© The British Computer Society 2020. All rights reserved.
For permissions, please e-mail: journals.permissions@oup.com
Advance Access publication on 9 May 2020
doi: 10.1093/comjnl/bxaa031

From Relation Algebra to Semi-join Algebra: An Approach to Graph Query Optimization

Jelle Hellings[1], Catherine L. Pilachowski[2], Dirk Van Gucht[2], Marc Gyssens[3,*] and Yuqing Wu[4]

[1]*Exploratory Systems Lab, Department of Computer Science, University of California, Davis, CA 95616-8562, USA, and Hasselt University, Faculty of Sciences, Martelarenlaan 42, 3500 Hasselt, Belgium*
[2]*Indiana University, Luddy School of Informatics, Computing and Engineering, 919 E 10th Street, Bloomington, IN 47408, USA*
[3]*Hasselt University, Faculty of Sciences, Data Science Institute, Martelarenlaan 42, 3500 Hasselt, Belgium*
[4]*Pomona College, 185 E 6th Street, Claremont, CA 91711, USA*
**Corresponding author: marc.gyssens@uhasselt.be*

Many graph query languages rely on composition to navigate graphs and select nodes of interest, even though evaluating compositions of relations can be costly. Often, this need for composition can be reduced by rewriting toward queries using semi-joins instead, resulting in a significant reduction of the query evaluation cost. We study techniques to recognize and apply such rewritings. Concretely, we study the relationship between the expressive power of the relation algebras, which heavily rely on composition, and the semi-join algebras, which replace composition in favor of semi-joins. Our main result is that each fragment of the relation algebras where intersection and/or difference is only used on edges (and not on complex compositions) is expressively equivalent to a fragment of the semi-join algebras. This expressive equivalence holds for node queries evaluating to sets of nodes. For practical relevance, we exhibit constructive rules for rewriting relation algebra queries to semi-join algebra queries and prove that they lead to only a well-bounded increase in the number of steps needed to evaluate the rewritten queries. In addition, on sibling-ordered trees, we establish new relationships among the expressive power of Regular XPath, Conditional XPath, FO-logic and the semi-join algebra augmented with restricted fixpoint operators.

Keywords: graph query optimization; relation algebra; semi-join algebra; rewriting; transitive closure; restricted transitive closure; expressiveness

Received 3 May 2019; Revised 19 February 2020; Accepted 1 March 2020
Handling editor: Peter Wood

1. INTRODUCTION

The graph data model (representing labeled binary relations) is a versatile and natural data model for representing RDF data, social networks, gene and protein networks and other sources of data.

Example 1.1. In Fig. 1 we show a simple social network represented by graph data. Its nodes represent objects corresponding to persons and its edges represent various semantic relationships between these persons. Here, we have *ParentOf* and *FriendOf* relationships.

In Fig. 2 we show the same data represented as labeled binary relations.

To query such graph data, a multitude of navigational and pattern-based graph query languages have been proposed [1]. Our focus in this paper is primarily on navigational query languages, as even pattern-based query languages—such as

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

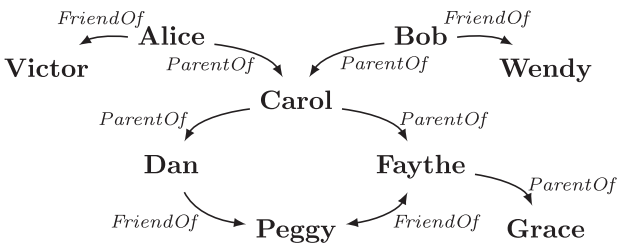

FIGURE 1. An example of social network graph data.

ParentOf

Alice	Carol
Bob	Carol
Carol	Dan
Carol	Faythe
Faythe	Grace

FriendOf

Alice	Victor
Bob	Wendy
Dan	Peggy
Faythe	Peggy
Peggy	Faythe

FIGURE 2. Labeled binary relations representing the graph data in Fig. 1.

SPARQL [2], Cypher [3] and Gremlin [4]—often rely on (sub)queries that are essentially navigational in nature [1].

Many navigational query languages use, at their core, a fragment of the relation algebra of Tarski [5], augmented with the Kleene-star operator (transitive closure). Examples of such relation-algebra-inspired query languages include XPath and its many formalizations [6–12], GXPath [13], the (nested) regular path queries [14] and the navigational expressions [15–19].

In query languages, graph navigation is primarily supported by *composition* ($\circ$). To see this, consider the query defined by the following relation algebra expression:

$$\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{FriendOf}.$$

This query searches for all pairs of people (m, n) such that n is a friend of m's grandchild. When applied to the data shown in Fig. 1, it returns the binary relation $\{(\text{Alice}, \text{Peggy}), (\text{Bob}, \text{Peggy})\}$. As another example, consider the expression

$$\mathit{GgpAndFriends} \equiv \pi_1[\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}] \circ \mathit{FriendOf},$$

which defines the query that yields the set of all pairs of great-grandparents and their friends. Indeed, in this expression, the projection

$$\pi_1[\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}]$$

returns the set of pairs of the form (m, m) where m is a great-grandparent, i.e. the pairs $\{(\text{Alice}, \text{Alice}), (\text{Bob}, \text{Bob})\}$. When this relation is composed with the *FriendOf* relation, we get the desired result $\{(\text{Alice}, \text{Victor}), (\text{Bob}, \text{Wendy})\}$.

These examples illustrate that the composition operator captures the intent of graph navigation in a simple and intuitive way, which explains why many graph query languages rely on it. In the setting of big data, this use of composition for graph navigation has a major drawback, however. Computing query results by evaluating each of the compositions involved is costly, both in terms of runtime and in terms of memory requirements. This is already evident from the above example: the subquery $\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}$ yields pairs of great-grandparents and great-grandchildren. After computing these pairs, the projection-step will discard all computed information on the great-grandchildren. Consequently, evaluating the query *GgpAndFriends* by evaluating each of the operators involved is a relative wasteful process and a more efficient approach would be to find all great-grandparents without also computing all great-grandchildren.

We notice that the above example reflects typical constructs used in graph query languages. Indeed, languages, such as XPath, GXPath and the Nested RPQs allow for the selection of nodes based on some navigational conditions (e.g. the '$[q]$' construct in XPath, with q a query). In our work, such node selections are represented by the *projection* π_1.

For relatively simple queries such as *GgpAndFriends*, we can add operators to the relation algebra to enable the direct expression of more efficient query evaluation approaches at a high level. One way to do so is by adding the semi-join operators $\ltimes$ and $\rtimes$. Rather than computing the composition of relations, semi-joins only determine the pairs that are involved in such compositions. In particular, if R and S are binary relations, then the left semi-join $\mathrm{R} \ltimes \mathrm{S}$ determines the pairs in R that can be composed with pairs in S, i.e. $\{(m, n) \in \mathrm{R} \mid \exists z\ (n, z) \in \mathrm{S}\}$ and the right semi-join $\mathrm{R} \rtimes \mathrm{S}$ determines the pairs in S that can be composed with pairs in R, i.e. $\{(m, n) \in \mathrm{S} \mid \exists z\ (z, m) \in \mathrm{R}\}$. We can now rewrite the query *GgpAndFriends* into the path-equivalent query

$$\pi_1[\mathit{ParentOf} \ltimes (\mathit{ParentOf} \ltimes \mathit{ParentOf})] \rtimes \mathit{FriendOf}$$

using semi-joins instead of compositions. The main advantage of replacing compositions by semi-joins in the above query is easy to see: evaluation of the resulting query by evaluating each operation involved will yield small and efficient-to-compute intermediate results, whereas evaluation of the original query can yield intermediate results of quadratic size. As social networks and other graph datasets tend to be extremely large, the original composition-based approach is unacceptably expensive, whereas the semi-join-based approach might be feasible. This also holds in general, as it is well-known that evaluating semi-joins is more efficient than evaluating compositions, even in the worst case [20].

To achieve these improvements in practice, we can add the semi-join operators to appropriate query languages. This puts the burden of efficient query evaluation on the users, however. Observe that, in the above rewriting, we needed both the left

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

and right semi-join operators. With respect to the former, we additionally had to insert parenthesis to control the order of evaluation of this non-associative operator. So, even in this simple example, the resulting expression becomes less intuitive and harder to write. Therefore, we believe that in modern graph database systems, which use declarative high-level graph query languages, such rewritings should be performed *for* the users, rather than *by* the users.

Here, we study ways to apply these semi-join optimizations automatically. More concretely, we study how fragments of the relation algebra relate to fragments of the semi-join algebra, the latter obtained by replacing composition by semi-joins and the Kleene-star by appropriate less-costly forms of fixpoint iteration.

To the best of our knowledge, we are the first to study the relationships between the expressive power of the relation algebra and the semi-join algebra comprehensively. We should point out that the study of semi-joins has already received attention in the setting of Codd's relational algebra [21–25]. In this setting, the semi-join version of the relational algebra is studied as a query language that has limited expressive power, cheap query evaluation and for which many decision problems are decidable.

In the design and implementation of relational database systems, *basic* semi-join rewrite rules are well-known and the automatic usage of semi-join steps plays an important role in the efficient evaluation of distributed joins [26] and in Yannakakis's algorithm for evaluating acyclic joins [27, 28]. In both cases, these semi-join steps are used as *reducers* that provide a preprocessing step aimed at reducing the size of intermediate relations before joining them. A similar reducer-based role for the semi-join has also been studied in the context of the multiset relational algebra [29]. This focus on using the semi-join as a reducer sharply contrasts with our usage, as we aim at eliminating compositions altogether in favor of semi-joins.

The main contributions of this paper are of both theoretical and practical interest. First, the main theoretical results are as follows:

1. We show that the semi-join algebra has the same expressive power as FO[2], first-order logic in which formulae have at most two variables. Since the relation algebra has the same expressive power as FO[3] [5, 17], the semi-join algebra has less expressive power than the relation algebra [30–34].
2. To further establish the relationships in the expressive power of the relation algebra and the semi-join algebra, we investigate how the relative expressive power of fragments of the relation algebra compares to the relative expressive power of fragments of the semi-join algebra. We do so by showing that expressions of the form $\pi_j[e]$, $j \in \{1, 2\}$, can be rewritten into path-equivalent expressions in the semi-join algebra whenever the expression does not use intersection or difference. We call this path equivalence of projection-expressions *projection equivalence*.
3. To extend the above results to the setting in which also the Kleene-star operator occurs, we introduce a simple semi-join-style form of fixpoint iteration. We show that this form of fixpoint iteration can be expressed in $L^2_{\infty\omega}$, the infinitary first-order logic extension of FO[2]. We also show that expressions that use the Kleene-star operator can be rewritten into projection-equivalent expressions in the semi-join algebra augmented with this fixpoint operator.
4. The above-mentioned rewritings only put restrictions on the usage of intersection and difference. To show that these restrictions are not too severe, we prove that not all expressions using both composition and intersection are expressible in FO[2]. We identify syntactical restrictions on the usage of intersection and difference in the relation algebra and show that the resulting language fragments have exactly the same expressive power as the semi-join algebra with respect to projection equivalence. From these results, it follows that intersection and difference only provide limited expressive power in the semi-join algebra.
5. In the setting of finite sibling-ordered trees [9], we use the above results to strengthen the well-known collapse of first-order logic on sibling-ordered trees (FO^{tree}) to Conditional XPath (a fragment of the relation algebra) by proving a projection-equivalent collapse of FO^{tree} queries to the semi-join algebra.
6. Finally, we investigate how the newly introduced notion of projection equivalence compares to the standard notions of path equivalence and Boolean equivalence. In particular, we also strengthen the known Boolean equivalences between fragments of the relation algebra when querying graphs [35] to projection equivalences.

To put the above theory in practice, we also study how rewriting from the relation algebra to the semi-join algebra can be utilized for graph query optimization. With this aim, our main results are as follows:

1. We propose an algorithm for the efficient evaluation of the simple form of fixpoint iteration that we introduce, and we argue that the semi-join and fixpoint operators can be considered more efficient than compositions and Kleene-stars.
2. We revisit the analysis of the semi-join rewrite rules and add to this analysis by providing strict bounds on the size of rewritten expressions. We show that if the input of the semi-join rewrite rules is an expression of length s that uses at most u union-operators, then application of the rewrite rules we propose yields a rewritten expression that can be evaluated in at most $s + u \leq 2 \cdot s$

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

evaluation steps, demonstrating the practical feasibility of our rewriting techniques.

3. We show that the semi-join rewrite rules provide strict guarantees on the *data complexity*—the complexity in terms of the size of the graph—of evaluating queries by evaluating each of the operators involved. The application of the semi-join rewrite rules can decrease the data complexity of query evaluation significantly, and will never increase it.
4. Finally, we also identify the identity, diversity and coprojection operators as expensive to evaluate. We show how common usage of these operators can be replaced by specialized, efficient to evaluate, operators.

This work is a revised and extended version of the paper 'From relation algebra to semi-join algebra: an approach for graph query optimization' presented at the *16th International Symposium on Database Programming Languages*, Munich, Germany (DBPL 2017) [36]. In comparison to this previous work, we have added a reflection on projection equivalence (Section 4.4) and looked at other expensive operators besides composition (Section 6.5). We have also provided proof details of the results presented.

2. GRAPH DATA MODEL AND QUERIES

We use an edge-labeled graph data model. A *graph* is a triple $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$, with $\mathcal{V}$ a finite set of nodes, Σ a finite set of edge labels, and $\mathbf{E} : \Sigma \to 2^{\mathcal{V}\times\mathcal{V}}$ a function mapping edge labels to edge relations.

In our setting, a query q maps a graph to a set of node tuples. We write $[\![q]\!]_{\mathcal{G}}$ to denote the *evaluation* of q on graph $\mathcal{G}$. We can interpret a query q as a *Boolean query*, in which case $[\![q]\!]_{\mathcal{G}} \neq \emptyset$ represents `true`. For simplicity, we assume that queries always yield binary relations (sets of node-pairs, $[\![q]\!]_{\mathcal{G}} \subseteq \mathcal{V} \times \mathcal{V}$). From this perspective, queries that in spirit return nodes will return identical pairs of nodes. Therefore, we refer to queries whose output is a subset of $\{(n, n) \mid n \in \mathcal{V}\}$ as *node queries*.

Finally, if R is a binary relation, then we denote by $\mathrm{R}|_1$ the set of nodes $\{m \mid \exists n\ (m, n) \in \mathrm{R}\}$ and by $\mathrm{R}|_2$ the set of nodes $\{n \mid \exists m\ (m, n) \in \mathrm{R}\}$. The relation R is typically obtained by the evaluation of a query on a graph. For example, if q is a query and $\mathcal{G}$ a graph, then we write $[\![q]\!]_{\mathcal{G}}|_1$ to denote $\{m \mid \exists n\ (m, n) \in [\![q]\!]_{\mathcal{G}}\}$.

2.1. Equivalence notions

In this work, we prove relationships between query languages using rewrite rules. To reason about the soundness of these rewrite rules, we need notions of expression equivalence. We consider four such notions: the traditional path-equivalence and Boolean-equivalence, which have been studied in great detail [15, 16, 18, 19, 35, 37, 38], and left-projection-equivalence and right-projection-equivalence, which we introduce to study the rewrite rules we provide.

DEFINITION 2.1. *Let q_1 and q_2 be queries. We say that q_1 and q_2 are*

1. *path-equivalent, denoted by $q_1 \equiv_{\text{path}} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}} = [\![q_2]\!]_{\mathcal{G}}$;*[1]
2. *Boolean-equivalent, denoted by $q_1 \equiv_{\text{bool}} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}} = \emptyset$ if and only if $[\![q_2]\!]_{\mathcal{G}} = \emptyset$;*
3. *left-projection-equivalent, denoted by $q_1 \equiv_{\pi_1} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}}|_1 = [\![q_2]\!]_{\mathcal{G}}|_1$; and*
4. *right-projection-equivalent, denoted by $q_1 \equiv_{\pi_2} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}}|_2 = [\![q_2]\!]_{\mathcal{G}}|_2$.*

By definition, expressions that are path-equivalent must also be left-projection-equivalent and right-projection-equivalent. Expressions that are left-projection-equivalent or right-projection-equivalent must also be Boolean-equivalent. The converse is generally not true. (We will look at this in more detail in Section 4.4). We observe that path-equivalence, left-projection-equivalence and right-projection-equivalence coincide on the class of node expressions.

Next, we illustrate these equivalence notions:

EXAMPLE 2.1. Looking back at the example queries used in the Introduction, we have

$$\begin{aligned} &\mathit{ParentOf} \ltimes (\mathit{ParentOf} \ltimes \mathit{ParentOf}) \\ &\quad \equiv_{\pi_1} \mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}. \end{aligned}$$

These expressions are not path-equivalent, however. We also have

$$\begin{aligned} &\pi_1[\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}] \circ \mathit{FriendOf} \\ &\quad \equiv_{\text{path}} \pi_1[\mathit{ParentOf} \ltimes (\mathit{ParentOf} \ltimes \mathit{ParentOf})] \rtimes \mathit{FriendOf}. \end{aligned}$$

2.2. Expressive power

The equivalence notions introduced in the previous section extend naturally to subsumption and equivalence notions between classes of expressions.

DEFINITION 2.2. *Let sem $\in \{\text{path}, \pi_1, \pi_2, \text{bool}\}$. We say that the class of queries $\mathcal{L}_1$ is sem-subsumed by the class of queries $\mathcal{L}_2$, denoted by $\mathcal{L}_1 \preceq_{sem} \mathcal{L}_2$, if every query in $\mathcal{L}_1$ is sem-equivalent to a query in $\mathcal{L}_2$. We say that the classes of queries $\mathcal{L}_1$ and $\mathcal{L}_2$ are sem-equivalent, denoted by $\mathcal{L}_1 \equiv_{sem} \mathcal{L}_2$, if $\mathcal{L}_1 \preceq_{sem} \mathcal{L}_2$ and $\mathcal{L}_2 \preceq_{sem} \mathcal{L}_1$. We say that the classes of queries $\mathcal{L}_1$ and $\mathcal{L}_2$ are projection-equivalent if $\mathcal{L}_1 \equiv_{\pi_1} \mathcal{L}_2$ and $\mathcal{L}_1 \equiv_{\pi_2} \mathcal{L}_2$.*

[1] We follow the generally accepted use of the term *path-equivalence* for this notion for historical reasons, but must point out at the same time that this term is somewhat misleading, because it involves only 'sources' and 'targets' of a navigation, and not the path in the graph that this navigation might follow.

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

3. NAVIGATIONAL GRAPH QUERIES

The focus of this work is mainly on the relationship between the expressive power of the relation algebra and the semi-join algebra, which are algebraic representations of FO[3] and FO[2], respectively. We also include iteration—in the form of transitive closure—since iteration is essential in graph querying.

3.1. Relation algebra and the semi-join algebra

We first define the relation algebra and the semi-join algebra without iteration.

DEFINITION 3.1. *The graph expressions are defined by the grammar*

$$e := \emptyset \mid id \mid di \mid \ell \mid \ell^{\frown} \mid \pi_j[e] \mid \overline{\pi}_j[e] \mid e \circ e \mid e \ltimes e \mid e \rtimes e \mid e \cup e \mid e \cap e \mid e - e,$$

in which $\ell \in \Sigma$ and $j \in \{1,2\}$. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph and let e be an expression. The semantics of evaluation is defined as follows:

$$\begin{aligned}
[\![\emptyset]\!]_{\mathcal{G}} &= \emptyset;\\
[\![id]\!]_{\mathcal{G}} &= \{(m,m) \mid m \in \mathcal{V}\};\\
[\![di]\!]_{\mathcal{G}} &= \{(m,n) \mid m,n \in \mathcal{V} \wedge m \neq n\};\\
[\![\ell]\!]_{\mathcal{G}} &= \mathbf{E}(\ell);\\
[\![\ell^{\frown}]\!]_{\mathcal{G}} &= \{(n,m) \mid (m,n) \in \mathbf{E}(\ell)\};\\
[\![\pi_j[e]]\!]_{\mathcal{G}} &= \{(m,m) \mid m \in [\![e]\!]_{\mathcal{G}}|_j\};\\
[\![\overline{\pi}_j[e]]\!]_{\mathcal{G}} &= [\![id]\!]_{\mathcal{G}} - [\![\pi_j[e]]\!]_{\mathcal{G}};\\
[\![e_1 \circ e_2]\!]_{\mathcal{G}} &= \{(m,n) \mid \exists z\ (m,z) \in [\![e_1]\!]_{\mathcal{G}} \wedge (z,n) \in [\![e_2]\!]_{\mathcal{G}}\};\\
[\![e_1 \ltimes e_2]\!]_{\mathcal{G}} &= \{(m,n) \mid (m,n) \in [\![e_1]\!]_{\mathcal{G}} \wedge \exists z\ (n,z) \in [\![e_2]\!]_{\mathcal{G}}\};\\
[\![e_1 \rtimes e_2]\!]_{\mathcal{G}} &= \{(m,n) \mid (m,n) \in [\![e_2]\!]_{\mathcal{G}} \wedge \exists z\ (z,m) \in [\![e_1]\!]_{\mathcal{G}}\};\\
[\![e_1 \cup e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} \cup [\![e_2]\!]_{\mathcal{G}};\\
[\![e_1 \cap e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} \cap [\![e_2]\!]_{\mathcal{G}};\\
[\![e_1 - e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} - [\![e_2]\!]_{\mathcal{G}}.
\end{aligned}$$

We sometimes use the shorthand $all = id \cup di$, which evaluates to the set of all node pairs in a graph.

The relation algebra, which we denote by $\mathcal{N}_3$, allows every operator above except for the semi-joins ($\ltimes$ and $\rtimes$). The semi-join algebra, which we denote by $\mathcal{N}_2$, allows every operator above except for composition ($\circ$).

EXAMPLE 3.1. The expressions

$$e_1 = \pi_1[ParentOf \circ \overline{\pi}_1[OwnsPet] \circ ResearcherAt];$$
$$e_2 = \pi_1[ParentOf \ltimes (\overline{\pi}_1[OwnsPet] \ltimes ResearcherAt)]$$

both return people that are parents of researchers that do not own any pets. The expression e_1 is in $\mathcal{N}_3$ and the expression e_2 is in $\mathcal{N}_2$. Both expressions are node expressions. We have $e_1 \equiv_{\text{path}} e_2$.

3.2. Adding iteration

The relation algebra, as a graph query language, is often augmented with a general Kleene-star operator (transitive closure): if e is an expression, then so is $[e]^*$. The semantics of evaluation on graph $\mathcal{G}$ is defined as

$$[\![[e]^*]\!]_{\mathcal{G}} = \bigcup_{0 \leq i} [\![e^i]\!]_{\mathcal{G}}$$

with $e^0 = \text{id}$ and $e^k = e \circ e^{k-1}$. We denote the relation algebra, augmented with the Kleene-star, by $\mathcal{N}_3^*$.

As a semi-join-like counterpart of the Kleene-star, which itself is an iterated version of composition, we introduce a form of fixpoint iteration. We add the operator $\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]$ with $j \in \{1,2\}$, b an expression, e an expression and $\mathfrak{N}$ the single free variable of e that represents the set of nodes resulting from evaluating the fixpoint. We do not allow $\mathfrak{N}$ to occur anywhere else. The semantics of evaluating $\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]$ on graph $\mathcal{G}$ is defined next. We first define

$$s_i := \begin{cases} [\![b]\!]_{\mathcal{G}}|_j & \text{if } i = 0;\\ s_{i-1} \cup [\![e]\!]_{\mathcal{G}+s_{i-1}}|_j & \text{if } i > 0, \end{cases}$$

in which $\mathcal{G} + s_{i-1}$ is the graph $\mathcal{G}$ augmented with the edge relation $\{(n,n) \mid n \in s_{i-1}\}$ labeled with $\mathfrak{N}$. Due to monotonicity of $\cup$, there exists k, $k \leq |\mathcal{V}|$, such that $s_k = s_{k+1}$. We define $[\![\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]]\!]_{\mathcal{G}} = \{(n,n) \mid n \in s_k\}$.

EXAMPLE 3.2. As a first example, consider the expression

$$e_1 = \pi_1[[ParentOf]^* \circ FriendOf]$$

in $\mathcal{N}_3^*$, which returns (identical pairs of) all ancestors of persons having a friend (including these persons themselves). Next, consider the expression

$$e_2 = \mathsf{fp}_{1,\mathfrak{N}}[ParentOf \ltimes \mathfrak{N} \text{ union } FriendOf].$$

We have $e_1 \equiv_{\text{path}} e_2$. To illustrate this, we formally evaluate e_2 on the graph $\mathcal{G}$ shown in Fig. 3. (Notice this is a variation

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

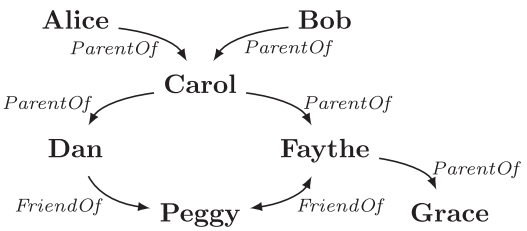

FIGURE 3. Variation of Fig. 1 used in Example 3.2.

on the graph shown in Fig. 1 obtained by removing Victor and Wendy.)

We have the following:

$$s_0 = \{\text{Dan}, \text{Faythe}, \text{Peggy}\};$$

$$s_1 = \{\text{Carol}, \text{Dan}, \text{Faythe}, \text{Peggy}\};$$

$$s_2 = \{\text{Alice}, \text{Bob}, \text{Carol}, \text{Dan}, \text{Faythe}, \text{Peggy}\};$$

$$s_3 = \{\text{Alice}, \text{Bob}, \text{Carol}, \text{Dan}, \text{Faythe}, \text{Peggy}\}.$$

Hence,

$$[\![e_2]\!]_{\mathcal{G}} = \{(\text{Alice}, \text{Alice}), (\text{Bob}, \text{Bob}), (\text{Carol}, \text{Carol}), (\text{Dan}, \text{Dan}), (\text{Faythe}, \text{Faythe}), (\text{Peggy}, \text{Peggy})\}.$$

Clearly, $[\![e_2]\!]_{\mathcal{G}} = [\![e_1]\!]_{\mathcal{G}}$.

As a second example, consider the expression

$$e_3 = \pi_1[[\mathit{ParentOf} \circ \overline{\pi}_1[\mathit{OwnsPet}]]^* \circ \mathit{ResearcherAt}]$$

in $\mathcal{N}_3^*$ which returns (identical pairs of) people that are ancestors of a chain of descendants that do not own pets and in which the youngest descendant is also a researcher. Now, let $e' = \mathit{ParentOf} \ltimes (\overline{\pi}_1[\mathit{OwnsPet}] \ltimes \mathfrak{N})$, and consider the expression

$$e_4 = \mathsf{fp}_{1,\mathfrak{N}}[e' \text{ union } \mathit{ResearcherAt}].$$

We have $e_3 \equiv_{\text{path}} e_4$.

Observe that both e_2 and e_4 have no free variables. In both cases, the variable $\mathfrak{N}$ has been bound within the scope of a fixpoint operator $\mathsf{fp}_{1,\mathfrak{N}}$.

We only introduce fixpoint iteration here as a less costly alternative to the Kleene-star. For this purpose, general fixpoints are too strong, however. Therefore, we put additional restrictions on the expression e used in $\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]$: if $j = 1$, then e must be *right-recursive* in $\mathfrak{N}$ and, if $j = 2$, then e must be *left-recursive* in $\mathfrak{N}$, which we inductively define next.

Let $dir \in \{\text{left}, \text{right}\}$. If $\mathfrak{N}$ is a variable, then the expression $\mathfrak{N}$ is *dir*-recursive in $\mathfrak{N}$. Expressions of the form $e = e_1 \cup e_2$ are *dir*-recursive in $\mathfrak{N}$ if e_1 and e_2 are *dir*-recursive in $\mathfrak{N}$. Expressions of the form $e = e_1 \ltimes e_2$ are right-recursive in $\mathfrak{N}$ if e_1 does not have free variables and e_2 is right-recursive in $\mathfrak{N}$. Expressions of the form $e = e_1 \rtimes e_2$ are left-recursive in $\mathfrak{N}$ if e_2 does not have free variables and e_1 is left-recursive in $\mathfrak{N}$. Finally, we consider fixpoint expressions that are nested within the scope of other fixpoint expressions. Expressions of the form $e = \mathsf{fp}_{1,\mathfrak{N}'}[e' \text{ union } b']$ are right-recursive in $\mathfrak{N}$ if b' is right-recursive in $\mathfrak{N}$ (while e' must not contain $\mathfrak{N}$ and must be right-recursive in $\mathfrak{N}'$). Likewise, expressions of the form $e = \mathsf{fp}_{2,\mathfrak{N}'}[e' \text{ union } b']$ are left-recursive in $\mathfrak{N}$ if b' is left-recursive in $\mathfrak{N}$.[2]

EXAMPLE 3.3. The expression $e = \mathit{ParentOf} \ltimes (\overline{\pi}_1[\mathit{OwnsPet}] \ltimes \mathfrak{N})$, as used in Example 3.2, is right-recursive. The expression $e' = \mathfrak{N} \rtimes \mathit{familyOf} \cup \mathfrak{N} \rtimes \mathit{FriendOf}$ is left-recursive. The expression

$$e'' = \mathsf{fp}_{2,\mathfrak{N}}[e' \text{ union } \mathit{OwnsPet}^\frown]$$

yields pet owners and people that are related to pet owners via friend and family relations.

We denote the semi-join algebra, augmented with this restricted form of fixpoint iteration, by $\mathcal{N}_2^{\mathsf{fp}}$.

3.3. Fragments of $\mathcal{N}_3^*$ and $\mathcal{N}_2^{\mathsf{fp}}$

We write $\mathscr{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, \cap, -\}$ to denote a set of operators in which π represents both π_1 and π_2, and, likewise, $\overline{\pi}$ represents both $\overline{\pi}_1$ and $\overline{\pi}_2$. By $\mathcal{N}_3(\mathscr{F})$ we denote the fragment of $\mathcal{N}_3$ that only allows $\emptyset$, id, $\ell \in \Sigma$, $\circ$, $\cup$ and all operators in $\mathscr{F}$, and by $\mathcal{N}_2(\mathscr{F})$ we denote the fragment of $\mathcal{N}_2$ that only allows $\emptyset$, id, $\ell \in \Sigma$, $\ltimes$, $\rtimes$, $\cup$ and all operators in $\mathscr{F}$. By $\mathcal{N}_3^*(\mathscr{F})$, we denote the fragment of $\mathcal{N}_3^*$ that allows the Kleene-star and the operators allowed by $\mathcal{N}_3(\mathscr{F})$. Finally, by $\mathcal{N}_2^{\mathsf{fp}}(\mathscr{F})$, we denote the fragment of $\mathcal{N}_2^{\mathsf{fp}}$ that allows the fixpoint iterator and the operators allowed by $\mathcal{N}_2(\mathscr{F})$.

3.4. Relationships with first-order logic

To express graph queries, we can also use standard first-order logic formulae over graphs [32]. These formulae adhere to the grammar

$$\varphi := x \triangleq y \mid \ell(x, y) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\, \varphi,$$

[2] The concepts of left-recursive and right-recursive expressions are closely related to concepts in formal languages [39]. Indeed, all expressions we allow can be mapped to concepts in *context-free grammars*: node variables map to non-terminals, unions map to individual grammar rules for a non-terminal and semi-joins map to the compositions within a single grammar rule. This is no coincidence: it is well known that a context-free grammar that is left-recursive or right-recursive can always be rewritten into a regular expression (using Kleene-star instead of recursion) [39].

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

in which $\triangleq$ denotes the equality operator, x and y are node variables, and $\ell \in \Sigma$. We interpret graphs $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ with $\Sigma = \{\ell_1, \ldots, \ell_{|\Sigma|}\}$ as structures $(\mathcal{V}; \ell_1, \ldots, \ell_{|\Sigma|})$ in which ℓ_i, $1 \leq i \leq |\Sigma|$, is interpreted as the binary relation $\mathbf{E}(\ell_i) \subseteq \mathcal{V} \times \mathcal{V}$. If $\varphi(x_1, \ldots, x_n)$ is a first-order logic formula with free variables $x_1, \ldots, x_n$, then we define $(\mathcal{V}; \ell_1, \ldots, \ell_{|\Sigma|}) \models \varphi(o_1, \ldots, o_n)$, with $o_1, \ldots, o_n \in \mathcal{V}$, in the standard manner and we define

$$[\![\varphi(x_1, \ldots, x_n)]\!]_{\mathcal{G}} = \{(o_1, \ldots, o_n) \mid (o_1, \ldots, o_n) \in \mathcal{V}^n \wedge (\mathcal{V}; \ell_1, \ldots, \ell_{|\Sigma|}) \models \varphi(o_1, \ldots, o_n)\}.$$

We write FO[k] to denote the first-order formulae with at most k variables. It is well-known that the relation algebra $\mathcal{N}_3$ has the same expressive power as FO[3] formulae with two free variables [5, 17]. Hence, $\mathcal{N}_3 \equiv_{\text{path}}$ FO[3]. Next, we look at the relationships between the semi-join algebra and first-order logic.

THEOREM 3.1. *$\mathcal{N}_2$ is path-equivalent to FO[2].*

Proof. The proof is based on the proof of the equivalence of FO[2] and the multi-dimensional modal logic MLR_2 [40, Section 2.3.1]. For the translation from FO[2] queries of the form $\varphi(v, w)$, with φ a FO[2] formula with free variables v and w, to expressions in $\mathcal{N}_2$, we use the following rewriting κ:

$$\begin{aligned}
\kappa(v \triangleq w) &= \text{id};\\
\kappa(w \triangleq v) &= \text{id};\\
\kappa(v \triangleq v) &= \text{all};\\
\kappa(w \triangleq w) &= \text{all};\\
\kappa(\ell(v, w)) &= \ell;\\
\kappa(\ell(w, v)) &= \ell^{\frown};\\
\kappa(\ell(v, v)) &= (\ell \cap \text{id}) \rtimes \text{all};\\
\kappa(\ell(w, w)) &= \text{all} \ltimes (\ell \cap \text{id});\\
\kappa(\neg\varphi) &= \text{all} - \kappa(\varphi);\\
\kappa(\varphi \vee \psi) &= \kappa(\varphi) \cup \kappa(\psi);\\
\kappa(\exists v\, \varphi) &= \text{all} \ltimes \kappa(\varphi[v, w/w, v]);\\
\kappa(\exists w\, \varphi) &= \kappa(\varphi[v, w/w, v]) \rtimes \text{all}.
\end{aligned}$$

In the above, the notation $\varphi[v, w/w, v]$, with φ an FO[2] formula, denotes the formula derived from φ by simultaneously substituting v for w and w for v (i.e. swapping v and w).

For the translation from expressions in $\mathcal{N}_2$ to FO[2] queries of the form $\varphi(v, w)$, with φ an FO[2] formula with free variables v and w, we use the following rewriting λ:

$$\begin{aligned}
\lambda(\text{id}) &= v \triangleq w;\\
\lambda(\text{di}) &= \neg(v \triangleq w);\\
\lambda(\ell) &= \ell(v, w);\\
\lambda(\ell^{\frown}) &= \ell(w, v);\\
\lambda(\pi_1[e]) &= v \triangleq w \wedge \exists w\, \lambda(e);\\
\lambda(\pi_2[e]) &= v \triangleq w \wedge \exists v\, \lambda(e);\\
\lambda(\overline{\pi}_1[e]) &= v \triangleq w \wedge \neg\exists w\, \lambda(e);\\
\lambda(\overline{\pi}_2[e]) &= v \triangleq w \wedge \neg\exists v\, \lambda(e);\\
\lambda(e_1 \ltimes e_2) &= \lambda(e_1) \wedge \exists v\, (v \triangleq w \wedge \exists w\, \lambda(e_2));\\
\lambda(e_1 \rtimes e_2) &= \lambda(e_2) \wedge \exists w\, (v \triangleq w \wedge \exists v\, \lambda(e_1));\\
\lambda(e_1 \cup e_2) &= \lambda(e_1) \vee \lambda(e_2);\\
\lambda(e_1 \cap e_2) &= \lambda(e_1) \wedge \lambda(e_2);\\
s\lambda(e_1 - e_2) &= \lambda(e_1) \wedge \neg\lambda(e_2).
\end{aligned}$$

In the rules for id, π and $\overline{\pi}$, we use the construction $v \triangleq w$ to enforce that the query evaluates to a node query.

Correctness of these translations follows from a straightforward structural induction argument. ■

The next example illustrates the rewriting κ as used in the above proof:

EXAMPLE 3.4. Consider the FO[2] query $\varphi(v, w)$ with

$$\varphi = v \triangleq w \wedge \exists w\, \mathit{ParentOf}(v, w).$$

This query yields identical pairs of parents. Notice that $e_1 \wedge e_2 = \neg(\neg e_1 \vee \neg e_2)$. Hence, φ is equivalent to

$$\varphi' = \neg(\neg(v \triangleq w) \vee \neg(\exists w\, \mathit{ParentOf}(v, w))).$$

Next, we rewrite φ' to an expression in $\mathcal{N}_2$:

$$\begin{aligned}
\kappa(\varphi') &= \kappa(\neg(\neg(v \triangleq w) \vee \neg(\exists w\, \mathit{ParentOf}(v, w))))\\
&= \text{all} - (\kappa(\neg(v \triangleq w) \vee \neg(\exists w\, \mathit{ParentOf}(v, w))))\\
&= \text{all} - (\kappa(\neg(v \triangleq w)) \cup (\neg(\exists w\, \mathit{ParentOf}(v, w))))\\
&= \text{all} - ((\text{all} - \kappa(v \triangleq w)) \cup (\text{all} - \kappa(\exists w\, \mathit{ParentOf}(v, w))))\\
&= \text{all} - ((\text{all} - \text{id}) \cup (\text{all} - (\kappa(\mathit{ParentOf}(w, v)) \rtimes \text{all})))\\
&= \text{all} - ((\text{all} - \text{id}) \cup (\text{all} - (\mathit{ParentOf}^{\frown} \rtimes \text{all}))).
\end{aligned}$$

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

Some straightforward simplifications yield:

$$\begin{aligned}\kappa(\varphi') &= \text{all} - (\text{di} \cup (\text{all} - (\mathit{ParentOf}^{\frown} \rtimes \text{all}))) \\ &= (\text{all} - \text{di}) \cap (\text{all} - (\text{all} - (\mathit{ParentOf}^{\frown} \rtimes \text{all}))) \\ &= \text{id} \cap (\mathit{ParentOf}^{\frown} \rtimes \text{all}) \\ &= \text{id} \cap (\pi_2[\mathit{ParentOf}^{\frown}] \rtimes \text{all}) \\ &= \text{id} \cap (\pi_1[\mathit{ParentOf}] \rtimes \text{all}) \\ &= \pi_1[\mathit{ParentOf}].\end{aligned}$$

We can generalize Theorem 3.1 to also cover fixpoints. It is well-known that first-order logic cannot express recursion such as provided by the Kleene-star or by fixpoints [32]. Instead, we consider the infinitary logic $\mathrm{L}^k_{\infty\omega}$, the standard infinitary first-order logic extension of FO[k] that allows conjunctions and disjunctions over arbitrary, possibly infinite, sets [31, 32]. The Kleene-star operator can be expressed in $\mathrm{L}^3_{\infty\omega}$ [31], and we have $\mathcal{N}^*_3 \preceq_{\text{path}} \mathrm{L}^3_{\infty\omega}$. For the relationship between $\mathrm{L}^2_{\infty\omega}$ and $\mathcal{N}^{\mathsf{fp}}_2$, we do not need to consider the full power of $\mathrm{L}^2_{\infty\omega}$. Instead, we restrict ourselves to adding to FO[2] an inflationary fixpoint of the form $[\mathbf{ifp}_{z,P}\ \varphi](z)$, with P a fresh monadic predicate defined by the fixpoint and $z \in \{v, w\}$. This inflationary fixpoint can be expressed in $\mathrm{L}^2_{\infty\omega}$ [31]. Next, we define the semantics of evaluating $[\mathbf{ifp}_{z,P}\ \varphi](z)$ on graph $\mathcal{G}$. We first define

$$s_i := \begin{cases} \emptyset & \text{if } i = 0; \\ s_{i-1} \cup \{o_z \mid (o_v, o_w) \in \mathcal{V}^2 \wedge \\ \qquad (\mathcal{V}; \ell_1, \ldots, \ell_{|\Sigma|}, P) \models \varphi(o_v, o_w)\} & \text{if } i > 0, \end{cases}$$

in which P is the monadic relation that evaluates to the set s_{i-1}. Due to monotonicity of $\cup$, there exists k, $k \leq |\mathcal{V}|$, such that $s_k = s_{k+1}$. Finally, we say that $(\mathcal{V}; \ell_1, \ldots, \ell_{|\Sigma|}) \models [\mathbf{ifp}_{z,P}\ \varphi](o)$ if $o \in s_k$. Notice that the semantics of the fixpoint operator fp we use is closely related to the semantics of the usual inflationary fixpoints we just introduced.

PROPOSITION 3.1. *$\mathcal{N}^{\mathsf{fp}}_2$ is path-subsumed by FO[2] to which inflationary fixpoints have been added.*

Proof. Let $\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]$ be a fixpoint expression in $\mathcal{N}^{\mathsf{fp}}_2$. We translate the fixpoint as follows:

$$\begin{aligned}\lambda(\mathsf{fp}_{1,\mathfrak{N}}[e \text{ union } b]) &= (v \triangleq w) \wedge \\ &\qquad [\mathbf{ifp}_{v,P}\ \exists w\ (\lambda(\pi_1[e]) \vee \lambda(\pi_1[b]))](v); \\ \lambda(\mathsf{fp}_{2,\mathfrak{N}}[e \text{ union } b]) &= (v \triangleq w) \wedge \\ &\qquad [\mathbf{ifp}_{w,P}\ \exists v\ (\lambda(\pi_2[e]) \vee \lambda(\pi_2[b]))](w),\end{aligned}$$

in which P is the fresh monadic predicate defined by the fixpoint, and we translate node variables $\mathfrak{N}$ by $\lambda(\mathfrak{N}) = (v \triangleq w) \wedge P(v)$. ■

From Proposition 3.1, we also conclude that $\mathcal{N}^{\mathsf{fp}}_2$ is path-subsumed by $\mathrm{L}^2_{\infty\omega}$, the two-variable fragment of infinitary logic [31, 34].

3.5. Relationships with other languages

The regular path queries (RPQs) use *regular expressions* to define the labeling of paths in the graph that are of interest [14] and are equivalent to the basic fragment $\mathcal{N}^*_3()$ of $\mathcal{N}^*_3$. This is no coincidence: the RPQs with and without the Kleene-star operator are often studied as a formalization of the 'greatest common divisor' of navigation in many practical graph and tree query languages, which also motivated our choice for the most basic languages, $\mathcal{N}_3()$ and $\mathcal{N}^*_3()$.

The RPQs can only be used to define very simple path-based intentional relationships. To strengthen the expressive power of the RPQs, several more expressive variants have been proposed and studied in the literature. The 2RPQs are obtained by adding converse ($^{\frown}$) [41] and the Nested RPQs are obtained by also adding projections (π_1 and π_2) [42]. Usually, expressions in these languages serve as binary predicates in a conjunctive query framework such as the CRPQs and C2RPQs [43, 44]. The Regular Queries are a particular powerful CRPQ-based language that can express all queries in $\mathcal{N}^*_3(^{\frown}, \pi, \cap, ^*)$ [45]. The RPQ-based query languages serve as the simplest of the relation algebras we study in this paper. The relation algebras themselves have also been studied in great detail with respect to graph querying [15, 16, 18, 19, 35, 37, 38]. Based on these results, the relationship in the expressive power of the relation algebras can be summarized as follows: we have $\mathcal{N}^*_3(\mathscr{F}_1) \preceq_{\text{path}} \mathcal{N}^*_3(\mathscr{F}_2)$ if and only if $\mathscr{F}_1 \subseteq C(\mathscr{F}_2)$, in which $C(\mathscr{F}_2)$ is obtained from $\mathscr{F}_2$ by adding any missing operators that are directly expressible using the operators already in $\mathscr{F}_2$ (e.g. adding $\cap$ when $- \in \mathscr{F}_2$ and adding π when $\overline{\pi} \in \mathscr{F}_2$).

As already argued in the Introduction, the relation algebra and its fragments are also at the core of many other graph query languages, including both navigational-based and pattern-based languages [1]. Examples include XPath (for querying XML data) [6–12, 46, 47], SPARQL (for querying RDF data) [2, 48], the graph query languages GXPath [13], Cypher [3] and Gremlin [4], and languages used for program verification such as PDL and KAT [49, 50]. Via the connections between the relation algebra and FO[2] we explore in this paper, there is also a close relation between the relation algebra and logics used in formal verification such as CTL, LTL and the μ-calculus [51]. We have summarized the query languages whose expressive power exactly match languages studied in this paper in Fig. 4.

4. REWRITING RELATION ALGEBRAS

In this section, we explore ways to automatically rewrite expressions with compositions and Kleene-stars into expressions with semi-joins and fixpoints. We start by identifying two situations in which the presence of a node expression, as a

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

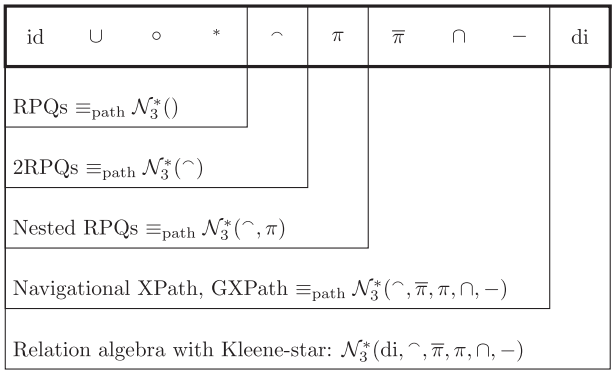

FIGURE 4. An overview of the relationships between graph query languages and path-equivalent fragments of the relation algebra. In this figure, each language is strictly more expressive than the languages it encloses.

subexpression of an expression e, allows for the elimination of composition from e in favor of semi-joins:

1. The expression e itself is a node expression due to the use of projection or coprojection at the outer level. An example is the expression $\pi_1[e_1 \circ e_2]$, where a straightforward rewriting yields $\pi_1[e_1 \circ e_2] \equiv_{\text{path}} \pi_1[e_1 \ltimes e_2]$.
2. The expression e is a composition $e_1 \circ e_2$, in which e_1 or e_2 is a node expression. An example is the expression $e_1 \circ \pi_1[e_2]$, where a straightforward rewriting yields $e_1 \circ \pi_1[e_2] \equiv_{\text{path}} e_1 \ltimes \pi_1[e_2]$.

Since the semantics of the Kleene-star is defined using composition, similar observations can be made with respect to the Kleene-star. The first observation above relies on the freedom to rewrite expressions to expressions that agree on either the first or the second column: hence, we are looking for left-projection-equivalent and right-projection-equivalent rewritings.

First, in Section 4.1, we introduce rewrite rules that support the rewriting of compositions and Kleene-stars to semi-joins and fixpoints in situations similar to the ones discussed above. Next, in Section 4.2, we look in detail to the implications of these rewrite rules with respect to the expressive power of the relation algebra. In Section 4.3, we look at the limitations of the rewrite rules and, in specific, at the roles of intersection and difference. Finally, in Section 4.4, we investigate how projection-equivalence, which is at the core of our rewrite rules, relates to the more standard path-equivalence and Boolean-equivalence.

4.1. Rewriting compositions and Kleene-stars

To support rewriting of compositions and Kleene-stars to semi-joins and fixpoints in situations similar to the ones discussed above, we will develop ways to rewrite (parts of) expressions in $\mathcal{N}_3^*$ to path-equivalent (parts of) expressions in $\mathcal{N}_2^{\mathsf{fp}}$. As a first step, we consider basic expressions built without using the composition, Kleene-star, semi-join and fixpoint operators.

DEFINITION 4.1. *The basic expressions are defined by the grammar*

$$e := \emptyset \mid id \mid di \mid \ell \mid \ell^{\frown} \mid \pi_j[e] \mid \overline{\pi}_j[e] \mid e \cup e \mid e \cap e \mid e - e,$$

in which $\ell \in \Sigma$ *and* $j \in \{1, 2\}$.

Observe that these basic expressions are both in $\mathcal{N}_3$ and $\mathcal{N}_2$. More specifically, every basic expression in $\mathcal{N}_3(\mathscr{F})$, $\mathscr{F} \subseteq \{\mathrm{di}, \frown, \pi, \overline{\pi}, \cap, -\}$, is also in $\mathcal{N}_2(\mathscr{F})$. To deal with composition and Kleene-star operators, we propose the rewrite rules of Fig. 5. We notice that these rewrite rules do not change basic expressions. We will argue later that $\tau(e) \equiv_{\text{path}} e$, $\tau_{\pi_1}(e) \equiv_{\pi_1} e$ and $\tau_{\pi_2}(e) \equiv_{\pi_2} e$.

EXAMPLE 4.1. Consider the expression

$$e = \pi_1[((WorksOn \circ WorksOn^{\frown}) \cap FriendOf) \circ EditorOf] \circ StudentOf.$$

This expression returns pairs of professors and their students, but only for professors that are friends with an editor with whom they collaborate on a project. For clarity, we abbreviate each edge label in e, resulting in $\pi_1[((W \circ W^{\frown}) \cap F) \circ E] \circ S$. We have the following:

$$\begin{aligned}
\tau(e) &= \tau_{\pi_2}(\pi_1[((W \circ W^{\frown}) \cap F) \circ E]) \rtimes \tau(S) \\
&= \pi_1[\tau_{\pi_1}(((W \circ W^{\frown}) \cap F) \circ E)] \rtimes S \\
&= \pi_1[\tau_{\circ_1}((W \circ W^{\frown}) \cap F; \tau_{\pi_1}(E))] \rtimes S \\
&= \pi_1[(\tau(W \circ W^{\frown}) \cap \tau(F)) \ltimes E] \rtimes S \\
&= \pi_1[((\tau(W) \circ \tau(W^{\frown})) \cap F) \ltimes E] \rtimes S \\
&= \pi_1[((W \circ W^{\frown}) \cap F) \ltimes E] \rtimes S.
\end{aligned}$$

We shall prove (Theorem 4.1) that e and $\tau(e)$ are path-equivalent. This rewriting results in an expression in which two out of three applications of composition are eliminated in favor of semi-joins. In Proposition 4.2 (Section 4.3), we shall show that the last remaining composition step is unavoidable.

When applied on expressions with subexpressions of the form $[e]^*$, the rewrite rules can introduce fixpoint operators. Consequently, certain rewrite rules yield subexpressions with free node variables. For these expressions, we have only defined the semantics within the scope of fixpoint iteration. Hence, we have not defined when these expressions are left-projection-equivalent or right-projection-equivalent. To enable reasoning about expressions with free node variables and proving soundness of the rewrite rules, we generalize Definition 4.1:

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

$$
\begin{aligned}
\tau(b) &= b \\
\tau(f_j[e]) &= f_j[\tau_{\pi_j}(e)] \\
\tau(e_1 \circ e_2) &= \circ_{\text{path}}(e_1; e_2) \\
\tau(e_1 \cup e_2) &= \tau(e_1) \cup \tau(e_2) \\
\tau(e_1 \oplus e_2) &= \tau(e_1) \oplus \tau(e_2) \\
\tau([e]^*) &= [\tau(e)]^*
\end{aligned}
$$

$$
\begin{aligned}
\tau_{\pi_i}(b) &= b \\
\tau_{\pi_i}(f_j[e]) &= f_j[\tau_{\pi_j}(e)] \\
\tau_{\pi_i}(e_1 \circ e_2) &= \circ_{\pi_i}(e_1; e_2) \\
\tau_{\pi_i}(e_1 \cup e_2) &= \tau_{\pi_i}(e_1) \cup \tau_{\pi_i}(e_2) \\
\tau_{\pi_i}(e_1 \oplus e_2) &= \tau(e_1) \oplus \tau(e_2) \\
\tau_{\pi_i}([e]^*) &= \text{id}
\end{aligned}
$$

$$
\begin{aligned}
\tau_{\circ_1}(b; \varepsilon) &= b \ltimes \varepsilon \\
\tau_{\circ_1}(f_j[e]; \varepsilon) &= f_j[\tau_{\pi_j}(e)] \ltimes \varepsilon \\
\tau_{\circ_1}(e_1 \circ e_2; \varepsilon) &= \tau_{\circ_1}(e_1; \tau_{\circ_1}(e_2; \varepsilon)) \\
\tau_{\circ_1}(e_1 \cup e_2; \varepsilon) &= \tau_{\circ_1}(e_1; \varepsilon) \cup \tau_{\circ_1}(e_2; \varepsilon) \\
\tau_{\circ_1}(e_1 \oplus e_2; \varepsilon) &= (\tau(e_1) \oplus \tau(e_2)) \ltimes \varepsilon \\
\tau_{\circ_1}([e]^*; \varepsilon) &= \mathsf{fp}_{\mathfrak{N},1}[\tau_{\circ_1}(e; \mathfrak{N}) \text{ union } \varepsilon]
\end{aligned}
$$

$$
\begin{aligned}
\tau_{\circ_2}(b; \varepsilon) &= \varepsilon \rtimes b \\
\tau_{\circ_2}(f_j[e]; \varepsilon) &= \varepsilon \rtimes f_j[\tau_{\pi_j}(e)] \\
\tau_{\circ_2}(e_1 \circ e_2; \varepsilon) &= \tau_{\circ_2}(e_2; \tau_{\circ_2}(e_1; \varepsilon)) \\
\tau_{\circ_2}(e_1 \cup e_2; \varepsilon) &= \tau_{\circ_2}(e_1; \varepsilon) \cup \tau_{\circ_2}(e_2; \varepsilon) \\
\tau_{\circ_2}(e_1 \oplus e_2; \varepsilon) &= \varepsilon \rtimes (\tau(e_1) \oplus \tau(e_2)) \\
\tau_{\circ_2}([e]^*; \varepsilon) &= \mathsf{fp}_{\mathfrak{N},2}[\tau_{\circ_2}(e; \mathfrak{N}) \text{ union } \varepsilon]
\end{aligned}
$$

$$
\circ_{\text{path}}(e_1; e_2) = \begin{cases} \tau(e_1) \circ \tau(e_2) & \text{if } e_1 \text{ and } e_2 \text{ are not node expressions;} \\ \tau(e_1) \ltimes \tau_{\pi_1}(e_2) & \text{if } e_2 \text{ is a node expression;} \\ \tau_{\pi_2}(e_1) \rtimes \tau(e_2) & \text{if } e_1 \text{ is a node expression.} \end{cases}
$$

$$
\circ_{\pi_i}(e_1; e_2) = \begin{cases} \tau_{\circ_1}(e_1; \tau_{\pi_1}(e_2)) & \text{if } i = 1; \\ \tau_{\circ_2}(e_2; \tau_{\pi_2}(e_1)) & \text{if } i = 2. \end{cases}
$$

FIGURE 5. Rewrite rules aimed at rewriting compositions to semi-joins and Kleene-stars to fixpoints. In these rules, b is a basic expression, ε is an already rewritten expression, $f \in \{\pi, \overline{\pi}\}$, $i \in \{1, 2\}$, $j \in \{1, 2\}$, $\oplus \in \{\cap, -\}$ and $\mathfrak{N}$ is a fresh variable.

DEFINITION 4.2. *Let e_1 and e_2 be expressions with a single free node variable $\mathfrak{N}$. We say that e_1 and e_2 are left-projection-equivalent, denoted by $e_1 \equiv_{\pi_1} e_2$, if, for every graph $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ and every $s \subseteq \mathcal{V}$, $[\![e_1]\!]_{\mathcal{G}+s}|_1 = [\![e_2]\!]_{\mathcal{G}+s}|_1$, in which $\mathcal{G} + s$ is the graph $\mathcal{G}$ augmented with the edge relation $\{(n, n) \mid n \in s\}$ labeled with $\mathfrak{N}$. Likewise, we say that e_1 and e_2 are right-projection-equivalent, denoted by $e_1 \equiv_{\pi_2} e_2$, if, for every graph $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ and every $s \subseteq \mathcal{V}$, $[\![e_1]\!]_{\mathcal{G}+s}|_2 = [\![e_2]\!]_{\mathcal{G}+s}|_2$.*

EXAMPLE 4.2. We have $(\mathit{ParentOf} \circ \mathit{FriendOf}) \ltimes \mathfrak{N} \equiv_{\pi_1} \mathit{ParentOf} \ltimes (\mathit{FriendOf} \ltimes \mathfrak{N})$. Indeed, for any possible node query we can fill in for $\mathfrak{N}$, the evaluation of both expressions will yield the same set of nodes in the first column.

To prove soundness of the rewrite rules in Fig. 5, we use the following:

LEMMA 1.16. *Let g, h, ϕ and ψ be expressions. We have:*

1. *If $g \equiv_{\pi_j} h$ with $j \in \{1, 2\}$, then $\pi_j[g] \equiv_{\text{path}} \pi_j[h]$ and $\overline{\pi}_j[g] \equiv_{\text{path}} \overline{\pi}_j[h]$.*
2. *Let $g \equiv_{\text{path}} h$. If $\phi \equiv_{\pi_1} \psi$, then $g \ltimes \phi \equiv_{\text{path}} h \ltimes \psi$, $g \circ \phi \equiv_{\pi_1} h \circ \psi$ and $g \circ \phi \equiv_{\pi_1} h \ltimes \psi$. If $\phi \equiv_{\pi_2} \psi$, then $\phi \rtimes g \equiv_{\text{path}} \psi \rtimes h$, $\phi \circ g \equiv_{\pi_2} \psi \circ h$ and $\phi \circ g \equiv_{\pi_2} \psi \rtimes h$.*
3. *Let $g \equiv_{\text{path}} h$ and let ϕ be a node expression. If $\phi \equiv_{\pi_1} \psi$, then $g \circ \phi \equiv_{\text{path}} h \ltimes \psi$. If $\phi \equiv_{\pi_2} \psi$, then $\phi \circ g \equiv_{\text{path}} \psi \rtimes h$.*
4. *If $g \equiv_{sem} h$ and $\phi \equiv_{sem} \psi$ with $sem \in \{\text{path}, \pi_1, \pi_2\}$, then $g \cup \phi \equiv_{sem} h \cup \psi$.*
5. *If $g \equiv_{\text{path}} h$ and $\phi \equiv_{\text{path}} \psi$, then $g \cap \phi \equiv_{\text{path}} h \cap \psi$ and $g - \phi \equiv_{\text{path}} h - \psi$.*
6. *$id \equiv_{\pi_1} [\phi]^*$ and $id \equiv_{\pi_2} [\phi]^*$.*
7. *If $g \ltimes \mathfrak{N} \equiv_{\pi_1} h$ and $\phi \equiv_{\pi_1} \psi$, then $[g]^* \circ \phi \equiv_{\pi_1} \mathsf{fp}_{1,\mathfrak{N}}[h \text{ union } \psi]$. If $\mathfrak{N} \rtimes g \equiv_{\pi_2} e[h]$ and $\phi \equiv_{\pi_2} \psi$, then $\phi \circ [g]^* \equiv_{\pi_2} \mathsf{fp}_{2,\mathfrak{N}}[h \text{ union } \psi]$.*

Proof. Statements 1–6 follow from the semantics of the operators involved. We prove the first case of Statement 7; the second case is analogous. Assume $g \ltimes \mathfrak{N} \equiv_{\pi_1} h$ and $\phi \equiv_{\pi_1} \psi$. We shall prove $[g]^* \circ \phi \equiv_{\pi_1} \mathsf{fp}_{1,\mathfrak{N}}[h \text{ union } \psi]$ using induction on the stages of the evaluation of the fixpoint operator fp. By s_i, we denote the i-th stage of the evaluation of the fixpoint and, by e_i, we denote the expression $(g^0 \cup \cdots \cup g^i) \circ \phi$. By induction, we shall prove that $[\![e_i]\!]_{\mathcal{G}}|_1 = s_i$. The base case is $i = 0$. We have $e_0 = g^0 \circ \phi \equiv_{\text{path}} \text{id} \circ \phi \equiv_{\text{path}} \phi \equiv_{\pi_1} \psi$. Hence, $[\![e_0]\!]_{\mathcal{G}}|_1 = [\![\phi]\!]_{\mathcal{G}}|_1 = [\![\psi]\!]_{\mathcal{G}}|_1 = s_0$. Now assume that, for every j with $0 \leq j < i$, $[\![e_j]\!]_{\mathcal{G}}|_1 = s_j$. We have

$$
\begin{aligned}
e_i &\equiv_{\text{path}} (g^0 \cup \cdots \cup g^i) \circ \phi \\
&\equiv_{\text{path}} (g^0 \cup \cdots \cup g^{i-1}) \circ \phi \cup g \circ \qquad ((g^0 \cup \cdots \cup g^{i-1}) \circ \phi) \\
&\equiv_{\text{path}} e_{i-1} \cup (g \circ e_{i-1}) \\
&\equiv_{\pi_1} e_{i-1} \cup (g \ltimes e_{i-1}).
\end{aligned}
$$

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

Due to the induction hypothesis, we have $[\![e_{i-1}]\!]_{\mathcal{G}}|_1 = s_{i-1}$. Hence, during computation of s_i, we have

$$\begin{aligned} s_i &= s_{i-1} \cup [\![h]\!]_{\mathcal{G}+s_{i-1}}|_1 \\ &= [\![e_{i-1}]\!]_{\mathcal{G}}|_1 \cup [\![g \ltimes \mathfrak{N}]\!]_{\mathcal{G}+s_{i-1}}|_1. \end{aligned}$$

By the semantics of $\mathcal{G} + s_{i-1}$ and $[\![e_{i-1}]\!]_{\mathcal{G}}|_1 = s_{i-1}$, we can replace $\mathfrak{N}$ by e_{i-1}, resulting in

$$\begin{aligned} s_i &= [\![e_{i-1} \cup (g \ltimes e_{i-1})]\!]_{\mathcal{G}}|_1 \\ &= [\![e_i]\!]_{\mathcal{G}}|_1. \end{aligned}$$

By the semantics of the iteration operators, there exists $k, 0 \leq k$, such that $s_k = s_{k+1}$ and $[\![e_k]\!]_{\mathcal{G}} = [\![e_{k+1}]\!]_{\mathcal{G}}$. Hence, we conclude $[g]^* \circ \phi \equiv_{\pi_1} \mathsf{fp}_{1,\mathfrak{N}}[h \text{ union } \psi]$. ■

The rewrite rules of Fig. 5 depend on the ability to determine if an expression e is a node expression. This is hard to determine without evaluation of e. We can, however, use the semantics of $\mathcal{N}_3, \mathcal{N}_3^*, \mathcal{N}_2$ and $\mathcal{N}_2^{\mathsf{fp}}$ to define a predicate $\mathsf{ns}(e)$ that evaluates to `true` *only if* the expression e is a node expression:[3]

$$\begin{aligned} \mathsf{ns}(\emptyset) = \mathsf{ns}(\mathrm{id}) &= \texttt{true}; \\ \mathsf{ns}(\mathrm{di}) &= \texttt{false}; \\ \mathsf{ns}(\ell) = \mathsf{ns}(\ell^{\frown}) &= \texttt{false}; \\ \mathsf{ns}(f_j[e]) &= \texttt{true}; \\ \mathsf{ns}(e_1 \circ e_2) &= \mathsf{ns}(e_1) \wedge \mathsf{ns}(e_2); \\ \mathsf{ns}(e_1 \ltimes e_2) = \mathsf{ns}(e_2 \rtimes e_1) &= \mathsf{ns}(e_1); \\ \mathsf{ns}(e_1 \cup e_2) &= \mathsf{ns}(e_1) \wedge \mathsf{ns}(e_2); \\ \mathsf{ns}(e_1 \cap e_2) &= \mathsf{ns}(e_1) \vee \mathsf{ns}(e_2); \\ \mathsf{ns}(e_1 - e_2) &= \mathsf{ns}(e_1); \\ \mathsf{ns}([e]^*) &= \mathsf{ns}(e); \\ \mathsf{ns}(\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]) &= \texttt{true}, \end{aligned}$$

in which $f \in \{\pi, \overline{\pi}\}$ and $j \in \{1, 2\}$. Using Lemma 4.1 and induction on the length of relation algebra expressions, we can establish the following main result about the soundness of the rewrite rules in Fig. 5:

THEOREM 4.1. *Let e be an expression in $\mathcal{N}_3^*$. We have $\tau(e) \equiv_{\text{path}} e$, $\tau_{\pi_1}(e) \equiv_{\pi_1} e$ and $\tau_{\pi_2}(e) \equiv_{\pi_2} e$.*

[3] For some node queries e, we have $\mathsf{ns}(e) = \texttt{false}$. We will not treat these queries as node queries during rewriting, but as normal queries. This prevents the application of some of the rewrite techniques used to eliminate composition and Kleene-star, but does not invalidate the end result.

The rewrite rules of Fig. 5 are sound, as stated in Theorem 4.1, but not complete, as illustrated by the following example:

EXAMPLE 4.3. Consider the expression

$$e = (\mathit{FriendOf} \cap (\mathit{FriendOf} \circ \mathit{FriendOf})) - \text{all}.$$

Due to the presence of intersection, the rewritings $\tau(e)$, $\tau_{\pi_1}(e)$ and $\tau_{\pi_2}(e)$ do not result in an expression in $\mathcal{N}_2$ or $\mathcal{N}_2^{\mathsf{fp}}$. Since e always evaluates to $\emptyset$, however, we have $e \equiv_{\text{path}} \emptyset$, and $\emptyset$ is, by definition, in $\mathcal{N}_2$ and $\mathcal{N}_2^{\mathsf{fp}}$.

Remark 1.1 Before closing Section 4.1, we wish to point attention to the order in which operators are evaluated in expressions. We can express this order explicitly using parentheses. Consider, for example,

$$e = \mathit{ParentOf} \circ (\mathit{FriendOf} \circ \mathit{OwnsPet}).$$

The parentheses in this expression suggest to first evaluate the composition $\mathit{FriendOf} \circ \mathit{OwnsPet}$, after which the remaining composition is evaluated. Due to associativity of composition, this expression is path-equivalent to

$$e' = (\mathit{ParentOf} \circ \mathit{FriendOf}) \circ \mathit{OwnsPet},$$

in which the order of evaluation is swapped. Therefore, the parentheses can be omitted in as far as only the meaning of the query is concerned. However, the order of evaluating the compositions (which are joins) greatly influences the performance of query evaluation. Hence, many optimization techniques employed by database systems aim at picking the right (or a sufficiently good) order of evaluation [20, 27, 28, 52–57].

The semi-join operator is not associative, however. Indeed, the order in which semi-joins are evaluated is usually fixed, and cannot be changed without also changing the meaning of the query. Compare, on the one hand, the query

$$e_1 = \mathit{ParentOf} \ltimes (\mathit{FriendOf} \ltimes \mathit{OwnsPet}),$$

which yields those parent-child-pairs in which the child has a friend, and that friend owns a pet and, on the other hand, the query

$$e_2 = (\mathit{ParentOf} \ltimes \mathit{FriendOf}) \ltimes \mathit{OwnsPet},$$

which yields parent-child-pairs in which the child has a friend and that same child (not the friend) has a pet. These queries clearly have a different meaning. In particular, only $e \equiv_{\pi_1} e_1$. Since semi-joins leave no choice in the order in which they are evaluated, there is also no point in considering different semi-join evaluation orderings for efficiency reasons. Fortunately, the semi-join is—overall—a much cheaper operator to evaluate than composition.

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

Finally, we want to point out that the rewrite rules of Fig. 5 are designed to yield the correct parentheses for the semi-join evaluation. To illustrate this, we rewrite both $\pi_1[e]$ and $\pi_1[e']$. For clarity, we use straightforward abbreviations for the edge labels used in e and e'. We have

$$\begin{aligned}
\tau(\pi_1[e]) &= \tau(\pi_1[P \circ (F \circ O)]) \\
&= \pi_1[\tau_{\pi_1}(P \circ (F \circ O))] \\
&= \pi_1[\tau_{\circ_1}(P; \tau_{\pi_1}(F \circ O))] \\
&= \pi_1[\tau_{\circ_1}(P; \tau_{\circ_1}(F; \tau_{\pi_1}(O)))] \\
&= \pi_1[\tau_{\circ_1}(P; \tau_{\circ_1}(F; O))] \\
&= \pi_1[\tau_{\circ_1}(P; F \ltimes O)] = \pi_1[P \ltimes (F \ltimes O)]; \\
\tau(\pi_1[e']) &= \tau(\pi_1[(P \circ F) \circ O]) \\
&= \pi_1[\tau_{\pi_1}((P \circ F) \circ O)] \\
&= \pi_1[\tau_{\circ_1}(P \circ F; \tau_{\pi_1}(O))] \\
&= \pi_1[\tau_{\circ_1}(P \circ F; O)] \\
&= \pi_1[\tau_{\circ_1}(P; \tau_{\circ_1}(F; O))] \\
&= \pi_1[\tau_{\circ_1}(P; F \ltimes O)] = \pi_1[P \ltimes (F \ltimes O)].
\end{aligned}$$

As one can see, rewriting $\pi_1[e]$ and $\pi_1[e']$ yields identical semi-join expressions, as intended.

4.2. Relative expressive power of $\mathcal{N}_3$ and $\mathcal{N}_2^{\mathsf{fp}}$

The rewrite rules of Fig. 5 do not fully rewrite every expression in $\mathcal{N}_3^*$ to $\mathcal{N}_2^{\mathsf{fp}}$. To better understand the limitations of those rewrite rules, we will take a closer look at how they rewrite fragments of $\mathcal{N}_3^*$. Before we do so, we first study the reverse: expressing the semi-join algebra using the relation algebra. To do so, we propose the rewrite rules of Fig. 6.

Using a straightforward induction on the length of expressions, in which the base cases are basic expressions and Lemma 4.1 is used to prove the inductive cases, we conclude the following:

PROPOSITION 4.1. *Let $\{\pi\} \subseteq \mathscr{F} \subseteq \{di, \frown, \pi, \overline{\pi}, \cap, -\}$ and let e be an expression.*

1. *If e is in $\mathcal{N}_2(\mathscr{F})$, then $e \equiv_{\text{path}} \rho(e)$ and $\rho(e)$ is in $\mathcal{N}_3(\mathscr{F})$;*
2. *If e is in $\mathcal{N}_2^{\mathsf{fp}}(\mathscr{F})$, then $e \equiv_{\text{path}} \rho(e)$ and $\rho(e)$ is in $\mathcal{N}_3^*(\mathscr{F})$.*

A careful analysis of the rewrite rules of Fig. 5, Theorem 4.1 and Proposition 4.1 allows us to conclude

THEOREM 4.2. *Let $\{\pi\} \subseteq \mathscr{F} \subseteq \{di, \frown, \pi, \overline{\pi}\}$. We have*

1. *$\mathcal{N}_2(\mathscr{F}) \preceq_{\text{path}} \mathcal{N}_3(\mathscr{F})$ and $\mathcal{N}_2(\mathscr{F}) \equiv_\pi \mathcal{N}_3(\mathscr{F})$;*
2. *$\mathcal{N}_2^{\mathsf{fp}}(\mathscr{F}) \preceq_{\text{path}} \mathcal{N}_3^*(\mathscr{F})$ and $\mathcal{N}_2^{\mathsf{fp}}(\mathscr{F}) \equiv_\pi \mathcal{N}_3^*(\mathscr{F})$.*

$$\begin{aligned}
\rho(b) &= b; \\
\rho(f_j[e]) &= f_j[\rho(e)]; \\
\rho(e_1 \ltimes e_2) &= \rho(e_1) \circ \pi_1[\rho(e_2)]; \\
\rho(e_1 \rtimes e_2) &= \pi_2[\rho(e_1)] \circ \rho(e_2); \\
\rho(e_1 \cup e_2) &= \rho(e_1) \cup \rho(e_2); \\
\rho(e_1 \oplus e_2) &= \rho(e_1) \oplus \rho(e_2); \\
\rho(\mathsf{fp}_{1,\mathfrak{N}}[e \text{ union } b]) &= \pi_1[[\rho_{\text{right-}\mathfrak{N}}(e)]^* \circ \rho(b)]; \\
\rho(\mathsf{fp}_{2,\mathfrak{N}}[e \text{ union } b]) &= \pi_2[\rho(b) \circ [\rho_{\text{left-}\mathfrak{N}}(e)]^*]; \\
\\
\rho_{dir\text{-}\mathfrak{N}}(\mathfrak{N}) &= \text{id}; \\
\rho_{dir\text{-}\mathfrak{N}}(e_1 \cup e_2) &= \rho_{dir\text{-}\mathfrak{N}}(e_1) \cup \rho_{dir\text{-}\mathfrak{N}}(e_2); \\
\rho_{\text{left-}\mathfrak{N}}(e_1 \rtimes e_2) &= \rho_{\text{left-}\mathfrak{N}}(e_1) \circ \rho(e_2); \\
\rho_{\text{right-}\mathfrak{N}}(e_1 \ltimes e_2) &= \rho(e_1) \circ \rho_{\text{right-}\mathfrak{N}}(e_2); \\
\rho_{\text{right-}\mathfrak{N}}(\mathsf{fp}_{1,\mathfrak{N}'}[e' \text{ union } b']) &= [\rho_{\text{right-}\mathfrak{N}'}(e')]^* \circ \\
&\qquad \rho_{\text{right-}\mathfrak{N}}(b'); \\
\rho_{\text{left-}\mathfrak{N}}(\mathsf{fp}_{2,\mathfrak{N}'}[e' \text{ union } b']) &= \rho_{\text{left-}\mathfrak{N}}(b') \circ [\rho_{\text{left-}\mathfrak{N}'}(e')]^*.
\end{aligned}$$

FIGURE 6. Rewrite rules aimed at rewriting semi-joins to compositions and fixpoints to Kleene-stars. In these rules, b is a basic expression, $f \in \{\pi, \overline{\pi}\}$, $j \in \{1, 2\}$, $\oplus \in \{\cap, -\}$ and $dir \in \{\text{left}, \text{right}\}$.

4.3. The role of intersection and difference

Observe that Theorem 4.2 does not cover the cases where intersection or difference are involved. This is a very severe restriction, since intersection and difference are allowed in the semi-join algebra. Careful analysis of the rewrite rules of Fig. 5 reveals that rewriting intersection and difference only causes issues in conjunction with compositions, but not when used in basic expressions. As a consequence, we can extend Theorem 4.2 slightly.

DEFINITION 4.3. *Let $\mathscr{F} \subseteq \{di, \frown, \pi, \overline{\pi}, \cap, -\}$. We write $\mathcal{B}_3(\mathscr{F})$, $\mathcal{B}_3^*(\mathscr{F})$, $\mathcal{B}_2(\mathscr{F})$ and $\mathcal{B}_2^{\mathsf{fp}}(\mathscr{F})$ to denote the basic fragments of $\mathcal{N}_3(\mathscr{F})$, $\mathcal{N}_3^*(\mathscr{F})$, $\mathcal{N}_2(\mathscr{F})$ and $\mathcal{N}_2^{\mathsf{fp}}(\mathscr{F})$, respectively, in which intersection and difference occur in basic expressions only.*

Below, we write $\underline{\mathscr{F}}$, $\mathscr{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, \cap, -\}$, to denote the fragment $\mathscr{F}$ to which $\cap$ is added if $- \in \mathscr{F}$ and $\overline{\pi}$ is added if $\pi, - \in \mathscr{F}$.

THEOREM 4.3. *Let $\{\pi\} \subseteq \mathscr{F} \subseteq \{di, \frown, \pi, \overline{\pi}, \cap, -\}$. We have*

1. *$\mathcal{B}_2(\mathscr{F}) \preceq_{\text{path}} \mathcal{B}_3(\mathscr{F})$, $\mathcal{B}_2(\mathscr{F}) \equiv_\pi \mathcal{B}_3(\mathscr{F})$ and $\mathcal{B}_2(\mathscr{F}) \equiv_{\text{path}} \mathcal{N}_2(\underline{\mathscr{F}})$;*
2. *$\mathcal{B}_2^{\mathsf{fp}}(\mathscr{F}) \preceq_{\text{path}} \mathcal{B}_3^*(\mathscr{F})$, $\mathcal{B}_2^{\mathsf{fp}}(\mathscr{F}) \equiv_\pi \mathcal{B}_3^*(\mathscr{F})$ and $\mathcal{B}_2^{\mathsf{fp}}(\mathscr{F}) \equiv_{\text{path}} \mathcal{N}_2^{\mathsf{fp}}(\underline{\mathscr{F}})$.*

Proof. It suffices to observe that in the semi-join algebra we may assume without loss of generality that intersection and

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

difference occur in basic expressions only, since we can push down intersection and difference through projections, coprojections, semi-joins and unions. We push down intersection using the following properties ($j \in \{1, 2\}$):

$$
\begin{aligned}
e \cap \pi_j[e'] &\equiv_{\text{path}} \pi_j[e'] \cap e \equiv_{\text{path}} (e \cap \text{id}) \ltimes \pi_j[e'];\\
e \cap \overline{\pi}_j[e'] &\equiv_{\text{path}} \overline{\pi}_j[e'] \cap e \equiv_{\text{path}} (e \cap \text{id}) \ltimes \overline{\pi}_j[e'];\\
e \cap (e_1 \ltimes e_2) &\equiv_{\text{path}} (e_1 \ltimes e_2) \cap e \equiv_{\text{path}} (e \cap e_1) \ltimes e_2;\\
e \cap (e_1 \rtimes e_2) &\equiv_{\text{path}} (e_1 \rtimes e_2) \cap e \equiv_{\text{path}} e_1 \rtimes (e \cap e_2);\\
e \cap (e_1 \cup e_2) &\equiv_{\text{path}} (e_1 \cup e_2) \cap e\\
&\equiv_{\text{path}} (e \cap e_1) \cup (e \cap e_2);\\
e \cap (e_1 - e_2) &\equiv_{\text{path}} (e_1 - e_2) \cap e \equiv_{\text{path}} (e \cap e_1) - e_2,
\end{aligned}
$$

We push down difference using the following properties:

$$
\begin{aligned}
e - \pi_j[e'] &\equiv_{\text{path}} (e \cap \text{di}) \cup ((e \cap \text{id}) \ltimes \overline{\pi}_j[\pi_j[e']]);\\
e - \overline{\pi}_j[e'] &\equiv_{\text{path}} (e \cap \text{di}) \cup ((e \cap \text{id}) \ltimes \overline{\pi}_j[\overline{\pi}_j[e']]);\\
\pi_j[e'] - e &\equiv_{\text{path}} \pi_j[e'] \ltimes \overline{\pi}_j[e \cap \text{id}];\\
\overline{\pi}_j[e'] - e &\equiv_{\text{path}} \overline{\pi}_j[e'] \ltimes \overline{\pi}_j[e \cap \text{id}];\\
e - (e_1 \ltimes e_2) &\equiv_{\text{path}} (e - e_1) \cup ((e \cap e_1) \ltimes \overline{\pi}_1[e_2]);\\
(e_1 \ltimes e_2) - e &\equiv_{\text{path}} (e_1 - e) \ltimes e_2;\\
e - (e_1 \rtimes e_2) &\equiv_{\text{path}} (e - e_2) \cup (\overline{\pi}_2[e_1] \rtimes (e \cap e_2));\\
(e_1 \rtimes e_2) - e &\equiv_{\text{path}} e_1 \rtimes (e_2 - e);\\
e - (e_1 \cup e_2) &\equiv_{\text{path}} (e - e_1) - e_2;\\
(e_1 \cup e_2) - e &\equiv_{\text{path}} (e_1 - e) \cup (e_2 - e).
\end{aligned}
$$

We observe that fixpoints are node expressions and we can treat them as if they were projections. By repeatedly pushing down intersection and difference until this is no longer possible, all intersections and differences occur in basic expressions only. ■

Theorem 4.2 and 4.3 also imply a collapse of semi-join algebra fragments to the corresponding basic semi-join algebra fragments:

COROLLARY 4.1. Let $\{\pi\} \subseteq \mathscr{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, \cap, -\}$. We have

1. $\mathcal{N}_2(\mathscr{F}) \preceq_{\text{path}} \mathcal{B}_3(\underline{\mathscr{F}})$ and $\mathcal{N}_2(\mathscr{F}) \equiv_\pi \mathcal{B}_3(\underline{\mathscr{F}})$;
2. $\mathcal{N}_2^{\mathsf{fp}}(\mathscr{F}) \preceq_{\text{path}} \mathcal{B}_3^*(\underline{\mathscr{F}})$ and $\mathcal{N}_2^{\mathsf{fp}}(\mathscr{F}) \equiv_\pi \mathcal{B}_3^*(\underline{\mathscr{F}})$.

Corollary 4.1 does not generalize to a collapse of relation algebra fragments to the corresponding basic relation algebra fragments. More generally, we prove that all basic fragments of $\mathcal{N}_3$ that include intersection have less expressive power than their non-basic counterparts:

FIGURE 7. On the *left*, two 3-cycle graph $\mathcal{G}_{3,3}$. On the *right*, single 4-cycle graph $\mathcal{G}_4$. All edges are assumed to be labeled ℓ.

PROPOSITION 4.2. *Let $\mathscr{F} \subseteq \{di, \frown, \pi, \overline{\pi}, \cap, -\}$ with $\cap \in \underline{\mathscr{F}}$. We have*

1. *$\mathcal{N}_3(\mathscr{F}) \not\preceq_{\text{bool}} \mathcal{B}_3(\underline{\mathscr{F}})$ and $\mathcal{N}_3(\mathscr{F}) \not\preceq_{\text{bool}} \mathcal{N}_2(\underline{\mathscr{F}})$;*
2. *$\mathcal{N}_3^*(\mathscr{F}) \not\preceq_{\text{bool}} \mathcal{B}_3^*(\underline{\mathscr{F}})$ and $\mathcal{N}_3(\mathscr{F}) \not\preceq_{\text{bool}} \mathcal{N}_2^{\mathsf{fp}}(\underline{\mathscr{F}})$.*

Proof. Consider the expression $e' = (\ell \circ \ell) \cap \ell$. This expression is based on the part of e in Example 4.1 that could not be rewritten without using composition. The expression e' has an occurrence of intersection beyond the scope of basic expressions. We claim that, for any $\mathscr{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, \cap, -\}$, no expression in $\mathcal{B}_3^*(\mathscr{F})$ is path-equivalent, left-projection-equivalent, right-projection-equivalent or Boolean-equivalent to e'.

To show this, consider the graphs $\mathcal{G}_{3,3}$ and $\mathcal{G}_4$ of Fig. 7 and observe that $[\![e']\!]_{\mathcal{G}_{3,3}} \neq \emptyset$ while $[\![e']\!]_{\mathcal{G}_4} = \emptyset$. To show that no expression in $\mathcal{B}_3^*(\mathscr{F})$ can distinguish between $\mathcal{G}_{3,3}$ and $\mathcal{G}_4$, it suffices to show that no expression in $\mathcal{N}_2^{\mathsf{fp}}$ can distinguish between $\mathcal{G}_{3,3}$ and $\mathcal{G}_4$ (Corollary 4.1). We can do so using standard two-pebble game results for the FO[2]-variant of the infinitary finite variable logics [31, Example 3.10]. ■

On graphs, the intersection and difference operators used only within the basic fragments of $\mathcal{N}_3$ still have useful roles, as shown in the next example.

EXAMPLE 4.4. Consider again the relationships *FriendOf* and *WorksWith* and consider the following basic expressions:

$$
\begin{aligned}
e_1 &= \mathit{FriendOf} \cap \mathit{WorksWith};\\
e_2 &= \mathit{FriendOf} - \mathit{WorksWith};\\
e_3 &= \mathit{WorksWith} - \text{id};\\
e_4 &= \mathit{WorksWith} \cap \text{di}.
\end{aligned}
$$

Expression e_1 yields work-friends, whereas expression e_2 yields non-work-friends. These examples show how intersection and difference can be used to select specific combinations of edge labels. Both expression e_3 and e_4 yields people who work with each other, while excluding self-loops (people that work with themselves).

4.4. The role of projection equivalence

In this paper, we introduced projection equivalence to study rewriting and simplifying expressions while keeping either the

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

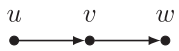

FIGURE 8. A chain of length three. All edges are assumed to be labeled ℓ.

first projection or second projection of the rewritten expression equivalent to the original expression. One can ask how projection equivalence relates to the more commonly studied path equivalence and Boolean equivalence.

PROPOSITION 4.3. *Let* $\mathscr{F}_1, \mathscr{F}_2 \subseteq \{di, \frown, \pi, \overline{\pi}, \cap, -\}$ *and* $X \in \{\mathcal{N}_3, \mathcal{N}_3^*\}$. *We have* $X(\mathscr{F}_1) \preceq_\pi X(\mathscr{F}_2)$ *if and only if* $X(\mathscr{F}_1) \preceq_{\text{bool}} X(\mathscr{F}_2)$.

Proof. We always have that $X(\mathscr{F}_1) \preceq_\pi X(\mathscr{F}_2)$ implies $X(\mathscr{F}_1) \preceq_{\text{bool}} X(\mathscr{F}_2)$. To prove that $X(\mathscr{F}_1) \preceq_{\text{bool}} X(\mathscr{F}_2)$ implies $X(\mathscr{F}_1) \preceq_\pi X(\mathscr{F}_2)$, we distinguish two cases:

1. $X(\mathscr{F}_1) \preceq_{\text{path}} X(\mathscr{F}_2)$. Then both $X(\mathscr{F}_1) \preceq_\pi X(\mathscr{F}_2)$ and $X(\mathscr{F}_1) \preceq_{\text{bool}} X(\mathscr{F}_2)$.
2. $X(\mathscr{F}_1) \not\preceq_{\text{path}} X(\mathscr{F}_2)$ and $X(\mathscr{F}_1) \preceq_{\text{bool}} X(\mathscr{F}_2)$. In this case, there exists a fragment $\mathscr{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}\}$ such that $\mathscr{F}_1 = \mathscr{F} \cup \{\frown\}$ and $\mathscr{F}_2 = \mathscr{F} \cup \{\pi\}$. The proof of $X(\mathscr{F}_1) \preceq_{\text{bool}} X(\mathscr{F}_2)$ in Fletcher et al. [15, Proof of Proposition 4.2] reveals that, for every expression e in $X(\mathscr{F}_1)$, there exist expressions e_1 and e_2 in $X(\mathscr{F}_2)$ such that $e_1 \equiv_{\text{path}} \pi_1[e]$ and $e_2 \equiv_{\text{path}} \pi_2[e]$. Hence, by definition, $X(\mathscr{F}_1) \preceq_\pi X(\mathscr{F}_2)$. ■

We observe that Proposition 4.3 implies that we may not conclude from $\mathcal{L}_1 \equiv_\pi \mathcal{L}_2$ that $\mathcal{L}_1 \equiv_{\text{path}} \mathcal{L}_2$. Next, we show that we may not conclude from $\mathcal{L}_1 \equiv_{\text{bool}} \mathcal{L}_2$ that $\mathcal{L}_1 \equiv_\pi \mathcal{L}_2$.

PROPOSITION 4.4. *Let* $X \in \{\mathcal{N}_3, \mathcal{N}_3^*\}$. *On chains, we have* $X(\pi) \preceq_{\text{bool}} X()$ *and* $X(\pi) \not\preceq_\pi X()$.

Proof. By a result from Hellings et al. [18, Theorem 4], we have $X(\pi) \preceq_{\text{bool}} X()$. We prove $X(\pi) \not\preceq_\pi X()$ via a simple counterexample. Consider the query $e = \pi_2[\ell] \circ \pi_1[\ell]$ evaluated on the chain $\mathcal{C}$ of Fig. 8. This query will return the node-pair (v, v). An exhaustive search shows that no expression in $X()$ is j-projection-equivalent to $e, j \in \{1, 2\}$. ■

Using Proposition 4.3 and Proposition 4.4, we conclude the following:

COROLLARY 4.2. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be query languages. We have that $\mathcal{L}_1 \equiv_{\text{bool}} \mathcal{L}_2$ does not imply $\mathcal{L}_1 \equiv_\pi \mathcal{L}_2$ and $\mathcal{L}_1 \equiv_\pi \mathcal{L}_2$ does not imply $\mathcal{L}_1 \equiv_{\text{path}} \mathcal{L}_2$.

5. QUERYING SIBLING-ORDERED TREES

The above expressiveness results have interesting implications for the relationship between, on the one hand, Regular XPath and Conditional XPath [9–12], and, on the other hand, first-order logic evaluated on node-labeled sibling-ordered trees, which we detail next. On finite node-labeled sibling-ordered trees, Conditional XPath is path-equivalent to FO^{tree}: first-order logic on tree structures represented by a descendant and a following-sibling relation, unary node-label predicates and equality [9, Proposition 2.7]. We shall prove that this collapse can be sharpened to a projection-equivalence collapse of FO^{tree} queries to $\mathcal{N}_2^{\text{fp}}$. For the readers not familiar with Conditional XPath and Regular XPath, we first provide a brief introduction to these languages.

Conditional XPath is a syntactical fragment of Regular XPath, and Regular XPath is a query language for querying node-labeled sibling-ordered XML data [9]. Regular XPath distinguishes path formulae, which evaluate to binary relations, and node formulae, which evaluate to unary relations (sets of nodes). Path formulae are defined by the grammar[4]

$$\mathsf{p_wff} = \mathit{Edge} \mid \mathsf{p_wff} \circ \mathsf{p_wff} \mid \mathsf{p_wff} \cup \mathsf{p_wff} \mid [\mathsf{p_wff}]^* \mid ?\mathsf{n_wff},$$

in which $\mathit{Edge} \in \{\mathit{Child}, \mathit{Parent}, \mathit{Left}, \mathit{Right}\}$ denotes the edge relations (the parent-child axis and the ordered sibling axis), $\mathsf{n_wff}$ is a node formula and $?\mathsf{n_wff}$ interprets the node formulae as a binary relation. Node formulae are defined by the grammar

$$\mathsf{n_wff} = \ell \mid \text{id} \mid \pi_1[\mathsf{p_wff}] \mid \overline{\mathsf{n_wff}} \mid \mathsf{n_wff} \cup \mathsf{n_wff} \mid \mathsf{n_wff} \cap \mathsf{n_wff},$$

in which ℓ denotes a node label.

As a first step toward proving the collapse of Boolean FO^{tree} queries to the semi-join algebra, we claim that Regular XPath is path-equivalent to $\mathcal{N}_3^*(\frown, \overline{\pi})$. We prove this claim by rewriting path formulae to expressions in $\mathcal{N}_3^*(\frown, \overline{\pi})$ and node formulae to node expressions in $\mathcal{N}_3^*(\frown, \overline{\pi})$. We represent node labels using edge labels. These choices result in a straightforward rewriting $\tau_{\mathsf{p_wff}}(\mathsf{p_wff})$ for path formulae $\mathsf{p_wff}$. For rewritings involving node formulae, we have

$$\begin{aligned}
\tau_{\mathsf{p_wff}}(?\mathsf{n_wff}) &= \pi_1[\tau_{\mathsf{n_wff}}(\mathsf{n_wff})];\\
\tau_{\mathsf{n_wff}}(\ell) &= \ell;\\
\tau_{\mathsf{n_wff}}(\text{id}) &= \text{id};\\
\tau_{\mathsf{n_wff}}(\pi_1[\mathsf{p_wff}]) &= \pi_1[\tau_{\mathsf{p_wff}}(\mathsf{p_wff})];\\
\tau_{\mathsf{n_wff}}(\overline{\mathsf{n_wff}}) &= \overline{\pi}_1[\tau_{\mathsf{n_wff}}(\mathsf{n_wff})];\\
\tau_{\mathsf{n_wff}}(\mathsf{n_wff}_1 \cup \mathsf{n_wff}_2) &= \tau_{\mathsf{n_wff}}(\mathsf{n_wff}_1) \cup \tau_{\mathsf{n_wff}}(\mathsf{n_wff}_2);\\
\tau_{\mathsf{n_wff}}(\mathsf{n_wff}_1 \cap \mathsf{n_wff}_2) &= \tau_{\mathsf{n_wff}}(\mathsf{n_wff}_1) \circ \tau_{\mathsf{n_wff}}(\mathsf{n_wff}_2).
\end{aligned}$$

4 We have slightly adapted the Regular XPath syntax to better match the syntax of $\mathcal{N}_3^*$.

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

As Conditional XPath is a restriction of Regular XPath in which the Kleene-star can only be applied to *steps* instead of generic expressions,[5] we conclude the following:

PROPOSITION 5.1. *With respect to queries yielding binary relations evaluated on finite node-labeled sibling-ordered trees, we have Regular XPath* $\equiv_{\text{path}} \mathcal{N}_3^*(\frown, \overline{\pi})$*, Conditional XPath* $\preceq_{\text{path}} \mathcal{N}_3^*(\frown, \overline{\pi})$ *and* $\mathcal{N}_3^*(\frown, \overline{\pi}) \not\preceq_{\text{bool}}$ *Conditional XPath.*

Proof. To translate from Regular XPath to $\mathcal{N}_3^*(\frown, \overline{\pi})$, we use the rewrite rules $\tau_{\mathsf{p_wff}}(\mathsf{p_wff})$. For the other direction, we adapt the above rewrite rules. The only difficulty in this are subexpressions of the form $\pi_2[e]$ and $\overline{\pi}_2[e]$. We deal with these subexpressions by rewriting them toward $\pi_1[[e]^{-1}]$ and $\overline{\pi}_1[[e]^{-1}]$, respectively, in which $[e]^{-1}$ is the converse of e, which can be expressed using $\frown$. The remainder of the statement of the Proposition follows from the well-known relationships between Regular XPath and Conditional XPath [9]. ■

We combine Theorem 4.2, Proposition 5.1 and a result from Marx [9]:

COROLLARY 5.2. With respect to queries yielding binary relations evaluated on finite node-labeled sibling-ordered trees, we have Regular XPath $\equiv_{\pi} \mathcal{N}_2^{\mathsf{fp}}(\frown, \overline{\pi})$, Conditional XPath $\preceq_{\pi} \mathcal{N}_2^{\mathsf{fp}}(\frown, \overline{\pi})$ and $\mathcal{N}_2^{\mathsf{fp}}(\frown, \overline{\pi}) \not\preceq_{\text{bool}}$ Conditional XPath.

Finally, we combine Corollary 5.2 with the collapse of FO^{tree} to Conditional XPath [9, Proposition 2.7] to conclude the following:

PROPOSITION 5.2. *With respect to queries yielding binary relations evaluated on finite node-labeled sibling-ordered trees, we have* $FO^{tree} \preceq_{\pi} \mathcal{N}_2^{\mathsf{fp}}(\frown, \overline{\pi})$.

Unfortunately, it is not possible to strengthen Proposition 5.2 by showing that one can translate Conditional XPath to the two-variable fragment of FO^{tree} via $\mathcal{N}_2^{\mathsf{fp}}(\frown, \overline{\pi})$. This follows from the simple fact that the two-variable fragment of FO^{tree} cannot express basic Conditional XPath constructions, including the edge relations *Child* and *Right*, and step-based conditional iteration via the descendant and the following-sibling relations.

6. OPTIMIZING GRAPH QUERIES

In Section 4, we studied the relationship between the relation algebra and the semi-join algebra using composition-to-semi-join rewrite rules. In this Section, we will review these rewrite rules and investigate their usefulness with respect to *graph query optimization*.

To do so, we introduce in Section 6.1 a simple classification of the complexity of evaluating the operators in the relation algebra and the semi-join algebra. Next, in Section 6.2, we show that also the fixpoint operator we use can be evaluated efficiently. In Section 6.3, we look at the size of expressions rewritten using the introduced rewrite rules. In Section 6.4, we further fine-tune the rewrite rules aimed at improving graph query optimizations. In Section 6.5, we look at all other operators we identified in Section 6.1 as being expensive, and show how their common usages can be optimized. Finally, in Section 6.6, we conclude on the query optimization potential of the rewrite rules we introduced.

6.1. The cost of evaluating operators

Most operators in the relation algebra and semi-join algebra can easily be evaluated using specialized versions of the many query evaluation algorithms that are used in traditional relational database management systems [20, 27, 52–57]. The only exception is the evaluation of the fixpoint operators, which we discuss in-depth in Section 6.2. Here, we focus on giving a cost model for the other operators. A detailed cost model for the evaluation cost of queries evaluation involves many aspects and is outside the scope of this work. Luckily, the worst-case cost of each operator we consider is almost entirely determined by the size of the evaluation result of its operands and the size of its result, even if we use naive algorithms for evaluating these operators [27, 52, 57]. We categorize the operators in three complexity classes.[6]

DEFINITION 6.1. *An operator is expression-linear if it is guaranteed to yield a result linearly upper-bounded by the size of the evaluation result of its operands. An operator is node-linear if it is not expression-linear, but still is guaranteed to yield a result linearly upper-bounded by the number of nodes (independent of the size of the evaluation result of any operands). An operator is non-linear if it does not fall in the above two categories.*

We deem expression-linear operators to be the least expensive and the non-linear operators to be the most expensive. We classify the relation algebra and semi-join algebra operators as follows:

LEMMA 6.1. *The operators* π_1, π_2, $\ltimes$, $\rtimes$, $\cup$, $\cap$ *and* $-$ *are expression-linear, the operators id,* $\overline{\pi}_1$ *and* $\overline{\pi}_2$ *are node-linear, and the operators di,* $\circ$ *and* * *are non-linear.*

[5] A step is an edge relation to which, optionally, a test is applied of the form ?n_wff.

[6] The categorization is similar to the categorization of Leinders and Van den Bussche [20].

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

Proof. To show that the operators id, $\overline{\pi}_1$ and $\overline{\pi}_2$ are node-linear, it suffices to provide an example showing that they are not expression-linear. To show that the operators di, $\circ$ and * are non-linear, we provide an example showing that they are not expression-linear and node-linear. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph with $\mathcal{V} = \{m, n_1, \ldots, n_{|\mathcal{V}|-1}\}$, $\Sigma = \{\ell, \ell'\}$, $\mathbf{E}(\ell) = \{(m, n_i), (n_i, m) \mid 1 \leq i \leq |\mathcal{V}| - 1\}$ and $\mathbf{E}(\ell') = \{(m, m)\}$. We have

$$\begin{aligned}
[\![\mathrm{id}]\!]_{\mathcal{G}} &= \{(n', n') \mid n' \in \mathcal{V}\};\\
[\![\mathrm{di}]\!]_{\mathcal{G}} &= \mathcal{V}^2 - \{(v, v) \mid v \in \mathcal{V}\};\\
[\![\overline{\pi}_1[\ell']]\!]_{\mathcal{G}} &= [\![\overline{\pi}_2[\ell']]\!]_{\mathcal{G}} = \{(n_i, n_i) \mid 1 \leq i \leq |\mathcal{V}| - 1\};\\
[\![\ell \circ \ell]\!]_{\mathcal{G}} &= \mathcal{V}^2 - \mathbf{E}(\ell);\\
[\![[\ell]^*]\!]_{\mathcal{G}} &= \mathcal{V}^2.
\end{aligned}$$

Observe that $|[\![\mathrm{id}]\!]_{\mathcal{G}}| = |\mathcal{V}|$, $|[\![\mathrm{di}]\!]_{\mathcal{G}}| = |\mathcal{V}|^2 - |\mathcal{V}|$, $|[\![\overline{\pi}_1[\ell']]\!]_{\mathcal{G}}| = |\mathcal{V}| - 1$ and $|[\![\ell \circ \ell]\!]_{\mathcal{G}}| = |\mathcal{V}|^2 - 2 \cdot (|\mathcal{V}| - 1)$. ■

Lemma 6.1 illustrates why we consider composition and the Kleene-star to be expensive and why also identity, diversity and coprojections are to be avoided. Next, we shall look at the complexity of fixpoint iteration.

6.2. Efficient evaluation of fixpoints

Due to the restrictions put on fixpoints, they can be evaluated very efficiently, which we will show next. Let $f = \mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]$ be an expression without free variables. The complexity of evaluating f is a function of the *recursion steps of* f and the cost of evaluating the *non-recursive terms of* f, which we define next.

We define $\mathcal{R}(e)$ by

$$\begin{aligned}
\mathcal{R}(\mathfrak{N}) &= 1;\\
\mathcal{R}(e_1 \ltimes e_2) &= \mathcal{R}(e_2 \rtimes e_1) = 1 + \mathcal{R}(e_1);\\
\mathcal{R}(e_1 \cup e_2) &= \mathcal{R}(e_1) + \mathcal{R}(e_2) + 1;\\
\mathcal{R}(\mathsf{fp}_{j,\mathfrak{N}'}[e' \text{ union } b']) &= \mathcal{R}(b') + \mathcal{R}(e') + 1,
\end{aligned}$$

with $\mathfrak{N}$ a variable, and we define the recursion steps of f by $\mathcal{R}(f) = \mathcal{R}(e)$. We define the multiset $\mathcal{T}(e)$ by

$$\begin{aligned}
\mathcal{T}(\mathfrak{N}) &= [\,];\\
\mathcal{T}(e_1 \ltimes e_2) &= \mathcal{T}(e_2 \rtimes e_1) = [e_2] + \mathcal{T}(e_1);\\
\mathcal{T}(e_1 \cup e_2) &= \mathcal{T}(e_1) + \mathcal{T}(e_2);\\
\mathcal{T}(\mathsf{fp}_{j,\mathfrak{N}'}[e' \text{ union } b']) &= \mathcal{T}(b') + \mathcal{T}(e').
\end{aligned}$$

with $\mathfrak{N}$ a variable, and we define the non-recursive terms of f by $\mathcal{T}(f) = [b] + \mathcal{T}(e)$.

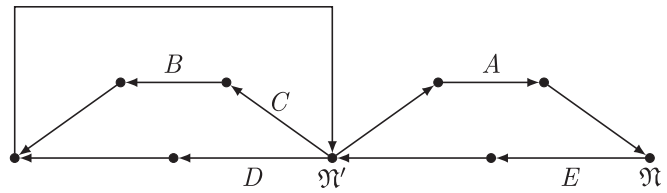

FIGURE 9. Graph representing the expression $e = \mathsf{fp}_{1,\mathfrak{N}}[A \ltimes e' \text{ union } F]$ with $e' = \mathsf{fp}_{1,\mathfrak{N}'}[(B \ltimes (C \ltimes \mathfrak{N}')) \cup (D \ltimes \mathfrak{N}') \text{ union } E \ltimes \mathfrak{N}]$. This graph is obtained by applying the graph representation construction of the proof of Proposition 6.1 on e.

PROPOSITION 6.1. *Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph and let $f = \mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]$ be an expression without free variables. The worst-case cost for evaluating $[\![f]\!]_{\mathcal{G}}$ is $\mathcal{O}(\mathcal{R}(f) \cdot n + s + c)$, in which*

$$\begin{aligned}
n &= \max\{|[\![t]\!]_{\mathcal{G}}|_j \mid t \in \mathcal{T}(f)\};\\
s &= \sum\{|[\![t]\!]_{\mathcal{G}}| \mid e[t] \in \mathcal{T}(f)\}
\end{aligned}$$

and c is the total cost of evaluating the expressions in $\mathcal{T}(f)$.

Proof. We observe that expression e does not use negation on the path toward the variable $\mathfrak{N}$: we only allow union, semi-joins and fixpoints, and we do not allow difference and coprojections. Hence, if we interpret the expressions in $\mathcal{T}(f)$ as pre-computed edge labels, then the restricted language we consider is expressible in a subset of the alternation-free μ-calculus, for which very efficient evaluation algorithms exist [58].

Using these algorithms, we sketch how to efficiently evaluate the fixpoint expression f when $j = 1$. The case for $j = 2$ is analogous. To evaluate the fixpoint expression f, we first translate the expression into a graph representation. We do so by making edge-connections between expressions in the following way.

1. Add an unlabeled connection from the expression e to the expression $\mathfrak{N}$.
2. For any right-recursive subexpression $e_1 \ltimes e_2$, add a connection labeled e_1 from the expression e_2 to the expression $e_1 \ltimes e_2$.
3. For any right-recursive subexpression $e_1 \cup e_2$, add unlabeled connections from the expressions e_1 and e_2 to the expression $e_1 \cup e_2$.
4. For any right-recursive subexpression $\mathsf{fp}_{1,\mathfrak{N}'}[e' \text{ union } b']$, add an unlabeled connection from the expression $\mathfrak{N}'$ to the expression $\mathsf{fp}_{1,\mathfrak{N}'}[e' \text{ union } b']$ and from the expression b' to the expression $\mathfrak{N}'$.

Figure 9 provides an example of the resulting graph representation of a fixpoint expression.

The graph representation is used for a message-passing evaluation algorithm in which each expression-node maintains a

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

set of received graph-nodes. When an expression-node receives a graph-node v it has not yet received, then it sends v to every expression-node to which it has an unlabeled connection and it sends w to every expression-node to which it has a connection labeled e' with $(w, v) \in [\![e']\!]_{\mathcal{G}}$. We initialize this process by sending each graph-node in $[\![b]\!]_{\mathcal{G}}|_1$ to the expression-node $\mathfrak{N}$. Let S be the set of all graph-nodes received by $\mathfrak{N}$ after all messages have been processed. We have $[\![f]\!]_{\mathcal{G}} = \{(v, v) \mid v \in S\}$.

Over every unlabeled connection, at most n messages are sent and over every connection labeled with e, at most $|[\![e]\!]_{\mathcal{G}}|$ messages are sent. The number of expression-nodes and the number of unlabeled connections are both worst-case $\mathcal{O}(\mathcal{R}(f))$. Furthermore, for every non-recursive term in $\mathcal{T}(f)$, there is exactly one labeled connection. Hence, at most $\mathcal{O}(\mathcal{R}(f) \cdot n + s)$ messages need to be sent.

To achieve the stated complexity in the expected case, we implement the graph-node set at each expression-node by a hash table and we store $[\![e']\!]_{\mathcal{G}}$, necessary for evaluating the connections labeled e', as a hash table in which the second column of $[\![e']\!]_{\mathcal{G}}$ is the search key. Using the approach of Cleaveland and Steffen [58], one can reach the stated complexity in the worst case. ■

Fixpoints do not really suit the classification we used for the other operators, as its operands cannot be evaluated independently due to the recursion involved. To obtain a classification in the spirit of Definition 6.1, we can view the set of non-recursive terms $\mathcal{T}(f)$ as the operands of a fixpoint. Using this view, we can classify the fixpoint operator as an *expression-linear* operator.

6.3. Size-complexity of rewritten expressions

We already claimed soundness of the rewrite rules of Fig. 5. To claim their usability for query optimization, we will analyze the complexity of the expression resulting from the rewrite next. We do so in terms of the expression size, the number of steps needed for evaluation and the complexity of the operators involved. In this analysis, we use the following terminology:

DEFINITION 6.2. *The size of an expression e, denoted by $\|e\|$, is the number of operations in e. We have*

$$\|\emptyset\| = \|id\| = \|di\| = \|\ell\| = \|\ell^{\frown}\| = 0;$$
$$\|f_j[e]\| = 1 + \|e\|;$$
$$\|e_1 \otimes e_2\| = 1 + \|e_1\| + \|e_2\|;$$
$$\|e_1 \oplus e_2\| = 1 + \|e_1\| + \|e_2\|;$$
$$\|[e]^*\| = 1 + \|e\|;$$
$$\|\mathfrak{N}\| = 0;$$
$$\|\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]\| = 1 + \|e\| + \|b\|,$$

with $f \in \{\pi, \overline{\pi}\}$, $j \in \{1, 2\}$, $\otimes \in \{\circ, \ltimes, \rtimes\}$ and $\oplus \in \{\cup, \cap, -\}$. The subexpression set of e, denoted by $\mathcal{S}(e)$, is the set of all unique non-atomic subexpressions that must be evaluated:

$$\mathcal{S}(\emptyset) = \mathcal{S}(id) = \mathcal{S}(di) = \mathcal{S}(\ell) = \mathcal{S}(\ell^{\frown}) = \emptyset;$$
$$\mathcal{S}(f_j[e]) = \{f_j[e]\} \cup \mathcal{S}(e);$$
$$\mathcal{S}(e_1 \otimes e_2) = \{e_1 \otimes e_2\} \cup \mathcal{S}(e_1) \cup \mathcal{S}(e_2);$$
$$\mathcal{S}(e_1 \oplus e_2) = \{e_1 \oplus e_2\} \cup \mathcal{S}(e_1) \cup \mathcal{S}(e_2);$$
$$\mathcal{S}([e]^*) = \{[e]^*\} \cup \mathcal{S}(e);$$
$$\mathcal{S}(\mathfrak{N}) = \emptyset;$$
$$\mathcal{S}(\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]) = \{\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]\} \cup \mathcal{S}(e) \cup \mathcal{S}(b).$$

The evaluation size of e is defined by eval-steps$(e) = |\mathcal{S}(e)|$.

EXAMPLE 6.1. Consider the expression

$$e = ((\ell \circ \ell) \circ (\ell \circ \ell)) \circ ((\ell \circ \ell) \circ (\ell \circ \ell)).$$

We have $\|e\| = 7$, we have eval-steps$(e) = 3$. Indeed, this expression can be evaluated in three steps, namely by first evaluating $e_1 = \ell \circ \ell$, next $e_2 = e_1 \circ e_1$ and, finally, $e = e_2 \circ e_2$. Now consider the expression

$$e' = \ell \circ (\ell \circ (\ell \circ (\ell \circ (\ell \circ (\ell \circ (\ell \circ \ell)))))),$$

for which we have $e \equiv_{\text{path}} e'$ and $\|e'\| = \text{eval-steps}(e') = 7$.

Next, we characterize the impact of the rewrite rule of Fig. 5 on the evaluation size and expression size of rewritten expressions. Observe that the only rewrite rules of Fig. 5 that increases the expression size significantly are the rewrite rules $\tau_{\circ_i}(e_1 \cup e_2; \varepsilon) = \tau_{\circ_i}(e_1; \varepsilon) \cup \tau_{\circ_1}(e_2; \varepsilon)$, $i \in \{1, 2\}$, as these rewrite rules duplicate the expression ε. By $\mathsf{u}(\tau(e))$, $\mathsf{u}(\tau_{\pi_j}(e))$ and $\mathsf{u}(\tau_{\circ_j}(e; \varepsilon))$, $j \in \{1, 2\}$, we denote the number of times the rewrite rules $\tau_{\circ_i}(e_1 \cup e_2; \varepsilon) = \tau_{\circ_i}(e_1; \varepsilon) \cup \tau_{\circ_1}(e_2; \varepsilon)$, $i \in \{1, 2\}$, have been applied in the rewriting of e using $\tau(e)$, $\tau_{\pi_j}(e)$, or $\tau_{\circ_j}(e; \varepsilon')$, respectively. We have the following:

THEOREM 6.1. *Let e be an expression in $\mathcal{N}_3^*$.*

1. *We have* eval-steps$(\tau(e)) \leq \mathsf{u}(\tau(e)) + \|e\|$ *and* $\|\tau(e)\| = \Theta(\|e\| \cdot 2^{\mathsf{u}(\tau(e))})$ *in the worst case.*
2. *Let $i \in \{1, 2\}$. We have* eval-steps$(\tau_{\pi_i}(e)) \leq \mathsf{u}(\tau_{\pi_i}(e)) + \|e\|$ *and* $\|\tau_{\pi_i}(e)\| = \Theta(\|e\| \cdot 2^{\mathsf{u}(\tau_{\pi_i}(e))})$ *in the worst case.*

Proof. We first prove $\|\tau(e)\| = \Omega(\|e\| \cdot 2^{\mathsf{u}(\tau(e))})$ in the worst case. In the worst case, we have $\mathsf{u}(\tau(e)) = \Theta(\|e\|)$. Let $e = \pi_1[(\ell_1 \circ \ell_2 \cup \ell_2 \circ \ell_1)^p \circ \ell^{p+1}]$. We have $\|(\ell_1 \circ \ell_2 \cup \ell_2 \circ \ell_1)^p\| = 4p-1$, $\|\ell^{p+1}\| = p$, $\|e\| = 5p$ and $\mathsf{u}(\tau(e)) = p$. This expression is rewritten into $e' = \pi_1[e_1]$ with, for i, $1 \leq i < p$, $e_i = \ell_1 \ltimes (\ell_2 \ltimes e_{i+1}) \cup \ell_2 \ltimes (\ell_1 \ltimes e_{i+1})$ and $e_p = \ell \ltimes (\ell \ltimes (\cdots \ltimes \ell) \ldots)$.

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

We have $\|e'\| = 1 + \|e_1\|$ with $\|e_i\| = 5 + 2 \cdot \|e_{i+1}\|$ and $\|e_p\| = p$. Hence,

$$\begin{aligned}\|e'\| &= 1 + 5 \cdot (2^0 + 2^1 + \cdots + 2^{p-2}) + p \cdot 2^{p-1} \\ &\geq (p \cdot 2^p)/2 \\ &= ((\|e\|/5) \cdot 2^{(\|e\|/5)})/2 = \Omega(\|e\| \cdot 2^{\mathsf{u}(\tau(e))}).\end{aligned}$$

To prove that $\|\tau_{\pi_i}(e)\| = \Omega(\|e\| \cdot 2^{\mathsf{u}(\tau_{\pi_i}(e))})$ in the worst case, it now suffices to observe that $\tau_{\pi_i}(e) = \tau(e)$.

To prove the remainder of the Theorem, it suffices to show that $\tau(e)$, $\tau_{\pi_i}(e)$, $\tau_{\circ_1}(e;\varepsilon)$ and $\tau_{\circ_1}(e;\varepsilon)$, with e an expression in $\mathcal{N}_3^*$ and ε an expression, satisfy the following conditions:

1. If $x = \tau(e)$ and $u = \mathsf{u}(\tau(e))$, then $x \equiv_{\text{path}} e$, $\|x\| \leq \|e\| \cdot 2^u$ and eval-steps$(x) \leq u + \|e\|$.
2. If $i \in \{1, 2\}$, $x = \tau_{\pi_i}(e)$ and $u = \mathsf{u}(\tau_{\pi_i}(e))$, then $x \equiv_{\pi_i} e$, $\|x\| \leq \|e\| \cdot 2^u$ and eval-steps$(x) \leq u + \|e\|$.
3. If ε is non-recursive or right-recursive in variable $\mathfrak{N}$, $x = \tau_{\circ_1}(e;\varepsilon)$ and $u = \mathsf{u}(\tau_{\circ_1}(e;\varepsilon))$, then $x \equiv_{\pi_1} e \ltimes \varepsilon$, $\|x\| \leq (\|e\| + \|\varepsilon\| + 1) \cdot 2^u$ and $\mathcal{S}(x) = \mathcal{S}(\varepsilon) \cup T$ with $|T| \leq \|e\| + u + 1$. If ε is right-recursive in variable $\mathfrak{N}$, then so is x, else x is non-recursive.
4. If ε is non-recursive or left-recursive in variable $\mathfrak{N}$, $x = \tau_{\circ_2}(e;\varepsilon)$ and $u = \mathsf{u}(\tau_{\circ_2}(e;\varepsilon))$, then $x \equiv_{\pi_2} \varepsilon \rtimes e$, $\|x\| \leq (\|e\| + \|\varepsilon\| + 1) \cdot 2^u$ and $\mathcal{S}(x) = \mathcal{S}(\varepsilon) \cup T$ with $|T| \leq \|e\| + u + 1$. If ε is left-recursive in variable $\mathfrak{N}$, then so is x, else x is non-recursive.

These properties are straightforward to prove using induction on the length of e. The base cases are the basic expressions and Lemma 4.1 is used to prove the inductive steps. ■

6.4. Fine-tuning the semi-join rewriting

The rules of Fig. 5 have been introduced with the purpose of proving relationships between fragments of the relation algebra and the semi-join algebra in terms of expressiveness rather than for optimizing query evaluation. It is therefore to be expected that these rewrite rules may not always improve query evaluation performance at every evaluation step, as is illustrated next.

EXAMPLE 6.2. Consider the expression $e = \pi_1[\ell_1 \circ \ell_2 \circ \ell_3]$. We have $\tau(e) = \pi_1[\ell_1 \ltimes (\ell_2 \ltimes \ell_3)]$. Now consider the graph $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ with $\mathcal{V} = \{m, n_1, \ldots, n_{|\mathcal{V}|-1}\}$, $\Sigma = \{\ell_1, \ell_2, \ell_3\}$, $\mathbf{E}(\ell_1) = \{(v, v) \mid v \in \mathcal{V}\}$, $\mathbf{E}(\ell_2) = \{(m, n_i) \mid 1 \leq i \leq |\mathcal{V}| - 1\}$ and $\mathbf{E}(\ell_3) = \{(n_i, m) \mid 1 \leq i \leq |\mathcal{V}| - 1\}$. This graph is visualized in Fig. 10.

We shall argue that evaluation of e by first evaluating the composition and then evaluating the projection is less costly than evaluation of $\tau(e)$. Observe that we have

$$[\![\ell_2 \circ \ell_3]\!]_{\mathcal{G}} = \{(m, m)\};$$

$$[\![\ell_2 \ltimes \ell_3]\!]_{\mathcal{G}} = \mathbf{E}(\ell_2).$$

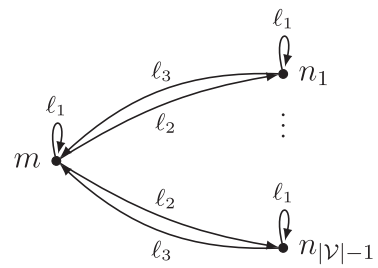

FIGURE 10. A graph with three edge labels, ℓ_1, ℓ_2, ℓ_3. This graph is used in Example 6.2.

Due to the intermediate result of evaluating $\ell_2 \circ \ell_3$ being much smaller than the intermediate result of evaluating $\ell_2 \ltimes \ell_3$, the follow-up composition of ℓ_1 with the intermediate query result $\{(m, m)\}$ will be much cheaper than the follow-up semi-join of ℓ_1 with the intermediate query result $\mathbf{E}(\ell_2)$. Moreover, in this specific example, algorithms for computing the compositions involved can easily achieve comparable performance to algorithms for computing the semi-joins involved.

In the following, we explore how the rewrite rules of Fig. 5 can be adjusted and used for graph query optimization. Remember that the cost of all operators is influenced primarily by the size of the evaluation results of its operands. From this observation, the issue shown in Example 6.2 can be easily explained: the rewrite rules of Fig. 5 can rewrite an expression e into an expression e' such that $|[\![e]\!]_{\mathcal{G}}| \ll |[\![e']\!]_{\mathcal{G}}|$. As a first step to alleviate this issue, we adjust the rewrite rules of Fig. 5 in such a way that the resulting rules provide the following strong guarantee: rewriting an expression e always yields and expression whose evaluation result, on any graph, is guaranteed to be upper-bounded by the evaluation result of e. We do so by modifying $\tau_{\circ_i}(e;\varepsilon)$ by replacing rewritings of the form $g \ltimes \varepsilon$ by $\pi_1[g \ltimes \varepsilon]$ and of the form $\varepsilon \rtimes g$ by $\pi_2[\varepsilon \rtimes g]$.

PROPOSITION 6.2. *Let $\mathcal{G}$ be a graph, let e be an expression in $\mathcal{N}_3^*$ and let ε be an expression. If we use the rewrite rules of Fig. 5 with the above modifications, then $\tau_{\circ_1}(e;\varepsilon)$ and $\tau_{\circ_2}(e;\varepsilon)$ are node expressions and their evaluation yields the smallest possible sets such that $[\![\tau_{\circ_1}(e;\varepsilon)]\!]_{\mathcal{G}}|_1 = [\![e \ltimes \varepsilon]\!]_{\mathcal{G}}|_1$ and $[\![\tau_{\circ_2}(e;\varepsilon)]\!]_{\mathcal{G}}|_2 = [\![\varepsilon \rtimes e]\!]_{\mathcal{G}}|_2$.*

With a minimal modification to the rewrite rules for $\tau_{\pi_i}(e)$, $i \in \{1, 2\}$, we can also guarantee that $\tau_{\pi_i}(e)$ minimizes intermediate evaluation results in the same way as the modified version of $\tau_{\circ_i}(e;\varepsilon)$ does. We do so by, additionally, applying the following two changes to $\tau_{\pi_i}(e)$:

$$\tau_{\pi_i}(b) = \pi_i[b];$$

$$\tau_{\pi_i}(e_1 \oplus e_2) = \pi_i[\tau(e_1) \oplus \tau(e_2)],$$

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

in which b is a basic expression and $\oplus \in \{\cap, -\}$. Using a straightforward induction argument, we obtain

PROPOSITION 6.3. *Let $\mathcal{G}$ be a graph and let e be an expression in $\mathcal{N}_3^*$. If we use the rewrite rules of Fig. 5 with the above modifications, then $\tau_{\pi_1}(e)$ and $\tau_{\pi_2}(e)$ are node expressions and their evaluation yields the smallest possible sets such that $[\![\tau_{\pi_1}(e)]\!]_{\mathcal{G}}|_1 = [\![e]\!]_{\mathcal{G}}|_1$ and $[\![\tau_{\pi_2}(e)]\!]_{\mathcal{G}}|_2 = [\![e]\!]_{\mathcal{G}}|_2$.*

From Proposition 6.3, we conclude:

COROLLARY 6.1. Let $\mathcal{G}$ be a graph, let e be an expression in $\mathcal{N}_3^*$ and $i \in \{1, 2\}$. If we use the rewrite rules of Fig. 5 with the above modifications, then we have $|[\![\tau_{\pi_i}(e)]\!]_{\mathcal{G}}| \leq |[\![e]\!]_{\mathcal{G}}|$.

Notice that the modified rewrite rules for $\tau_{\circ_i}(e; \varepsilon)$ and $\tau_{\pi_i}(e)$ will result in additional projection steps. In practice, we can easily eliminate such extra evaluation steps by integrating projection steps into the algorithms for evaluating the other operators, e.g. by not only using a general-purpose semi-join algorithm returning a binary relation, but also using semi-join algorithms computing only the first or second column of this binary relation. These specialized single-column operators can be evaluated at least as efficiently as the original operators and, consequently, including projection steps within other operators is frequently employed in practical query evaluation engines [27, 53–57]. This makes an attractive proposition to eliminate every usage of the projection operators. Hence, even though strictly speaking the number of evaluation steps slightly increases by the above changes, this does not translate into an increase in the evaluation cost of the resultant rewritten expressions if these operators are properly implemented.

6.5. Dealing with other expensive operators

The semi-join rewrite rules introduced are aimed at optimizing query evaluation for common usages of compositions and Kleene-stars in cases where their full expressive power is unnecessary. In Section 6.1, we argued that also identity, diversity and coprojections are to be avoided. Next, we recognize common usages in which the full expressive power of identity, diversity and coprojections is unnecessary. To enable optimization of query evaluation in these common cases, we introduce the selection operators $\sigma_=$ and $\sigma_\neq$ and the anti-semi-join operators $\bar{\ltimes}$ and $\bar{\rtimes}$. The semantics of these operators is defined by

$$
\begin{aligned}
[\![\sigma_=(e)]\!]_{\mathcal{G}} &= \{(n_1, n_2) \mid (n_1, n_2) \in [\![e]\!]_{\mathcal{G}} \wedge (n_1 = n_2)\};\\
[\![\sigma_\neq(e)]\!]_{\mathcal{G}} &= \{(n_1, n_2) \mid (n_1, n_2) \in [\![e]\!]_{\mathcal{G}} \wedge (n_1 \neq n_2)\};\\
[\![e_1 \bar{\ltimes} e_2]\!]_{\mathcal{G}} &= \{(m, n) \mid (m, n) \in [\![e_1]\!]_{\mathcal{G}} \wedge\\
&\qquad \neg\exists z\ (n, z) \in [\![e_2]\!]_{\mathcal{G}}\};\\
[\![e_1 \bar{\rtimes} e_2]\!]_{\mathcal{G}} &= \{(m, n) \mid (m, n) \in [\![e_2]\!]_{\mathcal{G}} \wedge\\
&\qquad \neg\exists z\ (z, m) \in [\![e_1]\!]_{\mathcal{G}}\}.
\end{aligned}
$$

Before we use these newly introduced operators, we classify these operators in the style of Lemma 6.1.

LEMMA 6.2. *The operators $\sigma_=$, $\sigma_\neq$, $\bar{\ltimes}$ and $\bar{\rtimes}$ are expression-linear.*

Next, we look at ways to rewrite common usages of identity and diversity. As illustrated by Example 4.4, common usages of identity and diversity involve intersection and difference. In these usages, identity and diversity are used to restrict query results to keep only node pairs of the form (n, n) or to restrict query results to filter out node pairs of the form (n, n). In these use cases, we can introduce selection operators:

PROPOSITION 6.4. *Let e be an expression. We have*

1. $e \cap id \equiv_{\text{path}} id \cap e \equiv_{\text{path}} e - di \equiv_{\text{path}} \sigma_=(e)$;
2. $e \cap di \equiv_{\text{path}} di \cap e \equiv_{\text{path}} e - id \equiv_{\text{path}} \sigma_\neq(e)$.

We observe that expression of the form $e - \text{di}$ and $\sigma_=(e)$ are node expressions. Hence, we can add these cases to the definition of $\mathsf{ns}(\cdot)$. Besides the above usages of identity, we observe that compositions and semi-joins with identity (id) can always be eliminated.

Finally, we look at ways to rewrite common usages of coprojections. In Section 3.4, we observed that $e_1 \circ \pi_1[e_2] \equiv_{\text{path}} e_1 \ltimes \pi_1[e_2] \equiv_{\text{path}} e_1 \ltimes e_2$. We use anti-semi-joins to generalize these rewritings to also cover coprojections:

PROPOSITION 6.5. *Let e and e' be expressions. We have*

1. $e \circ \overline{\pi}_1[e'] \equiv_{\text{path}} e \bar{\ltimes} e'$ *and* $e \circ \overline{\pi}_2[e'] \equiv_{\text{path}} e \bar{\ltimes} \pi_2[e']$;
2. $\overline{\pi}_1[e'] \circ e \equiv_{\text{path}} \pi_1[e'] \bar{\rtimes} e$ *and* $\overline{\pi}_2[e'] \circ e \equiv_{\text{path}} e' \bar{\rtimes} e$.

If, additionally, e is a node expression, then also

1. $e \cap \overline{\pi}_1[e'] \equiv_{\text{path}} e \bar{\ltimes} e'$ *and* $e \cap \overline{\pi}_2[e'] \equiv_{\text{path}} e \bar{\ltimes} \pi_2[e']$;
2. $e - \overline{\pi}_1[e'] \equiv_{\text{path}} e \ltimes e'$ *and* $e - \overline{\pi}_2[e'] \equiv_{\text{path}} e \ltimes \pi_2[e']$.

6.6. Data-complexity of rewritten expressions

We will consider the complexity of evaluating expressions e and the complexity of evaluating the rewritten expression $\tau(e)$ (using the rewrite rules of Section 6.3). To do so, we use the usual query evaluation complexity framework [59]. The worst-case complexity of evaluating any expression e' is $\mathcal{O}(\text{eval-steps}(e') \cdot c)$, where $\text{eval-steps}(e')$ is the number of evaluation steps and c is the maximum cost for performing a single evaluation step. Hence, the *query complexity*—the cost of evaluating an expression in terms of the size of the expression given a fixed graph—is $\mathcal{O}(\text{eval-steps}(e))$ and the *data complexity*—the cost of evaluating a query in terms of the size of the graph given a fixed query—is $\mathcal{O}(c)$.

We can verify that the rewrite rules $\tau(e)$, $\tau_{\pi_1}(e)$ and $\tau_{\pi_2}(e)$ reduce the data-complexity in two distinct ways. First, the number of expensive non-linear operators (composition and

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

Kleene-star) is reduced in favor of cheaper expression-linear operators (possibly reducing c, but never increasing it). Second, by Corollary 6.1, the size of evaluation results for subexpressions is minimized whenever possible, reducing the cost of evaluating non-rewritten operators.

The cost of the reduction in the data-complexity of evaluating an expression optimized by $\tau(e)$, $\tau_{\pi_1}(e)$ or $\tau_{\pi_2}(e)$ is an increase in the query-complexity of evaluating the optimized expression. This increase in the query-complexity is caused by an increase in the evaluation size and, in the worst case, this is an exponential increase:

EXAMPLE 6.3. Consider the expressions e and $e' = \tau_{\pi_1}(e)$ of Example 6.1. By Theorem 4.1, we have $e' \equiv_{\pi_1} e'$. During rewriting, the expression size did not increase, while the evaluation size did sharply increase: we have $\|e\| = \|e'\| = 7$, eval-steps$(e) = 3$ and eval-steps$(e') = 7$. As a consequence, evaluating e and e' by evaluating each of the operators involved is possible in worst-case $\mathcal{O}(3 \cdot |\mathbf{E}(\ell)|^2)$ and $\mathcal{O}(7 \cdot |\mathbf{E}(\ell)|)$, respectively. Hence, any increase in the query-complexity is accompanied by a sharp decrease in the data-complexity.

Even with a worst-case exponential increase in the query-complexity, Theorem 4.1 guarantees that the query complexity is linearly upper-bounded by the size of the original query. Hence, when queries are small and the data graphs are large, which is usually the case, the increase of the query complexity is a good trade-off if the data complexity decreases significantly.

7. CONCLUSION AND FUTURE WORK

The main theme of this paper is the relationship between the relation algebra and the semi-join algebra. We studied these relationships with query optimization in mind: we aimed at rewriting relation algebra expressions containing costly composition and Kleene-star operators into semi-join algebra expressions containing less costly semi-join and fixpoint operators. To do so, we identified sufficient conditions on relation algebra expressions that allow us to perform such rewritings and we have shown that these conditions are not too restrictive.

To make the theory applicable, we presented rewrite rules which can be used to rewrite (parts of) relation algebra expressions that satisfy the conditions identified. In addition, we have provided a complexity analysis that shows that our rewrite rules lead to only a well-bounded increase in the number of steps needed to evaluate the rewritten queries (while, at the same time, strictly reducing the number of costly composition and Kleene-star operations).

Since the relation algebra and the semi-join algebra correspond to FO[3] and FO[2], respectively, our rewrite rules also provided new insights into the relationship between these first-order logics. In addition, by specializing our results to node-labeled sibling-ordered trees, we were able to obtain new insights in the relationship between the expressive power of full first-order logic and FO[2] on such trees.

Even though our rewrite rules do not completely solve the efficient query evaluation problem for the relation algebra, we have been able to verify that our rewrite rules capture optimizations that have not yet been fully exploited by existing database systems (see, e.g. Hellings [60, Chapter 11]). Hence, a practical empirical study of a query evaluation system that combines our rewrite rules with other query optimization and evaluation techniques is an obvious avenue of further research. To illustrate this, we consider graph querying in social networks such as visualized in Fig. 1. For users, a core feature of social networks is to connect with old and new friends. To support this, several social networks will suggest new friends to users by suggesting friends-of-friends that are not already friends. These friends-of-friends can be retrieved via the query

$$\mathit{SuggestFriends} = (\mathit{FriendOf} \circ \mathit{FriendOf}) - (\mathit{FriendOf} \cup \mathrm{id}).$$

If we want to provide Alice with friend suggestions, we simply evaluate the above query and then select all nodes m such that the pair (Alice, m) is in the result of query *SuggestFriends*. This approach is far from optimal: we ran a complex query on the social network, after which we selected and used only a very small portion of the retrieved data. As social networks tend to be extremely large graphs, this approach is unacceptably expensive.

A more practical query language would allow us to *select* the node Alice within the query, which allows the query optimizer to take advantage of the selection to simplify query evaluation. To illustrate this, we can consider using a simple *node-selection operator* $\langle \mathit{Alice} \rangle$ to retrieve friend suggestions for Alice:

$$\mathit{SuggestAliceFriends} = \pi_2[\langle \mathit{Alice} \rangle \circ ((\mathit{FriendOf} \circ \mathit{FriendOf}) - (\mathit{FriendOf} \cup \mathrm{id}))].$$

We observe that evaluating the query *SuggestAliceFriends* as-is will be as inefficient as the original approach. A standard approach toward query optimization in existing database system is to push down selections. We can do so in this example by pushing the $\langle \mathit{Alice} \rangle$-selection through composition and difference. These selections enable additional semi-join rewrites and, consequently, the above *SuggestAliceFriends*-query can be optimized to the path-equivalent expression

$$\pi_2[(\langle \mathit{Alice} \rangle \rtimes \mathit{FriendOf}) \rtimes (\mathit{FriendOf} \bar{\ltimes} \pi_2[(\langle \mathit{Alice} \rangle \rtimes \mathit{FriendOf}) \cup \langle \mathit{Alice} \rangle])],$$

which is straightforward to evaluate in a highly efficient manner. Our initial look at combining our semi-joins rewrite rules with traditional query optimization techniques shows many

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

promising opportunities, including rewrite techniques that can be used to establish the above rewriting [60].

Other directions for future work include the following:

1. In this paper, we have introduced rewriting techniques for the semi-join algebra specialized to binary relations. The semi-join algebra has also been studied extensively for traditional relational databases (see, e.g. [22–25]) and several of its expressiveness properties and query evaluation benefits have been identified and used in practice. We plan to investigate how our techniques can be generalized and be of benefit for relational database query optimization and evaluation.
2. The intersection and difference operators limit the applicability of our rewriting techniques. For several restricted graph structures, well-known collapse results exist to eliminate intersection and difference (see e.g. [6, 18, 47]). Unfortunately, these known elimination results blow up the size of the resulting query excessively. Still, it is worthwhile to investigate whether more practical approaches exist to eliminate intersection and differences in the scope of semi-join based query optimization. At this point, we also notice that the combination of composition, intersection and difference leads to *cyclic joins*, for which it is known that only multi-way join algorithms can answer these optimally [61, 62].[7]
3. As argued in the Introduction, the strength of the graph query language $\mathcal{N}_3^*$ is navigation. Beyond navigation, the expressive power of $\mathcal{N}_3^*$ is rather limited, even with respect to basic counting. The language is, for example, incapable of expressing that nodes have a minimum number of incoming or outgoing edges. For FO[2] and FO[3] these limitations are usually lifted by adding so-called counting quantifiers [31]. Such counting quantifiers can also be added to $\mathcal{N}_3^*$ and $\mathcal{N}_2^{\mathsf{fp}}$. It is open to determine to which extent our rewrite rules and query optimization results are generalizable toward these languages.

Acknowledgments

The material is in part based upon work supported by the National Science Foundation under Grant No. NSF 1438990.

REFERENCES

[1] Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J. and Vrgoč, D. (2017) Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50, 68:1–68:40.

[2] Harris, S. and Seaborne, A. (2013) SPARQL 1.1 Query Language. *W3C Recommendation*. W3C, Cambridge, MA, USA, http://www.w3.org/TR/2013/REC-sparql11-query-20130321.

[3] Neo4j Team. (2019) The Neo4j Cypher Manual v3.5. Neo4j, Inc., San Francisco, CA, USA and Malmö, Sweden, https://neo4j.com/docs/cypher-manual/current/.

[4] Apache TinkerPop. (2019) Apache TinkerPop Documentation v3.4.4. The Apache Software Foundation, Wakefield, MA, USA, https://tinkerpop.apache.org/docs/current/reference/.

[5] Tarski, A. (1941) On the calculus of relations. *J. Symb. Log.*, 6, 73–89.

[6] Benedikt, M., Fan, W. and Kuper, G. (2005) Structural properties of XPath fragments. *Theoret. Comput. Sci.*, 336, 3–31.

[7] Benedikt, M. and Koch, C. (2009) XPath leashed. *ACM Comput. Surv.*, 41, 3:1–3:54.

[8] Clark, J. and DeRose, S. (1999) XML Path Language (XPath) v1.0. *W3C Recommendation*. W3C, Cambridge, MA, USA, http://www.w3.org/TR/1999/REC-xpath-19991116.

[9] Marx, M. (2005) Conditional XPath. *ACM Trans. Database Syst.*, 30, 929–959.

[10] Marx, M. and de Rijke, M. (2005) Semantic characterizations of navigational XPath. *SIGMOD Record*, 34, 41–46.

[11] ten Cate, B. (2006) The expressivity of XPath with transitive closure. *Proc. of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Chicago, IL, USA, June 26–28, pp. 328–337. ACM, New York, NY, USA.

[12] ten Cate, B. and Marx, M. (2007) Navigational XPath: Calculus and algebra. *SIGMOD Record*, 36, 19–26.

[13] Libkin, L., Martens, W. and Vrgoč, D. (2013) Querying graph databases with XPath. *Proc. of the 16th Int. Conf. on Database Theory*, Genoa, Italy, March 18–22, pp. 129–140. ACM, New York, NY, USA.

[14] Cruz, I.F., Mendelzon, A.O. and Wood, P.T. (1987) A graphical query language supporting recursion. *Proc. of the 1987 ACM SIGMOD Int. Conf. on Management of Data*, San Francisco, CA, USA, May 27–29, pp. 323–330. ACM, New York, NY, USA.

[15] Fletcher, G.H.L., Gyssens, M., Leinders, D., Surinx, D., Van den Bussche, J., Van Gucht, D., Vansummeren, S. and Wu, Y. (2015) Relative expressive power of navigational querying on graphs. *Inf. Sci.*, 298, 390–406.

[16] Fletcher, G.H.L., Gyssens, M., Leinders, D., Van den Bussche, J., Van Gucht, D., Vansummeren, S. and Wu, Y. (2015) The impact of transitive closure on the expressiveness of navigational query languages on unlabeled graphs. *Ann. Math. Artif. Intell.*, 73, 167–203.

[17] Givant, S. (2006) The calculus of relations as a foundation for mathematics. *J. Autom. Reason.*, 37, 277–322.

[18] Hellings, J., Gyssens, M., Wu, Y., Van Gucht, D., Van den Bussche, J., Vansummeren, S. and Fletcher, G.H.L. (2015) Relative expressive power of downward fragments of navigational query languages on trees and chains. *Proc. of the 15th Symposium on Database Programming Languages*, Pittsburgh, PA, USA, October 25–30, pp. 59–68. ACM, New York, NY, USA.

[19] Surinx, D., Fletcher, G.H.L., Gyssens, M., Leinders, D., Van den Bussche, J., Van Gucht, D., Vansummeren, S. and Wu, Y. (2015) Relative expressive power of navigational querying on graphs using transitive closure. *Log. J. IGPL*, 23, 759–788.

[7] Cyclic joins not only have a huge effect on the complexity of query evaluation and the complexity of distributed query evaluation [26, 61], but also on many other related problems that conceptually involve joins, e.g. checking integrity of relational data with respect to join dependencies [63].

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

[20] Leinders, D. and Van den Bussche, J. (2007) On the complexity of division and set joins in the relational algebra. *J. Comput. Syst. Sci.*, 73, 538–549. Special Issue: Database Theory 2005.

[21] Codd, E.F. (1970) A relational model of data for large shared data banks. *Communications of the ACM, **13***.

[22] Leinders, D. (2008) The semijoin algebra. PhD thesis, *Hasselt University and Transnational University of Limburg*.

[23] Leinders, D., Tyszkiewicz, J. and Van den Bussche, J. (2004) On the expressive power of semijoin queries. *Inform. Process. Lett.*, 91, 93–98.

[24] Leinders, D., Marx, M., Tyszkiewicz, J. and Van den Bussche, J. (2005) The semijoin algebra and the guarded fragment. *J. Log. Lang. Inf.*, 14, 331–343.

[25] Klausner, A. and Goodman, N. (1985) Multirelations: Semantice and languages. *Proc. of the 11th Int. Conf. on Very Large Data Bases, Stockholm*, Sweden, August 21–23, pp. 251–258. Morgan Kaufmann, Burlington, MA, USA.

[26] Bernstein, P.A. and Chiu, D.-M.W. (1981) Using semi-joins to solve relational queries. *J. ACM*, 28, 25–40.

[27] Ullman, J.D. (1990) *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA.

[28] Yannakakis, M. (1981) Algorithms for acyclic database schemes. *Proc. of the 7th Int. Conf. on Very Large Data Bases*, Cannes, France, September 9–11, pp. 82–94. IEEE Computer Society, Washington, DC, USA.

[29] Seshadri, P., Hellerstein, J.M., Pirahesh, H., Leung, T.Y.C., Ramakrishnan, R., Srivastava, D., Stuckey, P.J. and Sudarshan, S. (1996) Cost-based optimization for magic: Algebra and implementation. *SIGMOD Record*, 25, 435–446.

[30] Grädel, E. and Otto, M. (1999) On logics with two variables. *Theoret. Comput. Sci.*, 224, 73–113.

[31] Grohe, M. (1998) Finite variable logics in descriptive complexity theory. *Bull. Symb. Log.*, 4, 345–398.

[32] Libkin, L. (2004) *Elements of Finite Model Theory*. Springer, Berlin-Heidelberg, Germany.

[33] Otto, M. (1999) Bounded variable logics: Two, three, and more. *Arch. Math. Logic*, 38, 235–256.

[34] Otto, M. (1996) The expressive power of fixed-point logic with counting. *J. Symb. Log.*, 61, 147–176.

[35] Fletcher, G.H.L., Gyssens, M., Leinders, D., Van den Bussche, J., Van Gucht, D., Vansummeren, S. and Wu, Y. (2011) Relative expressive power of navigational querying on graphs. *Proc. of the 14th Int. Conf. on Database Theory*, Uppsala, Sweden, March 21–24, pp. 197–207. ACM, New York, NY.

[36] Hellings, J., Pilachowski, C.L., Van Gucht, D., Gyssens, M. and Wu, Y. (2017) From relation algebra to semi-join algebra: An approach for graph query optimization. *Proc. of the 16th Int. Symposium on Database Programming Languages*, Munich, Germany, September 1, pp. 5:1–5:10. ACM, New York, NY, USA.

[37] Fletcher, G.H.L., Gyssens, M., Leinders, D., Van den Bussche, J., Van Gucht, D., Vansummeren, S. and Wu, Y. (2012) The impact of transitive closure on the boolean expressiveness of navigational query languages on graphs. *Proc. of the 7th Int. Symposium on Foundations of Information and Knowledge Systems*, Kiel, Germany, March 5–9, Lecture Notes in Computer Science, vol. 7153, pp. 124–143. Springer, Berlin-Heidelberg, Germany.

[38] Hellings, J., Wu, Y., Gyssens, M. and Van Gucht, D. (2018) The power of Tarski's relation algebra on trees. *Proc. of the 10th Int. Symposium on Foundations of Information and Knowledge Systems*, Budapest, Hungary, May 14–18, Lecture Notes in Computer Science, vol. 10833, pp. 244–264. Springer, Berlin-Heidelberg, Germany.

[39] Linz, P. (2012) *An Introduction to Formal Languages and Automata* (5th edn). Jones and Bartlett Publishers, Inc., Sudbury, MA, USA.

[40] Marx, M. and Venema, Y. (1997) Multi-Dimensional Modal Logic. In *Applied Logic Series, vol. 4*. Springer Netherlands, Dordrecht, the Netherlands.

[41] Barceló, P. (2013) Querying graph databases. *Proc. of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, New York, NY, USA, June 22–27, pp. 175–188. ACM, New York, NY, USA.

[42] Barceló, P., Pérez, J., and Reutter, J. L. (2012) Relative expressiveness of nested regular expressions. *Proc. of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management*, Ouro Preto, Brazil, June 27–30, pp. 180–195. CEUR Workshop Proceedings, vol. 866, CEUR-WS.org, RWTH Aachen, Germany.

[43] Consens, M.P. and Mendelzon, A.O. (1990) GraphLog: A visual formalism for real life recursion. *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Nashville, TN, USA, April 2–4, pp. 404–416. ACM, New York, NY, USA.

[44] Calvanese, D., Giacomo, G.D., Lenzerini, M. and Vardi, M.Y. (2000) Containment of conjunctive regular path queries with inverse. *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning*, Breckenridge, CO, USA, April 11–15, pp. 176–185. Morgan Kaufmann, Burlington, MA, USA.

[45] Reutter, J.L., Romero, M. and Vardi, M.Y. (2017) Regular queries on graph databases. *Theory Comput. Syst.*, 61, 31–83.

[46] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. (2006) Extensible markup language (XML) 1.1, 2nd edition. *W3C recommendation*. W3C, Cambridge, MA, USA, http://www.w3.org/TR/2006/REC-xml11-20060816.

[47] Wu, Y., Van Gucht, D., Gyssens, M. and Paredaens, J. (2011) A study of a positive fragment of path queries: Expressiveness, normal form and minimization. *Comput. J.*, 54, 1091–1118.

[48] Schreiber, G. and Raimond, Y. (2014) RDF 1.1 primer. *W3C working group note*. W3C, Cambridge, MA, USA, http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624.

[49] Fischer, M.J. and Ladner, R.E. (1979) Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18, 194–211.

[50] Kozen, D. (1997) Kleene algebra with tests. *ACM Trans. Programming Lang. Syst.*, 19, 427–443.

[51] Clarke, E.M., Grumberg, O. and Peled, D. (1999) *Model Checking*. The MIT Press, Cambridge, MA, USA.

[52] Silberschatz, A., Korth, H.F. and Sudarshan, S. (2011) *Database System Concepts* (6th edn). McGraw-Hill, New York, NY, USA.

[53] Ioannidis, Y.E. (1996) Query optimization. *ACM Comput. Surveys*, 28, 121–123.

[54] Chaudhuri, S. (1998) An overview of query optimization in relational systems. *Proc. of the 17th ACM SIGACT-SIGMOD-*

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022

SIGART Symposium on Principles of Database Systems, Seattle, Washington, June 1–3, pp. 34–43. ACM, New York, NY, USA.

[55] Mannino, M.V., Chu, P. and Sager, T. (1988) Statistical profile estimation in database systems. *ACM Comput. Surveys*, 20, 191–221.

[56] Jarke, M. and Koch, J. (1984) Query optimization in database systems. *ACM Comput. Surveys*, 16, 111–152.

[57] Graefe, G. (1993) Query evaluation techniques for large databases. *ACM Comput. Surveys*, 25, 73–169.

[58] Cleaveland, R. and Steffen, B. (1993) A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Form. Methods System Design*, 2, 121–147.

[59] Vardi, M.Y. (1982) The complexity of relational query languages (extended abstract). *Proc. of the 14th Annual ACM Symposium on Theory of Computing*, San Francisco, CA, USA, May 5–7, pp. 137–146. ACM, New York, NY, USA.

[60] Hellings, J. (2018) On Tarski's Relation Algebra: querying trees and chains and the semi-join algebra. PhD thesis, *Hasselt University and Transnational University of Limburg, Hasselt*, Belgium.

[61] Atserias, A., Grohe, M. and Marx, D. (2013) Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42, 1737–1767.

[62] Veldhuizen, T. L. (2014) Leapfrog triejoin: A simple, worst-case optimal join algorithm. *Proc. of the 17th Int. Conf. on Database Theory*, Athens, Greece, March 24–28, pp. 96–106. OpenProceedings.org, University of Konstanz, Germany.

[63] Gyssens, M. (1986) On the complexity of join dependencies. *ACM Trans. on Database Syst.*, 11, 81–108.

Downloaded from https://academic.oup.com/comjnl/article/64/5/789/5827080 by Hasselt Universiteit user on 15 November 2022