An Algebra and Equivalences to Transform Graph Patterns in Neo4j

Jürgen Hölsch and Michael Grossniklaus
Department of Computer and Information Science, University of Konstanz
P.O. Box 188, 78457 Konstanz, Germany
{juergen.hoelsch,michael.grossniklaus}@uni-konstanz.de

ABSTRACT

Modern query optimizers of relational database systems embody more than three decades of research and practice in the area of data management and processing. Key advances include algebraic query transformation, intelligent search space pruning, and modular optimizer architectures. Surprisingly, many of these contributions seem to have been overlooked in the emerging field of graph databases so far. In particular, we believe that query optimization based on a general graph algebra and its equivalences can greatly improve on the current state of the art. Although some graph algebras have already been proposed, they have often been developed in a context, in which a relational database system is used as a backend to process graph data. As a consequence, these algebras are typically tightly coupled to the relational algebra, making them unsuitable for native graph databases. While we support the approach of extending the relational algebra, we argue that graph-specific operations should be defined at a higher level, independent of the database backend. In this paper, we introduce such a general graph algebra and corresponding equivalences. We demonstrate how it can be used to optimize Cypher queries in the setting of the Neo4j native graph database.

1. INTRODUCTION

The management and processing of graph data is gaining importance in several application domains such as social network analysis [3, 11, 18], the Semantic Web [4], and biological networks [17]. As a consequence, numerous graph database systems such as Neo4j, DEX/Sparksee, and OrientDB have recently emerged. With respect to their architecture, graph databases can be classified into two groups. Systems belonging to the first group map graphs to existing relational or non-relational database backends, wheareas the native graph database systems of the second group implement custom backends to store the data. While this design choice provides the freedom to address requirements that are specific to graph data by tailor-made solutions, it also poses the challenge of applying the lessons learnt in over 30 years of designing and developing relational database systems to these new graph database systems.

In this paper, we focus on query optimization in the setting of the Neo4j native graph database. We have chosen this concrete setting for several reasons. First, native graph databases are still very heterogeneous in terms of functionality and architecture. Faced with this situation, we follow a "bottom-up" approach that addresses the problems of one system and is later generalized to other systems. Second, Neo4j supports Cypher, a mature declarative graph query language that opens up similar optimization opportunities as SQL in relational database systems. Finally, Neo4j's current query processor provides ample opportunity for improvement, as we demonstrate in the following motivating example.

Consider the following Cypher query on a movie database[1] provided by Neo4j, which we use for examples throughout this paper. The query returns all movies in which "Al Pacino" and "Robert De Niro" act in.

```
MATCH (x:Actor)- ->(y)<- -(z:Actor)
WHERE x.name = "Al Pacino" AND z.name = "Robert De Niro"
RETURN *
```

In the Neo4j Community Edition Version 2.3.1, this query is evaluated using the following query plan. As in relational database systems, the evaluation order is from the leaves to the root.

```
      Filter[x.name="Al Pacino"]
                  |
         Expand(All)[y<- -x]
                  |
         Expand(All)[z- ->y]
                  |
NodeIndexSeek[z.name="Robert De Niro"]
```

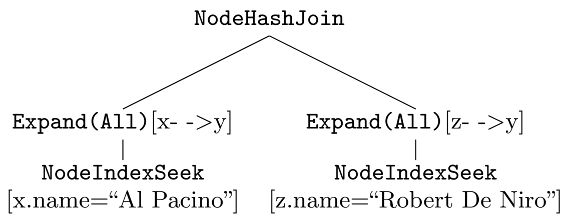
First, an index is accessed to determine the actor node corresponding to "Robert De Niro". Next, the Expand(All) operator is applied to get the movies in which Robert De Niro acts in. The Expand(All) operator returns all neighbors of a given node. For the movie nodes retrieved in this way, the graph is further traversed to find the actors that starred in the same movies as Robert De Niro. Finally, the actor node corresponding to "Al Pacino" is selected. This execution plan results in 1855 database accesses. In Neo4j, a database access is called "database hit". In this work, we will also use the term database hit to refer to a database access and use it as our primary metric to quantify the cost of a plan.

[1]http://neo4j.com/developer/example-data/ (November 19, 2015)

However, the execution plan that Neo4j produces for this query is not the most efficient one. In the following, we extend the query with hints that tell Neo4j to use indexes.

MATCH (x:Actor)- ->(y)<- -(z:Actor)
USING INDEX x:Actor(name)
USING INDEX z:Actor(name)
WHERE x.name = "Al Pacino" AND z.name = "Robert De Niro"
RETURN *

With the help of these hints, Neo4j's optimizer returns a different execution plan, which is given below.

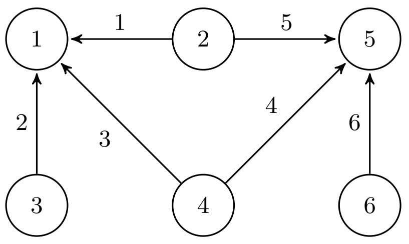


Both the "Al Pacino" and "Robert De Niro" actor node are now determined by an index access. Then, the neighboring movie nodes are retrieved for these two actor nodes. The result of these operations are two sets of subgraphs, in which each subgraph consists of exactly one edge. In the last step, the two subgraph sets are joined by matching the movie nodes. Compared to the first plan, this second plan is 95% less expensive as it only requires 90 database hits.

There are two obstacles that could prevent Neo4j from choosing the more efficient plan given the original query. First, Neo4j might not consider the second plan because it is unable to enumerate it. Second, even if it finds the second plan, Neo4j might not recognize it as less expensive because it lacks the cost model to accurately estimate the number of required database hits. In this work, we address the first of these two obstacles by outlining how the well-known technique of algebraic query optimization can be applied to Cypher graph patterns in Neo4j.

We are aware of existing proposals to define a graph algebra and corresponding equivalences in order to build graph query optimizers following the blueprint established by relational database systems. However, most of these algebras have been defined for graph database systems that use a relational database system as a backend. Therefore, they typically make assumptions about the relational representation of graphs or are tightly coupled to the operators of the relational algebra. While we believe in the general approach of extending the relational algebra to support graph data processing, we argue these extensions need to be defined at a higher level in order to support native graph databases such as Neo4j. The specific contributions of this paper are as follows.

- A data model that uses property graphs to represent the graph database, while so-called graph relations model the input and output of operators (Section 2).
- An algebra that defines two new high-level operators for representing graph patterns in Cypher (Section 3).
- Equivalence rules together with proofs of correctness that specify how expressions of our algebra can be transformed (Section 4).
- New algebraic optimization techniques for Cypher queries at the logical level (Section 5).



Node ID	Label	Name	Title
1	Movie	-	Heat
2	Actor	Al Pacino	-
3	Actor	Robert De Niro	-
4	Director	Michael Mann	-
5	Movie	-	The Insider
6	Actor	Russel Crowe	-

Figure 1: An example property graph (top) where only node and edge IDs are illustrated. The attributes and labels of nodes can be found in the table (bottom).

We position our work w.r.t. related approaches in Section 6. Section 7 gives concluding remarks. Since the results presented in this paper are the first step of a larger research effort, we also outline future work in that section.

2. DATA MODEL

Before defining our graph algebra, we first introduce the data model on which it is based. Existing graph data models can broadly be categorized into two classes, *i.e.*, data models based on property graphs and the RDF data model. Whereas property graph data models can assign properties to nodes and edges, properties in RDF are represented as additional nodes. Most existing graph algebra proposals have been defined in the context of RDF. Since RDF triples can easily be mapped to relations, these algebras are often only slight extensions of the relational algebra. Unfortunately, these algebras are typically unsuited to express queries on property graphs. In this work, we will therefore focus on the property graph model, which is also used by Neo4j. More specifically, we limit this work to directed graphs in which nodes have only one label.

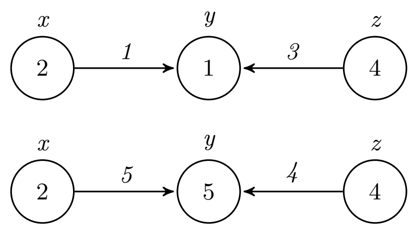
Definition 2.1 (Property Graph) *Let $G = (V, E, \Sigma_v, \Sigma_e, A_v, A_e, \lambda, l_v, l_e)$ be a property graph, where V is a set of nodes, E is a set of edges, Σ_v is a set of node labels, and Σ_e is a set of edge labels. In addition, A_v is a set of node properties and A_e is a set of edge properties. Let D be a set of atomic domains. A property $a_i \in A_v$ is a function $a_i : V \to D_i \cup \{\epsilon\}$ which assigns a property value from a domain $D_i \in D$ to a node $v \in V$, if v has property a_i, otherwise $a_i(v)$ returns ϵ. A property $a_j \in A_e$ is a function $a_j : E \to D_j \cup \{\epsilon\}$ which assigns a property value from a domain $D_j \in D$ to an edge $e \in E$, if e has property a_j, otherwise $a_j(e)$ returns ϵ. Furthermore, $\lambda : E \to V \times V$ is a function assigning nodes to edges, $l_v : V \to \Sigma_v$ is a function assigning labels to nodes, and $l_e : E \to \Sigma_e$ is a function assigning labels to edges.*

As we will explain in Section 3, the input and output of our algebra operators are subgraphs. Therefore, we need to define how a subgraph G' of G is represented. Since we want to combine our new operators with the existing operators of the relational algebra, we represent subgraphs as relations. For this purpose, we introduce the concept of *graph relations*. A graph relation is a relation that only contains columns corresponding to nodes or edges. Example 2.1 illustrates the concept of graph relations.

Example 2.1 *The Cypher query below returns the movies directed by Michael Mann in which Al Pacino acted.*

`MATCH` (x:Actor)- ->(y)<- -(z:Director)
`WHERE` x.name = “Al Pacino” `AND` z.name = “Michael Mann”
`RETURN` *

The following two subgraphs of the graph shown in Figure 1 match the pattern defined in the Cypher query above. In order to illustrate the matching, the variables x,y and z are included in the subgraphs.



These subgraphs are represented by the following graph relation.

x	xy	y	zy	z
2	*1*	*1*	*3*	*4*
2	*5*	*5*	*4*	*4*

The column names in the schema of a graph relation are used to access the nodes and edges of a subgraph (cf. Section 3). The column “xy” represents the edge from node x to y and the column “zy” represents the edge from node z to y.

Definition 2.2 (Graph Relation) *Given a property graph G, a relation R is a graph relation, if the following holds:*

$$\forall A \in attr(R) : dom(A) = V \vee E,$$

where $attr(R)$ is the set of attributes of R, $dom(A)$ is the domain of attribute A, V are nodes of G, and E are edges of G.

The attributes of a graph relation only contain node or edge identifiers that reference nodes or edges of the graph G.

Using graph relations as input and output of our algebra operators has two advantages. First, as mentioned before, the operators of the relational algebra (*e.g.*, selection and projection) can be reused. Second, graph relations are independent of the underlying graph representations. For example, instead of basing them on property graphs, they can also be defined in the scope of the RDF data model in order to represent SPARQL queries in our algebra.

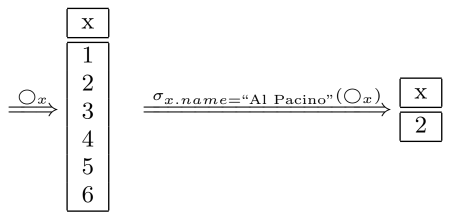
3. ALGEBRA

Having described the graph data model that we assume, we now extend the relational algebra with two new graph-specific operators. First, we introduce the GetNodes operator which returns a graph relation containing all nodes of the underlying graph G. Therefore, leaves of an operator tree of our algebra have to be GetNodes operators. Second, we introduce the Expand operator which adds two new columns to the current graph relation where one column contains all neighbors of a specific node and the other column contains the corresponding edges. The Expand operator makes it possible to formally describe the order in which nodes are traversed to find a specific pattern.

Before we define the GetNodes and Expand operator, we have to clarify how property and label values of nodes and edges are accessed in the selection condition of a selection operator. As described in the previous section, graph relations only contain node and edge identifier, whereas the property and label data is stored as a property graph. Assuming that x is a column of a graph relation, we use the notation “$x.a$” in selection conditions to express the access to the corresponding value of property a in the property graph. If x represents a node, then we can access the label of x with the term “$x.l_v$” (or $x.l_e$, if x is an edge).

We now introduce the GetNodes operator, which is denoted by $\bigcirc_x$, where x is the name of the newly created column. Before defining the operator, Example 3.1 demonstrates the operation of GetNodes.

Example 3.1 *Consider the graph G from Figure 1. The expression $\bigcirc_x$ returns the graph relation below (left) containing the IDs of all nodes of G. The expression $\sigma_{x.name=\text{“Al Pacino”}}(\bigcirc_x)$ returns the following table on the right.*



Definition 3.1 (GetNodes Operator) *Consider a property graph G with a set of nodes V. The GetNodes operator $\bigcirc_x$ returns a graph relation with a single attribute x which contains the nodes of G.*

$$val(\bigcirc_x) = V$$

$$sch(\bigcirc_x) = \langle x \rangle$$

Note that $val(R)$ is the set of tuples of R and $sch(R)$ is the schema of R.

Next, we define the Expand operator. The Expand operator is used to return the neighbors of a given node. Therefore, a sequence of Expand operators specifies a search order in which a graph is traversed to find a specific pattern. Example 3.2 illustrates the idea behind this operator.

Example 3.2 *Consider graph G of Figure 1. Assume we want to find all subgraphs in G that match the pattern defined in Example 2.1. First, the expression below returns the actor node with the name “Al Pacino”.*

$$\sigma_{x.l_v=\text{“Actor”} \wedge x.name=\text{“Al Pacino”}}(\bigcirc_x)$$

Then, we need to find the neighbors of x that are connected by an outgoing *edge. Therefore, we use the* ExpandOut *operator denoted by $\uparrow_x^y$, where y is the new column that is* *added to the current graph relation containing the neighbors of nodes from x.*

$$\uparrow_x^y (\sigma_{x.l_v=\text{"Actor"} \wedge x.name=\text{"Al Pacino"}}(\bigcirc_x))$$

The expression above returns the following graph relation, in which column xy contains the edges from x to y.

x	xy	y
2	1	1
2	5	5

As a next step, we need to retrieve the director nodes that are connected to nodes contained in y. In order to do so, we again use the Expand *operator. However, we now need to return the nodes that are connected by incoming edges to nodes of y. Hence, we use the* ExpandIn *operator denoted by* $\downarrow_y^z$*.*

$$\downarrow_y^z (\uparrow_x^y (\sigma_{x.l_v=\text{"Actor"} \wedge x.name=\text{"Al Pacino"}}(\bigcirc_x)))$$

The expression above returns the following graph relation, in which zy represents edges from z to y.

x	xy	y	zy	z
2	1	1	1	2
2	1	1	2	3
2	1	1	3	4
2	5	5	5	2
2	5	5	4	4
2	5	5	6	6

As shown in this example, the Expand *operator has two directions:* ExpandOut *is used for outgoing edges (denoted by ↑), whereas* ExpandIn *is used for ingoing edges (denoted by ↓). The variable given at the bottom of the* ExpandOut *or* ExpandIn *operator references an existing column of the graph relation contained in the input. The variable given at the top of the* ExpandOut *or* ExpandIn *operator is the name of the newly created column.*
In the last step of the example query, we select the subgraphs that contain the director"Michael Mann".

$$\sigma_{z.l_v=\text{"Director"} \wedge z.name=\text{"Michael Mann"}}(\downarrow_y^z (\uparrow_x^y (\sigma_{x.l_v=\text{"Actor"} \wedge x.name=\text{"Al Pacino"}}(\bigcirc_x))))$$

As a result, we get the following graph relation, which is equal to the graph relation obtained in Example 2.1.

x	xy	y	zy	z
2	1	1	3	4
2	5	5	4	4

Definition 3.2 (Expand Operator) *Let R be a graph relation and $x \in attr(R)$ an attribute. The* ExpandOut *operator $\uparrow_x^y (R)$ adds a new column y to R containing the nodes that can be reached by an outgoing edge from nodes of x. In addition, a column xy is added to R containing the corresponding edges from nodes of x to nodes of y. The symbol ∥ denotes the concatenation of schema tuples.*

$$val(\uparrow_x^y (R)) = \{ \langle t, e, v \rangle \mid t \in val(R) \wedge e \in E \wedge \lambda(e) = (t.x, v)\}$$

$$sch(\uparrow_x^y (R)) = sch(R) \parallel \langle xy, y \rangle$$

The ExpandIn *operator $\downarrow_x^y (R)$ adds a new column y to R containing the nodes that can be reached by an ingoing edge from nodes of x. In addition, a column yx is added to R containing the corresponding edges from nodes of y to nodes of x.*

$$val(\downarrow_x^y (R)) = \{ \langle t, e, v \rangle \mid t \in val(R) \wedge e \in E \wedge \lambda(e) = (v, t.x)\}$$

$$sch(\downarrow_x^y (R)) = sch(R) \parallel \langle yx, y \rangle$$

The column name for edges is a concatenation of the column names of the corresponding nodes. The column name of an edge also indicates the direction of an edge. For example, "xy" is an edge from x to y and "yx" is an edge from y to x.

The components of a schema tuple $s \in sch(R)$ and a data tuple $t \in val(R)$ are ordered. However, in this work, we say that two schemata are equal if they have the same set of attributes. Therefore, two tuples are equal if they have the same values in each of their attributes.

In Section 5, we demonstrate how the Expand operator can be directly mapped to the physical `Expand(All)` operator of Neo4j. Despite this direct correspondence, Expand is a logical operator since in a different storage backend it will be evaluated by another operator, for example in the case of a relational database system, as a join of edge tables.

4. EQUIVALENCE RULES

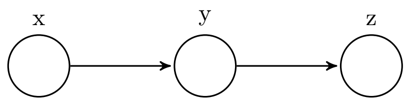
Based on the GetNodes and Expand operator that we introduced in the previous section, we define and prove the equivalence rules of our algebra. In this section, we only describe equivalence rules that are used in the optimizations in Section 5, which serve as a proof-of-concept for our approach. The definition of a complete set of equivalence rules is subject to future work.

Assume that we want to find all patterns, where a node (denoted with x) is connected by an outgoing edge to another node (denoted by y). This pattern can be traversed either by starting at x nodes and getting the outgoing edges or by starting at y nodes and getting the incoming edges. Rule 4.1 states this simple fact.

$$\uparrow_x^y (\bigcirc_x) \equiv \downarrow_y^x (\bigcirc_y) \tag{4.1}$$

Proof. Both expressions have the same set of attributes: $attr(\uparrow_x^y (\bigcirc_x)) = attr(\downarrow_y^x (\bigcirc_y)) = \{x, xy, y\}$. In addition, we have to show that $val(\uparrow_x^y (\bigcirc_x)) \subseteq val(\downarrow_y^x (\bigcirc_y)$ and $val(\downarrow_y^x (\bigcirc_y)) \subseteq val(\uparrow_x^y (\bigcirc_x))$ holds. Let $t \in val(\uparrow_x^y (\bigcirc_x))$ be a tuple, then there is an edge e with $\lambda(e) = (t.x, t.y)$. As a consequence, $t \in val(\downarrow_y^x (\bigcirc_y))$ holds. Therefore, $val(\uparrow_x^y (\bigcirc_x)) \subseteq val(\downarrow_y^x (\bigcirc_y)$ holds. $val(\downarrow_y^x (\bigcirc_y)) \subseteq val(\uparrow_x^y (\bigcirc_x))$ can shown analogously and is thus omitted. □

Assume that we have the pattern as above, but that we additionally want to find nodes (denoted by z) that are connected by outgoing edges starting from y nodes. The following graph shows such a pattern.



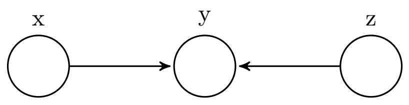
Rule 4.2 states that this pattern can be traversed by starting at z nodes and determining the incoming edges.

$$\uparrow_y^z (\uparrow_x^y (\bigcirc_x)) \equiv \downarrow_y^x (\downarrow_z^y (\bigcirc_z)) \tag{4.2}$$

Proof. Again, we have to show that both expressions return the same tuples. Consider a tuple $t \in val(\uparrow_y^z (\uparrow_x^y (\bigcirc_x)))$.

Then, there is an edge e with $\lambda(e) = (t.y, t.z)$. From this it follows that $t[y,z] \in val(\downarrow_z^y (\bigcirc_z))$, where $t[y,z]$ is the projection of t on the attributes y and z. Since $t \in val(\uparrow_y^z (\uparrow_x^y (\bigcirc_x)))$ holds, it also follows that there is an edge e' with $\lambda(e') = (t.x, t.y)$. Therefore, $t \in val(\uparrow_y^z (\uparrow_x^y (\bigcirc_x)))$ holds. $val(\downarrow_y^x (\downarrow_z^y (\bigcirc_z))) \subseteq val(\uparrow_y^z (\uparrow_x^y (\bigcirc_x)))$ is shown analogously. □

Furthermore, consider the following pattern in which y nodes have ingoing edges from x and z nodes.



Rule 4.3 states that this pattern can also be traversed by starting at z nodes.

$$\downarrow_y^z (\downarrow_y^x (\bigcirc_y)) \equiv \downarrow_y^x (\uparrow_z^y (\bigcirc_z)) \tag{4.3}$$

As the structure of the proof for Rule 4.3 is very similar to the proof for Rule 4.2, we do not include a proof for Rule 4.3 at this point.

The Rules 4.1, 4.2, and 4.3 describe the basic cases how a pattern can be traversed. For patterns with more than three nodes, we introduce Rule 4.4. Rule 4.4 states that the order of two Expand operators can be interchanged if neither of the operators accesses an attribute that is introduced by the other operator. Formally, the rule is defined as follows. Consider two Expand operators $\alpha, \beta \in \{\uparrow, \downarrow\}$, a graph relation E, and the attributes $a, c \in attr(\mathrm{E})$. Then, the following equivalence holds.

$$\beta_c^d(\alpha_a^b(\mathrm{E})) \equiv \alpha_a^b(\beta_c^d(\mathrm{E})) \tag{4.4}$$

Proof. As a precondition for the attributes $a, c \in attr(\mathrm{E})$, it has to hold that $a \neq d$ and $b \neq c$. Since the direction of β_c^d and α_a^b is not changed by switching their evaluation order, $sch(\beta_c^d(\alpha_a^b(\mathrm{E}))) = sch(\alpha_a^b(\beta_c^d(\mathrm{E})))$ holds. Suppose $t \in val(\beta_c^d(\alpha_a^b(\mathrm{E})))$. It follows that there is an edge between $t.c$ and $t.d$ where $c \in attr(\mathrm{E})$. As a consequence, $t[c, d, attr(\mathrm{E})] \in val(\beta_c^d(\mathrm{E}))$. From $t \in val(\beta_c^d(\alpha_a^b(\mathrm{E})))$ it follows that there is an edge between $t.a$ and $t.b$ where $a \in attr(\mathrm{E})$. Therefore, $t \in \alpha_a^b(\beta_c^d(\mathrm{E}))$ holds. The other direction can be shown analogously. □

Up to now, we only looked at traversals that start at a single node. However, a pattern can also be traversed by starting at multiple nodes and joining the resulting subpatterns. For example, consider the previous pattern, in which y nodes have ingoing edges from x and z nodes. We could traverse this pattern by starting at x and z nodes. Then, we determine the y nodes that are connected by outgoing edges to the x nodes. For the z nodes, we also determine the y nodes that are connected by outgoing edges. As a result, we get a set of subgraphs with x and y nodes connected to each other and a set of subgraphs with z and y nodes connected to each other. As a last step, we have to join the subgraphs that have the same y nodes. Rule 4.5 below gives the equivalence between these two alternative evaluation strategies.

$$\downarrow_y^z (\uparrow_x^y (\mathrm{E})) \equiv \uparrow_x^y (\mathrm{E}) \bowtie \uparrow_z^y (\bigcirc_z) \tag{4.5}$$

Proof. Consider $t \in val(\downarrow_y^z (\uparrow_x^y (\mathrm{E})))$. Then, there is an edge e with $\lambda(e) = (t.x, t.y)$. Consequently, $t[x,y] \in val(\uparrow_x^y (\mathrm{E}))$ holds. Since $t \in val(\downarrow_y^z (\uparrow_x^y (\mathrm{E})))$ holds, it also follows that there is an edge e' with $\lambda(e') = (t.z, t.y)$. Therefore, $t[y,z] \in val(\uparrow_z^y (\bigcirc_z))$ holds. $t[x,y]$ and $t[y,z]$ have the same value in y. As a consequence, $t \in val(\uparrow_x^y (\mathrm{E}) \bowtie \uparrow_z^y (\bigcirc_z))$ holds. The other direction can be shown analogously. □

Pushing selections down in the operator tree of a query evaluation plan is an important and well-known heuristic to reduce the size of intermediate results. Therefore, we define Rule 4.6, which states that the order of a selection and the Expand operator can be interchanged if no variable in the selection condition is introduced by the Expand operator. More formally, let $\alpha \in \{\uparrow, \downarrow\}$ be an Expand operator and F a selection condition. Without loss of generality, assume that α creates a new column y. If F does not contain properties/labels of node y or the edge introduced by α, then the following equivalence holds.

$$\sigma_{\mathrm{F}}(\alpha_x^y(\mathrm{E})) \equiv \alpha_x^y(\sigma_{\mathrm{F}}(\mathrm{E})) \tag{4.6}$$

The proof for Rule 4.6 is trivial and is therefore omitted.

5. OPTIMIZATIONS

In this section, we demonstrate how Cypher queries can be optimized at the logical level by transforming them using the equivalences of Section 4. In order to demonstrate the potential benefits of this approach, we will present example queries for which the optimizer of the Neo4j Community Edition Version 2.3.1 does not find the best execution plan.

First, we show how the Cypher query that we used as a motivating example in the introduction can be transformed into a more efficient query. The following algebra expression is a logical representation of the Cypher query given in the introduction.

$$\sigma_{z.l_v=\text{“Actor”} \wedge z.name=\text{“Robert De Niro”}}(\downarrow_y^z (\uparrow_x^y (\sigma_{x.l_v=\text{“Actor”} \wedge x.name=\text{“Al Pacino”}}(\bigcirc_x))))$$

This statement first selects all actor nodes with the name “Al Pacino”. Afterwards, the neighbors connected by outgoing edges are determined. For each of the resulting nodes of this ExpandOut operation the neighbors connected by incoming edges are determined and only the neighbors with the name “Robert De Niro” are selected. In order to transform this logical plan into a physical plan, a cost model would be required, which we plan to develop in future work. For evaluation purposes, we therefore try to manually find a physical plan that has the same evaluation order as our logical plan. A physical plan that satisfies this requirement is the first execution plan given in the introduction.

Since the current form of the logical algebra expression does not yield the most efficient execution strategy, we transform it into an alternative expression by applying the equivalence rules defined in Section 4. The original expression begins the evaluation at a single actor node and then expands this node twice. As a consequence, we retrieve all actors that starred in all movies in which the first actor also starred. Since we are ultimately only looking for one actor, this execution strategy does not use the most selective access path and we end up loading to much data. A better strategy would be to retrieve both actor nodes separately, expand them to get the nodes corresponding to movies the actors starred in, and then perform a join on these movie nodes. This execution strategy is more efficient because it avoids accessing the set of all person nodes related to movies of “Al Pacino”, which is much larger than the sets of movies nodes related to “Al Pacino” and “Robert De Niro”.

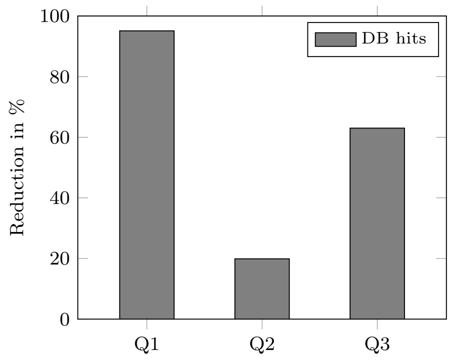


Figure 2: Reduction of the DB hits in %.

In order to algebraically transform the original expression into the more efficient expression, we apply the following rules. First, we use Rule 4.5 to introduce a join.

$$\overset{4.5}{\equiv} \sigma_{z.l_v=\text{“Actor”}\wedge z.name=\text{“Robert De Niro”}}\big(\uparrow_x^y (\sigma_{x.l_v=\text{“Actor”}\wedge x.name=\text{“Al Pacino”}}(\bigcirc_x)) \bowtie \uparrow_z^y (\bigcirc_z)\big)$$

As a next step, we push the outer selection into the join. Note that this transformation can be done using an existing equivalence rule from the relational algebra.

$$\equiv \uparrow_x^y (\sigma_{x.l_v=\text{“Actor”}\wedge x.name=\text{“Al Pacino”}}(\bigcirc_x)) \bowtie \sigma_{z.l_v=\text{“Actor”}\wedge z.name=\text{“Robert De Niro”}}\big(\uparrow_z^y (\bigcirc_z)\big)$$

As a last step, we move the selection into the Expand operator by applying Rule 4.6.

$$\overset{4.6}{\equiv} \uparrow_x^y (\sigma_{x.l_v=\text{“Actor”}\wedge x.name=\text{“Al Pacino”}}(\bigcirc_x)) \bowtie \uparrow_z^y (\sigma_{z.l_v=\text{“Actor”}\wedge z.name=\text{“Robert De Niro”}}(\bigcirc_z))$$

The evaluation order of the resulting expression is the same as in the execution plan of the transformed Cypher query given in the introduction. In order to force Neo4j to execute the query using this plan, we add the `USING INDEX` hint. Compared to the original query, the number of database hits is reduced by about 95% (*cf.* Figure 2, Q1).

The next Cypher query, which we use as an example to demonstrate the transformation-based optimization supported by our algebra, is given below. The query returns all subgraphs in which a person gives a five star rating to a movie in which “Al Pacino” acted.

```
MATCH (x:Person)-[e:RATED]->(y)<- -(z)
WHERE e.stars = 5 AND z.name = “Al Pacino”
RETURN *
```

In order to execute this query, Neo4j uses the following execution plan.

```
Filter[e.stars=5 ∧ x.l_v =“Person”]
|
Expand(All)[y<-[e:RATED]-x]
|
Expand(All)[z- ->y]
|
Filter[z.name=“Al Pacino”]
|
AllNodesScan[z]
```

Again, this is not the most efficient execution plan because it accesses all nodes of the graph in the first step. As in the previous example, Neo4j can be forced to use a better execution plan by including the hint “`USING SCAN` x:Person”. This more efficient execution plan is given below.

```
Filter[z.name=“Al Pacino”]
|
Expand(All)[y<- -z]
|
Filter[e.stars=5]
|
Expand(All)[x-[e:RATED]->y]
|
NodeByLabelScan[x:Person]
```

Instead of scanning all nodes of the graph, this execution plan only accesses nodes with the label “Person”. Therefore, the number of database hits is reduced by about 20% (*cf.* Figure 2, Q2). Since the optimizer of Neo4j is unable to find this execution plan without hints, we show how our algebra and its equivalence rules can be used to enumerate plans with alternative evaluation orders at the logical level.

We begin with a logical representation of the Cypher query that has the same evaluation order as the original execution plan.

$$\sigma_{xy.l_e=\text{“RATED”}\wedge xy.stars=5\wedge x.l_v=\text{“Person”}}\big(\downarrow_y^x \big(\uparrow_z^y (\sigma_{z.name=\text{“Al Pacino”}}(\bigcirc_z))\big)\big)$$

Next, we move the inner selection out of the Expand operators.

$$\overset{4.6}{\equiv} \sigma_{xy.l_e=\text{“RATED”}\wedge xy.stars=5\wedge x.l_v=\text{“Person”}}\big(\sigma_{z.name=\text{“Al Pacino”}}(\downarrow_y^x (\uparrow_z^y (\bigcirc_z)))\big)$$

Then, the order of the edge traversals is changed.

$$\overset{4.1}{\equiv} \sigma_{xy.l_e=\text{“RATED”}\wedge xy.stars=5\wedge x.l_v=\text{“Person”}}\big(\sigma_{z.name=\text{“Al Pacino”}}(\downarrow_y^x (\downarrow_y^z (\bigcirc_y)))\big)$$

$$\overset{4.3}{\equiv} \sigma_{xy.l_e=\text{“RATED”}\wedge xy.stars=5\wedge x.l_v=\text{“Person”}}\big(\sigma_{z.name=\text{“Al Pacino”}}(\downarrow_y^z (\uparrow_x^y (\bigcirc_x)))\big)$$

Finally, the outer selection is pushed down.

$$\equiv \sigma_{z.name=\text{“Al Pacino”}}\big(\downarrow_y^z \big(\sigma_{xy.l_e=\text{“RATED”}\wedge xy.stars=5}\big(\uparrow_x^y (\sigma_{x.l_v=\text{“Person”}}(\bigcirc_x))\big)\big)\big)$$

This logical plan has the same evaluation order as the alternative execution plan that we introduced above.

The last Cypher query that we use as an example to illustrate the benefits of our approach returns all movies, in which “Al Pacino” acted and which “Michael Mann” directed.

```
MATCH (x:Actor)- ->(y)<- -(z:Director)
WHERE x.name = “Al Pacino” AND z.name = “Michael Mann”
RETURN y
```

For this example query, Neo4j uses the execution plan given below.

```
Filter[z.name=“Michael Mann” ∧ z.l_v =“Director”]
|
Expand(All)[y<- -z]
|
Expand(All)[x- ->y]
|
NodeIndexSeek[x.name=“Al Pacino”]
```

The pattern search begins traversing the graph at the “Al Pacino” actor node. However, it is more efficient to start at the "Michael Mann" director node. By adding "`USING INDEX` z:Director(name)" to the Cypher query, Neo4j produces the following execution plan that corresponds to this improved execution strategy.

```
Filter[x.name="Al Pacino" ∧ x.l_v ="Actor"]
                 |
        Expand(All)[y<- -x]
                 |
        Expand(All)[z- ->y]
                 |
 NodeIndexSeek[z.name="Michael Mann"]
```

This alternative plan reduces the number of database hits by about 60% (*cf.* Figure 2, Q3). In order to change the evaluation order of the first plan into the evaluation order of the alternative plan, the corresponding logical expressions can be transformed as follows.

$$\pi_y(\sigma_{z.name=\text{“Michael Mann”}\wedge\ z.l_v=\text{“Director”}}(\downarrow_y^z(\uparrow_x^y(\sigma_{x.name=\text{“Al Pacino”}\wedge x.l_v=\text{“Actor”}}(\bigcirc_x)))))$$

As a first step, we move the inner selection out of the Expand operators.

$$\overset{4.6}{\equiv}\pi_y(\sigma_{z.name=\text{“Michael Mann”}\wedge\ z.l_v=\text{“Director”}}(\sigma_{x.name=\text{“Al Pacino”}\wedge x.l_v=\text{“Actor”}}(\downarrow_y^z(\uparrow_x^y(\bigcirc_x)))))$$

Then, the order of the node traversals can be changed.

$$\overset{4.1}{\equiv}\pi_y(\sigma_{z.name=\text{“Michael Mann”}\wedge\ z.l_v=\text{“Director”}}(\sigma_{x.name=\text{“Al Pacino”}\wedge x.l_v=\text{“Actor”}}(\downarrow_y^z(\downarrow_y^x(\bigcirc_y)))))$$

$$\overset{4.3}{\equiv}\pi_y(\sigma_{z.name=\text{“Michael Mann”}\wedge\ z.l_v=\text{“Director”}}(\sigma_{x.name=\text{“Al Pacino”}\wedge x.l_v=\text{“Actor”}}(\downarrow_y^x(\uparrow_z^y(\bigcirc_z)))))$$

Finally, the outer selection is pushed down.

$$\overset{4.6}{\equiv}\pi_y(\sigma_{x.name=\text{“Al Pacino”}\wedge x.l_v=\text{“Actor”}}(\downarrow_y^x(\uparrow_z^y(\sigma_{z.name=\text{“Michael Mann”}\wedge\ z.l_v=\text{“Director”}}(\bigcirc_z)))))$$

The evaluation order of this logical expression is equivalent to the execution order of the physical plan that Neo4j produces if the hint "`USING INDEX` z:Director(name)" is added to the original Cypher query as described above.

6. RELATED WORK

Numerous graph query languages have been proposed in the literature [6, 8, 10, 13, 15, 17]. However, in contrast to relational database systems, a standard query language for graph databases has yet to emerge. The lack of such a common query language might be a reason why there are still relatively few works on graph query optimization.

Schmidt *et al.* [20] study the foundations of query optimization in the context of SPARQL. They introduce algebraic rewriting rules as well as semantic optimizations on the SPARQL code level. Yakovets *et al.* [21] work on a cost-based optimization of SPARQL property paths. In their approach, an execution plan is composed of finite automata defining the search order of patterns in the graph. He and Singh [15] introduce an algorithm for finding patterns in a graph. A building block in this algorithm is a procedure that determines the search order of nodes based on a cardinality estimation.

To the best of our knowledge, there is also no common algebra that is general enough to represent the graph queries of all proposed languages. Cyganiak [9] describes how simple SPARQL graph patterns can be mapped to relational algebra expressions. Harris and Shadbolt [14] and Chebotko *et al.* [5] also show how a subset of SPARQL can be represented by relational algebra expressions. However, by using the standard relational algebra, a graph traversal has to be represented as a sequence of joins. As joins work on logical addresses, these approaches are not well suited for native graph databases such as Neo4j, which use physical addresses (pointers) to traverse neighboring nodes. The Expand operator proposed in this paper is independent of the underlying data storage and processing model. It is therefore a first step towards a general graph algebra.

Sakr *et al.* [19] introduce the query language G-SPARQL, which extends SPARQL for querying property graphs. In addition, the authors show how a property graph can be stored in a relational database system by adapting a decomposed storage model (DSM) [1, 7]. The authors also define a set of algebraic operators that are used to represent G-SPARQL queries. First, they introduce simple operators to retrieve neighboring nodes or to retrieve property values of nodes and edges. Second, they define more complex operators as, for example, an operator to compute shortest paths. As a consequence, the algebra of Sakr *et al.* is a better foundation for graph databases than the pure relational algebra. Nevertheless, their algebra is still limited. For a given node, only outgoing edges can be traversed, wheras native graph databases such as Neo4j, also support the retrieval of neighboring nodes that are connected by incoming edges. Our Expand operator can represent both directions and is therefore more general. Additionally, Sakr *et al.* do not propose any equivalence rules for their algebra that can be used to enumerate different traversal orders.

He and Singh [15] introduce an algebra for property graphs, which is also based on the relational algebra. They introduce a composition operator that is used to generate a new graph from a matched graph. In addition, the authors generalize the selection operator to graph pattern matching. Therefore, on the logical level, the pattern matching is represented by a single selection operator for which the authors propose an access method. Due to this coarse granularity, the algebra of He and Singh is not an ideal basis for query plan search space exploration in a transformation-based query optimizer.

7. CONCLUSION AND FUTURE WORK

In this work, we presented how the relational algebra can be extended with a general Expand operator in order to enable the transformation-based enumeration of different traversal patterns for finding a pattern in a graph G. The input and output of our algebra operators are so-called graph relations. The tuples of a graph relation represent subgraphs of G matching a given pattern. Additionally, we introduced a set of equivalence rules and gave proofs for their correctness. In the context of Neo4j, we demonstrated how these equivalences can be used to algebraically transform Cypher query at the logical level. Finally, we illustrated the potential performance benefits of this approach by comparing the database hits of the query evaluation plan found to Neo4j to one resulting from applying our equivalence rules.

Our long-term goal is to design and develop a transformation-based query optimizer for graph databases. Although the work presented in this paper is a first step towards this goal, there remain open challenges that we will address in future work. First, the algebra presented in this paper is restricted to simple pattern matching functionalities. However, in graph query languages such as Cypher, it is possible to search for variable length patterns or to aggregate nodes. Therefore, we plan to integrate these features in our algebra in a next step.

Second, apart from enumerating alternative query plans in the search space, a query optimizer also needs to assign a cost to these plans in order to choose the most efficient one. While plan enumeration is supported by a general graph algebra together with its equivalences, a corresponding cost model has yet to be developed. Note that reduction of database hits reported in Section 5 are measured empirically by executing the two versions of the query in Neo4j, rather than estimated by an analytical cost model.

In order to build this query optimizer, we plan to use the Cascades framework defined by Graefe [12]. On the one hand, Cascades is based on a flexible and modular architecture, which facilitates the integration of new operators and transformation rules. On the other hand, the Cascades framework is used in commercial systems such as Microsoft SQL Server. With this implementation as a foundation, we can both study how graph-specific cardinality statistics affect query evaluation and integrate advanced search space pruning strategies [2, 16].

Acknowledgement

This work is supported by Grant No. GR 4497/2 of the Deutsche Forschungsgemeinschaft (DFG). Additionally, the authors would like to thank Leonard Wörteler for his feedback on an early version of the graph algebra presented in this paper.

8. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management using Vertical Partitioning. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 411–422, 2007.

[2] R. Ahmed, R. Sen, M. Poess, and S. Chakkappen. Of Snowstorms and Bushy Trees. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1452–1461, 2014.

[3] S. Amer-Yahia, L. V. S. Lakshmanan, and C. Yu. SocialScope: Enabling Information Discovery on Social Content Sites. In *Proc. Intl. Conf. on Innovative Database Research (CIDR)*, pages 525–528, 2009.

[4] M. Arenas and J. Pérez. Querying Semantic Web Data with SPARQL. In *Proc. Intl. Symp. on Principles of Database Systems (PODS)*, pages 305–316, 2011.

[5] A. Chebotko, S. Lu, and F. Fotouhi. Semantics Preserving SPARQL-to-SQL Translation. *Data Knowledge Engineering*, 68:973–1000, 2009.

[6] M. P. Consens and A. O. Mendelzo. GraphLog: A Visual Formalism for Real Life Recursion. In *Proc. Intl. Symp. on Principles of Database Systems (PODS)*, pages 404–416, 1990.

[7] G. P. Copeland and S. N. Khoshafian. A Decomposition Storage Model. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 268–279, 1985.

[8] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A Graphical Query Language Supporting Recursion. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 323–330, 1987.

[9] R. Cyganiak. A Relational Algebra for SPARQL. Technical report, HP Labs, Bristol, UK, 2005.

[10] A. Dries, S. Nijssen, and L. D. Raedt. A Query Language for Analyzing Networks. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 484–494, 2009.

[11] W. Fan. Graph Pattern Matching Revised for Social Network Analysis. In *Proc. Intl. Conf. on Database Theory (ICDT)*, pages 8–21, 2012.

[12] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, 18:19–29, 1995.

[13] R. H. Güting. GraphDB: Modeling and Querying Graphs in Databases. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 297–308, 1994.

[14] S. Harris and N. Shadbolt. SPARQL Query Processing with Conventional Relational Database Systems. In *Proc. Intl. Conf. on Web Information Systems Engineering (WISE)*, pages 235–244, 2005.

[15] H. He and A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 405–418, 2008.

[16] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 168–177, 1991.

[17] U. Leser. A Query Language for Biological Networks. *Bioinformatics*, 21:33–39, 2005.

[18] M. S. Martín, C. Gutiérrez, and P. T. Wood. SNQL: A Social Network Query and Transformation Language. In *Proc. Intl. Workshop on Foundations of Data Management (AMW)*, 2011.

[19] S. Sakr, S. Elnikety, and Y. He. G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 335–344, 2012.

[20] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL Query Optimization. In *Proc. Intl. Conf. on Database Theory (ICDT)*, pages 4–33, 2010.

[21] N. Yakovets, P. Godfrey, and J. Gryz. WAVEGUIDE: Evaluating SPARQL Property Path Queries. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, pages 525–528, 2015.