. M. A. Rodriguez. A collectively generated model of the world. In *Collective intelligence: creating a prosperous world at peace*, pages 261–264. 2008.
16. M. A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In *DBPL*, pages 1–10, 2015.
17. M. A. Rodriguez and P. Neubauer. The graph traversal pattern. In *Graph Data Management: Techniques and Applications*, pages 29–46. 2011.
18. M. Rudolf et al. The graph story of the SAP HANA database. In *BTW*, 2013.
19. S. Sakr, S. Elnikety, and Y. He. G-SPARQL: a hybrid engine for querying large attributed graphs. In *CIKM*, pages 335–344, 2012.
20. G. Szárnyas et al. IncQuery-D: A distributed incremental model query framework in the cloud. In *MODELS*, pages 653–669, 2014.
21. G. Szárnyas et al. The Train Benchmark: Cross-technology performance evaluation of continuous model validation. *Softw. Syst. Model.*, 2017.
22. G. Szárnyas, J. Maginecz, and D. Varró. Evaluation of optimization strategies for incremental graph queries. *Periodica Polytechnica, EECS*, 2017.
23. G. Szárnyas and J. Marton. Formalisation of openCypher queries in relational algebra. Technical report, Budapest University of Technology and Economics, 2017. `http://hdl.handle.net/10890/5395`.
24. W3C. Resource Description Framework. `https://www.w3.org/RDF/`, 2014.