Eindhoven University of Technology

MASTER

Regular queries with memory from theory to practice

Mulder, Thomas

Award date:
2019

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 08. Jul. 2025

Regular Queries with Memory:

From Theory to Practice

Master Thesis

Thomas Mulder

Committee members:

Dr. Nikolay Yakovets (Supervisor)

Dr. George Fletcher (Supervisor)

Dr. Wouter Duivesteijn

Department of Mathematics & Computer Science

Database Group

Eindhoven University of Technology

April 15, 2019

Abstract

We introduce a plan space and enumeration procedure for regular queries with memory (RQMs), based on existing k-register automata and WavePlans. Furthermore, we analyse the complexity of the plan space and enumeration procedure and show that cost-based enumeration using dynamic programming is possible. Additionally, we experimentally justify the proposed plan space by showing that in a substantial number of cases an enumeration procedure that considers the topological- and data constraints of an RQM separately misses out on optimal plans, and that planning the data constraints of an RQM, even for a fixed topological plan can produce order of magnitude performance differences.

Contents

List of Figures **4**

1 Introduction **6**

2 Related Work **10**

3 Preliminaries **11**

3.1 Data Graph 11

3.2 Path queries 12

3.2.1 Regular path queries 12

3.2.2 Regular queries with memory 13

3.2.3 Complexity and expressive power 16

4 Query Planning **17**

4.1 Plan operations 17

4.1.1 Projection 17

4.1.2 Assignments and conditions 18

4.1.3 Local and global data joins 20

4.2 k-Register Waveplans 20

4.2.1 Waveplans 20

4.3 Plan Space 23

5 Plan Enumeration **25**

5.1 Rule-based enumeration 26

5.1.1 Sub-expressions of REMs 26

5.1.2 Projection graph 28

5.1.3 Regular enumeration rules 29
5.1.4 Data enumeration rules 33
5.2 Cost-based enumeration 34
5.2.1 Cost estimation 34
5.2.2 Optimal substructure 35
5.2.3 Complexity of cost-based enumeration 36

6 Plan Execution **41**
6.1 Search routine 41
6.1.1 Crank routine 42

7 Experimental Setup **46**
7.1 Implementation 46
7.1.1 Planning 46
7.1.2 Execution 46
7.1.3 Hardware 47
7.2 Edge walks 48
7.3 Input 48
7.3.1 Data graph 48
7.3.2 Workload 49
7.4 Planning orthogonality 51
7.4.1 Improvement ratio 52
7.5 Data planning performance 53

8 Experimental Results **54**
8.1 Planning orthogonality 54
8.2 Data planning performance 54

9 Conclusion **57**

References **59**

List of Figures

1.1 Movie property graph (a) and a data graph based on that property graph (b) . . 7

4.1 A WavePlan for `likes` · `directedBy` · `dealsWith` 22

4.2 A k-register waveplan for `likes` · $\downarrow x_0$.`directedBy` · `dealsWith` · `produced`$[x_0^=]$ · `likedBy` 22

5.1 Projection graph for $e = a \cdot \downarrow x_0.b \cdot c[5^= \wedge x_0^=] \cdot d$. 29

5.2 Enumeration rules for the regular part of expressions. 30

5.3 A naive WavePlan for $(ab)^+$. 32

5.4 Enumeration rules for adding data joins. 34

5.5 Sub-expressions of regular expression $r = abcdef$. 38

6.1 A k-register waveplan for the running example query. 43

6.2 Execution of the plan from Figure 6.1. 44

7.1 Overview of the proposed system. 47

7.2 Serialization and categorization of a monetary value. 49

7.3 Graph representation of pattern $(a \cdot \downarrow x_0.b[x_0^=] \cdot c)^+$. 50

7.4 Graph representation of pattern $(\downarrow x_0.a \cdot b[x_0^=])^+$. 51

7.5 k-Register waveplan for e_1. 52

7.6 k-Register waveplan for e_2. 52

7.7 Sub-optimal k-register waveplan for e_3 52

7.8 Optimal k-register waveplan for e_3 53

7.9 Obtaining the sets of optimal plans for an RQM and its regular part. 53

8.1 Order of magnitude difference between best- and worst edge walks. 55

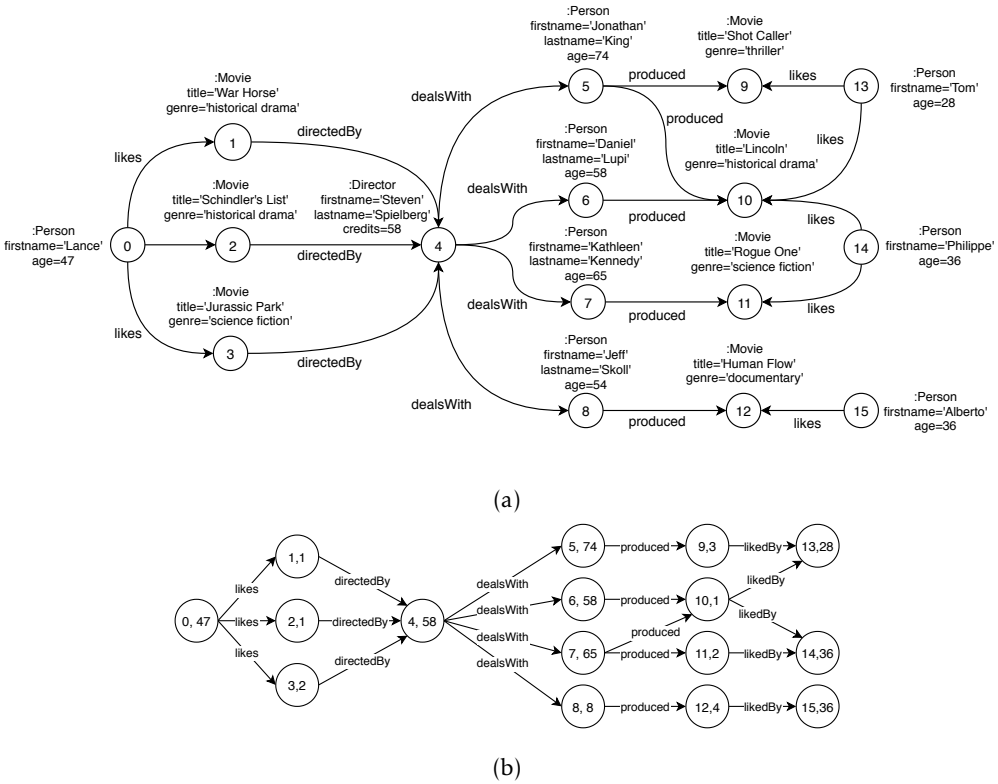
8.2 Relative frequency of ODE for subsets with shared topological order. 56

1 Introduction

Motivation. The graph database model has seen a strong, renewed interest in database research in recent years [1]. It elegantly captures the connectivity that exists in many types of data nowadays, which is something that is difficult to achieve in many traditional database management systems. Examples of such connected data are social networks (i.e. groups of people and the relationships amongst them), information networks (i.e. citation information on scientific papers or internet hypermedia), technological networks (i.e. computer networks, airline routes or power grids) and biological networks occurring in genomics, chemical structure and the relationships between species, for instance [2].

Part of a graph database model are the set of operations that can be performed on such a model. Querying is one such operation. Queries on a graph database model can be divided into two categories; queries on the *data* stored in the vertices of a graph, and queries on the *topology* of the graph. The former category is conceptually no different from traditional relation queries, whereas the latter is very different from- and hard to express in traditional relational queries. Even more valuable and interesting queries to ask of a graph database are those queries that combine both of these categories. That is, queries that express constraints on *topology and data*. [3][4][5].

Consider as an example the *property graph* presented in Figure 1.1a. This property graph is part of a movie database, capturing relations between persons, directors and movies. Such a property graph can be converted to a *data graph* where we allow only a single integer data value to be associated with each vertex. In this case, we associate each vertex of type `:Person` with its `age` property, and the `:Director` vertex with its `credits`. Additionally, the `genre` property of the `:Movie` type can be mapped to integer values 1, 2, 3 and 4 for values `historical drama`, `science fiction`, `thriller` and `documentary`, respectively. The resulting data graph is shown in Figure 1.1b. Such queries on data and topology often take the shape of path queries, where



(a)

(b)

Figure 1.1: Movie property graph (a) and a data graph based on that property graph (b)

we want to find all pairs of vertices u, v such that there exists a path from u to v where all edges and vertices along this path satisfy the constraints expressed in the path query. One class of these queries is known as regular path queries (RPQs) since the constraints are expressed through a regular language [1].

An example of a regular path query on a movie database is "Find all pairs u, v of people such that u likes a movie directed by someone who has worked with a producer that produced a movie that v likes". On the graphs in Figure 1.1 this would result in pairs $(0,13)$, $(0,14)$ and $(0,15)$.

These RPQs have been studied intensively over recent years, supplying a rich body of theoretical work [1]. One recent work suggests the extension of RPQs with memory (RQM) [3]. That is, besides expressing constraints on the edges and vertices through a regular language, the regular language is extended so as to allow the storage of data encountered along a path and comparison against the stored values later on.

For example, an RQM could ask that the movies liked by u and v are of the same genre, by storing the data values in vertices 1, 2 and 3 and comparing them against the data values in vertices 9, 10, 11 and 12. This would reduce the result to pairs $(0,13)$, $(0,14)$.

Problem statement. Query planning is a vital part of any database system and concerns translating a query into a multitude of *plans* that can be used to evaluate that query. Such a set of plans is called a *plan space*. Generating plans within a plan space is known as *enumeration* of that plan space. A plan in the broadest sense is an ordering of operations that, when executed, results in the answer to a query.

Query planning in the context of RQMs has not been studied thus far. While [3] introduces k-register automata as a way of representing RQMs, such an automaton represents only a single ordering of the operations necessary to evaluate a query.

Hence, the problem we study is that of representing RQMs in a way that allows a rich, yet tractable plan space to be defined and enumerated.

Contributions. We introduce k-register waveplans as a way of representing plans for RQMs, which are an extension of Waveplans as introduced in [6].

Additionally, we provide a plan space for RQMs, analyze its complexity and experimentally justify the use of this particular plan space, by showing that plans with the same structure *with respect to topology* but a different structure *with respect to data* display order of magnitude

performance differences. Moreover, a plan space and enumeration procedure that does not consider topology and data together misses out on finding optimal plans in a substantial proportion of cases. Hence, planning for topology and data *both* is essential.
Furthermore, we outline an enumeration procedure for the given plan space, analyze its complexity and present a brief sketch of a cost-model that allows for cost-based enumeration based on dynamic programming.

Organization. The remainder of this thesis is organized as follows; Chapter 2 provides a brief discussion of other research regarding querying graphs using registers. Chapter 3 provides the theoretical preliminaries from [3] on which this thesis builds directly. Chapter 4 introduces k-register waveplans and a plan space thereof in the context of RQMs. Chapter 5 provides a procedure for enumerating the proposed plan space and analyzes its complexity. Chapter 6 concludes the theoretical contributions by outlining an execution procedure that solves the evaluation problem of RQMs given an automaton as a plan. Chapter 7 acknowledges the way in which the proposed system was prototyped, and defines how the choice for the proposed plan space is empirically justified. Finally, Chapters 8 and 9 present the outcome of this empirical evaluation and summarize the theoretical- and experimental contributions.

2 Related Work

The expressiveness and complexities of graph query languages is an active research topic. Many languages and fragments thereof have been studied for queries regarding topological constraints. Few have considered the data encountered along a path.

Besides [3], on which this work is directly based, [4] compares the expressiveness of REMs as proposed in [3] to Walk Logic (WL), originally presented by Hellings et al. [5], concluding that while REMs are rather limited in terms of expressive power compared to WL, the latter is intractable due to its data complexity being non-elementary.

Subsequently, [4] proposes Register Logic (RL), that closes REMs under Boolean combinations and existential quantification over vertices, paths and register assignment. A fragment of this language, called NRL^+, is based on *nested* REMs (NREM), which extend the expressive power of REMs while remaining tractable since the combined complexity is shown to be in $PSPACE$.

An example, as given in [4], of a query that can be expressed in NRL^+, but not as an REM is: find pairs of vertices u and v such that there is a vertex w and a path π from u to v in which each vertex is connected to w.

They allow a vertex's identifier to be assigned to a register, which is equivalent to assuming all vertices have distinct data values in the data graph model. Hence, registers are used to test the connectedness of w.

3 Preliminaries

This chapter covers the theoretical preliminaries from [3] necessary to build the proposed system. First, we cover the data in terms of the data graph model and data paths. Second, a formal definition for regular queries with memory is provided.

3.1 Data Graph

A data graph is a collection of vertices and edges such that each vertex is associated with one integer value, called a *data value*. Additionally, edges are associated with a *label*. The set of all labels is referred to as an *alphabet*. Hence, this model is an abstracting of more commonplace graph models, such as property graphs. Figure 1.1 shows an example of a property graph, and a data graph that can be obtained from it. Formally, a data graph is defined as follows:

Definition 3.1.1. (Data Graph) A *data graph* over Σ and $\mathcal{D}$ is a tuple $G = \langle V, E, \rho \rangle$ with

- V a finite set of vertices, and
- $E \subseteq V \times \Sigma \times V$ a set of labeled edges, and
- $\rho : V \to \mathcal{D}$ a function that assigns a data value to each vertex in V.

where Σ is a finite labeling *alphabet* and $\mathcal{D} \subset \mathbb{Z}$ is a finite set of *data values*.

To express the connectedness of vertices within a graph we use the concept of a path. A path is a sequence of alternating vertices and edge labels, always starting- and ending with a vertex, such that all triples consisting of two vertices and an edge label represent an edge in the graph. Formally, a path is defined as:

Definition 3.1.2. (Path) A *path* from vertex v_1 to v_n in a graph $G = \langle V, E \rangle$ is a sequence

$$\pi = v_1 a_1 v_2 a_2 v_3 ... v_{n-1} a_{n-1} v_n$$

such that $(v_i, a_i, v_{i+1}) \in E$ for $1 \leq i < n$.

The notation $\lambda(\pi)$ is used to denote the *word* $a_1 ... a_{n-1}$ which is the sequence of labels along π. An example of a path is the sequence $\pi = 0$ `likes` 1 `directedBy` 4 in Figure 1.1b, which is a path from 0 to 4.

The notion of a path can be modified to cover data graphs, by replacing vertices with their data value. Formally, a data path is defined as:

Definition 3.1.3. (Data Path) Let $\pi = v_1 a_1 v_2 ... v_{n-1} a_{n-1} v_n$ be a path from v_1 to v_n in a data graph. The *data path* corresponding to π is

$$w_\pi = \rho(v_1) a_1 \rho(v_2) a_2 \rho(v_3) ... \rho(v_{n-1}) a_{n-1} \rho(v_n)$$

Thus, a data path is defined as a sequence of alternating data values and labels, always starting and ending with data values. The notation $\lambda(w_\pi)$ is used to denote the word $a_1 ... a_{n-1}$ for data paths as it is for normal paths.
The data path corresponding to $\pi = 0$ `likes` 1 `directedBy` 4 is $w_\pi = 47$ `likes` 1 `directedBy` 58.

3.2 Path queries

Queries on graphs come in many different forms. The particular form under consideration here, is an instance of a *path query*.

3.2.1 Regular path queries

A *path query* asks for all pairs (u, v) in G such that there exists a path π with a particular word $\lambda(\pi)$. *Regular path queries* (RPQs) are path queries where the word along the path must be part of a *regular language* L, rather than equal to a particular word. An RPQ Q is an expression of the form

$$Q = x \xrightarrow{L} y$$

where L is a regular language over Σ, typically represented as a *regular expression*, and x and y are variables that bind to vertices. Formally, the answer $Q(G)$ to an RPQ Q on graph G is the set of pairs of vertices (u, v) such that there is a path π from u to v with $\lambda(\pi) \in L$.

3.2.2 Regular queries with memory

Libkin et. al. [3] studies several language formalisms that can be used to extend RPQs by taking into account data values along a path, in terms of their relative expressive power and complexity. They conclude that *register automata* are the only formalism that can offer an interesting increase in expressive power compared to standard RPQs, while maintaining an acceptable complexity. A more detailed discussion of the expressive power and complexity follows at the end of this chapter.

Conditions on data

Register automata move from one state to another by reading the appropriate letter from a finite alphabet and comparing the currently read data value to the ones previously stored in the registers. The version of register automata under consideration here will use Boolean conditions to compare data values, rather than only checking for equality. To define such conditions formally, assume that, for each $k > 0$, we have variables $x_1, ..., x_k$. Then conditions in $\mathcal{C}_k$ are given by the grammar:

$$c \;:=\; x_i^{=} \mid x_i^{\neq} \mid z^{=} \mid z^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \quad 1 \leq i \leq k$$

where z is a data value from $\mathcal{D}$, also referred to as the *constant*. Let $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$, where $\perp$ is a special symbol signifying that the register is empty. The satisfaction of a condition is defined with respect to a data value $d \in \mathcal{D}$ and a tuple $\tau = (d_1, ..., d_k) \in \mathcal{D}_\perp^k$ as follows:

- $d, \tau \models x_i^{=}$ if and only if $d = d_i$,
- $d, \tau \models x_i^{\neq}$ if and only if $d \neq d_i$,
- $d, \tau \models z^{=}$ if and only if $d = z$,
- $d, \tau \models z^{\neq}$ if and only if $d \neq z$,
- $d, \tau \models c_1 \wedge c_2$ if and only if $d, \tau \models c_1$ and $d, \tau \models c_2$,
- $d, \tau \models c_1 \vee c_2$ if and only if $d, \tau \models c_1$ or $d, \tau \models c_2$, and

- $d, \tau \models \neg c$ if and only if $d, \tau \not\models c$.

The symbol ϵ is used as a shorthand for a condition that is true for any valuation and data value.

Regular expressions with memory

It is convenient to define an extension of regular expressions that allows data values to be stored in registers and incorporates the conditions previously defined. The regular language in a query can subsequently be defined by an expression rather than directly by a register automaton, which is impractical to write.
Typically, regular expressions for RPQs involve three operators; disjunction, conjunction and closure. Hence, a regular expression with memory (REM) consists of these operators, an assignment operator that denotes storing a data value in a number of registers and a condition operator that compares a data value against a Boolean condition. Formally, a regular expression with memory is defined as:

Definition 3.2.1. (Regular Expression with Memory) Let Σ be a finite alphabet and $x_1, ..., x_k$ a set of variables. Then *regular expressions with memory* (REMs) are defined by the grammar:

$$e \;:=\; \epsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow \overline{x}.e,$$

where a ranges over alphabet symbols, c over conditions in $\mathcal{C}_k$, and $\overline{x}$ over tuples of variables from $x_1, ..., x_k$. Furthermore, $e + e$ denotes disjunction, $e \cdot e$ conjunction, e^+ closure, $e[c]$ the application of condition c and $\downarrow \overline{x}.e$ the assignment of the current data value into registers i for all $x_i \in \overline{x}$.

A regular expression with memory e is *well formed* if it satisfies two conditions:

- Subexpressions e^+, $e[c]$ and $\downarrow \overline{x}.e$ are not allowed if e reduces to ϵ. Formally, e reduces to ϵ if it is ϵ, or it is $e_1 + e_2$ or $e_1 \cdot e_2$ or e_1^+ or $e_1[c]$ or $\downarrow \overline{x}.e_1$, where e_1 and e_2 reduce to ϵ.
- No variable appears in a condition before it appears in $\downarrow \overline{x}$.

In the remainder of this work, all REMs are assumed to be well formed.

Since a regular language that supports assignments to- and comparisons against registers can be defined in terms of an REM, regular path queries can now be extended to support these operations as well. Regular queries with memory are formally defined as:

Definition 3.2.2. (Regular query with memory) A *regular query with memory* (RQM) is an expression $Q = x \xrightarrow{e} y$, where e is a regular expression with memory. Given a data graph G, the result of the query $Q(G)$ consists of pairs of vertices (u, v) such that there is a data path w from u to v that belongs to the language $L(e)$ generated by e.

Recall the regular path query "Find all pairs u, v of people such that u likes a movie directed by someone who has worked with a producer that produced a movie that v likes", and the regular query with memory which additionally requires that the two movies involved are of the same genre. These queries can now be expressed formally as:

$$Q = x \xrightarrow{e} y, \quad \text{with}$$
$$e = \text{likes} \cdot \text{directedBy} \cdot \text{dealsWith} \cdot \text{produced} \cdot \text{likedBy}$$

and,

$$Q' = x \xrightarrow{e'} y, \quad \text{with}$$
$$e' = \text{likes} \cdot \downarrow x_0.\text{directedBy} \cdot \text{dealsWith} \cdot \text{produced}[x_0^=] \cdot \text{likedBy}$$

where one register, identified by variable x_0, is used to store the genre of the movie.

Expression semantics

In order to define the semantics of REMs, it is convenient to first define the *concatenation* of two data paths $w = d_1 a_1 ... a_{n-1} d_n$ and $w' = d_n a_n ... a_{m-1} d_m$. Notice that it is only defined if the last data value of w equals the first data value of w'. The definition naturally extends to concatenation of several data paths. If $w = w_1 ... w_l$, then $w_1 ... w_l$ is referred to as a a *splitting* of a data path w into $w_1, ..., w_l$.

The semantics of REMs is defined by means of a relation $(e, w, \sigma) \vdash \sigma'$, where e is a REM, w is a data path and both σ and σ' are k-tuples over $\mathcal{D} \cup \{\perp\}$. The intuition is as follows: starting with a memory configuration σ (i.e. values of $x_1, ..., x_k$), parsing w according to e yields a memory configuration σ'. The language of e is then defined as

$$L(e) \ = \ \{w \mid (e, w, \overline{\perp}) \vdash \sigma \text{ for some } \sigma\},$$

where $\overline{\perp}$ is the tuple of k values $\perp$.

The relation $\vdash$ is defined inductively on the structure of expressions. Recall that the empty word corresponds to a data path with a single data value d (i.e. a single vertex in a data graph). The notation $\sigma_{\overline{x}=d}$ is used for the valuation obtained from σ by setting all the variables in $\overline{x}$ to d:

- $(\epsilon, w, \sigma) \vdash \sigma'$ if and only if $w = d$ for some $d \in \mathcal{D}$ and $\sigma' = \sigma$,
- $(a, w, \sigma) \vdash \sigma'$ if and only if $w = d_1 a d_2$ and $\sigma' = \sigma$,
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ if and only if there is a splitting $w = w_1 \cdot w_2$ of w and a valuation σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$,
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ if and only if $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$,
- $(e^+, w, \sigma) \vdash \sigma'$ if and only if there is a splitting $w = w_1 ... w_m$ of w and valuations $\sigma = \sigma_0, \sigma_1, ..., \sigma_m = \sigma'$ such that $(e, w_i, \sigma_{i-1}) \vdash \sigma_i$ for all $i \in [m]$,
- $(\downarrow \overline{x}.e, w, \sigma) \vdash \sigma'$ if and only if $(e, w, \sigma_{\overline{x}=d}) \vdash \sigma'$, where d is the first data value of w, and
- $(e[c], w, \sigma) \vdash \sigma'$ if and only if $(e, w, \sigma) \vdash \sigma'$ and $\sigma', d \models c$, where d is the last data value of w.

Take note that in the last item it is required that σ', and not σ, satisfies c. The reason for this is that the initial assignment might change before reaching the end of the expression and this change should be reflected when we check that condition c holds.

3.2.3 Complexity and expressive power

Having defined regular queries with memory, it is useful to return to the issues of expressive power and complexity, so as to observe the motivation for studying RQMs in the first place. It is straightforward to see that RQMs properly subsume RPQs in terms of expressive power, provided that the regular expressions used in such RPQs allow only disjunction, conjunction and closure.

The increase in expressive power of being able to write constraints on data values using registers comes at the cost of increasing the combined complexity from P to NP-complete for RQMs over finite languages and to $PSPACE$-complete for RQMs in general. Fortunately, the $PSPACE$ combined complexity means that these queries remain tractable, which is not the case for some similar formalisms [3][4].

4 Query Planning

The necessity of query planning is well-known throughout DBMS research. Planning regular queries with memory comes with its very particular set of challenges, however. The aim of this chapter is to establish (1) which operations RQM plans must capture beyond those present in k-register automata and (2) which formalism can effectively represent these operations. Additionally, we will present a brief overview- and analysis of the *plan space*. That is, the set of plans that is considered for an arbitrary RQM.

We conclude that RQM plans must capture assignments, conditions and projection, as well as the order of assignments and conditions at a particular stage of a plan. Furthermore, we will see that an extension of WavePlans we call k-register waveplans that can capture these operations. Finally, we consider the plan space $\mathcal{P}_{SRP}$ and show that $O(|r|^n c^{|}r|)$ is an upper-bound on the complexity of $\mathcal{P}_{SRP}$.

4.1 Plan operations

4.1.1 Projection

For the sake of efficient execution of a query, the size of the *intermediate result* much be kept as small as possible. It is convenient to take a relational perspective on the size of an intermediate result, not least because the proposed plan execution will be built upon a relational system. That is, it is helpful to think of the size of the intermediate result in terms of *width* (i.e. size of a single tuple) and *height* (i.e. the number of tuples) of a table of intermediary results.

For regular path queries, the width can be kept very small. In fact, it can be kept constant at a value of two, storing only the end-points of an intermediary result. Such a pair is often referred to as a *subject-object* pair. Little can be done to reduce the height, since it is subject to the query and the data.

The following example will illustrate that, unfortunately, storing only subject-object pairs in our intermediary results is not feasible for RQMs.

Consider the REM $e = \text{likes} \cdot \downarrow x_0.\text{directedBy} \cdot \text{dealsWith} \cdot \text{produced}[x_0^=] \cdot \text{likedBy}$ from our running example. Here, we store the genre of a movie in the 0th register. Given this REM, such a data value can only come from (1) the target vertex of an edge labeled `likes` or (2) the source vertex of an edge labeled `directedBy`. Should we only store subject-object pairs, the assignment $\downarrow x_0$ will have to be evaluated together with edges labeled `likes` or `directedBy`, since if the subject-object pair corresponding to `likes` were joined with those of `directedBy`, the reference to the necessary vertex would be lost, since it is neither the subject not the object, but actually the join predicate.

The plan space is thus severely restricted should one only use subject-object pairs in the evaluation of RQMs. We will see examples later that further illustrate why plans outside of this restricted space might be preferable.

In order to obtain an unrestricted plan space, we employ *limited projection push-down*. Returning to our example, we could, instead of a subject-object pair, store a triple of vertices such that `likes` can be joined with `directedBy` before applying the assignment. Notice that once the assignment has been made, we can once again store a subject-object pair. Hence, the *width* of our intermediate results will be *flexible*, and deciding which vertices to keep will be part of planning an RQM.

Thus we arrive at the first operation that RQM plans must capture; *projection*. This might seem straight-forward since query planning in general involves planning projection operators, but the automata-based representations of regular queries we have seen thus far do not involve projection while only end-points of paths are stored.

4.1.2 Assignments and conditions

In order to compare data values against a register, such a register must first be assigned a value. Recall that the semantics of an assignment operation $\downarrow \overline{x}.e$ are defined as setting the value in the ith register equal to the first data value in a data path that satisfies e for all $x_i \in \overline{x}$. This works when an assignment is always performed together the first label in e, but it breaks when we introduce flexible intermediate results. Hence, for assignments, and subsequently also for conditions, there must be some additional bookkeeping to make sure the correct data values are used. To this end we introduce the notion of a *step* in a path.

Steps

When considering only subject-object pairs, we always deal with the end-points of an intermediate result. For a tuple consisting of three or more vertices this is not the case. After all, a triple could be used to store the result of a path consisting of two labels, in which case it holds the end-points and the vertices on which the two labels join. However, it could just as well be used to store the result of a path consisting of three labels, for which one of the intermediary vertices has been projected out, while the other has not. Hence, we will have to keep track of which part of the path it is actually storing. To this end, we will store m-tuples with $2 \leq m \leq n+2$ where n is the number of concatenations in the input REM. Such tuples will consists of pairs (s, v) where $s \in \mathbb{N}$ uniquely identifies a *step* in a path and v a vertex.
Consider again the path π = 0 `likes` 1 `directedBy` 4. A 3-tuple that stores this path is $t = ((0,0),(1,1),(2,4))$
Assignments and conditions must hence take into account a step to which they are to be applied. We will refer to such assignments and conditions as *wavefront assignments* and *wavefront conditions*, respectively. They are defined formally as:

Definition 4.1.1. (Wavefront assignment) An *wavefront assignment* is a tuple $a = \langle s, r \rangle$ where

- $0 \leq s \leq n+1$ is the step of a pair (s, v) such that *a depends on v*, and
- $0 \leq r \leq k$ is the register that *a provides* (i.e. assigns a value).

Let $e = \downarrow \overline{x}.a \cdot \downarrow \overline{x'}.b$ be an REM with $\overline{x} = (x_0)$ and $\overline{x'} = (x_1, x_2)$. This assignment will produce three wavefront assignments: $a_1 = \langle 0, 0 \rangle$, $a_2 = \langle 1, 1 \rangle$ and $a_2 = \langle 1, 2 \rangle$, since $\overline{x}$ depends on the vertex in step zero (i.e. the source vertex of a) and provides register zero, while $\overline{x'}$ depends on step one and provides registers one and two.

Definition 4.1.2. (Wavefront condition) A *wavefront condition* is a tuple $o = \langle s, c \rangle$ where

- $0 \leq s \leq n+1$ is the step of a pair (s, v) such that *o depends on v*, and
- c a condition.

Let $R_o \subseteq \{0, ..., k\}$ denote the set of registers that o depends on. That is, the set of registers that are referred to in c. Notice that $R_o = \emptyset$ for conditions that do not contain sub-expressions of the form $x_i^=$ or $x_i^{\neq}$. Also, consider a condition $c := x_0^= \wedge x_1^=$. It will produce a wavefront condition o with $R_o = \{0, 1\}$.

We will use the term *data join* to refer to assignments and conditions in general. This is because, in any realistic DBMS scenario, the data and topology of a data graph are going to be *decomposed*. Hence, assignments and conditions are essentially just joins against a data-structure capturing data, rather than topology.

4.1.3 Local and global data joins

A final observation to make before introducing a formalism that can represent query plans incorporating the ideas sketched above, is that data joins can be either *local* or *global*. That is, a join with data can be performed before, or after a join on topology.
Consider once more the running example where the genre of a movie that person u likes and person w directed is stored in a register. A choice must be made to either lookup and store the movies u likes, or lookup and store movies that w directed. Suppose we chose the first option. Without the need to store the genre in a register, that would be that. The only remaining course of action is to join the movies u likes with movies w directed on the predicate that they are the same movie. With the assignment however, there is a second choice to be made. Namely, do we first join on the edge labels and subsequently assign the register its value, or do we assign register values first, and then perform the join on the edge labels. In the first case, we say that the assignment is *global*, whereas in the second case it is *local*. The same principle applies to conditions. Hence, a plan must be able to represent a set of local- and a set of global data joins associated with the evaluation of an edge label.

4.2 k-Register Waveplans

Having provided a sketch of the peculiarities of planning RQMs, and introducing some of the necessary concepts such as projection, local- and global assignments and conditions, and steps, we can introduce a formalism that can represent plans for RQMs. We will extend WavePlans from Yakovets et al. [6] into something we call *k-register waveplans*.

4.2.1 Waveplans

WavePlans are introduced in [6] to provide a rich plan space for RPQs. A WavePlan consists of one or more automata, called *wavefronts*. WavePlans posses multiple features that produce a rich plan space for RPQs. We consider two of these features, namely *inverse transitions* and

transitions over views.

Inverse transitions

An inverse transition is a transition that expands an intermediate result by *prepending* tuples from the graph or cache rather than appending them.
Consider REM e = `likes` · `directedBy` · `dealsWith`, which is the first part of our running example. Let $I_1 = \{(0,1),(0,2),(0,3)\}$ be the set of tuples corresponding to the path π_1 = `likes`, $I_2 = \{(1,4),(2,4),(3,4)\}$ be the set of tuples corresponding to the path π_2 = `directedBy` and $I_3 = \{(4,5),(4,6),(4,7),(4,8)\}$ the set of tuples corresponding to the path π_3 = `dealsWith`. A regular transition might append I_3 to I_2, yielding $I_4 = \{(1,5),(1,6),...,(3,7),(3,8)\}$, which is the result corresponding to the path π_4 = `directedBy` · `dealsWith`. Conversely, an inverse transition might prepend I_1 to I_2, yielding $I_5 = \{(0,4)\}$ which corresponds to π_5 = `likes` · `directedBy`.
To denote inverse transitions, we add a dot to the label of a transition. For example, the label `dealsWith` · denotes a normal transition, whereas · `likes` denotes an inverse transition.

Transitions over views

A transition over a view expands an intermediate result from the *cache* rather than from the graph.
Consider again the sets I_1 and I_4 from the previous section. If both of these sets are cached during the execution of a plan, a transition over a view can expand I_1 by appending I_4. Similarly, for sets I_3 and I_5, a transition over a view can expand I_3 by prepending I_5. Notice that the set that is being appended- or prepended now represents a path with more than one label.
Let Q be the set of states in a WavePlan, where every state $q \in Q$ has an associated label l_q. Let $L = \{l_q \mid q \in Q\}$. Transitions over views can be represented by taking a label from L rather than from Σ. This denotes that the output of a state q with label l_q is used as the input for a transition labeled l_q. Figure 4.1 shows a WavePlan consisting of two wavefronts wf_0 and wf_1 for the expression `likes` · `directedBy` · `dealsWith`. Where wf_0 represents the view `likes` · `directedBy`, wf_1 prepends the result of this view in the transition from state 1 to 2.

With inverse transitions and transitions over views defined, we can introduce k-register wavefronts. Formally, the definition of k-register wavefront is as follows:

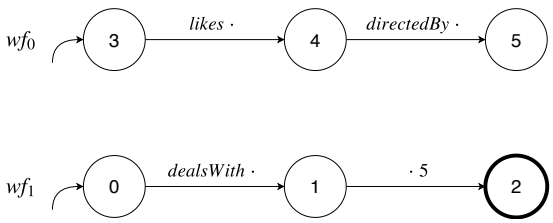


Figure 4.1: A WavePlan for `likes` · `directedBy` · `dealsWith`

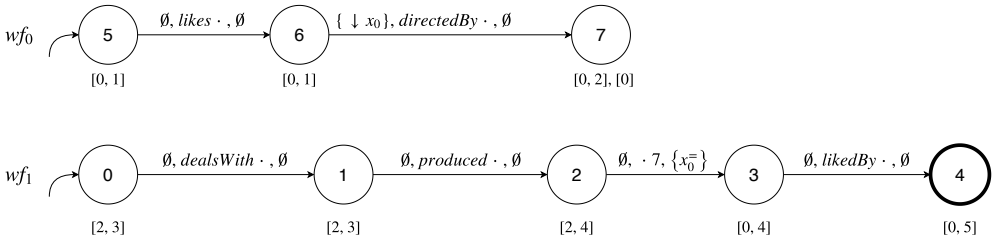


Figure 4.2: A k-register waveplan for `likes` · $\downarrow x_0$.`directedBy` · `dealsWith` · `produced`$[x_0^=]$ · `likedBy`

Definition 4.2.1. (k-Register Wavefront) Let Σ be a finite labeling alphabet, $\mathcal{D} \subset \mathbb{Z}$ a finite set of data values and Δ a finite set of data joins. A *k-register wavefront* is a tuple $\mathcal{A} = \langle Q, q_0, F, \delta, \tau_0 \rangle$ with

- Q a finite set of states where
  - every $q \in Q$ has a label l_q,
  - every $q \in Q$ has an associated set $P_q^v \subset \mathbb{N}$ called a *vertex projection*,
  - every $q \in Q$ has an associated set $P_q^r \subset \mathbb{N}$ called a *register projection*,
- $q_0 \in Q$ the starting state,
- $F \subseteq Q$ a set of accepting states,
- $\delta \subseteq Q \times \Delta \times \Sigma^{\cdot} \times \Delta \times Q$ a transition relation, and
- $\tau_0 \in \mathcal{C}_\perp^k$ the initial configuration of the registers.

where $\Sigma^{\cdot} = \bigcup_{a \in \Sigma \cup L} \{a\cdot, \cdot a\}$ is a shorthand for the extended set of labels containing all labels in Σ and a label for each state $q \in Q$.

Figure 4.2 shows a k-register waveplan for the whole query in our running example. The k-register wavefront wf_0 computes the result of `likes` $\cdot \downarrow x_0$.`directedBy`. Notice that the assignment $\downarrow x_0$ is local on the transition from 6 to 7. Furthermore, the result is a pair of vertices for steps 0 and 2, as well as the 0th register, as indicated by the vertex- and register projections $[0, 2]$ and $[0]$, respectively. Empty register projections have been omitted.
The behaviour of wf_1 is relatively straight-forward, except perhaps for the transition from state 2 to state 3. Here, the result of wf_0 is prepended to that of `dealsWith` $\cdot$ `produced`, after which the global condition $x_0^=$ is applied.

4.3 Plan Space

For any given RQM there are multiple k-register waveplans that evaluate it. We call the set of such waveplans the *plan space*.
Consider the *standard plan space* $\mathcal{P}_{SWP}$ for RPQs as presented in [6]. Notice that any such RPQ is also an RQM, since the grammar for REMs covers that of regular expressions as present in [6]. Let e be an arbitrary REM. We will refer to r as the *regular part of* e to mean the RPQ obtained by removing any assignments and conditions from e.
A naive way of obtaining a plan for an RQM with expression e is to take the plans in $\mathcal{P}_{SWP}$ for its regular part r and add on the necessary data joins in a prescribed manner. This could be done by adding *full* projections to each state. That is, to never project out any of the steps or registers. Additionally, all data joins would be added to each transition reaching a final state. Thus, we obtain plans which keep full intermediate results around until the end of evaluation, at which point the assignments and conditions are performed. Such a plan could subsequently be optimised by *pushing* data joins back to the earliest transition that covers their dependencies, and then to reduce the projections as much as possible. In this way, every plan in $\mathcal{P}_{SWP}$ will lead to exactly one plan in the naive plan space for RQMs.
A more comprehensive plan space considers, for each plan in $\mathcal{P}_{SWP}$, all possible legal placements of the data joins. For each of these plans, projections are minimised. We will refer to this plan space as the *standard regular query with memory plan space*, or $\mathcal{P}_{SRP}$. Notice that $\mathcal{P}_{SRP}$ is considerably richer than the naive plan space.
The need for this richer plan space comes from the observation that the planning of an RQM's regular part and its data joins are *orthogonal*. That is, a plan $p \in \mathcal{P}_{SWP}$ that is optimal with respect to the regular part r of some REM e in $Q : x \xrightarrow{e} y$ need not be the basis of a plan for Q

that is optimal with respect to both the regular part- and the data joins of e.

Plan space complexity

Ideally, the complexity of the plan space $\mathcal{P}_{SRP}$ for an arbitrary RQM $Q := x \xrightarrow{e} y$ would be defined in terms of two parameters; the size $|r|$ of the regular part of e and the number of data joins n. However, we will show with an example that such a general definition is not possible. Consider the REMs $e = a \cdot b[5^{=}]$ and $e' = a[5^{=}] \cdot b$. For both REMs, the regular expression obtained by removing all data joins is $r = a \cdot b$. Hence, the size of the regular part and the number of data joins are the same for e and e'. However, the number of plans for e and e' is not the same. For e, the condition $5^{=}$ can be evaluated locally with b or globally with $a \cdot b$. For e', the condition can be evaluated locally with a, globally with $a \cdot b$ *but also locally with* b, since the target vertex of a on which $5^{=}$ is evaluated is also the source vertex of b.
Hence, the size of the plan space for an REM e is inherently dependent on the structure of e, not just the size of its regular part and the number of data joins involved. In order to at least obtain an upper-bound on the complexity of the plan space, we can assume that any data join can be evaluated at any point in the plan. Clearly, this set of plans properly subsumes $\mathcal{P}_{SRP}$ since it includes all legal placements of data joins, as well as many illegal ones. From WaveGuide we know that $\mathcal{P}(r) = \Theta(c^{|r|})$ for some constant c is the number of plans for a regular expression r. If a data join can be evaluated at any point in a plan, than the number of plans $\mathcal{P}(r, n)$ for regular expression r with n data joins is larger than $\mathcal{P}(r)$ by a factor $f(t, n)$ that is the number of ways to distribute n data joins over t transitions. Since t is directly proportional to $|r|$ we have that $f(t, n) = O(|r|^n)$ and so also that $\mathcal{P}(r, n) = O(|r|^n c^{|r|})$.

5 Plan Enumeration

This chapter covers the enumeration procedure of the proposed plan space and its complexity. Additionally, it covers an outline of cost-based enumeration. We will see that while dynamic programming can still be used to develop a cost-based enumerator and reduce the complexity of the procedure, it is still exponential in the total number of assignments and conditions, at $O(|e|^2|r|^6 8^n)$.

We have outlined a plan space $\mathcal{P}_{SRP}$ for regular queries with memory. This plan space has a very high complexity of $O(|r|^n c^{|r|})$ where $|r|$ is the size of the regular part of an RQM, n is the number of data joins in that RQM and c is a constant. The process of constructing plans in a plan space is known as *enumeration*.

We distinguish two kinds of enumeration, namely (1) *exhaustive* enumeration, and (2) *cost-based* enumeration. Exhaustive enumeration of a plan space means constructing and storing all plans within that space. Clearly, this is an expensive procedure given the typical complexity of plan spaces.

Cost-based enumeration drastically reduces the complexity of enumerating a plan space by employing *dynamic programming*. Let r_1 and r_2 be sub-expressions of a regular expression r such that r_1 and r_2 can be combined to form r. The *optimal substructure* property of plans for r_1 and r_2 means that combining optimal plans for r_1 and r_2 yields an optimal plan for r. Notice that a plan is optimal only with respect to some *estimated* cost based on a *cost model*.

Both kinds of enumeration are *bottom-up*, meaning that plans are computed for increasingly larger sub-expressions of the input REM.

While we will give a brief outline of a cost model of RQMs on data graphs, argue that plans for RQMs posses optimal substructure and analyse the complexity of cost-based enumeration for RQMs, the effectiveness of cost-based enumeration with the given cost model will not be

evaluated experimentally. Recall that we aim to experimentally justify choosing $\mathcal{P}_{SRP}$ as a plan space for RQMs. Hence, we must examine the performance gains that can be achieved by planning for the regular part- and data joins of an RQM, both. This means comparing the performance of all plans within $\mathcal{P}_{SRP}$ and thus means applying exhaustive enumeration.

5.1 Rule-based enumeration

We will extend WaveGuide's [6] enumeration procedure to cover $\mathcal{P}_{SRP}$. The procedure will be *rule-based*, and construct plans for increasingly larger *sub-expressions* of an input REM e.
That is, starting with sub-expressions of size one (i.e. single labels), particular rules will construct plans for these sub-expressions. Subsequently, these plans are used in the next iteration of the enumeration, when plans are constructed for sub-expressions of size two. Thus plans are generated in a *bottom-up* manner.
Recall the distinction between the regular part of an REM and its data joins. There will be a set of rules RER that construct plans for sub-expressions where the size of the regular part increases. That is, given sub-expressions e_1 and e_2 with regular parts r_1 and r_2 respectively, the sum of the sizes of r_1 and r_2 is larger than the size of the regular part for which plans were constructed in the previous iteration.
Conversely, the set of rules DER contains rules that construct plans for sub-expressions where the number of data joins increases.

5.1.1 Sub-expressions of REMs

Sub-expressions of an REM e are subdivided in terms of *size*. The size $|e|$ of an REM e can be defined recursively as:

- 1 if $e = \epsilon$ or $e = a$ for some $a \in \Sigma^{\cdot}$,
- $|e_1| + |e_2|$ if $e = e_1 + e_2$ or $e = e_1 \cdot e_2$,
- $|e_1| + 1$ if $e = e_1^+$,
- $|e_1| + |c|$ if $e = e_1[c]$, and
- $|e_1| + |\overline{x}|$ if $e = \downarrow \overline{x}.e_1$.

where $|\overline{x}|$ is simply the size of the tuple $\overline{x}$. The definition of the size $|c|$ of a condition c is not so straight-forward. To see why, consider the conditions $c_1 = x_0^= \wedge 5^{\neq} \wedge 6^{\neq}$ and $c_2 = x_0^= \vee$

Size	Sub-expressions
6	$\downarrow \overline{x}.a \cdot b \cdot c[(x_0^= \vee x_1^= \vee 5^{\neq}) \wedge (x_0^= \vee x_1^= \vee 6^{\neq})]$
5	$\downarrow \overline{x}.a \cdot b \cdot c$
4	$\downarrow x_0.a \cdot b \cdot c \quad \downarrow x_1.a \cdot b \cdot c \quad \downarrow \overline{x}.a \cdot b$
3	$a \cdot b \cdot c \quad \downarrow \overline{x}.a \quad \downarrow x_0.a \cdot b \quad \downarrow x_1.a \cdot b$
2	$a \cdot b \quad b \cdot c \quad \downarrow x_0.a \quad \downarrow x_1.a$
1	$a \quad b \quad c$

Table 5.1: Sub-expressions and their sizes for $\downarrow \overline{x}.a \cdot b \cdot c[x_0^= \vee x_1^= \vee (5^{\neq} \vee 6^{\neq})]$ with $\overline{x} = (x_0, x_1)$

$5^= \vee 6^=$. The three literals in c_1 can be evaluated separately, since c_1 forms a conjunction. The literals naturally all depend on the same vertex, but $x_0^=$ also requires the 0th register. Since more selective conditions always improve query execution performance, and conjunctions of conditions are at least as selective as either of the conjuncts, it is optimal to combine literals with the same dependencies. Hence, $|c_1| = 2$. Conversely, the three literals in c_2 cannot be evaluated separately, due to the fact that they are in disjunction with one another. Hence, c_2 must be treated as a whole and $|c_2| = 1$. Let c' be a condition equivalent to c such that c' is in *conjunctive normal form* and all literals with the same dependencies in c' are nested together. Then $|c| = |c'|$ can be defined recursively as:

- 1 if $c' \in \{x_i^=, x_i^{\neq}, z^=, z^{\neq}\}$,
- $\max(|c_1|, |c_2|)$ if $c' = c_1 \wedge c_2$ and the dependencies of c_1 and c_2 are the same,
- $|c_1| + |c_2|$ is $c' = c_1 \wedge c_2$ and the dependencies of c_1 and c_2 are not the same,
- 1 if $c' = c_1 \vee c_2$, and
- $|c_1|$ if $c' = \neg c_1$.

Consider the REM $e = \downarrow \overline{x}.a \cdot b \cdot c[x_0^= \vee x_1^= \vee (5^{\neq} \vee 6^{\neq})]$ with $\overline{x} = (x_0, x_1)$. The conjunctive normal form of the conditions is $(x_0^= \vee x_1^= \vee 5^{\neq}) \wedge (x_0^= \vee x_1^= \vee 6^{\neq})$. Notice that the size of the conjunctive normal form condition is one. Table 5.1 shows the sub-expressions of sizes 1 through $|e| = 6$ of e.

5.1.2 Projection graph

Before discussing the enumeration rules, it is helpful to take a step back and consider how we might determine the projections for a state in a k-register wavefront. A naive approach would keep the full projection in each state, and minimize them once the plan has been completed. While this approach works for exhaustive enumeration, where all plans are generated and stored, it will not work for cost-based enumeration. The cost estimation involved relies on *cardinality estimates* for the input relations and output relation of a join. These estimates are influenced by the projections. Hence, while constructing plans in a cost-based fashion, we must be able to decide at every iteration what the projections will be *in the final plan*.
To this end, we introduce the *projection graph*, which is a graph that captures the dependencies between the various literals in an REM e and allows us to decide for any sub-expression of e what the projection should be.

Definition 5.1.1. (Projection Graph) A *projection graph* over regular expression with memory e is a tuple $\mathcal{P}_e = \langle V_1, V_2, V_3, V_4, V_5, E_1, E_2, E_3, E_4 \rangle$ with

- V_1 a set of vertices corresponding to the labels in e,
- V_2 a set of vertices corresponding to the steps storing vertices in e,
- V_3 a set of vertices corresponding to assignments in e,
- V_4 a set of vertices corresponding to the steps storing register values in e,
- V_5 a set of vertices corresponding to conditions in e,
- $E_i \subseteq V_i \times V_{i+1}$ sets of edges for $0 < i < 3$, and
- $E_4 \subseteq (V_2 \cup V_4) \times V_5$ the final set of edges

The idea behind the projection graph is that there is a vertex for every literal in e, where a literal is either a label (V_1), assignment (V_3) or condition (V_5), as well as a vertex for every step that a literal *provides* (V_2) or *depends on* (V_4). The edges capture the relations between the literals, and by performing some operations on the sets of vertices and edges we will be able to compute the projection for sub-expressions.

As an example, consider the REM $e = a \cdot \downarrow x_0 . b \cdot c[5^= \wedge x_0^=] \cdot d$. The projection graph for this REM is given in Figure 5.1. We can see that there is one vertex for each of the label literals a,

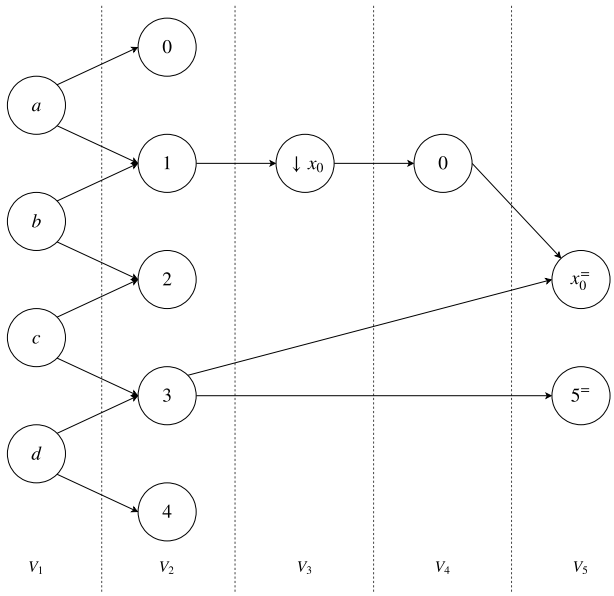


Figure 5.1: Projection graph for $e = a \cdot \downarrow x_0 . b \cdot c[5^= \wedge x_0^=] \cdot d$.

b, c and d, as well as for the assignment- and condition literals $\downarrow x_0$, $5^=$ and $x_0^=$. Additionally, there are step vertices for steps 0, 1, 2, 3 and 4 which will store vertices, and step 0 which will store register values. Observe that the resulting graph is always a 5-partite graph, hence the structure of the tuple $\mathcal{P}_e$.

To obtain the projection for a sub-expression e' of e given $\mathcal{P}_e$ we apply the procedure detailed in Algorithm 1. The sets I and J are the subsets of V_1 and V_3 that are reachable from literals in e', respectively. The sets P and R are the vertex- and register projections, respectively. The vertex projection P always contains the minimum- and maximum elements in I, which are the end-points of any intermediate result. Loop [6-9] computes the subsets T of V_3 reachable from vertices in $I \backslash P$. If this subset is not empty or contained within L, it means that the step v will still be required in a subsequent plan. Hence, v is added to P. Similarly, loop [10-13] computes the subsets T of V_5 reachable from J, and adds the required steps to R.

5.1.3 Regular enumeration rules

The set of regular enumeration rules *RER* is schematically represented in Figure 5.2. This set of rules is very similar to the set of rules *ER* from [6]. In fact, only the *projections* added to new states created by a rule are introduced. We write P to denote the vertex projection, and R to denote the regular projection.

rule		k-register waveplan	precondition			
id	*description*		e_1	e_2	*op*	*seed*
AA	atom append	p: U P, R $e_1 \cdot$ P, R	$\lvert e_1 \rvert = 1$	null	null	null
AP	atom prepend	p: U P, R $\cdot e_2$ P, R	$\lvert e_1 \rvert = 1$	null	null	null

(a) Enumeration rules for labels.

rule		k-register waveplan	precondition			
id	*description*		e_1	e_2	*op*	*seed*
CC	concat compound	p: p_1 d_1 e_1 $W_{e_2} \cdot$ P, R	$\lvert e_1 \rvert > 1$	$\lvert e_2 \rvert > 1$ $d_2 = U$	$\cdot$	null
CCF	concat compound flip	p: p_1 d_2 e_2 $\cdot W_{e_1}$ P, R	$\lvert e_1 \rvert > 1$ $d_1 = U$	$\lvert e_2 \rvert > 1$	$\cdot$	null
CP	concat pipe	p: p_1 d_1 e_1 $e_2 \cdot$ P, R	$\lvert e_1 \rvert > 0$	$\lvert e_2 \rvert = 1$	$\cdot$	null
CPF	concat pipe flip	p: p_1 d_2 e_2 $\cdot e_1$ P, R	$\lvert e_1 \rvert = 1$	$\lvert e_2 \rvert > 0$	$\cdot$	null
DP	direct pipeline	p: p_1 d_1 e_1 ε e_2 p_2	$\lvert e_1 \rvert > 1$	$d_2 = e_1$	$\cdot$	null
IP	inverse pipeline	p: p_1 d_2 e_1 ε e_2 p_2	$d_1 = e_2$	$\lvert e_2 \rvert > 1$	$\cdot$	null

(b) Enumeration rules for concatenations.

rule		k-register waveplan	precondition			
id	*description*		e_1	e_2	*op*	*seed*
KP	kleene plus	p: d P_{e_1}, R_{e_1} ε p_1 e_1 ε ε P_{e_1}, R_{e_1}	$d_1 = d \cdot (e_1)^+$ $d_1 = (e_1)^+ \cdot d$	null	$+$	null

(c) Enumeration rules for Kleene closures.

rule		k-register waveplan	precondition			
id	*description*		e_1	e_2	*op*	*seed*
ASDP	absorb seed direct pipe	p: d P, R $e_1 \cdot$ P, R	$\lvert e_1 \rvert = 1$	null	null	d
ASIP	absorb seed inverse pipe	p: d P, R $\cdot e_2$ P, R	null	$\lvert e_2 \rvert = 1$	null	d
ASDC	absorb seed direct compound	p: d $P_{W_{e_1}}, R_{W_{e_1}}$ $W_{e_1} \cdot$ $P_{W_{e_1}}, R_{W_{e_1}}$	$\lvert e_1 \rvert > 1$ $d_1 = U$	null	null	d
ASIC	absorb seed inverse compound	p: d $P_{W_{e_2}}, R_{W_{e_2}}$ $W_{e_2} \cdot$ $P_{W_{e_2}}, R_{W_{e_2}}$	null	$\lvert e_2 \rvert > 1$ $d_2 = U$	null	d

(d) Enumeration rules for seed passing.

Figure 5.2: Enumeration rules for the regular part of expressions.

Algorithm 1 Projection($e, \mathcal{P}_e$)

1: Let L be the set of literals in e
2: $I = \bigcup_{l \in L \cap V_1} \{v \in V_2 \mid (l, v) \in E_1\}$
3: $J = \bigcup_{l \in L \cap V_3} \{v \in V_4 \mid (l, v) \in E_3\}$
4: $P = \{\min(I), \max(I)\}$
5: $R = \emptyset$
6: **for** $v \in I \setminus P$ **do**
7: $\quad T = \{t \in (V_3 \cup V_5) \mid (v, t) \in (E_2 \cup E_4)\}$
8: $\quad$ **if** $T \not\subset L \wedge T \neq \emptyset$ **then**
9: $\quad\quad P = P \cup \{v\}$
10: **for** $v \in J$ **do**
11: $\quad T = \{t \in V_5 \mid (v, t) \in E_4\}$
12: $\quad$ **if** $T \not\subset L \wedge T \neq \emptyset$ **then**
13: $\quad\quad R = R \cup \{v\}$
14: **return** (P, R)

Rule preconditions

Each rule in *RER* has a number of *preconditions* on the input sub-expressions e_1 and e_2, operator *op* and seed *seed*. The role of a seed and *seed passing* will be discussed subsequently. Suffice it to say that rules often require a particular operator (like $\cdot$ or $^+$), particular sizes $|e_1|$ and $|e_2|$ of sub-expressions e_1 and e_2 or a particular seed d.
A rule $r \in RER$ is applied if and only if its preconditions are satisfied by the input parameters.

Seed passing

Seed passing is a mechanism introduced in [6] that makes sure that plans involving closures are generated correctly.
Consider the regular expression $(ab)^+$. Applying the rules for labels, concatenation and Kleene closure naively might result in the WavePlan in Figure 5.3. The problem with this WavePlan is that it is not *strict*. That is, the transitions forming the closure are not all appending nor all prepending. Performing the closure once should yield a result for the expression *abab*, but this plan will actually produce a result for *aabb* due to first prepending *a* and then appending *b* to the intermediate result for *ab*.

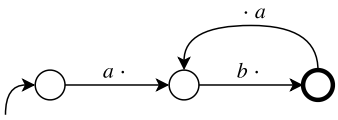


Figure 5.3: A naive WavePlan for $(ab)^+$.

For a more detailed discussion on strictness and seed passing we refer to [6]. Since seed passing and strictness are not of specific importance with regard to planning with respect to data, suffice it to say that seed passing makes sure that plans for closures are generated strict and so makes sure that the enumeration produces only plans corresponding to the input expression.

Label rules

The subset of *RER* that deals with a single input sub-expression consisting of a label is composed of *atom append* (AA) and *atom prepend* (AP).
Both rules introduce two new states; a starting state and an accepting state. Furthermore, they create a single transition from the starting to the accepting state that has the input label and either expands the result by appending (AA) or prepending (AP). Finally, the projections P and R are computed by the `Projection` routine with input e_1.

Concatenation rules

The subset of *RER* that deals with concatenations of sub-expressions is composed of *concat compound* (CC), *concat compound flip* (CCF), *concat pipe* (CP), *concat pipe flip* (CPF), *direct pipeline* (DP) and *inverse pipeline* (IP).
The first two rules, CC and CCF, deal with the case where both input sub-expressions e_1 and e_2 are larger than one, and so plans p_1 and p_2 will have already been constructed in a previous iteration. A plan p that combines e_1 and e_2 is constructed by adding a new accepting state to p_1 and transitions from the old accepting states in p_1 to the new accepting state. These transitions are transitions over a view, where the view is implemented by the wavefront W_{s_1} from p_1, or W_{s_2} from p_2 for CCF and CC, respectively.
The third and fourth rules, CP and CPF, perform a similar procedure, but here one of the sub-expressions e_1 and e_2 is of size one. Hence, normal transitions with label e_1 or e_2 suffice instead of transitions over views.
The last two rules, DP and IP, connect existing plans p_1 and p_2 for sub-expressions e_1 and e_2

with an ϵ-transition, provided that there is a match between the seed of one sub-expression and the other sub-expression itself, which means that the output of one plan is valid input to the next.

Finally, there is the issue of projection. Since the last two rules do not introduce new states, no new projections have to be computed. In the other four cases, projections P and R are computed by the `Projection` routine with as input the *super-expression* of e_1 and e_2. That is, the expression obtained from combining e_1 and e_2, which is the expression we are building plans for in the current iteration of the enumeration.

Kleene closure rules

There is one rule in *RER* that deals with Kleene closures, name Kleene plus (KP). As the name implies we consider only the closure over at least one instantiation of a regular expression. To allow for zero or more instantiation (i.e. Kleene star), an ϵ label can be used in the input REM. This rule closes an existing plan p_1 for e_1 by adding two states and three ϵ-transitions.

The projections P_{e_1} and R_{e_1} are the projections from an accepting state in p_1. Notice that an arbitrary accepting state suffices, since all accepting states must have the same projections, or a plan would lead to inconsistent results. Additionally, notice that the new starting- and accepting state have the same projections.

Seed passing rules

The final subset of rules in *RER* consists of *absorb seed direct pipe* (ASDP), *absorb seed inverse pipe* (ASIP), *absorb seed direct compound* (ASDC) and *absorb seed inverse compound* (ASIC). Once more, for a detailed discussion of the role of seed passing we refer back to [6]. With respect to planning for data, it suffices to say that all four rules introduce two new states; a starting- and an accepting state. For ASDP and ASIP the projections P and R are computed by the `Projection` routine given e_1. Conversely, the projections $P_{W_{e_1}}$, $R_{W_{e_1}}$, $P_{W_{e_2}}$ and $R_{W_{e_2}}$ are obtained from accepting states in W_{e_1} and W_{e_2}, respectively, for ASDC and ASIC, respectively.

5.1.4 Data enumeration rules

The set of data enumeration rules *DER* is schematically represented in Figure 5.4. It consists of rules *add local data join* (ALDJ) and *add global data join* (AGDJ). These rules have an additional input parameter; a data join δ. For ALDJ, δ is added to the set of local data joins D_l for all

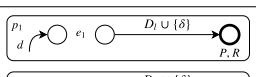
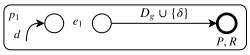
rule		k-register waveplan	precondition				
id	*description*		e_1	e_2	*op*	*seed*	*data join*
ALDJ	add local data join	p: 	$\|e_1\| > 0$	null	null	null	δ
AGDJ	add global data join	p: 	$\|e_1\| > 1$	null	null	null	δ

Figure 5.4: Enumeration rules for adding data joins.

transitions reaching an accepting state in plan p_1 for sub-expression e_1. Alternatively, for AGDJ, δ is added to the set of global data joins D_g. Notice that ALDJ requires that $|e_1| > 0$ since a data join must be added to an expression containing at least one label. Similarly, AGDJ requires that $|e_1| > 1$ since a global data join only makes sense for an expression consisting of at least two labels.

5.2 Cost-based enumeration

Having defined the rules and sub-expressions on which the enumeration procedure is based, we can give an outline of a cost model for RQMs over data graphs, argue that plans for RQMs posses optimal substructure and analyse the complexity of cost-based enumeration.

5.2.1 Cost estimation

With query plans defined as k-register waveplans, we can briefly discussed cost estimation. Estimating the cost of a plan is essential if we are to build an enumerator that is capable of finding good plans. Indeed, the only guarantee of the quality of our enumeration will be that it returns a plan that is optimal *with respect to the cost estimate*.

Cost estimation on the context of RQMs is a topic that could take up an entire thesis, however. Currently, our goals are to show the challenges in planning RQMs and showing the *relative performance* of plans within a certain plan space. That is, optimal performance of enumeration nor execution in an absolute sense is not part of our goals. Hence, we will be content with a minimal extension of the cost estimation as presented in [6].

Duplication due to data

The one necessary extension of cost estimation from [6] to cover RQMs has to do with *cardinality estimation*. Returning to our running example, suppose that we are given a plan that joins

edges labeled `likes` and `directedBy`, estimating the cost of such a plan depends on the size, or cardinality, of the result of `likes` $\cdot$ `directedBy`. Consider again the data graph in Figure 1.1b. The cardinality of `likes` $\cdot$ `directedBy` is one, since $(0,4)$ is the only distinct pair in the result. The fact that there are three ways of reaching 4 from 0 does not matter for the cardinality. However, consider the cardinality of `likes` $\cdot \downarrow x_0$.`directedBy`. Now, the data values in vertices $1, 2$ and 3 play a distinctive role. That is, while $(0,4)$ is still the only distinct *pair of vertices* in the result, the value in the 0th register can be either 1 or 2. Thus, the cardinality of `likes` $\cdot \downarrow x_0$.`directedBy` is two, instead of one. To capture this discrepancy, we introduce a coefficient to the estimate called a *duplication factor*. A duplication factor $d_{a \cdot b}$ is defined as

$$d_{a \cdot b} = \frac{n}{\max(n-m, 1)} \tag{5.1}$$

where n is the number of distinct vertices that have an incoming edge labeled a and an outgoing edge labeled b, and m is the number of distinct data values among those same vertices.
Multiplying the estimate for `likes` $\cdot$ `directedBy` by $d_{\texttt{likes} \cdot \texttt{directedBy}}$ will give us an estimate for the cardinality of `likes` $\cdot \downarrow x_0$.`directedBy`. This will allow us to build a cost-based enumerator that is at least *aware* of data joins, even though its estimates will not be particularly accurate. It is the topic of future work to flesh out cost estimation for RQMs.

5.2.2 Optimal substructure

To efficiently enumerate a plan space that's exponential in the size of the regular expression, WaveGuide introduces a cost-based approach using dynamic programming. The key observation underpinning this approach is that the enumeration problem has the *optimal sub-structure* property. That is, a plan for $r = a \cdot b$ that is not optimal cannot be the basis for an optimal plan for $r' = a \cdot b \cdot c$. In other words, by storing only one optimal plan for each sub-expression of an input expression r and combining these plans in a bottom-up fashion to produce plans for larger sub-expression, the complexity of the enumeration procedure can be reduced, while still guaranteeing that the final plan is optimal.

Let P be a *problem* to which the solution is the generation of an *optimal* plan (with respect to cost estimation) for a given regular expression with memory e. Then, optimal solutions to *sub-problems* S_i are the lowest cost plans for sub-expressions s_i of e. Since there are many different ways of combining sub-expressions s_i to obtain e, we must show two things to prove optimal substructure, namely:

1. a construction of a solution of P from optimal solutions for S_i is optimal, and
2. during enumeration, all possible combinations of sub-problems S_i resulting in P are explored.

Claim (1) is shown in [6] by structural induction on a fixed parse tree of e, and split into cases based on the operator in the parse tree. To extend this argument, we must consider the assignment- and condition operators.
Suppose that $e = \downarrow \overline{x}.e_1$. Let p_1 denote the optimal plan for e_1. The optimality of a solution for e is subject only to the cost of performing the assignment to the result of e_1. Since p_1 is an optimal plan for e_1, applying the assignment to its result cannot be more expensive than applying it to the result of any plan p for e_1 that is sub-optimal. Thus, a plan that applies the assignment $\overline{x}$ to the result of p_1 is optimal for e.
Suppose that $e = e_1[c]$. Let p_1 denote the optimal plan for e_1. Again, the optimality of a solution for e is subject only to the cost of evaluating c over the result of e_1. Therefore, an optimal plan for e can be constructed from p_1.

Combining the argument from [6] that we consider all combinations s_1 and s_2 for e such that $|s_1| + |s_2| = |e|$, with the observation that the ALDJ and AGDJ rules consider all plans for sub-expression s_1 of e such that s_1 and e have the same regular part r and s_1 has $n-1$ data joins whereas e has n, results in the conclusion that we do indeed consider all combinations of problems S_i resulting in P.

5.2.3 Complexity of cost-based enumeration

The key difference between the complexity of WaveGuide's enumeration procedure and the one proposed here, lies in the number of sub-expressions for which plans are generated. In case of regular expressions, this number is polynomial in the size of the expression. Figure 5.5 shows the sub-expressions for regular expression $r = abcdef$. Since the number of sub-expressions at the bottom of the pyramid is $|r| = 6$ and every other level has one fewer sub-expression than the level below, we can write the number of sub-expressions of a regular expression r of size $|r|$ as:

$$\sum_{i=0}^{|r|} |r| - i = \frac{|r|^2 + |r|}{2} \tag{5.2}$$

For regular expressions with memory, there is an additional number of sub-expressions, namely those formed by adding a data join. Here, we run into the same difficulty as we did when

analyzing the plan space complexity; the number of sub-expressions of an REM is dependent on its structure. We can make two assumptions that will allow us to derive an upper-bound on the number of sub-expressions of an REM in terms of the size $|r|$ of its regular part and the number of data joins n in it.

First, assume that the data join can be evaluated together with two labels. That is, assume that none of the data joins are assignments on the first vertex, or conditions on the last vertex of a path.

Second, assume that the number of labels before the two labels a data join can be evaluated with is the same as the number of labels after.

In case of $r = abcdef$ this means we assume any data join can be evaluated together with c, d or any sub-expression containing c or d. This subset consists of those sub-expressions inside the pentagon in Figure 5.5.

To see that these assumptions lead to an upper-bound, consider what happens if we should change the labels with which a data join can be evaluated. For example, consider a data join that can be evaluated with b or c. This change would *shift* the pentagon to the left, enabling b and ab as sub-expressions with which to evaluate our data join, but disabling d, de and def.

Choosing the middle two labels maximizes the size of the pentagon, and thus the number of sub-expressions with which a data join can be evaluated. From equation 5.2 we can derive that the number of sub-expressions outside our maximized pentagon is:

$$2\left(\frac{\left(\frac{|r|-2}{2}\right)^2 + \left(\frac{|r|-2}{2}\right)}{2}\right) = 2\left(\frac{|r|^2 - 2|r|}{8}\right) \tag{5.3}$$

And so the difference, which is the number of sub-expression with which a data join can be evaluated is:

$$\frac{|r|^2 + |r|}{2} - 2\left(\frac{|r|^2 - 2|r|}{8}\right) = \frac{|r|^2 + 4|r|}{4} \tag{5.4}$$

Each of these sub-expressions will spawn an additional:

$$\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1 \tag{5.5}$$

sub-expressions, since there are $2^n - 1$ ways to add 1 up to n data joins to a sub-expression that has none. Combining equations 5.2, 5.4 and 5.5 yields an expression for the maximal number of sub-expressions of any REM with regular part r and n data joins:

$$\frac{|r|^2 + |r|}{2} + \frac{|r|^2 + 4|r|}{4}\left(2^n - 1\right) \tag{5.6}$$

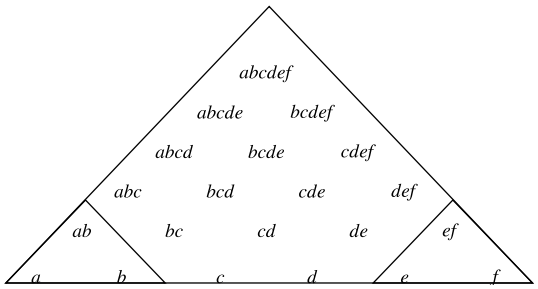


Figure 5.5: Sub-expressions of regular expression $r = abcdef$.

Asymptotically, this means that there are $O(|r|^2 2^n)$ sub-expressions for an expression e with regular part r and n data joins. This increased number of sub-expression will drastically impact the complexity of the enumeration procedure. Algorithms 2 and 1 provide pseudo-code for the routine that generates plans with data joins and the overall enumeration procedure, respectively.

The sub-routines `seedConst`, `seedKleene`, `getPlans` and `memoizeMin` are not changed from [6], except for the minor detail that set of plans that `getPlans` iterates over is now called RER instead of ER. Because of the increase in the number of sub-expressions from $O(|r|^2)$ to $O(|r|^2 2^n)$ the complexities of `seedKleene` and `memoizeMin` change to $O(|r|2^n)$ and $O(|r|^2 2^n)$, respectively. The complexities for the other two routines are unchanged.

The complexity of the new sub-routine `getPlansWithData` is $O(|r|^2 n)$ since there are $O(|r|)$ seeds, and the rules in DER must consider all sub-expressions with the same regular part r and $n-1$ data joins, of which there are $O(n)$.

To analyse the complexity of the overall enumeration procedure, we will have to investigate the loops L[1-9], L[10-30], L[11-30], L[13-30], L[14-20] and L[25-30]. Starting with the inner-most loops, the complexity of L[14-20] is $O(|r|^4 2^{2n})$ since there are $O(|r|^2 2^n)$ sub-expressions s_2 and `memoizeMin` is the most expensive sub-routine within the loop. The same bound and argument hold for L[25-30]. Since loops L[14-20] and L[25-30] are equally expensive, and lines 23 and 24 are less expensive, the complexity of L[13-30] becomes $O(|r|^6 2^{3n})$. Hence, loops L[11-30] and L[10-13] will have complexities of $O(|e||r|^6 2^{3n})$ and $O(|e|^2|r|^6 2^{3n})$, respectively. The bound on L[1-9] follows from the fact that there are $O(|r|)$ expressions s of size one, since data joins play no role here, and is therefore $O(|r|^3 2^n)$. Using the fact that $|e| \geq |r|$ yields an overall complexity of $O(|e|^2|r|^6 8^n)$.

Algorithm 2 getPlansWithData(e)

```
1:  Plans ps
2:  for each d ∈ getSeeds(e) do
3:      for each rule ∈ DER do
4:          Let D be the set of data joins in e
5:          for each δ ∈ D do
6:              if rule.precondition(e, null, null, null, δ) then
7:                  Let e′ be the REM with the same regular part as e
                    and set of data joins D\{δ}
8:                  p1 ← bestPlan(e′, d)
9:                  Plan p ← rule.genPlan(p1, null, null, δ)
10:                 ps.add(e)
11: return ps
```

Algorithm 3 WGEnum($Q = (x, e, y)$)

```
1:  for all s ⊆ e : |s| = 1 do
2:      seedConst(s, Q)
3:      seedKleene(s, Q)
4:      if s = a then
5:          P ← getPlans(a, null, null, null)
6:          memoizeMin(P, a_i)
7:      if s = a^+ then
8:          P ← getPlans(a, null, +, null)
9:          memoizeMin(P, a^+)
10: for 1 < l ≤ |e| do
11:     for 1 ≤ l_1 < l do
12:         l_2 = l − l_1
13:         for each s_1 ⊆ e : |s_1| = l_1 do
14:             for each s_2 ⊆ e : |s_2| = l_2 do
15:                 if s_1 · s_2 ⊆ e then
16:                     P ← getPlans(s_1, s_2, ·, null)
17:                     memoizeMin(P, s_1 · s_2)
18:                 if s_1 + s_2 ⊆ e then
19:                     P ← getPlans(s_1, s_2, +, null)
20:                     memoizeMin(P, s_1 + s_2)
21:             Let D be the set of data joins in s_1
22:             if D ≠ ∅ then
23:                 P ← getPlansWithData(s_1)
24:                 memoizeMin(P, s_1)
25:             for each s ⊆ e : |s| = l do
26:                 seedConst(s, Q)
27:                 seedKleene(s, Q)
28:                 if s = s_1^+ ⊆ e then
29:                     P ← getPlans(s_1, null, +, null)
30:                     memoizeMin(P, s_1^+)
31: return bestPlan(e, Q)
```

6 Plan Execution

With plans in $\mathcal{P}_{SRP}$ defined as k-register waveplans, we can present a procedure for executing such waveplans and thus answering regular queries with memory. This procedure will be an extension of [6], and concern a fix-point procedure of `crank`, `reduce` and `cache` routines, where intermediate results are expanded from the graph or a view, duplicates are removed and the new intermediate results are stored, respectively.

As with enumeration, the execution procedure is built on that presented in [6]. It concerns a *fix-point* procedure where the execution is carried out iteratively until no new results are produced. The terms execution and *search* are used interchangeably since the execution procedure is similar to the *breadth-first search* graph algorithm.
The input to the search is a data graph G and a k-register waveplan $\mathcal{A}_Q$ constructed from a given regular query with memory $Q = x \xrightarrow{e} y$. We say that the automaton $\mathcal{A}_Q$ *guides* the search, since the graph search and a run of the automaton are carried out in lockstep. That is, each transition in $\mathcal{A}_Q$ specifies the exact operations to be carried out in an iteration of the search, and the state of $\mathcal{A}_Q$ determines which transitions can be taken.
During the search, intermediate results take the shape of a triple consisting of two m-tuples as defined in Chapter 4 and a state in the k-register waveplan. For example, the triple (v, r, q) with $v = ((0, v_1), (1, v_2), (2, v_3))$ and $r = ((0, 2), (1, 5))$ denotes an intermediate result with vertices v_1, v_2 and v_3 for steps 0, 1 and 2, respectively, register values 2 and 5 for registers 0 and 1 and state q.

6.1 Search routine

The search starts by *cranking* the graph based on the waveplan $\mathcal{A}$. The structure and state of $\mathcal{A}$ determine the exact implementation of the `crank` routine. The tuples produced by cranking

the graph are stored in Δ_0^R, and subsequently *cached* in C_0.
While $|\Delta_i^R| > 0$ the last iteration produced new tuples, and so a fix-point has not yet been reached. We proceed by continuing to `crank`, `reduce` and `cache`.
The `crank` procedure now takes C_i as input parameter, since $\mathcal{A}$ might require expanding results from the cache.
The new cache stored in Δ_{i+1}^C will need to be *reduced*. That is, duplicate tuples within C_i and in $C_i \cup \Delta_{i+1}^C$ need to be removed from Δ_{i+1}^C in order to avoid unnecessary work and unbounded execution.
After the delta has been reduced and stored in Δ_{i+1}^R it can be cached into C_{i+1} and a new iteration may be started.
When $|\Delta_i^R| > 0$ no longer holds true, the last iteration did not produce any new results and a fix-point has been reached. The result of the regular query with memory Q can now be extracted from C which is the union of all caches produced by the search. To do this, triples $(v, r, q) \in C$ can be filtered on the predicate $q \in F$, where F is the set of accepting states in $\mathcal{A}$.

Algorithm 4 GuidedSearch($G, \mathcal{A}$)

1: $\Delta_0^R = \texttt{crank}(G, \emptyset, \mathcal{A})$
2: $C_0 = \texttt{cache}(\Delta_0^R, \emptyset)$
3: $i = 0$
4: **while** $|\Delta_i^R| > 0$ **do**
5: $\quad \Delta_{i+1}^C = \texttt{crank}(G, C_i, \mathcal{A})$
6: $\quad \Delta_{i+1}^R = \texttt{reduce}(\Delta_{i+1}^C, C_i)$
7: $\quad C_{i+1} = \texttt{cache}(\Delta_{i+1}^R, C_i)$
8: $\quad i = i + 1$
9: $C = \bigcup_{0 \leq j \leq i} C_j$
10: **return** $\texttt{extract}(C)$

6.1.1 Crank routine

The first application of the `crank` routine on line one iterates over all outgoing transitions from the starting states of $\mathcal{A}$. First, a look-up of all the edges in G that have the same edge label as the current transition's label is performed. Subsequently, local- and global data joins are applied, in that order. The resulting m-tuples are returned together with the state in $\mathcal{A}$ reached by the transition.

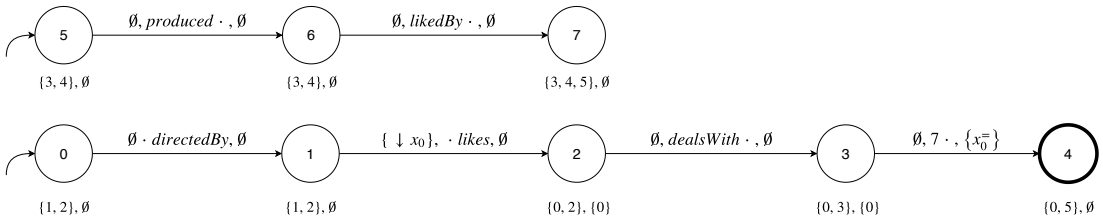


Figure 6.1: A k-register waveplan for the running example query.

Later applications of `crank` iterate over all outgoing transitions from states present in C_i, however. They too perform look-up and local data joins. After that though, results in C_i are expanded by the newly found tuples by either prepending or appending them, based on the transition's label. Only after this *join on topology* as happened, are the global data joins applied.

Let us consider the execution of a k-register waveplan for our running example, where we ask "Find all pairs u, v of people such that u likes a movie directed by someone who has worked with a producer that produced a movie that v likes, given that the movies have the same genre". Figure 6.1 shows such a plan. The execution of this plan is presented in Figure 6.2, where i indicates the iteration number of the execution routine. In the 0th iteration (i.e. before the `while`-loop), the `crank` routine is applied to the transitions from states 5 to 6, and 0 to 1. The first produces pairs of vertices for edges labeled `produced`. Similarly, the second produces pairs of vertices for edges labeled `directedBy`.

In the first iteration, transition $(6,7)$ is input to the `crank` routine. Notice that state 7 has a vertex projection $P_v = \{3,4,5\}$ since vertices for step 4 will be required later on to evaluate the condition $x_0^=$. Hence, this application of `crank` will produce triples, representing results of `produced` · `likedBy`. Transition $(1,2)$ also serves as input to the `crank` routine in this iteration. It produces pairs corresponding to `likes` · `directedBy`, as well as assigning a value to the 0th register. Since both vertex 1 and 2 have data value 1, there exists a duplicate entry with vertices $(0,2)$, register assignment (1) and state 2, which is subsequently removed by the `reduce` routine. The second iteration expands results for `likes` · `directedBy` by appending `dealsWith`. Notice that, due to the fact that register 0 cannot yet be projected out, there occurs some *duplication due to data* as discussed in Section 5.2.1

The third and final iteration appends the results from $(6,7)$ produced in the first iteration to the results from $(2,3)$ produced in the second iteration. Notice that the condition $x_0^=$, which expresses that the two movies involved are of the same genre, is in the set of global data joins

$i = 0$

crank(5, 6)

steps		*registers*	*state*
3	4		
5	9		6
6	10		6
7	10		6
7	11		6
8	12		6

crank(0, 1)

steps		*registers*	*state*
1	2		
1	4		1
2	4		1
3	4		1

$i = 1$

crank(6, 7)

steps		*registers*	*state*
3	4	5	
5	9	13	7
6	10	13	7
6	10	14	7
7	10	13	7
7	10	14	7
7	11	14	7
8	12	15	7

crank(1, 2)

steps		*registers*	*state*
0	2	0	
0	4	1	2
0	4	1	2
0	4	2	2

reduce(1, 2)

steps		*registers*	*state*
0	2	0	
0	4	1	2
0	4	2	2

$i = 2$

crank(2, 3)

steps		*registers*	*state*
0	3	0	
0	5	1	3
0	6	1	3
0	7	1	3
0	8	1	3
0	5	2	3
0	6	2	3
0	7	2	3
0	8	2	3

$i = 3$

crank(3, 4)

steps		*registers*	*state*
0	5		
0	13		4
0	14		4
0	13		4
0	14		4
0	14		4

reduce(3, 4)

steps		*registers*	*state*
0	5		
0	13		4
0	14		4

Figure 6.2: Execution of the plan from Figure 6.1.

on $(3,4)$ and is therefore evaluated after the results from $(2,3)$ and $(6,7)$ have been joined. It is important to recognize that $(2,3)$ produces steps 0, 3 and register 0, whereas $(6,7)$ produces steps 3, 4 and 5. Hence, all the required steps and registers are available to evaluate $x_0^=$ *before* the projection to steps 0 and 5 is applied.

7 Experimental Setup

This chapter briefly discusses the implementation of the proposed system. Furthermore, it formalizes the hypotheses central to this thesis, and describes the methodology by which to test them.
The hypotheses are twofold; (1) topology and data planning are orthogonal, and (2) data planning significantly impacts performance for fixed topology.

7.1 Implementation

By and large, the implementation of the proposed system is the same as that of WaveGuide [6]. We will reiterate some of the important aspects of the implementation regarding planning, execution and hardware, as well as indicating how the current implementation differs from WaveGuide.

7.1.1 Planning

The implementation regarding query planning consists of two main parts; *parsing* and *enumeration*. To parse regular expressions with memory the Java distribution of the ANTLR library is used. Subsequently, the enumeration is implemented in Java. These choices are inherited from WaveGuide.

7.1.2 Execution

Like WaveGuide, the current execution procedure is built using procedural SQL in PostgreSQL. PostgreSQL was originally chosen because it is open-source and has a well performing implementation of procedural SQL.
Transitions in a Waveplan are *deployed* to a database by translating them to a procedural SQL routine called `transition()`, and storing the code as data in a table `trans`. The code for a

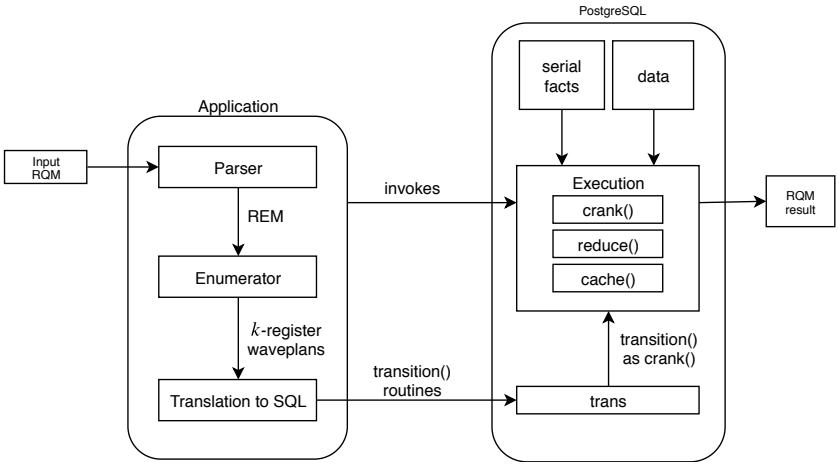


Figure 7.1: Overview of the proposed system.

particular transition is then injected into the main execution procedure as described in Chapter 6, thus implementing the `crank` routine.

The data graph is decomposed into two tables; `serialfacts` and `data`. The first table stores triples consisting of a *subject* (vertex), *predicate* (edge label) and *object* (vertex). This represents the topology of the graph. The second table stores pairs consisting of a subject (vertex) and a data value. This represents the data in the vertices of the graph.

Since intermediate results may consist of $n+k$ steps, with n the number of steps used to store vertices, and k the number of steps to store registers, the *cache* covers all $n+k$ steps, even though many steps will only contain `null` values. Other approaches using decomposition of a tuple into multiple rows were considered, but these proved inferior to storing the full projection in terms of performance.

Figure 7.1 shows an overview of the proposed system and the interactions between its various components.

7.1.3 Hardware

The hardware available for the experimental evaluation consists of a Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz, 8GB of RAM, a 332GB partition of a 7200RPM HDD running Ubuntu 18.04.1 LTS.

7.2 Edge walks

To asses the performance of k-register waveplans we will look at the number of *edge walks* that a plan makes. The number of edge walks is defined as the total number of unique edges of the input graph that the execution procedure must store. Since the `reduce` routine removes duplicate tuples, we know that each tuple that is cached represents exactly one edge walk. Thus, the number of edge walks is the total size of the cache after execution has finished.
The motivation for this choice of statistic is that it can be regarded as *implementation agnostic*. That is, its value does not depend on the implementation of particular operations. Nor does it depend on the hardware on which an experiment runs. A statistic like total execution time, for instance, would depend on these factors. Given that the logical operations necessary to execute a plan, as well as the order in which they are applied, is fixed, the execution routine must cache the same number of tuples, regardless of the way in which these tuples are obtained. Hence, with better implementation and hardware a plan will run faster, but it won't cache fewer tuples.
As an example, consider the execution presented in Figure 6.2. Operations `crank`(5, 6), `crank`(0, 1), `crank`(6, 7) and `crank`(2, 3) produce 5, 3, 7 and 8 tuples, respectively. Additionally, operations `reduce`(1, 2) and `reduce`(3, 4) produce 2 tuples each. Hence, the total number of tuples, and thus the total number of edge walks, is 27.

7.3 Input

The input to an experiment consists of a data graph G and a set of regular queries with memory, known as a *workload*.

7.3.1 Data graph

The data graph is extracted from the YAGO2 data set. YAGO2 is a knowledge base built from Wikipedia, GeoNames and WordNet [7]. This data has been serialized into triples consisting of a subject, predicate and object. Additionally, a subset of predicates has been identified as representing data values. One such predicate is `hasIncome`. The value found at the object of an edge labeled `hasIncome` thus concerns a monetary value, and can be serialized into a pair consisting of the subject and a data value. Since the grammar for REMs only allows for equality and inequality tests, data values will need to be *categorized* in order to avoid extremely selective queries. For monetary values, for instance, values are categorized with respect to their

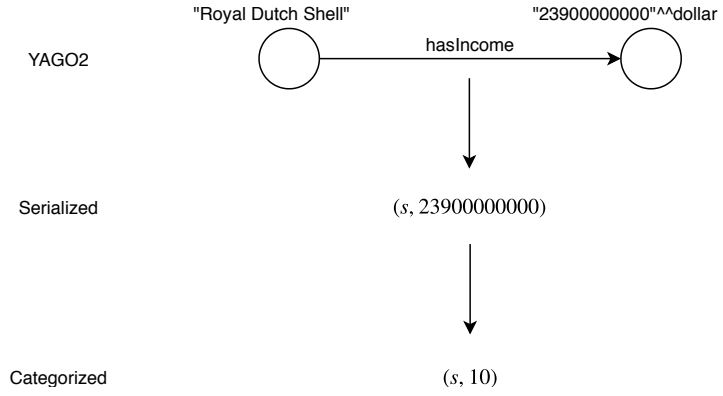


Figure 7.2: Serialization and categorization of a monetary value.

order of magnitude. That is, we take the base-10 logarithm of the integer value, assuming it is positive. Figure 7.2 shows an example of serialization and categorization for a triple with predicate `hasIncome`, where s is a variable representing the serialization of the subject.

7.3.2 Workload

The set of RQMs that serve as input to an experiment are obtained by first defining *patterns*, and then *mining* labels to make these patterns concrete.

Patterns

Patterns are regular expressions with memory, using placeholder labels like a, b and c. An example pattern is:

$$e = (a \cdot \downarrow x_0.b[x_0^{=}] \cdot c)^{+} \tag{7.1}$$

Two properties are important when defining patterns; (1) a pattern must contain at least one assignment and one condition that depends on an assignment, and (2) a condition that depends on an assignment must be inside of a Kleene plus. These properties are important, because if they are not met, an RQM that is defined by such an REM can be rewritten as a *conjunctive regular path query* (CRPQ). We abstain from providing a full comparison between the expressiveness and complexity of CRPQs compared to RQMs, but suffice it to say that since an RQM without use of assignments and conditions is an RPQ, it is therefore also a CRPQ. Furthermore, an RQM that does not check against a register within a closure can be thought of as a number of RPQs with a variable binding to each vertex where a register is used. Since the number of register

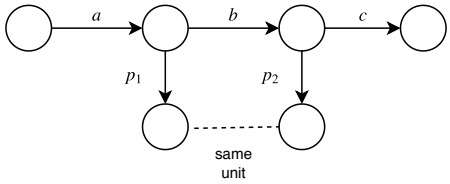


Figure 7.3: Graph representation of pattern $(a \cdot \downarrow x_0.b[x_0^=] \cdot c)^+$.

uses is finite, so is the number of RPQs. Hence, there is an equivalent CRPQ and a condition over the variables that produces the same result as such an RQM.

Mining

To obtain a concrete workload, we assign predicates from YAGO2 to the labels in a pattern. When looking for predicates from YAGO2 to assign to labels, it is helpful to think of a pattern as a graph. For instance, the pattern 7.1 can be mined using the graph displayed in Figure 7.3. Notice that, besides the topological structure of 7.1, there are also edges for the data values necessary to perform an assignment and evaluate a condition. For a query to make any sort of semantic sense, we require that p_1 and p_2 produce data values that have the same unit, say a monetary value. If this constraint were not in place, we might end up comparing a monetary value to a person's height, for instance. Such a comparison would negate the value of using real data, since data values might as well have been sampled if there is no need for a relation between them. Consider the following pattern:

$$e = (\downarrow x_0.a \cdot b[x_0^=])^+ \tag{7.2}$$

It's graph representation is given in Figure 7.4. Notice that in this case, the two edges capturing data values must not only have the same unit, but they must actually have the same predicate, since a data graph only allows a single data value to be associated with each vertex. We have found the data graph model and the YAGO2 data set to be rather irreconcilable on this point. That is, there was no combination of predicates that matched pattern 7.2 while also being *categorizable*. In summary then, fifty combinations of label a, b and c were mined for pattern 7.1 to comprise the experimental workload.

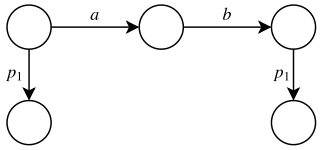


Figure 7.4: Graph representation of pattern $(\downarrow x_0.a \cdot b[x_0^=])^+$.

7.4 Planning orthogonality

Recall that a plan can be optimal *with respect to topology*, and *with respect to topology and data*. The first is referred to as being topologically optimal, the second as being overall optimal. Intuitively, this means that a plan that is optimal overall does not always have the same *topological order* as a plan that is topologically optimal.

Consider the following three regular expressions with memory, when applied to the data graph from Figure 1.1b.

$$e_1 = \texttt{likes} \cdot \texttt{directedBy} \cdot \texttt{dealsWith} \cdot \texttt{produced} \cdot \texttt{likedBy} \tag{7.3}$$

$$e_2 = \downarrow x_0.\texttt{likes} \cdot \texttt{directedBy} \cdot \texttt{dealsWith} \cdot \texttt{produced} \cdot \texttt{likedBy}[x_0^=] \tag{7.4}$$

$$e_3 = \texttt{likes} \cdot \downarrow x_0.\texttt{directedBy} \cdot \texttt{dealsWith} \cdot \texttt{produced}[x_0^=] \cdot \texttt{likedBy} \tag{7.5}$$

Notice that e_3 is the expression from our running example. Enumerating and executing the plans for e_1 yields the plan in Figure 7.5 as one of the optimal plans with respect to the number of edge walks, producing a total of 15.

Since e_1 is the regular part of both e_2 and e_3, we know that the order in which the labels are evaluated in the plan in Figure 7.5 is the optimal topological order for plans for e_2 and e_3.

To show that a plan for an REM with assignments and conditions that is optimal overall, can have the same topological order as a plan that is topologically optimal, consider the plan for expression e_2 in Figure 7.6. This plan has the same topological order as the plan in 7.5, which we saw is the optimal topological order. Since the 0th register can be assigned only one value (namely 47), there is no duplication due to data, and so the number of edge walks for this plan is also 15, meaning that it is optimal overall.

Finally, consider the plan for expression e_3 in Figure 7.7. This plan also has the optimal topological order. When applied to the data graph from our running example, this plans

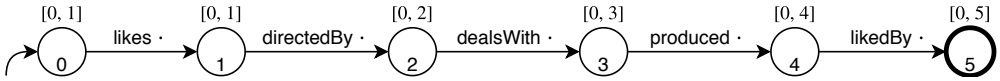


Figure 7.5: k-Register waveplan for e_1.

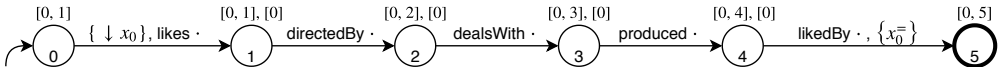


Figure 7.6: k-Register waveplan for e_2.

produces 21 edge walks. Conversely, the plan in Figure 7.8 also evaluates e_3, but does not have the optimal topological order. This order, when applied to e_1 would yield 27 edge walks. When applied to e_3, however, it only produces 14 edge walks, by being able to projection the 0th register out immediately after use. Hence, this plan is not optimal with respect to topology, but it is optimal with respect to topology and data.

7.4.1 Improvement ratio

We will investigate the orthogonality between topological optimality and overall optimality by looking at a statistic we call the *improvement ratio*. This statistic represents the proportion of a workload for which better plans were found by planning for topology and data both.

Let W be a workload consisting of regular queries with memory of the shape $Q := x \xrightarrow{e} y$. To obtain the improvement ratio ψ_W over W, we produce a regular expression r for every $Q \in W$ by removing all assignments and conditions from e. Enumerating the plans for both e and r yields sets of plans E and R, respectively. Then, executing all plans in E and R yields $E' \subseteq E$ and $R' \subseteq R$ which are the sets of optimal plans for e and r, respectively. Since r is the regular part of e, the plans in R' are topologically optimal for e. By counting the number of times that the intersection between E' and R' is empty, we count the number of times that a better plan was found by planning for topology and data both. For if the intersection is non-empty, it means that an optimal plan could have been found by planning with respect to data only for an optimal plan for r. By dividing the number of times a better plan was found by planning for

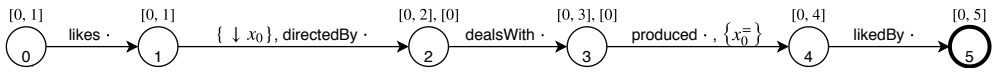


Figure 7.7: Sub-optimal k-register waveplan for e_3

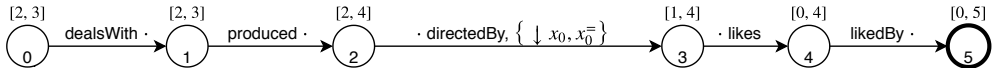


Figure 7.8: Optimal k-register waveplan for e_3

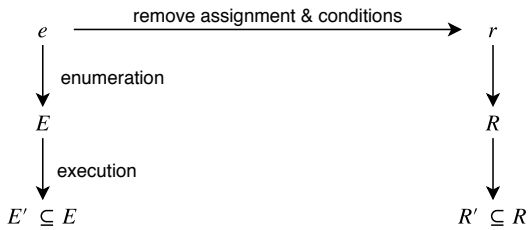


Figure 7.9: Obtaining the sets of optimal plans for an RQM and its regular part.

topology and data both by the size of the workload, we obtain the improvement ratio. Formally, this ratio can be expressed as:

$$\psi_W = \frac{1}{|W|} \sum_{Q \in W} (E' \cap R' = \emptyset) \tag{7.6}$$

where we assume that the predicate $(E' \cap R' = \emptyset)$ evaluates to zero or one. Figure 7.9 shows the process of obtaining E' and R' from Q schematically.

Hence, our first hypothesis can now be formulated as: there are regular queries with memory for which an overall optimal plan can only be found by planning with respect to topology and data both. Formally, this can be written as $\psi_W > 0$ for some workload W.

7.5 Data planning performance

Intuitively, the second hypothesis states that planning with respect to data is useful to begin with. Let E be the set of plans obtained from REM e, and $E^r \subset E$ a set of plans that share the topological order r. Then, any performance difference between plans $p, q \in E^r$ is due to the way the interactions with data have been planned. Hence, for every topological order r of an REM e we will investigate the subset E^r and look at the difference in order of magnitude of the best- and worst performing plans within that subset. Again, performance is expressed in the number of edge walks.

8 Experimental Results

The experiment results for the orthogonality of planning with respect to topology and data, as well as the results of planning with respect to data for fixed topology are summarized in this chapter.

8.1 Planning orthogonality

The improvement ratio ψ_W of the workload W mined based on pattern $e = (a \cdot \downarrow x_0.b[x_0^{=}] \cdot c)^+$ is $\psi_W = 0.16$. This means that for 16% of the queries in the workload an overall optimal plan did not have the same topological order as a topologically optimal plan. In other words, for just this pattern and workload, an overall optimal plan could not have been obtained by not considering topology and data together during planning.

8.2 Data planning performance

Figure 8.1 shows the relative frequency of the difference in order in magnitude (ODE) between the best performing (i.e. smallest number of edge walks) and worst performing (i.e. largest number of edge walks) plans within sets of plans that have a shared topological order. Hence, this difference in performance is solely due to planning with respect to data.
Twenty percent of sets display at least one order magnitude difference in the number of edge walks between the best- and worst performing plans for the pattern e.
Figure 8.2 breaks down the relative frequency of the ODE on a query-by-query level. That is, it shows for each query the relative frequency of subsets of plans with shared topological order that display a certain ODE. For example, for query number four, around 80% of subsets did not display an order of magnitude difference between the best- and worst performance plans,

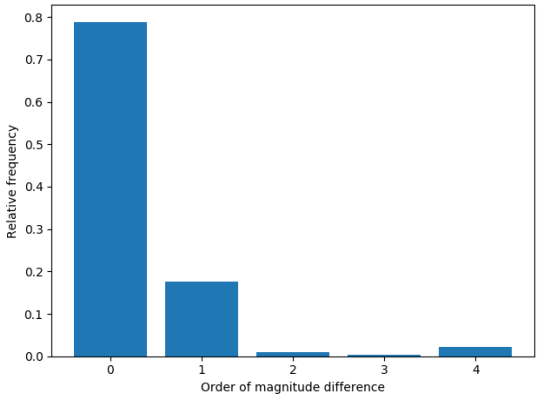


Figure 8.1: Order of magnitude difference between best- and worst edge walks.

whereas the other roughly 20% displayed one order of magnitude difference.

The extreme cases, in both figures, of three or four orders of magnitude difference for a given order must be regarded with some caution. Namely, there are queries within the workload for which the result is empty. Hence, a plan that can discard all tuples early on, may well cache orders of magnitude fewer tuples than a plan that discard its tuples later, even though the absolute number of tuples considered by the latter plan may not be particularly large.

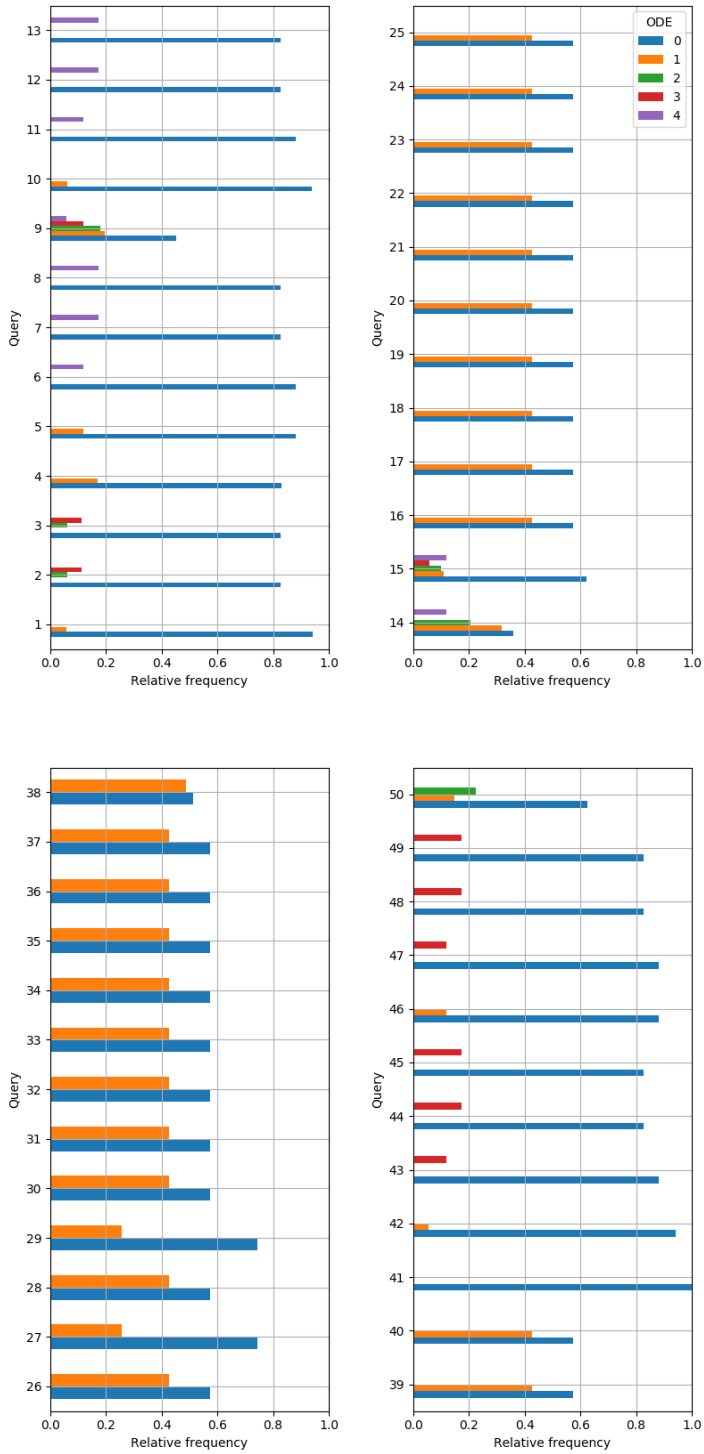


Figure 8.2: Relative frequency of ODE for subsets with shared topological order.

9 Conclusion

We have introduced k-register waveplans as a way of representing plans for RQMs. This approach combines the strengths of register automata [3] and WavePlans [6]. Additionally, we have investigated the peculiarities in planning RQMs, and have proposed a plan space $\mathcal{P}_{SRP}$ for RQMs, arguing that the complexity of this plan is $O(|r|^n c^{|}r|)$ where $|r|$ is the size of regular part of the input REM, n is the total number of assignments and conditions and c is a constant. An extension of the enumeration procedure as detailed in [6] has been proposed. This extension adds projection to the existing enumeration rules, and adds two new rules to cover both local- and global assignments and conditions. Additionally, the optimal substructure of the enumeration problem for RQMs has been shown, which together with the consideration of the effect that data has on cardinality estimation, can be used to implement cost-based enumeration. We have analyzes the complexity of the enumeration procedure and have concluded that it is $O(|e|^2|r|^6 8^n)$ where $|e|$ is the size of the input REM, $|r|$ is the size of its regular part and n is the total number of assignments and conditions in e.
Furthermore, we have extended the fix-point procedure used to execute WavePlans to cover plans with flexible intermediate result sizes.

On an empirical note, we have investigated two hypotheses; (1) topology and data planning are orthogonal, and (2) data planning significantly impacts performance for fixed topology plans.

The former hypothesis was formalized as $\psi_W > 0$ for some workload W, which states that we expect the proportion of plans that are optimal overall, but have a different topological order than the optimal topological order is larger than zero. In other words, we hypothesized that certain optimal plans can only be found by planning with respect to topology and data both.
The improvement ratio ψ_W over a workload W consisting of fifty variations of the pattern

$e = (a \cdot \downarrow x_0.b[x_0^=] \cdot c)^+$ obtained from- and evaluated over a data graph extracted from the YAGO2 data set, was found to be $\psi_W = 0.16$.
Hence, even for a simple pattern which uses only one register for a single assignment and a single condition, we could not have found an overall optimal plan without planning for topology and data together in 16% of the queries in W.

We have also found some evidence to support the second hypothesis. Indeed, plans with the same topological structure but different interactions with data can display order of magnitude differences in performance with respect to edge walks. Our results show that just over a fifth of all groups with the same topological structure produce order of magnitude differences in performance. While this result might seem small at first glance, it must again be taken into account that only one register was used, once in an assignment and once in a condition. We speculate that the effect on performance increases significantly when more registers are used. Hence, to already observe order of magnitude performance differences for a fifth of all groups with only one register, is actually a fairly strong result.

In summary, we conclude that there is some evidence that overall optimal plans can only be obtained by considering topology and data together for a substantial proportion of queries. Furthermore, we have seen that planning with respect to data can produce order of magnitude differences in performance for sets of queries with a shared topological order. Both of these conclusions were reached for a very modest query pattern, using only one register for one assignment and one condition, but also on a limited workload.
This conclusion encourages us to expand upon the methodology by (1) considering larger and more varied data sets, (2) building much larger workloads to test this hypothesis more rigorously and (3) expanding the grammar of the query language to be able to write more realistic queries.

References

[1] Angela Bonifati, G.H.L. Fletcher, Hannes Voigt, and N. Yakovets. *Querying graphs*. Morgan & Claypool Publishers, 2018.

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008.

[3] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, March 2016.

[4] Pablo Barceló, Gaëlle Fontaine, and Anthony Widjaja Lin. Expressive path queries on graph with data. *Logical Methods in Computer Science*, 11, 2013.

[5] Jelle Hellings, Bart Kuijpers, Jan Van den Bussche, and Xiaowang Zhang. Walk logic as a framework for path query languages on graph databases. In *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, pages 117–128, New York, NY, USA, 2013. ACM.

[6] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Query planning for evaluating SPARQL property paths. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1875–1889, 2016.

[7] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 01 2013.