

**OPTIMIZATION OF REGULAR PATH QUERIES IN GRAPH  
DATABASES**

NIKOLAY YAKOVETS

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE  
STUDIES  
IN PARTIAL FULFILMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE  
YORK UNIVERSITY  
TORONTO, ONTARIO  
AUGUST 2016

## Abstract

Regular path queries offer a powerful navigational mechanism in graph databases. Recently, there has been renewed interest in such queries in the context of the Semantic Web. The extension of SPARQL in version 1.1 with *property paths* offers a type of *regular path query* for RDF graph databases. While eminently useful, such queries are difficult to optimize and evaluate efficiently, however. We design and implement a cost-based optimizer we call WAVEGUIDE for SPARQL queries with property paths. WAVEGUIDE builds a query plan—which we call a *waveplan* (WP)—which *guides* the query evaluation. There are numerous choices in the construction of a plan, and a number of optimization methods, so the space of plans for a query can be quite large. Execution costs of plans for the same query can vary by orders of magnitude with the best plan often offering excellent performance. A WP’s costs can be estimated, which opens the way to cost-based optimization. We demonstrate that WAVEGUIDE properly subsumes existing techniques and that the new plans it adds are relevant. We analyze the effective plan space which is enabled

by WAVEGUIDE and design an efficient enumerator for it. We implement a prototype of a WAVEGUIDE cost-based optimizer on top of an open-source relational RDF store. Finally, we perform a comprehensive performance study of the state of the art for evaluation of SPARQL property paths and demonstrate the significant performance gains that WAVEGUIDE offers.

## Acknowledgements

First and foremost, I would like to thank my supervisors, Professors Jarek Gryz and Parke Godfrey. Their continuous support, deep insights, motivation and encouragement helped me in my research and writing of this dissertation.

Also, I would like to express my gratitude to the rest of my examination committee Prof. Aijun An, Prof. Suprakash Datta, and Prof. Ilijas Farah for their helpful comments and engaging conversation during my defense. Additionally, I would like to thank Prof. Ken Pu for taking time out from his busy schedule to serve as my external examiner.

I would like to thank my colleagues at York University, University of Toronto, and University of Waterloo for our debates in computer science and mathematics, exchange of skills and knowledge during my graduate program. Thanks also goes out to Ouma Jailpaul-Gill for being a great friend. I am grateful to Ulya Yigit, Seela Balkissoon, and Jason Keltz for all of their computer and technical assistance throughout my graduate program.

Last but not the least, I would like to thank my parents Alexander and Natalia for their love and support they provided me through my entire life. A very special appreciation goes out to my wife and best friend, Sasha, without whose love, encouragement, and humour, I would not have finished this dissertation.

# Table of Contents

<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Table of Contents</b>	<b>vi</b>
<b>List of Figures</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The Problem . . . . .	1
1.2 Motivation . . . . .	5
1.3 Goals . . . . .	6
1.4 Contributions . . . . .	8
1.5 Outline . . . . .	9
<b>2 Background &amp; Related Work</b>	<b>11</b>
2.1 Graph Data Models . . . . .	11

2.1.1	RDF . . . . .	15
2.1.2	Property Graph . . . . .	18
2.2	Graph Serialization . . . . .	20
2.2.1	Relational . . . . .	20
2.2.2	Native Graph Databases . . . . .	27
2.3	Path Queries . . . . .	29
2.3.1	Path Semantics . . . . .	31
2.3.2	Paths in SPARQL . . . . .	33
2.3.3	Paths in Cypher . . . . .	35
2.4	Query Planning . . . . .	35
2.5	Related Work . . . . .	37
2.5.1	FA Plans . . . . .	38
2.5.2	Alpha-RA Plans . . . . .	43
2.5.3	Index-based Evaluation . . . . .	46
2.5.4	Relational Query Optimizers . . . . .	48
<b>3</b>	<b>Methodology</b>	<b>53</b>
3.1	Motivation . . . . .	53
3.2	Graph Walk . . . . .	59
3.3	Query Plans . . . . .	60

3.4	Optimizer & Enumerator . . . . .	60
3.5	Implementation & Benchmarking . . . . .	61
<b>4</b>	<b>Graph Walk</b>	<b>63</b>
4.1	Wavefronts . . . . .	63
4.2	Expanding a Wavefront . . . . .	64
<b>5</b>	<b>Query Plans</b>	<b>69</b>
5.1	Guiding a Wavefront . . . . .	69
5.2	Wavefront Interaction . . . . .	71
5.3	<b>WAVEGUIDE</b> 's Plan Space . . . . .	75
<b>6</b>	<b>Optimizer &amp; Enumerator</b>	<b>77</b>
6.1	Cost Framework . . . . .	77
6.2	Search Cost Factors . . . . .	79
6.2.1	Search Sizes . . . . .	79
6.2.2	Solution Redundancy . . . . .	79
6.2.3	Sub-path Redundancy . . . . .	80
6.3	Plan Optimizations . . . . .	80
6.3.1	Choice of Wavefronts . . . . .	81
6.3.2	Reduce . . . . .	82
6.3.3	Threading . . . . .	83

6.3.4	Partial Caching . . . . .	83
6.3.5	Loop Caching . . . . .	84
6.4	Cost Analysis . . . . .	85
6.4.1	Cost of Threading . . . . .	86
6.4.2	Cost of Loop Caching . . . . .	88
6.5	Cardinality Estimator . . . . .	91
6.5.1	Synopsis Statistics . . . . .	94
6.5.2	Consistent Estimation . . . . .	98
6.6	Plan Enumerator . . . . .	99
6.6.1	Standard Plan Space . . . . .	102
6.6.2	Enumeration . . . . .	119
<b>7</b>	<b>Implementation &amp; Benchmarking</b>	<b>141</b>
7.1	Implementation . . . . .	141
7.1.1	Software: The System . . . . .	141
7.1.2	Hardware: Runtime . . . . .	146
7.1.3	Software: Runtime . . . . .	146
7.2	Methodology . . . . .	146
7.3	The Optimizations . . . . .	147
7.3.1	Threading . . . . .	147

7.3.2	Loop Caching . . . . .	149
7.3.3	Partial Loop Caching . . . . .	154
7.3.4	Combined Optimizations . . . . .	155
7.4	Comparison with Other Systems . . . . .	156
7.4.1	Transitive Closure . . . . .	157
7.4.2	Query Planning . . . . .	159
7.4.3	Query Planning vs. Transitive Closure . . . . .	160
<b>8</b>	<b>Conclusions</b>	<b>162</b>
8.1	Contributions . . . . .	163
8.2	Future Work . . . . .	164
8.2.1	Multiple & Conjunctive RPQs . . . . .	165
8.2.2	Better Cardinality Estimation . . . . .	168
8.2.3	A Richer Enumerator: Beyond Standard Waveplans . . . . .	170
8.3	In Summary . . . . .	173
<b>Bibliography</b>		<b>175</b>
<b>A Appendix</b>		<b>181</b>
A.1	Nomenclature . . . . .	181
A.2	Queries . . . . .	183
A.2.1	Threading . . . . .	183

A.2.2 Loop Caching . . . . .	184
A.2.3 Partial Loop Caching & State of the art Planning . . . . .	184
A.2.4 State of the art Planning . . . . .	185

## List of Figures

1.1	An example graph database.	3
2.1	Timeline of graph database models.	13
2.2	Graph representation of an RDF triple.	16
2.3	Example property vs. RDF graph.	19
2.4	RDF storage based on a single table.	21
2.5	RDF storage based on property tables.	23
2.6	RDF storage based on vertical partitioning.	26
2.7	Relational vs. native physical graph storage.	27
2.8	Different solution semantics in RPQs.	31
2.9	Query optimizer architecture.	35
2.10	An $\varepsilon$ -NFA and corresponding reduced NFA for $\mathcal{Q}_{1.3}$ .	40
2.11	Example run of an algorithm proposed by Kochut et al.	41
2.12	Example run of an algorithm proposed by Koschmieder et al.	43
2.13	A parse tree and $\alpha$ -RA tree for query $\mathcal{Q}_{1.3}$ .	44

3.1	Plan space classes. . . . .	54
3.2	Plans and corresponding DATALOG programs. . . . .	58
4.1	WAVEGUIDE's evaluation procedure. . . . .	64
4.2	Two waveplans, $P_1$ & $P_2$ , over graph $G$ , with an evaluation trace of $P_1$ . . . . .	67
5.1	Types of transitions used in a wavefront. . . . .	70
5.2	Waveplans for $abc$ expression. . . . .	74
5.3	Attempting to build waveplans for $(abc)^+$ expression. . . . .	74
6.1	Types of search cost factors. . . . .	81
6.2	Choosing the wavefronts. . . . .	82
6.3	Threading a shared sub-path. . . . .	84
6.4	Types of loop caching. . . . .	85
6.5	Lensing. . . . .	89
6.6	Cardinality estimation of a join. . . . .	92
6.7	Synopsis statistics for graph label frequencies. . . . .	94
6.8	Estimating join cardinality using synopsis. . . . .	96
6.9	Different ways of estimating the cardinality of $r = a/b/c/d$ . . . . .	98
6.10	Thompson Construction. . . . .	100
6.11	Recursive templates for a standard waveplan. . . . .	103

6.12	Plan space classes.	105
6.13	Example of SWP generation.	106
6.15	WAVEGUIDE's memoization sub-routine.	120
6.14	WAVEGUIDE's enumeration procedure.	121
6.16	WAVEGUIDE's plan generation sub-routine.	122
6.17	WAVEGUIDE's constant seed passing sub-routine.	122
6.18	WAVEGUIDE's Kleene seed passing sub-routine.	122
6.19	Enumeration rules for graph label expressions.	123
6.20	Enumeration rules for the concatenation operator.	124
6.21	Enumeration rule for the union operator.	124
6.22	Enumeration rules for Kleene closures.	125
6.23	Enumeration rules for <i>seed passing</i> .	125
6.24	A run of the enumeration algorithm for $Q = (x, (abc)^+, y)$ .	130
6.25	A plan template for Kleene plus $Q = (x, s_1^+, y)$ .	138
7.1	Overview of the prototype system.	142
7.2	Query plan designer.	144
7.3	Runtime visualizer and profiler.	145
7.4	Benchmarking Threading.	148
7.5	Benchmarking Loop Caching.	150
7.6	Effect of plans on query evaluation.	151

7.7	Benchmarking vs. state of the art. . . . .	157
8.1	Extended Synopsis. . . . .	168
8.2	A richer enumerator. . . . .	171
8.3	Example of $k$ -unrolling. . . . .	172

# 1 Introduction

## 1.1 The Problem

Graph data is becoming rapidly prevalent with the rise of the Semantic Web, social networks, and data-driven exploration in life sciences. There is a need for natural, expressive ways to query over these graphs. Standards are coming into place for this. The Resource Description Framework (RDF) [56] provides a data model for graph data. An RDF *store* is a set of *triples* that describes a directed, edge-labeled multi-graph. A triple,  $\langle s, r, o \rangle$ , denotes an *edge* from *node* “*s*” (the *subject*) to node “*o*” (the *object*), with the edge labeled by “*r*” (the *role*, also called *label* or *predicate*).

Correspondingly, the SPARQL query language [54] provides a formal means to query over RDF stores. A query defines sub-graph match criteria; its evaluation over an RDF store returns all embedded sub-graphs meeting the criteria. For example, the query

$$\text{"?friend :friendOf Charles"} \quad (\mathcal{Q}_{1.1})$$

evaluates to a list of people (nodes binding to variable “`?friend`”) who are friends of (role “`:friendOf`”) “Charles” (a named node, so a constant). This is a simple query, of course, and could be evaluated just by extracting the triples with “`r = :friendOf`” and “`o = Charles`”.

In its latest version, 1.1, SPARQL’s expressiveness is extended with *property paths* [29]. This effectively introduces the concept of *regular path queries* (RPQs)—well studied before the advent of RDF and SPARQL—into the query language. Instead of specifying the path of interest *explicitly* between nodes, one may now specify it *implicitly* via a *regular expression*. (This also means matching paths in the graph are not bounded in length by the query’s expression, while they are in SPARQL 1.0). For example, the query

$$\text{?friend :friendOf+ Charles .} \quad (\mathcal{Q}_{1.2})$$

evaluates to a list of people who are friends of “Charles”, or friends of people who are friends of “Charles”, and so forth (that is, a *transitive closure* over “`:friendOf`”).

While SPARQL provides the expressiveness we desire, such queries are more challenging to optimize well. Query  $\mathcal{Q}_{1.1}$  could be evaluated just by extracting the triples with “`r = :friendOf`” and “`o = Charles`”. For even a slightly more complicated query, however, it may not be straightforward to find a *plan* to evaluate it efficiently.

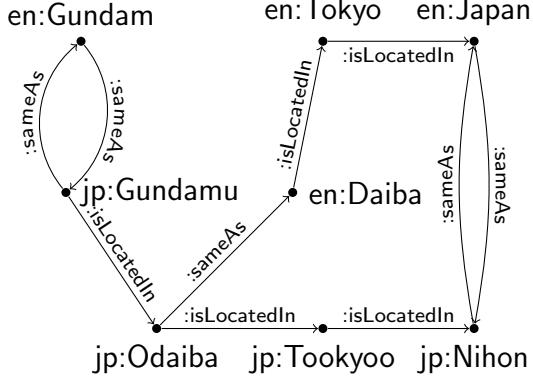


Figure 1.1: An example graph database.

For example, consider the knowledge base dataset of the Linked Open Data (LOD) cloud. LOD is a community effort which aims to interlink the structural information available in various datasets on the Web (such as Wikipedia, WordNet, and others), and make it available as a single RDF graph.

RPQs prove useful in querying such linked data by providing a convenient declarative mechanism which can be used to answer queries without prior knowledge of the underlying data paths.

**Example 1** Consider the part of a LOD graph database as presented in Fig. 1.1. This represents information the Gundam robot statue in Odaiba in Tokyo. The data has been integrated from two datasets, identified by the prefixes `en` and `jp`, standing for the English and Japanese Wikipedia, respectfully. The data entities between these two datasets are interlinked by using Web Ontology Language (OWL) ontology terms. Equivalent entities are connected with `owl:sameAs` edges. In this

case, the Japanese dataset contains richer spatial information related to the statue than does the English dataset.

Say a user wants to know in which country this *Gundam* statue is located. Since there are no direct `:isLocatedIn` edges outgoing from `en:Gundam`—as is often the case in linked data—the graph needs to be searched. During the search, equivalent data entities need to be resolved by following `:sameAs` edges. Likewise, a spatial hierarchy needs to be computed by following `:isLocatedIn` edges. This search can be defined by the following SPARQL query pattern:

$$\begin{aligned} \text{en:Gundam} \; (&\text{:sameAs}^*/\text{:isLocatedIn})^+ \\ &/\text{:sameAs}^* \; ?\text{place} \end{aligned} \quad (\mathcal{Q}_{1.3})$$

$\mathcal{Q}_{1.3}$  computes the spatial hierarchy starting from node `en:Gundam`, using information from both interlinked datasets to resolve equivalent entity closures.

Observe that  $\mathcal{Q}_{1.3}$  requires transitive closure over concatenation:

“`:sameAs*/:isLocatedIn`”

with respect to the graph. As we will see, even simple queries like this are significantly more challenging to optimize and plan for than queries without closures (e.g.,  $\mathcal{Q}_{1.1}$ ), and than queries with closures just over single labels (e.g.,  $\mathcal{Q}_{1.2}$ ).

## 1.2 Motivation

Property path evaluation is a tale of two methods: two quite different approaches appear in the literature. For RPQs, seminal work [48] that introduced the **G+** query language shows how to use a *finite state automaton* (**FA**) effectively as a plan to guide the *graph walk* for the query’s evaluation. We call this approach **FA**. Subsequent work on RPQs has followed this idea.

SPARQL with property paths is much more recent. Systems for SPARQL query evaluation have followed the second approach, based primarily on the seminal work of [44]. These extend the *relational algebra* to accommodate the translation of a property path’s regular expression, and then use dynamic programming over the (extended) relational-algebra parse to devise a plan. Added is an “ $\alpha$ ” operator, which provides the transitive closure over a relation to accommodate regular expressions’ Kleene star. VIRTUOSO [21], a leading SPARQL system, does just this. Thus, we call this approach  $\alpha$ -RA, the relational algebra extended by “ $\alpha$ ”.

Which approach is better? We shall show that the effective “plan spaces” that result from **FA** and  $\alpha$ -RA are incomparable. Sometimes, for a given query and graph, an **FA** plan will be the better choice. Other times, an  $\alpha$ -RA plan will be. Our goal is to formalize the notion of plan space for both, to be able to choose the best plan. We shall show that a richer plan space can be had that properly

subsumes FA and  $\alpha$ -RA, and offers more plans existing in neither (“mixed” plans), and that these sometimes are the best plans.

### 1.3 Goals

Our goal is to design, build, and test a viable cost-based query optimization and evaluation for SPARQL property paths over RDF stores that is on par with the state of the art for relational database systems. We name the approach that we develop WAVEGUIDE.

The first step of this endeavor is to define a *plan space*—the space of query plans which we name *waveplans* (WPs)—for SPARQL property path queries. We focus on single-path, property path queries, essentially the RPQ fragment of SPARQL 1.1. We consider a set semantics—the `distinct` directive in each query—and thus do not consider aggregation. A *waveplan* consists of a collection of (non-deterministic) finite automata for the property path and search directives which *guide* the query evaluation. With proper choice of plan, we can gain orders of magnitude performance improvement for many property path queries over real datasets, while maintaining comparable performance for other queries, as the leading SPARQL query engines as JENA [34] and VIRTUOSO [21]. We evince that planning is critical to evaluate SPARQL queries efficiently, and that choosing the right plan depends on the underlying graph data and thus ultimately must be cost-based. Waveplans

model a rich space of plans for path queries which encompass powerful optimization techniques.

The second step of this endeavor is to design an *enumerator* which will efficiently walk the space of waveplans. We will show that even for a single-path property path query (an RPQ), the number of plans which evaluate it is exponential in the size of the query. Fortunately, the nature of RPQs allows us to design an efficient enumerator to walk the plan space in time polynomial in the size of the query.

The plan enumerator must be coupled with a *cost estimator* which costs the plans in order to be able to select the best plan by estimated cost. For this, we develop a *cost model* to model the execution costs (at runtime) of evaluation steps with respect to a plan. To be able to apply this cost model to estimation, we also need *cardinality estimation* of intermediate stages of a plan in execution with respect to the graph. To aid in cardinality estimation, some degree of statistics of the graph must be maintained. We develop a cost model, a cardinality estimator, and *synopsis* graph statistics that support the cardinality estimation. Based on these, we construct the cost estimator that can be used in tandem with plan enumerator.

We then implement a WAVEGUIDE prototype and test it on real-world datasets with a micro-benchmark of regular path queries that we develop. We justify the plan space WAVEGUIDE approach provides, perform the benchmarking of the optimizations WAVEGUIDE offers, and compare the performance against leading graph

databases.

## 1.4 Contributions

We summarize the contributions of this dissertation in three main areas: novel *plan space* and *optimizations* for regular path queries; a cost-based *optimizer* which utilizes these optimizations; and design, implementation, and benchmarking of the proposed approach.

1. *Waveguide-plan space.*
  - (a) *Summarize* the state of the art for evaluation of RPQs and SPARQL property paths (§2.5). *Establish* why none suffices (§3.1).
  - (b) *Devise* the waveguide place space (§5). *Demonstrate* it *subsumes* the state of the art, and extends well beyond it (§5.3).
  - (c) *Present* the powerful optimizations offered by waveplans (§6.3). *Model* the *cost factors* that determine the efficiency of plans (§6.4).
2. *Build* a full-fledged *cost-based query optimizer* for SPARQL 1.1 for property paths (RPQ fragments of SPARQL 1.1).
  - (a) *Devise* a concrete *cost model* for WPs (§6.1).
  - (b) *Determine* an array of *statistics* that can be computed efficiently offline that can be used in conjunction with the cost model (§6.5).
  - (c) *Define* “WP” systematically to define formally the *space* of WPs for a

given query (§6.6.1).

- (d) *Design* an enumeration algorithm to walk dynamically the space of WPs to find the WP with least estimated cost (§6.6.2).

3. *Prototyping and performance study.*

- (a) *Provide* an evaluation framework (§3.2) and implement a prototype of a WAVEGUIDE system (§7.1).
- (b) *Benchmark* query plans for realistic queries over real RDF stores / graphs. *Substantiate* the optimizations of our approach (§7).
- (c) *Justify* the *necessity* of planning and the waveplan space (§7).

## 1.5 Outline

The dissertation is organized as follows. In Chapter 2, we provide an overview of graph data models with a focus on Resource Description Framework (RDF). We discuss graph serialization methods in relational and native graph databases. The semantics of evaluation of path queries is presented. Then we overview several techniques for query planning and discuss query optimization in relational databases. Finally, we present current methods of RPQ evaluation based on finite automata and relational algebra.

In Chapter 3, after the appropriate background and existing approaches have been established, we provide an overview of our proposed WAVEGUIDE method-

ology. In Chapter 4, we present the evaluation framework used in WAVEGUIDE. We define a *wavefront* — a basic unit of graph search used in WAVEGUIDE. In Chapter 5, we introduce *waveplans* as data structures used in *guiding* wavefronts. We discuss the plan space entailed by waveplans and compare it to the existing RPQ evaluation approaches.

In Chapter 6, we present a cost-based query optimizer based on WAVEGUIDE approach. We discuss cost framework, factors which drive the cost and optimizations (enabled by WAVEGUIDE) which are used to minimize the cost. We define waveplans formally and analyze the space of waveplans. Then, we design an enumerator to efficiently walk the waveplan space. Combined with cardinality estimator, enumerator sets the foundation for cost-based WAVEGUIDE’s query optimizer.

In Chapter 7, we present our prototype implementation of WAVEGUIDE system over RDF store backed by an open-source relational database system (POSTGRESQL). Then we demonstrate the experiments on large graph datasets which show the merit of the optimizations used in WAVEGUIDE. We also compare our system with the state of the art in SPARQL property path evaluation and show that our system demonstrates significant performance gains.

In Chapter 8, we finish by reiterating the dissertation’s key contributions, present several promising directions for future work, and conclude.

## 2 Background & Related Work

### 2.1 Graph Data Models

In many applications, information about relationships between data entities is as important as the data itself. In these areas, the ability to model, manipulate and issue queries over graph structures can be extremely useful. Graphs provide natural and easily comprehensible means to represent the information in many domains:

- *Social networks:* Graph nodes represent people or groups of people and edges are various social relationships such as friendships, collaborations, shared interests and interactions.
- *Life sciences:* Advances in the automation of data gathering has resulted in significant challenges in management and analysis of data used in life sciences.

In many situations, graphs offer an ideal data modeling tool. Consider, for example, an important research area such as genomics. There, datasets that describe gene regulation, chemical structure or metabolic pathways can be naturally represented as graphs.

- *Spatial*: Queries over many technological networks involve calculations that take into account spatial or geographical features of stored entities. Examples are transport networks such as highways, airline routes and railways, telecommunication infrastructure (phone and the Internet), electrical power grids and water delivery networks.
- *Information Flow*: Networks that involve some information flow such as World Wide Web (WWW), linguistic databases such as thesauri describing relations between word classes, specialized preference and peer-to-peer networks. Finally, graphs are the model of choice in representing the data on the Semantic Web.

Besides allowing natural modeling of the data, graphs enable specification of powerful query languages. Specifically, many of the applications that we described above require special graph operations that cannot be easily expressed as queries over classical data models such as the relational data model. In social networks, we use specialized measures such as distance between nodes, neighborhoods, size of connected components, clustering coefficients of vertices and networks. In biological networks, we are interested in pair correlations which are strong (nearest neighbor degrees) and interactions between proteins (connected components). Geospatial applications utilize specific geometric operations such as area or boundary computations, intersections, inclusions, metric measures such as distance between entities

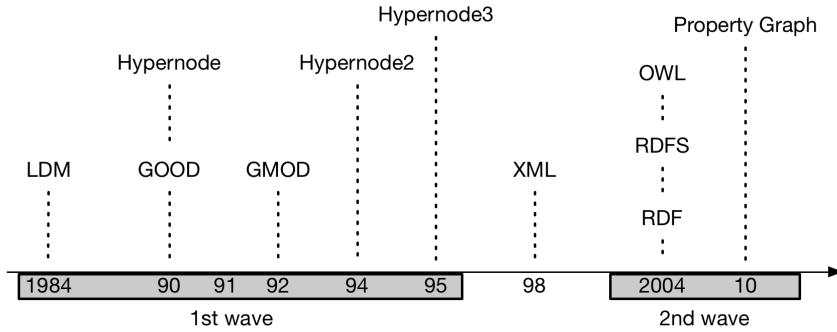


Figure 2.1: Timeline of graph database models.

and diameter of a cluster.

Over the years, graph databases have received attention both from academia and industry. In general, the research community has evolved two efforts for graph databases. Activity around graph data models flourished in the first half of the 90s, but then the topic almost disappeared. The rise of the Semantic Web in the late 2000s is attributed to the return of interest of research community to graph databases, which still remains strong today. A timeline showing influential graph data models is presented in Fig.2.1.

The Logical Data Model (LDM, [38]) is considered to be the first data model based on graphs. It was conceived in an attempt to generalize the relational, hierarchical and network models. In LDM, a schema is represented by a digraph which consists nodes of specific types and edges which represent the connections among data. The database instance, however, is represented as a table: an assignment of

values to terminal nodes in the schema. Both data and schema are graph-based in GOOD [25], an influential data model that inspired many offsprings. Specifically, GMOD [4] extended GOOD by proposing a number of concepts to deal with graph-oriented database user interfaces.

An interesting departure from using simple flat graphs as data representation is exhibited by the Hypernode family of data models [41–43]. Inspired by the practical issue of displaying the graph to the users in a clear and comprehensive way, these models utilize nested nodes, which are themselves graphs. This gives the ability to represent each real-world object as a separate database entity, thus enabling straightforward encapsulation of information.

In general, this early research established a solid theoretical base for graph-based data models: data and schema structures, data manipulation languages, and integrity constraints. Main motivations for this research were the generalization of classical data models, the limited expressive power of existing query languages, the need to improve the functionality of object-oriented systems, and work on graphical and visual interfaces. It should be noted, however, that most of these proposals lacked actual implementations.

### 2.1.1 RDF

The rise of the Semantic Web in the late 2000s is attributed to the return of interest of research community to graph databases. A new data model was required to accommodate distributed, heterogeneous, semi-structured and machine-readable metadata used on the Semantic Web. Resource Description Framework (RDF, [56]), a recommendation of W3C, was designed to handle such metadata. The main design goal of RDF is to support highly-distributed data without any assumptions about the particular application domain from which the data comes.

RDF views data as a set of *resources*, which are uniquely identified by their Internationalized Resource Identifiers (IRIs). RDF *statements* are used to describe relationships, called *properties*, between resources in the form of triples (*subject*, *property*, *object*). Properties are possible relations between resources; e.g., “created by” and “born in”. Objects can either be resources or *literals*, which are atomic values such as strings.

As an example, consider the following triple that states that Shakespeare wrote Hamlet:

(<http://lit.com/author#Shakespeare>, **wrote**, <http://lit.com/book#Hamlet>)

Another way to represent a triple  $(s, P, o)$  is as a logical formula  $P(s, o)$ , where the *binary predicate*  $P$  relates subject  $s$  to object  $o$ :



Figure 2.2: Graph representation of an RDF triple.

**wrote**(<http://lit.com/author#Shakespeare>, <http://lit.com/book#Hamlet>)

RDF can thus be represented as a graph by considering properties as labeled, directed edges between subject and object nodes, and between subjects and literals (as shown in Fig.2.2).

We formally define RDF triples as follows. Consider set  $I$  of all IRIs and  $L$  the set of all literals, and assume that these two sets are disjoint<sup>1</sup>. Then, an RDF triple is a tuple:  $(s, p, o) \in I \times I \times (I \cup L)$ , where  $s$  is the subject,  $p$  is the predicate, and  $o$  is the object. An RDF graph is a finite set of RDF triples.

RDF provides simple and flexible data model for the Semantic Web. In contrast to the relational model, in RDF the data is decomposed into triples representing the relationship between data elements explicitly. This allows for very simple data merging as one does not have to worry about matching the schemas from different data sources. Given the global identification mechanism of IRI, merging data is reduced to a union. This ability to easily process widely distributed and highly heterogeneous data is essential for RDF to be a successful Web language that aims to represent data on the Semantic Web.

---

<sup>1</sup>RDF also considers a special type of objects to describe *anonymous resources*, called blank nodes in the RDF data model. These are out of scope of this document.

RDFS is the schema language for RDF. Unlike other data models, there is no separation between the data and schema in RDF/RDFS. All schema information in RDFS is defined within RDF itself by introducing a set of distinguished terms which are prefixed with `rdfs:`. All these RDFS terms can be grouped based on their usage as follows.

- **Resources:** These self-explanatory terms are used to define resources and classes of resources: `rdfs:Resource`, `rdfs:Class`, `rdfs:Literal`, `rdfs:Datatype`.
- **Relationships:** These terms are used to define the relationships between resources: `rdfs:range`, `rdfs:domain`, `rdfs:subClassOf`, `rdfs:subPropertyOf`.
- **Non-modeling and Utilities:** These include `rdfs:label`, `rdfs:comment`, `rdfs:seeAlso` and `rdfs:isDefinedBy`.

One of the key uses of RDFS is *inferencing*. Given the explicitly stated information, RDFS type and property propagation rules allow determining other, related information that can be considered as if it had been explicitly stated. Inferencing is a powerful mechanism in information processing that allows a database to *reason* about the data. To facilitate reasoning, in addition to RDFS, other, more elaborate RDF schema languages such as RDFS-Plus [28] and OWL [46] were developed.

### 2.1.2 Property Graph

The Property Graph Model [57] was introduced in 2010 by Rodriguez et al. It has become a popular data model for *native* graph databases<sup>2</sup>. The main feature of the Property Graph Model is to allow node and edge attributes. The attributes are key/value pairs that are attached to nodes and edges. Rodriguez et al. argue that property graphs are popular in current graph databases due to their versatility. Specifically, by simply abandoning or adding particular features to the model, many common graph types can be expressed. For example, by dropping the attributes and restricting node/edge labels to IRIs, RDF graph can be generated.<sup>3</sup> Similarly, by using weights as edge attributes weighted graphs can be generated.

We formally define a property graph as a directed, labeled, attributed multi-graph. The edges are directed, nodes and edges are labeled, key-value attributes (or, *properties*) are associated with both nodes and edges, and there can be multiple edges between any two nodes.

Consider an example property graph is shown in Fig.2.3a that describes and expands on some interesting facts about William Shakespeare. The design choice made in property graph model is that not every datum needs to be “related” to an entity; e.g., name, age, and date of birth. Instead, such properties are encoded

---

<sup>2</sup>We talk more on this in §2.2.

<sup>3</sup>This is not completely true as RDF also uses blank or anonymous nodes.

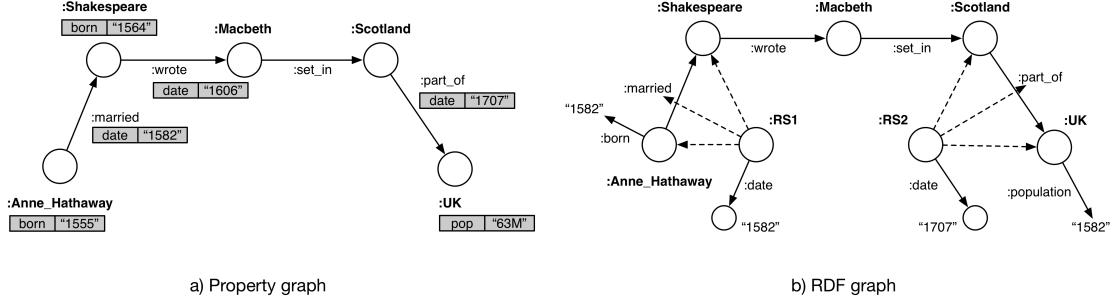


Figure 2.3: Example property vs. RDF graph.

as key/value attributes that are attached to the entity in question. Thus, property graphs provide finer granularity on the meaning of an edge as the key/value of the edge add extra information beyond the edge label.

On paper, a property graph is a *schema-less* data model. Rodriguez et al. state that ultimately, data is represented according to some schema whether that schema is explicit in the database, in the code interacting with the database, or simply left implicit. Often, such claims are justified by the need to support of evolving schemas in current “big data” applications. We think that, in this case, this argument is invalid, assuming that the consistency in the database is as important as the support for the flexible schema. The optional schema in RDF seems like a good example of what can be done in a distributed graph setting with relaxed schema structures.

A single advantage of property graph over RDF comes from the ability to add attributes to edges. This is useful in practice as many relationships between entities have literal properties. In RDF, in order to add a literal property to an edge, this

edge needs to be reified. This is an unnecessarily cumbersome procedure in the process of the modeling of the domain.<sup>4</sup>

## 2.2 Graph Serialization

With increasing interest in graph technologies, many developers have faced the challenge of how to rapidly implement an efficient and robust graph storage. While some opted for “native” solutions, many turned to relational databases in order to provide the core of an RDF storage system. This decision is not surprising given that relational technologies offer a four decade-worth of research and development, great query performance, and good scalability. However, unlike native solutions, relational implementations require intricate mechanisms that link graph and relational data models. The following subsections provide an overview of different proposals in this area of research.

### 2.2.1 Relational

#### 2.2.1.1 Single Table

A generic solution to store graph data in a relational database is to use a single 3-attribute table. In this table, for each triple in an RDF dataset, a single (**subject**,

---

<sup>4</sup>As illustrated in Figures 2.3a and 2.3b. In RDF, reified statements (e.g. **RS1** and **RS2**) need to be added in order to be able to state a literal fact about a relationship.

The figure consists of two tables. The top table is a triple store where each row represents a triple (Subject, Predicate, Object). The bottom table is a statement store where each row represents a statement with an ID and a URI/Literal.

Subject	Predicate	Object
http://..	http://..	"Literal"
http://..	http://..	http://..
http://..	http://..	"Literal2"
http://..	http://..	"Literal5"
http://..	http://..	http://..
http://..	http://..	"Literal2"
..	..	..

ID	URI/Literal
1	http://..
2	http://..
3	http://..
..	..

Figure 2.4: RDF storage based on a single table.

`(predicate, object)` tuple is stored. For data indexing and compression purposes, long strings such as IRIs may be replaced with short unique numerical identifiers as shown in Fig. 2.4.

All queries that are issued by an end user at an RDF endpoint are translated to SQL and sent to a backend database. Queries with more elaborate graph patterns, however, would require multiple self-joins over `Statements` table which can lead to significant performance issues. While single table storage implementation can efficiently handle highly heterogeneous data and single triple pattern queries, its biggest disadvantage is the performance penalty incurred by the self-joins associated with SPARQL queries that have elaborate graph patterns.

### 2.2.1.2 Property Tables

To reduce the number of self-joins, Wilkinson et al. devised an RDF storage method that utilizes *property tables* [63]. An idea behind this approach is the exploitation of the fact that many RDF datasets often have frequently occurring patterns of statements (*regularities*). Regularities are the properties that are frequently associated together with the majority of the resources in the dataset. For example, an employee dataset might include for each employee, an employee number, a name, location, and phone. Then, such common properties can be modeled as attributes of the same table, which is called a property table.

The attributes of a property table consist of a primary key of a resource (typically an IRI) and a set of properties that are commonly associated with the resources in the dataset. For example, one may notice that in the dataset presented in Fig.2.4 properties **Type**, **Title** and **Copyright** appear in the majority of the resources. Thus, these can be used to construct a property table shown in the middle of Fig.2.4.

When a property is not associated with a resource, it is reflected by a **NULL** value in the property table. In the example dataset, properties **Author**, **Artist** and **Language** are not attributes of the property table. These properties are associated with minority of the resources, therefore their inclusion in the property table would lead to a lot of **NULL** values, which is a waste of space. Triples excluded from the

**Property Table**

Subject	Predicate	Object
ID1	type	BookType
ID1	title	"XYZ"
ID1	author	"Fox, Joe"
ID1	copyright	"2001"
ID2	type	CDType
ID2	title	"ABC"
ID2	author	"Orr, Tim"
ID2	copyright	"1985"
ID2	language	"French"
ID3	type	BookType
ID3	title	"MNO"
ID3	language	"English"
ID3	language	"French"
ID4	type	DVDType
ID4	title	"DEF"
ID5	type	CDType
ID5	title	"GHI"
ID5	copyright	"1995"
ID6	type	BookType
ID6	copyright	"2004"

Subject	Type	Title	Copyright
ID1	BookType	"XYZ"	"2001"
ID2	CDType	"ABC"	"1985"
ID3	BookType	"MNO"	NULL
ID4	DVDType	"DEF"	NULL
ID5	CDType	"GHI"	"1995"
ID6	BookType	NULL	"2004"

Subject	Predicate	Object
ID1	author	"Fox, Joe"
ID2	artist	"Orr, Tim"
ID2	language	"French"
ID3	language	"English"
ID3	language	"French"

Subject	Predicate	Object
ID2	language	"French"
ID3	language	"English"
ID3	language	"English"
ID4	type	DVDType
ID4	title	"DEF"

Figure 2.5: RDF storage based on property tables.

property table are placed into *excess table*, which has (subject, predicate, object) format.

In the example in Fig. 2.5, the `language` property can be both “English” and “French” for multilingual media. Due to First Normal Form [2] requirement in the relational model, such multi-valued properties cannot be included in the property table, so would have to be placed into excess table.

An improvement on this RDF storage approach is to consider multiple property tables. Based on the statistical evaluation of a given RDF dataset, a property clustering algorithm can be used to decide on the number and configuration of

property tables. The schema information can also be used in order to identify the structure of property tables. In particular, RDFS classes are used to group resources with the same structure. Hence, it is worthwhile to use these classes as guidelines to build property tables. Such tables are called *property-class tables*; an example is given on the right side of Fig. 2.4.

The advantage of this RDF storage approach is that queries that involve clustered attributes can be answered without joins. This approach seems to work well on well-behaved and highly structured data, as then the schema information can be used to build efficiently the property tables.

However, while such data structure and query workloads may be dominant in some applications, in many situations this storage approach will underperform. For instance, a performance penalty is incurred when the data from different property tables needs to be combined. Since the main feature of RDF is that it was designed to model highly heterogeneous data, it is usual that most interesting queries would involve joins over property tables. One way to reduce the likelihood of such “bad” queries is to make the property tables wider. However, since RDF data is not structured, this will contribute to the sparseness of the property table, which, in turn, will impose a significant performance overhead.

The property table technique can significantly improve performance by dramatically reducing the number of self-joins when compared to the single table approach.

However, this scheme requires careful property clustering in order to create property tables that are not too wide, while still not being too narrow in order to be able to answer most queries directly.

#### 2.2.1.3 Vertical Partitioning

As an alternative to property tables, Abadi et al. proposed a vertically partitioned RDF storage approach [1]. In this *fully decomposed* storage model [15], each unique property in the RDF dataset gets a table. In each of these tables, the first column contains the subjects that define that property and the second column contains the objects for those subjects. An example decomposition is shown in Fig. 2.6.

The vertically partitioned approach has the following advantages over the property table technique. First, multi-valued attributes are fully supported and no longer problematic. In this storage model, if a subject has more than one object assigned via particular property, then each object is listed in a successive row in the corresponding table for that property. Second, if no subject is associated with an object for a particular property, then the corresponding row is simply omitted from the table. This allows for smaller tables, as **NULL** data is not explicitly stored. Finally, the vertically partitioned approach is simpler to implement as no property clustering algorithms are needed.

Despite its advantages, the vertically partitioned approach is not free from limi-

Subject	Predicate	Object
ID1	type	BookType
ID1	title	"XYZ"
ID1	author	"Fox, Joe"
ID1	copyright	"2001"
ID2	type	CDType
ID2	title	"ABC"
ID2	author	"Orr, Tim"
ID2	copyright	"1985"
ID2	language	"French"
ID3	type	BookType
ID3	title	"MNO"
ID3	language	"English"
ID3	language	"French"
ID4	type	DVDType
ID4	title	"DEF"
ID5	type	CDType
ID5	title	"GHI"
ID5	copyright	"1995"
ID6	type	BookType
ID6	copyright	"2004"

Type	
Subject	Object
ID1	BookType
ID2	CDType
ID3	BookType
ID4	DVDType
ID5	CDType
ID6	BookType

Title	
Subject	Object
ID1	"XYZ"
ID2	"ABC"
ID3	"MNO"
ID4	"DEF"
ID5	"GHI"

Copyright	
Subject	Object
ID1	"2001"
ID2	"1985"
ID5	"1995"
ID6	"2004"

Author	
Subject	Object
ID1	"Fox, Joe"

Artist	
Subject	Object
ID1	"Orr, Tim"

Language	
Subject	Object
ID2	"French"
ID3	"English"

Figure 2.6: RDF storage based on vertical partitioning.

tations. Queries with several properties would require more joins than the property table technique. Although, these joins are fast merge-joins, they are not free, and still slower than single table sequential access. Also, when an object or a subject is updated, all different tables for corresponding properties need to be updated as well.

In summary, the vertically partitioned approach provides similar performance to the property table technique while being simpler to design and implement. This approach performs even better in a column-oriented database. Despite its advan-

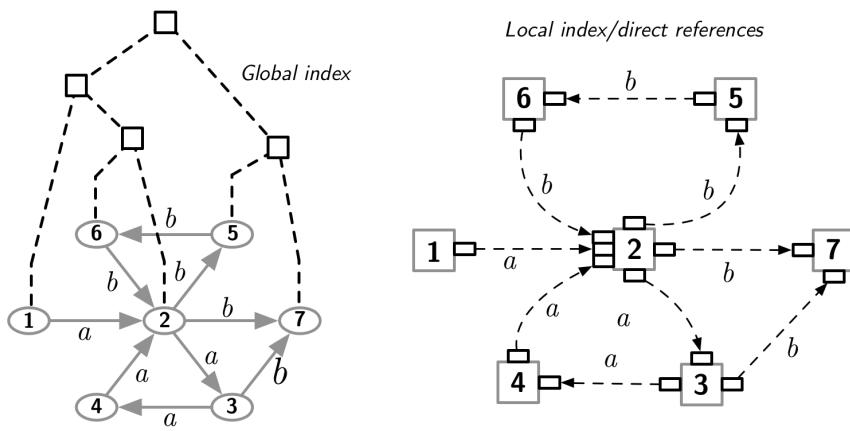


Figure 2.7: Relational vs. native physical graph storage.

tages, for some datasets and query workloads, vertically partitioned scheme can under perform when compared to single or property table approaches.

### 2.2.2 Native Graph Databases

We call a graph database *native* when it satisfies the *index-free adjacency* property: each node stores information about its neighbors as direct references or a local index in addition to an optional global index of connectedness between nodes. In contrast, in relational graph databases, a graph is represented as a globally indexed collection of triples.

At the root of many graph algorithms is the procedure called *graph walk*. A graph walk is a “traversal” of the nodes in a graph. During this walk, information about graph is processed, and, in general, an algorithm which solves a problem on

a graph can be computed.

Consider an execution of a walk over a graph that is stored in a relational database and a graph stored in a native database as presented in Fig.2.7. In order to make a “jump” from one node to another in the relational graph, an algorithm has to consult a global index to identify the neighbors of this node. Since most indexes are implemented as trees, the complexity of a jump, in this case, is  $O(\log n)$ , where  $n$  is a number of entries in the index; e.g., a number of nodes. On the other hand, in a native database, each node has direct references to its neighbors. This means that, in this case, each jump takes constant ( $O(1)$ ) time.

Hence, if an algorithm executes a graph walk that involves many jumps, its overall complexity will be lower in a native graph database. Further, native jumps are independent of the number of nodes in the graph, which becomes important as graph size increases.

Many graph databases today (Neo4j<sup>5</sup>, DEX<sup>6</sup>, and AllegroGraph<sup>7</sup>) implement index-free adjacency, and thus, can be considered being native graph databases.

---

<sup>5</sup><http://neo4j.org/>

<sup>6</sup><http://www.sparsity-technologies.com/>

<sup>7</sup><http://franz.com/agraph/allegrograph/>

## 2.3 Path Queries

Besides allowing for natural modeling of the data, graphs enable the specification of powerful query languages. We enumerate the desired features of graph query languages as follows<sup>8</sup>. The first type of query which is often used in information retrieval, the Semantic Web, and life sciences is *adjacency* query. Two nodes are adjacent if there is an edge between them. Similarly, two edges are adjacent if they share a common node. Such queries may test whether nodes/edges are adjacent; check the k-neighborhood of a node, or list all neighbors. The second type is *pattern matching* queries. These aim to find all sub-graphs in a database that are isomorphic to a given query pattern graph. Such queries are useful in many data retrieval tasks. *Summarization* queries are the third type. These queries summarize or operate on query results, and typically return a single value. For example, aggregation queries such as average, count, min, and max are included in this group. In addition, we include queries which allow functions that compute some properties of the graph or its elements such as the distance between nodes, the diameter of a graph, and node degree. The last type of query deals with problems related to *paths* or *reachability* in a graph. These queries identify or test if nodes are reachable by following a specified path. In general, we consider two types of paths: fixed and arbitrary-length. Fixed paths contain a predefined number of nodes and

---

<sup>8</sup>We borrow this classification from an excellent survey [5] by Angles et al.

edges. In contrast, arbitrary-length paths are not fixed in terms of the number of nodes and edges but specify more relaxed restrictions on paths, usually by using regular expressions. These are called *regular path queries* (RPQs). RPQs allow queries to evaluate regular expressions over graph data and have been recently included in the specification of popular graph-oriented query languages such as SPARQL [54] and Cypher [17].

A *graph database*  $G$  can be defined as  $\langle V, \Sigma, E \rangle$  for which  $V$  is a finite set of nodes (vertices),  $\Sigma$  is a finite *alphabet* (a set of *labels*), and  $E$  is a set of *directed, labeled edges*,  $E \subseteq V \times \Sigma \times V$ .

A *path* in a graph is defined as a sequence  $p = n_0a_0n_1\dots n_{k-1}a_{k-1}n_k$  such that  $n_i \in N$ , for  $0 \leq i \leq k$ , and  $\langle n_i, a_i, n_{i+1} \rangle \in E$ , for  $0 \leq i < k$ . The path-induced *path label*  $\lambda(p)$  is the string  $a_1a_2\dots a_k \in \Sigma^*$  (for which  $\Sigma^*$  is a set of all finite strings formed over  $\Sigma$ ). Each node  $n \in V$  is associated with an *empty path*,  $n$ , the path label of which is the empty string, denoted by  $\varepsilon$ .

A *regular expression* over alphabet  $\Sigma$  is defined inductively, as follows: 1. the empty string  $\varepsilon$  and each symbol  $r \in \Sigma$ ; and, 2. given regular expressions  $r$ ,  $r_1$ , and  $r_2$ , then (a) the concatenation  $r_1r_2$ , (b) the disjunction  $r_1|r_2$ , and (c) Kleene star  $r^*$ . The *regular language* defined by the regular expression  $r$  is denoted by  $L(r)$ . The regular language is defined inductively, as follows: 1.  $L(\varepsilon) = \{\varepsilon\}$  and  $L(a) = \{a\}$ , for each  $a \in \Sigma$ ; and, 2. for inductively combining strings, (a)  $L(r_1r_2) = L(r_1) \cdot L(r_2)$ ,

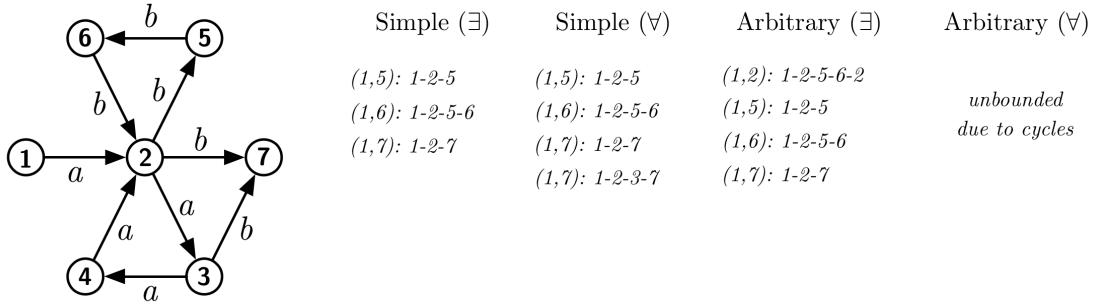


Figure 2.8: Different solution semantics in RPQs.

(b)  $L(r_1|r_2) = L(r_1) \cup L(r_2)$ , and (c)  $L(r*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i$ .

A *regular path query*  $Q$  is a tuple  $\langle x, r, y \rangle$  for which  $x$  and  $y$  are free variables (that range over nodes) and  $r$  is a regular expression. An *answer* to  $Q$  over graph  $G = \langle V, \Sigma, E \rangle$  is a pair  $\langle s, t \rangle \in V \times V$  such that there exists an arbitrary path  $p$  from node  $s$  to node  $t$  for which the path label  $\lambda(p)$  is in language  $L(r)$  ( $\lambda(p) \in L(r)$ ). The *answer set* of  $Q$  over graph  $G$  is the set of all answers of  $Q$  over  $G$ .

### 2.3.1 Path Semantics

The semantics of conforming paths can either be *simple* or *arbitrary*. A simple path cannot go through the same node twice, whereas arbitrary path does not have this restriction. In addition, the solution of an RPQ can be either a *bag* or a *set* of variable bindings. In literature, these are also known as *counting* ( $\forall$ ) and *existential* ( $\exists$ ) semantics for query solutions. For example, consider graph database

$G$  presented in Fig. 2.8. Suppose, we evaluate the query  $Q = \langle 1, a+b+, y \rangle$ , where  $x = 1$  is fixed and  $y$  is free. Then, the query solutions that correspond to different semantics are as follows:

- **Simple/ $\exists$ :** In this scenario, we match paths between nodes to simple paths and return only a single pair of nodes  $(s, t)$  even if there are multiple simple paths between  $s$  and  $t$ . In our example, we return three solutions in total.
- **Simple/ $\forall$ :** Here, we match simple paths, but *count* the solutions, i.e. return as many duplicate pairs  $(s, t)$  as there are conforming simple paths between nodes  $s$  and  $t$ . In our example, we return all the solutions from **Simple/ $\exists$** , plus a duplicate pair  $(1, 7)$  which correspond to two different simple paths in  $G$ .
- **Arbitrary/ $\exists$ :** This case is similar to **Simple/ $\exists$** , but we drop the requirement for paths to be simple, i.e. a path is conforming as long as its path-induced label belongs to the language defined by a given regular expression. In our example, we return all the solutions from **Simple/ $\exists$**  plus a pair  $(1, 2)$  which path satisfies  $L(a+b+)$ , but is not simple since node 2 is visited more than once.
- **Arbitrary/ $\forall$ :** In this scenario, we drop the requirement for paths to be simple, but require solutions to be counted. That is, when evaluating RPQs one can obtain several duplicates for the same solution, essentially one duplicate for

every different path in the graph satisfying the expression. Since graphs containing cycles may lead to an infinite number of paths, this scenario requires a cycle elimination procedure to be defined. Due to cycles in  $G$  in our example, the potential number of solutions is unbounded.

The semantics of RPQs have a great impact on the complexity of query evaluation and thus need to be considered carefully in the design of a query language.

### 2.3.2 Paths in SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) [29] is a query language for RDF. A SPARQL query consists of a set of *variables* and a *graph pattern* used for matching within the RDF graph. In the SPARQL 1.0 standard, graph patterns only allow simple navigation in the RDF graph, by matching nodes over fixed-length paths. Under the proposed SPARQL 1.1 standard, the W3C working group has greatly extended the navigational capabilities of SPARQL queries by allowing graph patterns that include regular expressions in the form of *property paths*, which enable matching of nodes over arbitrary-length paths, and which allow a limited form of negation.

We use the terminology from [7]. Consider the following pairwise disjoint, infinite sets:  $I$  (IRIs),  $B$  (blank nodes),  $L$  (literals) and  $V$  (variables). The proposed SPARQL 1.1 standard [29] defines a property path recursively as follows:

1. Any  $a \in I$  is a property path;
2. Given property paths  $p_1$  and  $p_2$ , expressions  $p_1/p_2$ ,  $p_1|p_2$ ,  $\hat{p}_1$ ,  $p_1^*$ ,  $p_1^+$  and  $p_1?$  are also property paths;
3. Given  $a_1, \dots, a_n \in I$ , expressions  $!a_1$ ,  $!\hat{a}_1$ ,  $!(a_1| \dots |a_n)$ ,  $!(\hat{a}_1| \dots | \hat{a}_n)$  and  $!(a_1| \dots |a_j|\hat{a}_{j+1}| \dots | \hat{a}_n)$  are property paths.

Hence, property paths are regular expressions over vocabulary  $I$  of all IRIs, for which “/” is concatenation, “|” disjunction, “ $\hat{\cdot}$ ” inversion, “ $*$ ” Kleene star (zero or more occurrences), “ $^+$ ” one or more occurrences, and “ $?$ ” zero or one occurrence.

Negated property paths are not supported, but negation on IRIs, inverted IRIs and disjunctions of combinations of IRIs and inverted IRIs is allowed. A property path triple is a tuple  $t$  of the form  $(u, p, v)$ , where  $u, v \in (I \cup V)$  and  $p$  is a property path. Such a triple is a graph pattern that matches all pairs of nodes  $\langle u, v \rangle$  in an RDF graph that are connected by paths that conform to  $p$ .

Initially, W3C had adopted simple path semantics for arbitrary-length property paths in SPARQL 1.1 for the “ $*$ ” and “ $^+$ ” operators. W3C had also required paths to be *counted*; i.e., to report the number of duplicate pairs  $(a, b)$  that correspond to a number of paths between  $a$  and  $b$  that conform to  $p$ . However, [7, 44] have shown that both of these requirements are computationally infeasible in many cases. These observations led W3C to drop both simple path and path counting requirements in favor of regular paths and existential semantics.

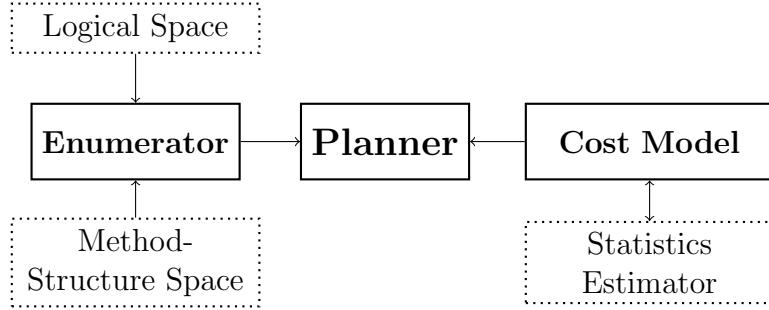


Figure 2.9: Query optimizer architecture.

### 2.3.3 Paths in Cypher

Cypher [17] is a graph-oriented declarative query language used in a popular native graph database Neo4j.<sup>9</sup> It is a powerful language that supports adjacency, path, pattern matching and summarization queries. Similar to SPARQL property paths, paths in Cypher are defined in terms of regular expressions, yet with slightly different syntax and evaluation semantics. Cypher path queries essentially represent a limited subset of RPQs with simple paths and counting semantics.

## 2.4 Query Planning

In general, a query can be executed in many different ways by the database engine. The strategy which is used by the database during query execution is encoded in a *query plan*. Often, the *costs* of such plans can vary by orders of magnitude, which motivates the problem of choosing the plan with the lowest possible execution cost.

---

<sup>9</sup><http://www.neo4j.com>

*Query optimization* is a large research area in the database field which attempts to find answers to this problem. It is an established area which has been surveyed extensively [32]. The scope of this document is to provide an overview of an architecture of a *query planner*, which is an essential part of any query optimizer.

Query planner is responsible for examining all possible execution plans for each query and selecting a *cheapest* plan to be executed. Candidate plans are considered in a certain order as provided by the *enumerator* module. Often, the space of candidate plans grows exponentially with respect to the size of the given query, making it infeasible to consider every single plan. Hence, it is the task of the enumerator to provide a sufficient number of candidate plans so that the plan close to optimal can be found, while pruning those candidate plans which are unlikely to have the lowest cost.

The plan space which is considered by the enumerator is determined by its *logical* and *method-structure* components. Given the formula which is obtained by parsing the query, logical space contains all different execution orders of formula's operators which can be followed to answer the query. This is called a *logical* plan. Given a logical plan, the concrete implementation choices for each of the operators are determined by the method-structure space, to produce a collection of *physical* plans.

A cost of a physical plan is estimated by a *cost model* module. This module

specifies the cost formulas which are used to estimate the cost of execution strategies used to evaluate operators in a physical plan. These cost formulas take into account the execution method used, the amount of resources available (such as processor, memory buffer pool, and disk space), the catalog information (such as available indexes and tuple sizes), and the statistical estimates gathered from participating datasets.

## 2.5 Related Work

We describe the methods used in the evaluation of regular path queries. Recall that RPQs have the general form  $Q = (x, L(r), y)$ , where  $x$  and  $y$  are free variables and  $L(r)$  is the regular language defined by regular expression  $r$ . In this work, we consider existential arbitrary path semantics  $\text{Arbitrary}/\exists$  of RPQ solutions.<sup>10</sup> Then, an evaluation of  $Q$  over a graph database  $G = (V, E)$  is a *set* of variable bindings  $(s, t)$  such that the arbitrary path between nodes  $s$  and  $t$  conforms to given expression  $r$ .

Depending on whether variables  $x$  and  $y$  are fixed or free, we consider three types of RPQs: open-ended, half-open, and reachability. Open-ended queries have both variables free, hence they identify all pairs of nodes in  $G$  that are connected with a path specified by the expression. These queries are often used in graph exploration

---

<sup>10</sup>As discussed in §2.3.1

and analytical contexts. Half-open queries fix one of the variables, while the other remains free. Such queries find all nodes reachable from a specific node in a graph by following a given path. Finally, both variables are fixed in reachability queries. The answer to this kind of a query is “yes” or “no” depending on the existence of a path between two given nodes that satisfies the regular expression.

The literature on path queries over graphs, as is pertinent to property paths, comes from two distinct sources: evaluation strategies for *regular path queries* (RPQs); and SPARQL platforms extended for version 1.1 to handle *property paths*. We consider each in turn.

### 2.5.1 FA Plans

Research on RPQs, which well precedes RDF and SPARQL, mostly focused on theoretical aspects, but little on performance issues for evaluating such queries in practice. The seminal work that introduced the G+ query language [48] exploited the natural observation that where there is a regular expression, there is a *finite automaton* (FA) that is a *recognizer* for it. They showed how to use finite state automata to direct search over the graph to evaluate an RPQ. In essence, an FA corresponding to the query’s regular expression provides a plan for its evaluation. Subsequent work on RPQs followed on this idea. Let us call this the FA approach.

Regular expressions are a formal notation for patterns that *generate* strings—

called *words*—over an *alphabet*. The set of words that a given regular expression can generate is called its *language*. The dual to generation is *recognition*. *Finite state automata* are the recognition counterpart to regular expressions. For any given regular expression, a finite state automaton—abbreviated as *finite automaton*—can be constructed that will *recognize* the words over the alphabet that belong to the expression’s language. Thus, an FA  $A$  can be constructed to recognize the language of a given regular expression  $r$ . One can construct one such FA by traversing the parse tree of  $r$  bottom up, and combining the automata that recognize sub-expressions of  $r$  into a composite automaton via the union, concatenation, and closure of the sub-automata as is appropriate.

**Example 2** Recall query  $\mathcal{Q}_{1.3}$  from Ex. 1. As shown in Fig. 2.10, an automaton construction for this query is a two-step procedure. First, traversing the parse tree of  $r$  bottom up, the  $\varepsilon$ -NFA is built up, by the base case and the inductive rules. Second, the resulting  $\varepsilon$ -NFA is then minimized to an NFA, which typically has the smaller size, and hence, is more efficient to process.

The first algorithm to use automata to evaluate regular expressions on graphs was presented in [48] as a part of an implementation of the **G+** query language. Given a graph database  $G = (V, E)$  and a query  $Q = (s, r, t)$  in which  $s$  and  $t$  are nodes in  $G$ , the algorithm proceeds as follows. The expression  $r$  is converted into a finite automaton  $A_Q$  by using the bottom-up traversal of parse tree of  $r$ , as

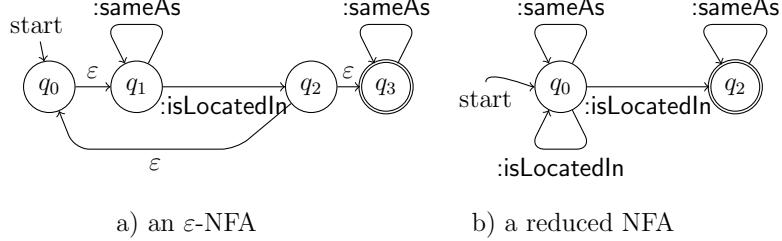


Figure 2.10: An  $\varepsilon$ -NFA and corresponding reduced NFA for  $\mathcal{Q}_{1.3}$ .

discussed. Then, the graph database  $G$  is converted to finite automaton  $A_G$  with graph nodes becoming automaton states and graph edges becoming transitions. Node  $x$  is assigned to be the initial state, and  $y$  is assigned to be the accepting state in  $A_G$ . Finally, given  $A_G$  and  $A_Q$ , a product automaton  $P = A_G \times A_Q$  is constructed.  $P$  is then tested for non-emptiness, which checks whether any accepting state can be reached from the initial state. If the language defined by  $P$  is not empty, then the answer for the reachability query  $(s, r, t)$  on graph  $G$  is “yes”; i.e., there exists a path between  $s$  and  $t$  in  $G$  that conforms to  $r$ . This idea of employing a product automaton for RPQ evaluation over graphs has been used in [14, 35, 37, 48, 53, 66]. We briefly discuss two main approaches in detail below.

Kochut et al. [35] proposed an evaluation method that is based on bidirectional breadth-first search (BFS) in the graph which works as follows. Given a reachability query, two automata are constructed. The first one accepts the regular language defined by the original path expression, while the second one accepts the reversed language, which is also regular. The path search uses the steps from the bidirectional

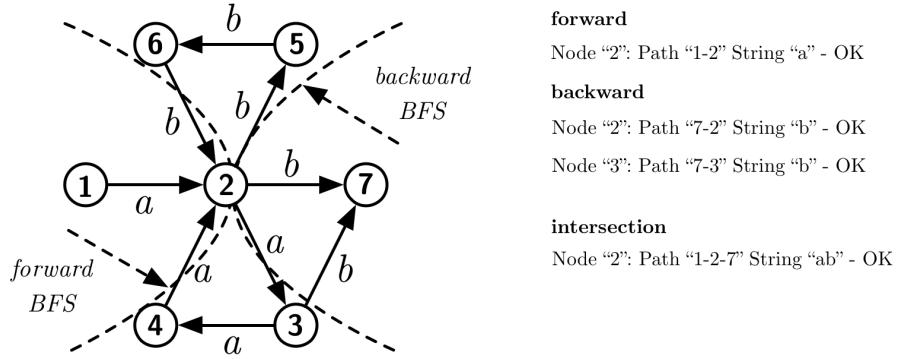


Figure 2.11: Example run of an algorithm proposed by Kochut et al.

tional BFS to expand the frontiers of entities used to connect paths. Before each entity is placed on the frontier for the next expansion, a check is performed if the partial path leading to it is not rejected by the appropriate automaton. This guarantees that the partial results which are not accepted by the automaton will not be expanded further. A candidate path is located when an entity from the forward frontier matches an entity from the reverse frontier. At this point, it is only known that the “forward” sub-path has not been rejected by the forward automaton and that the “reverse” sub-path has not been rejected by an automaton that accepts the reverse language. Before the concatenated path is returned, it must be accepted by the forward automaton, created from the original path expression. For half-open queries, a similar solution is used. In this case, only one automaton in conjunction with a standard breadth first search is used to grow a single frontier of entities. An

example execution of this algorithm on graph database  $G$  presented in Fig.2.8 and reachability query  $Q = (1, a+b+, 7)$  is shown in Fig. 2.11.

Koschmieder et al. [37] present an RPQ evaluation method that searches the graph while simultaneously advancing in the query automaton. This is achieved by exploring the graph using a breadth-first search strategy while marking the nodes in the graph with the states in the automaton. A search state (one specific point during the search process) consists of the current position in the graph (node identifier) and the current state of the automaton. For every state in the automaton, a *labeled follow set* is created. This construction consists of two lists. The first list contains all edge labels that are accepted in that state, and the second list shows into which states the automaton may transition for each label. Then, when traversing an edge during BFS, its label is checked whether it is in the follow set of the current state. In the positive case, the transition is made and new entries are added to the end of the list of search states to be processed. One search state is created for each entry in the list. For every node in the graph, the states in which a search passed this node are kept. This is used to find completed paths as well as to prevent cycles in the result paths. The search ends once the list of unprocessed search states is empty. An example run of this algorithm on graph database  $G$  presented in Fig.2.8 and half-open query  $Q = (1, a+b+, y)$  is shown in Fig. 2.12.

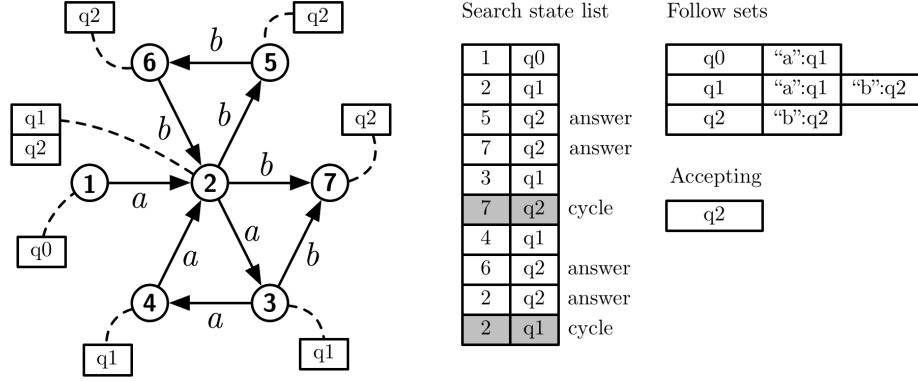


Figure 2.12: Example run of an algorithm proposed by Koschmieder et al.

### 2.5.2 Alpha-RA Plans

Work on evaluating property paths—much newer by virtue of the fact that the SPARQL 1.1 standard is quite recent—has mirrored the dynamic-programming approach behind the algorithm presented in the seminal work of [44]. This can be modeled by the *relational algebra* (RA) extended by an operator  $\alpha$  for *transitive closure* ( $\alpha$ -RA) [3]. The full power of relational algebra, extended with  $\alpha$ , can then be employed to devise an evaluation plan—an  $\alpha$ -RA-expression tree—based on the regular expression of the property path. This general approach is found behind many SPARQL platforms, as it follows relational techniques well. For example, VIRTUOSO [21], a leading SPARQL system which is also a well-established relational database system, extended their platform to accommodate property paths essentially by adding an “ $\alpha$ ” operator to the engine.

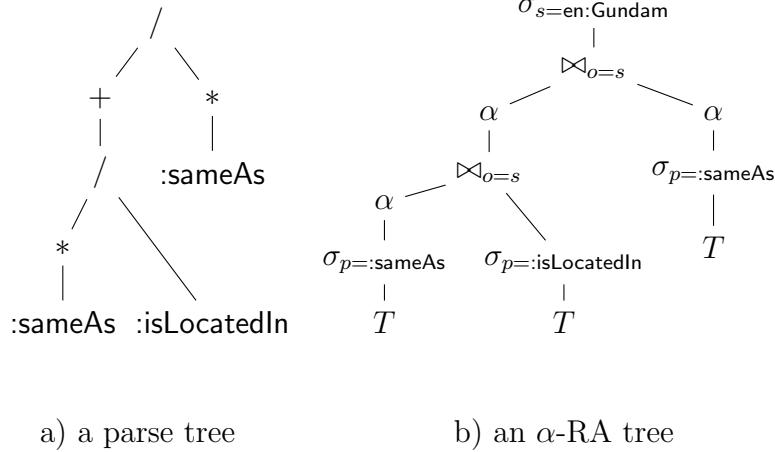


Figure 2.13: A parse tree and  $\alpha$ -RA tree for query  $\mathcal{Q}_{1.3}$ .

The  $\alpha$  operator computes the transitive closure of a relation. Let the graph database be represented as a relation of triples  $G(s, p, o)$ . Let  $T = \pi_{1,3}(G)$ ; thus  $T$  consists of pairs of nodes  $\langle s, o \rangle$  such that the pair is connected by a directed edge in the graph. Then  $\alpha$  applied to  $T$  computes the *least fixpoint*:

$$T^+ = T \cup \pi_{1,3}(T^+ \bowtie_{T^+.o=T.s} T) \quad (\mathcal{E}_{2.1})$$

Thus,  $\alpha(T)$  results in all pairs of nodes such that, for the nodes of each pair, there exists a path between them in the graph (denoted by)  $G$ . If we were to evaluate the fixpoint by a semi-naïve evaluation, each iteration of evaluation is over paths of length one greater than of the previous iteration. The process stops when no new pairs are added; i.e., the fixpoint has been reached.

Given the SPJRU (select-project-join-rename-union) relational algebra extended

with the  $\alpha$  operator, one can evaluate the RPQ  $Q = (x, L(r), y)$  over graph  $G = (V, E)$  by the algorithm proposed in [44]. This traverses the syntax tree of expression  $r$  bottom-up. Let  $s$  be the sub-expression of  $r$  represented by a given node in a parse tree.

The binary relation  $R_s \subseteq V \times V$  is computed so that node pair  $(u, v) \in R_s$  iff there exists a path from  $u$  to  $v$  in  $G$  matching  $s$ . The manner in which the relations are joined going bottom-up in a parse tree depends on the type of the node. The cases are as follows:

1. If  $s$  is a  $\Sigma$ -symbol, then  $R_s := \{(u, v) | (u, s, v) \in E\}$ .
2. If  $s = \varepsilon$ , then  $R_s := \{(u, u) | u \in V\}$ .
3. If  $s_1$  and  $s_2$  are sub-expressions and  $s = s_1 | s_2$ , then  $R_s = R_{s1} \cup R_{s2}$ .
4. If  $s_1$  and  $s_2$  are sub-expressions and  $s = s_1 \cdot s_2$ , then  $R_s = \pi_{1,3}(R_{s1} \bowtie_{R_{s1}.2=R_{s2}.1} R_{s2})$ .
5. If  $s = s_1^*$ , then  $R_s$  is the reflexive and transitive closure of  $R_{s1}$ , or  $R_s = \alpha(R_{s1}) \cup R_{s1}$ .
6. If  $s = s_1^+$ , then  $R_s$  is the transitive closure of  $R_{s1}$ , or  $R_s = \alpha(R_{s1})$ .

(Correctness of this algorithm is established in [44]). The correctness of this algorithm can be proved by performing a structural induction and showing that the following invariant holds for every relation  $R_s$  that is calculated: For each sub-expression  $s$  of  $r$ , we have  $(u, v) \in R_s \iff \exists$  path  $p$  in graph  $G$  from  $u$  to  $v$  such

that its induced label  $\lambda(p)$  belongs to language  $L(s)$ .

**Example 3** Given query  $\mathcal{Q}_{1.3}$  and the database  $G$  from Ex. 1, the corresponding  $\alpha$ -RA tree is shown in Fig. 2.13.

The  $\alpha$ -RA-based RPQ evaluation can be directly implemented in most relational databases and relational triple-stores. In [65], we proposed a method that translates RPQs as defined by SPARQL property paths into recursive SQL. A similar approach was used by Dey et al. [19] in the context of the evaluation of provenance-aware RPQs by a relational engine.

### 2.5.3 Index-based Evaluation

This approach, which is *orthogonal* to query planning, uses specialized data structures (e.g., *indexes*) in the evaluation of RPQs. In these methods, parts of regular expressions, which are frequent in a given query workload, are precomputed and stored in indexes to speed up the subsequent RPQ evaluation. One of the main challenges in using indexes in the evaluation of RPQs has to do with the presence of transitive closures. The size of a transitive closure may be orders of magnitude larger than the size of a graph it is computed on. Hence, one needs to be careful in designing the index structure which balances its construction time, storage space consumption, and speed of retrieval (and, hence, the effectiveness of RPQ evaluation).

In [27], authors implement RPQ evaluation in open-source RDF store RDF-3X [51] by using highly-compact FERRARI *reachability index* [60]. In this implementation, the FERRARI index provides a concise representation of the transitive closure of the graph. The compactness of the FERRARI is achieved by adaptively compressing the transitive closure using exact and approximate intervals. Then, the query processor of RDF-3X is modified to support property path queries through invocation of the FERRARI index for Kleene expressions in the given query. The main disadvantage of this approach is that it requires the construction of the index for all Kleene expressions which are used in the query workload. While this might be feasible in limited applications for Kleene expressions over single labels (as it might be few of them), the construction of the indexes for all possible permutations of labels in more complex expressions is less practical due to space limitations.

In [23], authors aim to overcome the requirement of construction of reachability indexes for all Kleene expressions in the given query workload. Instead, given RPQ is rewritten as a union of label paths, thus removing Kleene expressions from the query completely. Then, each path is evaluated by using precomputed  $k$ -path indexes, which index all paths up to length  $k$  in the graph.  $k$ -path histograms are used to optimize join ordering of  $k$ -path indexes during the evaluation. This approach essentially reduces Kleene fragments of regular path queries to unions of label concatenations. This requires *bounding* Kleene recursions to prespecified path

lengths. While often sufficient in many practical settings, this requirement might not be suitable for some applications when complete closure needs to be computed.

#### 2.5.4 Relational Query Optimizers

Research on query optimization has been largely focused in the context of relational databases. We briefly describe the research efforts in connection with each module in the query optimizer presented in §2.4.

A typical query in a relational database is written in SQL or its variants. In an optimizer, an SQL query is translated into a formula based on relational algebra which deals with operations on relations such as select ( $\sigma$ ), project ( $\pi$ ), and join ( $\bowtie$ ). In order to represent an evaluation order of relational formulas, syntax *trees* are used in which leaf nodes are participating relations and inner nodes are operations performed on these relations. For a given SQL query, many different relational formulas exist, each associated with its syntax tree, which evaluate it. These *equivalent* formulas can be obtained by repeatedly applying a number of *rewrites* based on algebraic properties of relational operators such as join reordering and selection pushdown. Hence, a logical plan space, in relational databases, consists of a collection of equivalent relational algebra trees.

Due to the exponential blow up of the logical plan space with respect to the size of the query, it is computationally infeasible to perform an exhaustive search

for an optimal operator tree. Hence, the task of the enumerator is to restrict the search space. One popular heuristic which is used in System R [11] is to only consider left-deep operator trees. In this scenario, an outer relation in all joins is always a base database relation and not an intermediate output of some relational operator. While this heuristic was used in early database optimizers as it reduced the plan space considerably, it has been shown [49] to miss an optimal plan in many practical situations. The consideration of a query graph has increased the complexity of the search due to expensive connectedness verification for all possible partitions of relations participating in a query. Hence, the enumerator should be “smart” to choose only those subsets of relations which are likely to span connected query subgraphs. Two classes of enumeration strategies which find an optimal relational operator tree have been considered: bottom-up enumeration via dynamic programming [49], and top-down enumeration through memoization [22]. Both strategies have been shown to have algorithms which are able to achieve comparable performance [22].

For each logical plan, many corresponding *physical* plans exist. A physical plan assigns concrete evaluation algorithms to be used for each relational operator specified in a logical plan. Given a physical plan, its evaluation cost can be estimated based on a *cost formula*. This formula is carefully tuned based on an evaluation algorithm, execution hardware and cardinalities of participating relations.

Cardinality and frequency distribution estimation of input and resulting relations is an important component of every query optimizer and an established research topic within an academic community (see [45] and [13] for extensive surveys).

A prevalent technique for estimating sizes of relations which is used by most commercial database systems is based on *histograms*. In a histogram of an attribute  $a$  of a relation  $R$ , the domain of values of  $a$  is partitioned into *buckets*. Within each bucket, a uniform distribution is assumed. For any bucket  $b$  in a histogram, if a value  $v_i \in b$ , then the frequency  $f_i$  of  $v_i$  is approximated by  $\sum_{v_j \in b} f_j / |b|$ . Such histogram is called *trivial* as it assumes the uniform distribution over the entire attribute domain. There are various histogram types which were proposed by researchers for estimation of attribute distributions in a relation. Early prototypes focused on trivial histograms [59], but had large errors since uniform distribution assumption rarely holds in real data. A better estimation is provided by *equi-width* histograms [36]. In these, regardless of the frequency of each attribute value in the data, the number of consecutive attribute values associated with each bucket is the same.

Another class of histograms, *equi-depth* [36], provide a lower worst-case and average error for a number of selection queries. In equi-depth histograms, the sum of frequencies of the attribute values associated with each bucket is the same,

regardless of the number of these attribute values in the data. In [50], multi-dimensional equi-width histograms are proposed which are suitable for answering multi-attribute selection queries.

In *serial* [33] histograms, the buckets group frequencies that are close to each other with no interleaving. These have been shown [33] to be optimal (under various criteria) for reducing the worst-case and the average error in equality selection and join queries. However, it takes time exponential in the number of buckets to identify the optimal histogram among all serial ones. Further, since there is typically no order correlation between attribute values and their frequencies, serial histograms tend to be quite large and require an additional index.

*End-biased* [33] histograms were introduced to address these shortcomings. In these, some number of highest frequencies and some number of lowest frequencies are maintained in individual buckets, while all the remaining middle frequencies are approximated together in a single bucket. Optimal end-biased diagrams take significantly less space, are easy to find in time linear to the number of buckets and are not too far away [33] from serial diagrams in cardinality estimation. Thus, many vendors are using end-biased histograms in their commercial RDBMS.

In addition to histograms, several other cardinality estimation techniques have been investigated. One class of methods [45] relies on approximating the frequency distributions by a parameterized mathematical distribution or a polynomial. These

approaches require very little overhead, but they might not be suitable for many types of real data which do not follow any mathematical function.

Another class of methods uses online *sampling* [52] in order to estimate the cardinalities of a query result. These methods typically produce highly accurate results, however at a cost of collecting and processing random samples of data. This cost is considerably higher than the cost of other estimation methods, which makes online sampling prohibitive in a real query optimization setting when estimates are required frequently.

## 3 Methodology

### 3.1 Motivation

Work on property path evaluation has been remiss in not drawing the connection to RPQs. How do the FA and  $\alpha$ -RA approaches compare? Does one subsume the other? Or are they *incomparable*? If so, a combined approach might be superior. A generalized approach might offer new plans that neither FA nor  $\alpha$ -RA can produce with superior performance.

Both the FA and  $\alpha$ -RA approaches effectively provide evaluation plans for property path queries. However, the *plan spaces* that are implicit in these approaches have not been considered. In FA, choosing a different (but still correct) automaton for the plan might offer a significantly more efficient plan. In systems taking the  $\alpha$ -RA approach, planning is done over the  $\alpha$ -RA expression tree that results from the property path's translation, but no planning specific to the semantics of property paths takes place.

The FA and  $\alpha$ -RA approaches each entail a *plan space*; that is, the plans collec-

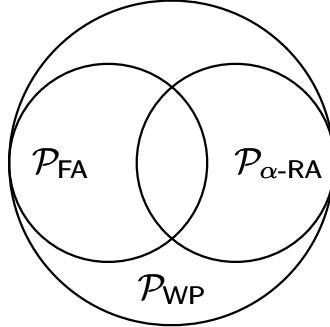


Figure 3.1: Plan space classes.

tively an approach produces over all possible property path queries. Let  $\mathcal{P}_{\text{FA}}$  and  $\mathcal{P}_{\alpha\text{-RA}}$  denote the plan spaces for FA and  $\alpha$ -RA, respectively. To understand how the approaches are related—for instance, whether one approach subsumes the other, or whether they are incomparable—we consider these plan spaces. The Venn diagram of how they are related is shown in Fig. 3.1.<sup>11</sup>

**Lemma 3.1**  $\mathcal{P}_{\text{FA}}$  and  $\mathcal{P}_{\alpha\text{-RA}}$  are incomparable ( $\mathcal{P}_{\text{FA}} - \mathcal{P}_{\alpha\text{-RA}} \neq \emptyset$  and  $\mathcal{P}_{\alpha\text{-RA}} - \mathcal{P}_{\text{FA}} \neq \emptyset$ ), but overlap ( $\mathcal{P}_{\text{FA}} \cap \mathcal{P}_{\alpha\text{-RA}} \neq \emptyset$ ).

**Proof:** Of course, we are taking liberties; the place spaces should be over the same *domain* of plans. As we have presented things, however, they are not; we have presented FA plans as automata and  $\alpha$ -RA plans as algebraic trees. To prove formally the lemma in Fig. 3.1, we would need to establish an isomorphism between

---

<sup>11</sup>The diagram's claim that the plan space of waveplans,  $\mathcal{P}_{\text{WP}}$ , properly subsumes both  $\mathcal{P}_{\text{FA}}$  and  $\mathcal{P}_{\alpha\text{-RA}}$  is taken up in §3.1.

$\text{FA}$  and  $\alpha\text{-RA}$  plans, or have a canonical form for plans to which each plan type could be mapped. This can be done [14]. The formalism for waveplans would suffice for this mapping. DATALOG, or the *relational algebra* extended by *while loops* (established to be expressively equivalent to DATALOG) [2], would provide an even more universal domain that would suffice.

Here, we formally establish that  $\text{FA}$  and  $\alpha\text{-RA}$  spaces are distinct by establishing a mapping of  $\text{FA}$  and  $\alpha\text{-RA}$  plans into DATALOG programs. Suppose, graph  $G$  is represented in DATALOG as a collection of ground facts  $p(s, o)$  for each edge in  $G$  from node  $s$  to node  $o$  labeled  $p$ . This forms an extensional database (EDB) for a given graph.

Consider query  $Q = (x, L(r), y)$  over labeled graph  $G = \langle V, \Sigma, E \rangle$ . First, we define the  $\alpha\text{-RA}$  generation procedure  $\text{GEN}_{\alpha\text{-RA}}$  as follows. Given regular expression  $r$  in query  $Q$ , we traverse the parse tree of  $r$  bottom-up. Let  $s$  be the subexpression of  $r$  represented by a given node in the parse tree. Then, we construct the DATALOG rules depending on the type of the node in the parse tree. The cases are as follows:

1. If  $s$  is a  $\Sigma$ -symbol, then we do nothing as it is already a part of an EDB.
2. If  $s_1$  and  $s_2$  are subexpressions and  $s = s_1/s_2$ , then we add:

$$s(X, Y) \leftarrow s_1(X, Z), s_2(Z, Y).$$

3. If  $s_1$  and  $s_2$  are subexpressions and  $s = s_1|s_2$ , then we add the following

DATALOG rules:

$$\begin{aligned}s(X, Y) &\leftarrow s_1(X, Y). \\ s(X, Y) &\leftarrow s_2(X, Y).\end{aligned}$$

4. If  $s = s_1^*$ , then we add:

$$\begin{aligned}s(X, X). \\ s(X, Y) &\leftarrow s(X, Z), s_1(Z, Y).\end{aligned}$$

5. If  $s = s_1^+$ , then we add:

$$\begin{aligned}s(X, Y) &\leftarrow s_1(X, Y). \\ s(X, Y) &\leftarrow s(X, Z), s_1(Z, Y).\end{aligned}$$

6. Finally, if  $s = r$ , then we extract query answers:

$$Q(X, Y) \leftarrow r(X, Y).$$

Next, we define the FA generation procedure  $\text{GEN}_{\text{FA}}$ . We construct a finite automaton  $A_r$  which accepts language  $L(r)$  defined by regular expression  $r$  in a given query  $Q$ . For each transition in  $A_r$  from state  $q_i$  to state  $q_j$  labeled  $p$ , we generate a DATALOG rule as follows:

$$q_j(X, Y) \leftarrow q_i(X, Z), p(Z, Y).$$

For the initial state  $q_0 \in A_r$ , we generate:

$$q_0(X, X).$$

And for each final state  $f \in A_r$ , we extract query answers with:

$$Q(X, Y) \leftarrow f(X, Y).$$

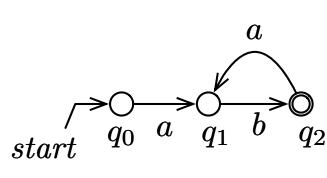
Both procedures  $\text{GEN}_{\alpha\text{-RA}}$  and  $\text{GEN}_{\text{FA}}$  generate DATALOG programs based on a given query.  $\text{GEN}_{\alpha\text{-RA}}$  generates a program based on a tree structure of a regular expression  $r$  and, hence, corresponds to the  $\alpha$ -RA-based evaluation methods. On the other hand,  $\text{GEN}_{\text{FA}}$  generates a program from a finite automaton which recognizes given regular expression  $r$  and, hence, corresponds to the FA-based evaluation methods. These DATALOG programs along with a method of *how* these programs are evaluated (e.g., via standard semi-naive bottom-up evaluation [2]) present a canonical form of evaluation plans to which both FA and  $\alpha$ -RA plans are mapped.

Consider the following *generic* property path query pattern:

$$?x \ (a/b)^+ ?y . \quad (\mathcal{Q}_{3.1})$$

We shall be using  $\mathcal{Q}_{3.1}$  as a prevalent example. Here, “a” and “b” are stand-ins for labels. It matches node-pairs that are connected by some path labeled **ab**, **bab**, or **babab**, and so forth. This is a quite simple property path query, but one that already demonstrates the complexities of planning.

The FA plan in Fig. 3.2a would be in  $\mathcal{P}_{\text{FA}}$  for  $\mathcal{Q}_{3.1}$ . There is no  $\alpha$ -RA plan that could be equivalent to it, however; none would ever evaluate **aba**, **ababa**, and so forth as state  $q_1$  does in the FA plan.  $\alpha$ -RA plans cannot compute transitive closure in a *pipelined* fashion as the FA plan is doing; the  $\alpha$  operator acts over an entire relation.



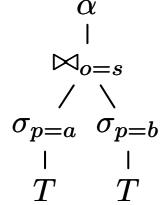
$q_0(X, X).$

$q_1(X, Y) \leftarrow q_0(X, Z), a(Z, Y).$      $s_1(X, Y) \leftarrow a(X, Z), b(Z, Y).$

$q_2(X, Y) \leftarrow q_1(X, Z), b(Z, Y).$      $s_2(X, Y) \leftarrow s_1(X, Y).$

$q_1(X, Y) \leftarrow q_2(X, Z), a(Z, Y).$      $s_2(X, Y) \leftarrow s_2(X, Z), s_1(Z, Y).$

$Q(X, Y) \leftarrow q_2(X, Y).$                  $Q(X, Y) \leftarrow s_2(X, Y).$



(a) FA DATALOG program.

(b)  $\alpha$ -RA DATALOG program.

Figure 3.2: Plans and corresponding DATALOG programs.

The  $\alpha$ -RA plan in Fig. 3.2b would be in  $\mathcal{P}_{\alpha\text{-RA}}$  for  $\mathcal{Q}_{3.1}$ . There is no FA plan that could be equivalent to it, however; no state transition in its automata can represent the “join” with **ab**. FA plans do not encompass *views*, materialized parts of the query that can be reused, while the  $\alpha$ -RA plan does by effectively materializing **ab** to join repeatedly on it.

Meanwhile, there are many plans in common between FA and  $\alpha$ -RA: for any query that is restricted to transitive closure over single labels, for example, will result in common FA and  $\alpha$ -RA plans.  $\square$

We aim to devise an RPQ evaluation strategy, which is a hybrid of automata and tree-based approaches. To achieve this, we identify the challenges which are

discussed in the following subsections.

## 3.2 Graph Walk

We formally define a framework that is based on an iterative search guided by an automaton. We model the search process by using *fixpoint* algebra, which is used in query evaluation in deductive databases. We capture a search cache as a collection of tuples  $\langle u, v, s \rangle$ , for which  $u$  and  $v$  are nodes in the graph and  $s$  is a state in the controlling automaton. Then, a search is a process of repeated application of operations `crank`, `reduce`, and `cache` on a search cache until a fixpoint is reached, i.e. no new search cache tuples are produced. In short, `crank` advances the search *wavefront*, simultaneously in the graph and the automaton, `reduce` prevents unbounded computation by checking for cycles, and `cache` adds newly discovered tuples to the cache. The main advantage of this model of the search is that it operates on a collection of tuples by using operations that can be encoded in procedural or recursive SQL. Hence, this framework can be implemented directly in most relational database systems, and be able to utilize efficient indexing and external storage support.

### 3.3 Query Plans

Recall that the construction of the automaton forces a certain order to the query evaluation, thus defying the purpose of query evaluation planning. To overcome this, in waveplans, we propose to use *wavefront automata* which employ two additional ways to perform the transitions: *inverse* transitions and transitions on *views*. In short, inverse transitions specify a wavefront that expands in the opposite direction to the edges of the graph. Transitions on views extend from the cache instead of from the graph. These offer powerful choices in WPs for guiding the search. The search can have *multiple* wavefronts from different starting points and directions. Each wavefront employs the cache to avoid unnecessary recomputation.

### 3.4 Optimizer & Enumerator

WAVEGUIDE plans model a rich space of plans for path queries which encompass powerful optimization techniques. Thus, RPQ evaluation needs a cost-based optimizer in order to be able to pick the plan with the lowest estimated cost. To build such optimizer, we need the following. First, we need to define WP systematically to define formally the *space* of WPs for a given query. Second, we devise a concrete *cost-model* for WPs. Third, we determine the array of *statistics* that can be computed efficiently offline that can be used in conjunction with the cost model.

Finally, we design an enumeration algorithm to walk dynamically the space of WPs to find the WP with least estimated cost.

### 3.5 Implementation & Benchmarking

Finally, to demonstrate the efficacy of our evaluation method, we focus on evaluation of SPARQL 1.1 property paths over large RDF graphs. We illustrate how our approach can be implemented effectively on a modern relational database system. An architecture of our prototype is designed as follows. There are two layers: *application* and *RDBMS*. The application layer provides a user front-end, preprocesses the graph data, parses user queries, and generates WPs. Specifically, for demonstration purposes, user front-end includes a graphical designer of waveplans and a profiling widget. This component shows current evaluation state of the system as a visualization of the graph and paths which were discovered during the iterations of the search so far. Statistical information such as the number of edge walks, the size of the cache and the number of reduced state tuples is also shown. The RDBMS layer provides postprocessing of graph data and performs an iterative graph search for the given WP. The iterative semi-naive bottom up evaluation procedure can be implemented as a procedural SQL program. Typically, this implementation, being local to data, is efficient and close to the performance of native SQL processing.

Finally, this prototype is benchmarked against a number of query workloads

on large graph datasets from various domains such as YAGO2s [64] and DBpedia [18]. We mine these graphs for regular path patterns which follow several simple templates. These patterns are used to construct a comprehensive benchmark of the optimizations enabled by WAVEGUIDE against state of the art for evaluation of RPQs and SPARQL property paths.

## 4 Graph Walk

WAVEGUIDE’s evaluation strategy is based on an *iterative search algorithm* — a *graph walk*, and variations thereof.

### 4.1 Wavefronts

In WAVEGUIDE, we perform efficiently path search while *simultaneously* recognizing the path expressions. WAVEGUIDE’s input is a graph database  $G$  and a *waveplan*  $P_Q$  which *guides* a number of search *wavefronts* that explore the given graph. We introduce the term *wavefront* to refer to a part of the plan that evaluates breadth-first during the evaluation and is essentially *pipelined*. This graph exploration, driven by an iterative search procedure, is inspired by the semi-naïve bottom-up strategy used in the evaluation of linear recursive expressions based on *fixpoint*, as is done for the  $\alpha$  operator for  $\alpha$ -RA, described in §2.5.2.

The key idea is, given a *seed* as a start, to expand repeatedly the search wavefronts until no new answers are produced; i.e., we reach a fixpoint. Each search

```

WAVEGUIDESEARCH ( $G$ ,  $P_Q$ )
1  $\Delta_0^R \leftarrow \text{seed}(G);$ 
2  $i \leftarrow 0;$ 
3 while  $|\Delta_i^R| \geq 0$  do
4    $\Delta_{i+1}^S \leftarrow \text{seed}(\Delta_i^R);$ 
5    $\Delta_{i+1}^C \leftarrow \text{crank}(\Delta_{i+1}^S, \Delta_i^R, G, C_i, P_Q);$ 
6    $\Delta_{i+1}^R \leftarrow \text{reduce}(\Delta_{i+1}^C, \Delta_i^R, C_i);$ 
7    $C_{i+1} \leftarrow \text{cache}(\Delta_{i+1}^R, C_i);$ 
8    $i \leftarrow i + 1;$ 
9 done;
10 return extract( $C_i$ );

```

Figure 4.1: WAVEGUIDE’s evaluation procedure.

wavefront is guided by an *automaton* in the plan, a finite state automaton based on an **NFA**. In Fig. 3.2a, the plan consists of a single wavefront automaton. We shall see examples below of multi-wavefront plans; e.g., plan  $P_2$  in Fig. 4.2. This is akin to the **FA** approach discussed in §2.5.1. Different, though, from **NFAs** which are used as recognizers of regular expressions on strings, wavefront automata have features directed to the evaluation of regular expressions over graphs.

## 4.2 Expanding a Wavefront

Each search wavefront has a *seed* as its initialization. The seed is the set of nodes in the graph from which this wavefront begins its search. A seed can be either *universal* or *restricted*. A wavefront with a universal seed conducts its search

*virtually*<sup>12</sup> starting from *every* node in a graph. A wavefront with a restricted seed starts the search from a fixed set of nodes. (A restricted seed is defined by the results of other wavefronts or by constants used in a query.) Graphically, a seed is represented as an incoming edge to starting state  $q_0$  of the wavefront. We use the label “ $U$ ” to denote a universal seed; any other label on this incoming edge denotes a restriction placed on the seed, thus a “restricted” seed.

The graph exploration (*walk*) is performed by an iterative procedure *guided* by a waveplan, as illustrated in Fig. 4.1. Consider the example from Fig. 4.2 with an example graph  $G$  and two plans  $P_1$  and  $P_2$  for evaluating query  $Q = (x, (ab)^+, y)$ . Waveplan  $P_1$  consists of a single wavefront automaton ( $W_1$ ), which is analogous to the plan that results from the FA method. Fig. 4.2 also presents the evaluation trace by the procedure in Fig. 4.1 using plan  $P_1$  over graph  $G$ .

During the search, intermediate results are kept in a *cache*, denoted  $C_i$  for iteration  $i$ . This is a collection of tuples  $\langle u, v, s \rangle$  for which  $u$  and  $v$  are nodes in  $G$  and  $s$  is a state in the plan. The newly discovered tuples found in the current iteration are denoted by *delta*  $\Delta_i$ . We use  $C_i$  and  $\Delta_i$  to eliminate answers already seen in the search. Delta tuples produced by `crank` are denoted  $\Delta^C$ , and those produced by `reduce` are  $\Delta^R$ .

To begin, all the universal seeds are initialized (but never materialized!);  $\Delta_0^R$  is

---

<sup>12</sup>Effectively, all nodes in a graph are never materialized due to *first-hop* optimization which restricts the starting nodes according to the transition labels originating from  $q_0$ .

assigned the set of  $\langle u, u, q_0 \rangle$  for all  $u \in N$ . Note that  $q_0$  is the starting state for all wavefronts with universal seeds. We then iterate, performing four operations per iteration:

1. **seed.** Determine the set of nodes from which to search.
2. **crank.** Perform a step of the fixpoint operation, the breadth-first search.
3. **reduce.** Eliminate node pairs seen before.
4. **cache:** Update the set of pairs of nodes seen already in the evaluation.

The iteration continues until the fixpoint is reached. Thus, this is a guided semi-naïve evaluation [2].

The **seed** step populates the restricted seeds, according to their respective seed conditions. The **crank** step transitions from the previous delta to the current,  $\Delta_i^R \rightarrow \Delta_{i+1}^C$ . For each node  $v$  in  $\langle u, v, s \rangle \in \Delta_i^R$ , for edge  $\langle v, a, w \rangle \in G$  and automaton transition  $\langle s, a, t \rangle \in W_1$ ,  $\langle u, w, t \rangle$  is added to  $\Delta_{i+1}^C$ . Thus **crank** advances the search *simultaneously* in the graph *and* in the automaton.

To prevent unbounded computation over cyclic graphs, the delta is *reduced*:  $\Delta_{i+1}^C$  is checked against both the previous delta  $\Delta_i^R$  and the cache  $C_i$ ; tuples that are seen in either  $\Delta_i^R$  or  $C_i$  are removed to produce  $\Delta_{i+1}^R$ . Lastly, the cache is updated  $C_{i+1}$  by adding the tuples in the reduced delta  $\Delta_{i+1}^R$  to it ( $C_i$ ). The iteration halts once  $\Delta^R$  is empty.

It can be established by structural induction that, for any tuple  $\langle u, v, s \rangle$  in the

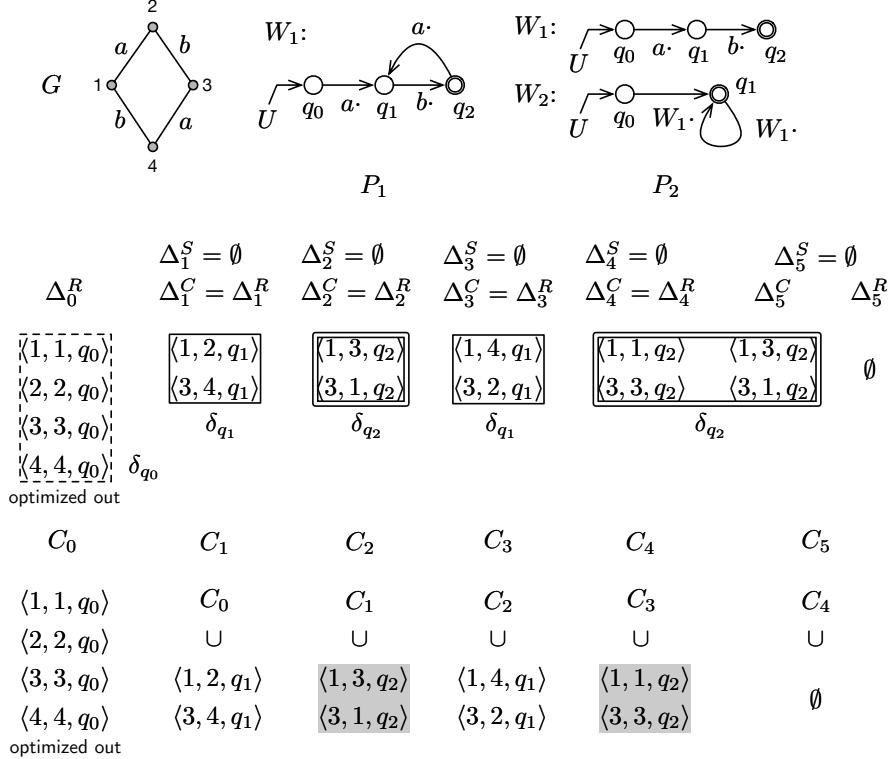


Figure 4.2: Two waveplans,  $P_1$  &  $P_2$ , over graph  $G$ , with an evaluation trace of  $P_1$ .

cache ( $C$ ) such that  $s$  is an accepting state, the pair of nodes  $\langle u, v \rangle$  must have a path between them in the graph that conforms to  $r$ . Thus, WAVEGUIDE produces the correct results. The answer set can be then *extracted* from the cache by selecting the tuples  $\langle u, v, s \rangle$  for all accepting states  $s$  of automaton  $W_1$ .

**Example 4** Consider the evaluation trace presented in Fig. 4.2. First, universal seeds in  $\Delta_0^R$  are optimized out by the first-hop optimization which builds  $\Delta_1^C$  directly by selecting the tuples from  $G$  which match the label (a) of the first transition  $\langle q_0, a \cdot, q_1 \rangle$  in  $P_1$ . The search proceeds by iterating **crank**, **reduce** and **cache** to produce

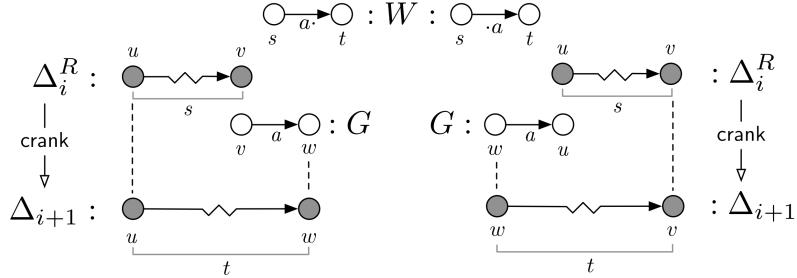
$\Delta_i^C$ ,  $\Delta_i^R$  and  $C_i$  at each iteration  $i$ . The search stops at the 5<sup>th</sup> iteration when reduced delta  $\Delta_5^R$  is empty. Those cache tuples in accepting state  $q_2$  (shaded) at the end are then extracted as the answer set.

## 5 Query Plans

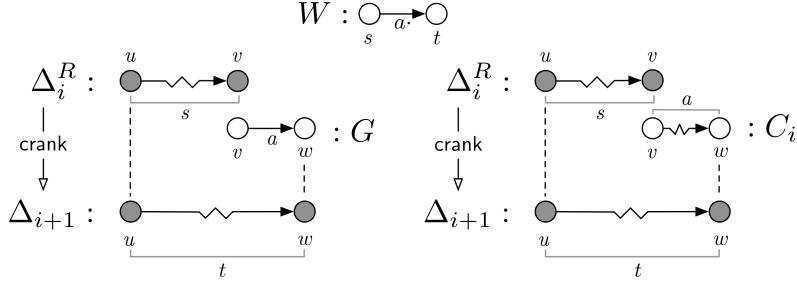
We are able to express complex query evaluation plans which involve multiple search *wavefronts* that iteratively explore the graph. A query will have a number of different plans that could be used to evaluate it, as we saw in §3.1. The space of WAVEGUIDE’s plans properly subsumes that of FA and  $\alpha$ -RA, providing us the best of both those approaches, along with useful plans that do not exist with FA or  $\alpha$ -RA (§5.3).

### 5.1 Guiding a Wavefront

The construction of the NFA forces an order to the query evaluation. A “wrong” choice of NFA can lead to an inefficient evaluation plan. In WAVEGUIDE, we aim to minimize the search space explored by considering the possible orders of graph exploration by search wavefronts. To achieve this, we use waveform automata which can use transitions that expand the waveform in the direction opposite to the direction of the edges of the graph.



(a) Prepending vs. appending by expanding the wavefront using tuples from graph  $G$ .



(b) Expanding the wavefront by appending and using the tuples from  $G$  (over the graph) vs. from search cache  $C$  (over the view).

Figure 5.1: Types of transitions used in a wavefront.

Consider a graph transition  $\langle s, l, t \rangle$  in a wavefront as illustrated in Fig. 5.1a.

Edge label  $l$  has a general form  $\cdot a$  or  $a \cdot$ , where  $a$  is an edge label in  $G$  and the position of a dot ‘·’ specifies a direction of a search wavefront and denotes prepend ( $\cdot a$ ) or append ( $a \cdot$ ) transition. A prepend wavefront expands in the opposite direction to the edges in the graph. Likewise, the append parameter guides a wavefront that expands in the same direction as the edges in the graph.

Hence, wavefronts enable automaton transitions that explore the graph in a specified direction. This allows a wavefront to initiate evaluation from any label

in the given regular expression and iteratively expand by appending or prepending path labels. This gives us the power to explore all different expansion orders of a single wavefront.

## 5.2 Wavefront Interaction

Often, the search space is constrained even further if several wavefronts are employed in the evaluation, each evaluating parts<sup>13</sup> of a given regular expression. WAVEGUIDE enables this by defining a number of automata, one for each search wavefront.

In addition to transitions over graph edge labels, waveplans allow transitions over *path views*, by using cached result sets produced by other wavefronts. Consider Fig. 5.1b and automaton transition  $\langle s, a, t \rangle$ . If  $a$  is an edge label in  $G$ , then this *graph* transition expands the wavefront by using the tuples from *graph*  $G$ . Otherwise, if  $a$  is a state in a wavefront then this *view* transition expands the wavefront by employing the tuples produced by a wavefront.

These new types of transitions offer powerful choices in WPs for guiding the search. The search can have *multiple* wavefronts originating from different starting points *and* expanding in different directions. Further, each wavefront can employ

---

<sup>13</sup>This decomposition of a regular expression can be determined by graph statistics such as cardinalities of its labels.

the cache through transitions over views to avoid recomputation. (This is also known as *memoization*.)

**Example 5** Consider the waveform search in Fig. 4.2 for a query  $Q$  with regular expression  $r = (ab)^+$ .  $P_1$  is a basic WP embodying an NFA that recognizes  $r$ . From  $P_1$ , we can design a more efficient WPP<sub>2</sub>: first, compute  $(ab)$  with waveform  $W_1$ ; and then use a loop-back path-view transition to compute the closure  $(ab)^+$  (with waveform  $W_2$ ). It can be shown that  $P_2$  explores a smaller search space than  $P_1$ .

Based on the discussion above, we are now ready to give formal definitions of the concepts introduced in WAVEGUIDE.

**Definition 5.1** Given graph  $G = (V, E)$ , we call seed  $S$  a set of node pairs  $\{(s, o) : s, o \in V\}$ . A seed can be defined by results of any waveform, by projection-constructions, or by an RPQ  $(x, r, y)$ .

**Definition 5.2** We define a pair of projection-construction operators  $\pi_s(S)$  and  $\pi_o(S)$  on given seed  $S$ . Operator  $\pi_s(S)$ , provided seed  $S = \{(s, o) : s, o \in V\}$  constructs a set of subject pairs  $\{(s, s) : s \in V\}$ . Similarly,  $\pi_o(S)$ , provided seed  $S = \{(s, o) : s, o \in V\}$  constructs a set of object pairs  $\{(o, o) : o \in V\}$ .

**Definition 5.3** We call seed  $U = \{(s, s) : s \in V\}$  a universal seed when  $s$  can be any node which has an incoming or outgoing edge in  $G$ . Formally,  $U$  is defined as

$RPQ$  ( $?x, \varepsilon, ?x$ ) which yeilds to all nodes in  $G$ . Here,  $\varepsilon$  is label of virtual empty self-loop edge which any node in a graph has.

**Definition 5.4** We call *wavefront* a tuple  $W_l = \langle l, S, q_0, Q, \delta, E, L, F \rangle$ , where

- $l$  is a *wavefront label*,
- $S$  is a *seed*,
- $Q$  is a *set of states*,
- $q_0$  is a *starting state*,  $q_0 \in Q$ ,
- a *transition function*  $\delta : Q \times ((E \cup L) \times \{\cdot\_, \cdot\}) \cup \{\varepsilon\} \rightarrow 2^Q$ ,
- $E$  is a *set of edge labels* in a given graph  $G = (V, E)$ ,
- $L$  is a *set of wavefront labels* distinct from  $E$ , and
- a *set of accepting states*  $F \subseteq Q$ .

**Definition 5.5** We call *waveplan*  $P$  an ordered set of wavefronts. Consider any pair of wavefronts  $W, W' \in P$  such that  $W = (l, S, q_0, Q, \delta, E, L, F)$  and  $W' = (l', S', q'_0, Q', \delta', E', L', F')$ . Then  $P$  defines an order  $<_P$  on wavefronts as follows:

$$\forall W, W' \in P \quad | \quad W <_P W' : l' \notin S \wedge l' \notin L$$

Given this order, lower wavefronts cannot use labels of higher wavefronts in their seeds or transitions. That is, such dependencies between wavefronts have no cycles as guaranteed by their topological order.

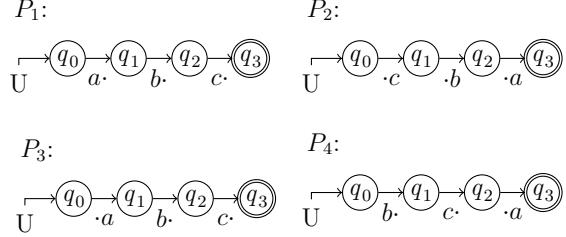


Figure 5.2: Waveplans for  $abc$  expression.

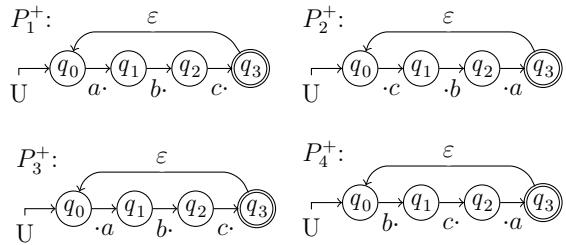


Figure 5.3: Attempting to build waveplans for  $(abc)^+$  expression.

**Definition 5.6** *Waveplan  $P$  is legal with respect to  $Q = (x, r, y)$  when, for any graph  $G$ , the evaluation of  $P$  over  $G$  produces a set of bindings  $\langle s, t \rangle$  such that there exists an arbitrary path  $p$  from node  $s$  to node  $t$  in  $G$  such that the path label  $\lambda(p)$  is in language  $L(r)$ .*

**Example 6** Consider waveplans  $P_1$ - $P_4$  shown in Fig. 5.2. Observe, that  $P_1$ - $P_4$  are legal with respect to regular expression  $r = abc$ . On the other hand, consider plans  $P_1^+$ - $P_4^+$  shown in Fig. 5.3. These plans were obtained by connecting accepting and starting states in  $P_1$ - $P_4$  with an  $\varepsilon$ -transition; i.e., establishing a loop-back pipeline in an attempt to produce plans for  $(abc)^+$  expression. Observe, however, that only

$P_1^+$  and  $P_2^+$  are legal w.r.t.  $(abc) +$  expression, whereas plans  $P_3^+$  and  $P_4^+$  are legal w.r.t.  $a + (bc) +$  expression.

### 5.3 WAVEGUIDE's Plan Space

We claim that the space of waveguide plans  $\mathcal{P}_{WP}$  subsumes that of the FA and  $\alpha$ -RA approaches, as the Venn diagram in Fig. 3.1 shows (and with the caveats as discussed in §3.1).

**Lemma 5.7**  $\mathcal{P}_{WP}$  properly subsumes the union of  $\mathcal{P}_{FA}$  and  $\mathcal{P}_{\alpha\text{-RA}}$  ( $\mathcal{P}_{WP} \supsetneq \mathcal{P}_{FA} \cup \mathcal{P}_{\alpha\text{-RA}}$ ).

**Proof:** That  $\mathcal{P}_{WP}$  subsumes each of  $\mathcal{P}_{FA}$  and  $\mathcal{P}_{\alpha\text{-RA}}$  is straightforward; we devised WP so that we could express both FA- and  $\alpha$ -RA- type plans. WP extends the FA model. WP encompasses  $\alpha$ -RA by the addition of *path views*; what the  $\alpha$  operator offers, transitive closure over an arbitrary relation, can be accomplished by view-labeled transitions.

$\mathcal{P}_{WP}$  properly subsumes the union of  $\mathcal{P}_{FA}$  and  $\mathcal{P}_{\alpha\text{-RA}}$  because there is a waveguide plan that corresponds to neither an FA plan nor an  $\alpha$ -RA plan. We have demonstrated that in the discussions above: any WP with multiple wavefronts and some wavefront with a long loop-back is such a plan; FA plans are essentially single wavefront by the FA model, and pipelined loop-backs are outside the scope of  $\alpha$ -RA.

Likewise, any **WP**, even single wavefront, that is “mixed”, that combines views and long loop-backs, corresponds to no **FA** plan and to no  $\alpha$ -**RA** plan. (In Fig. 6.4 below,  $P_2$  with partial loop-caching is such a plan.) These very types of waveguide plans that **FA** and  $\alpha$ -**RA** miss can be the most efficient plans.  $\square$

In §6.4, we explain why this rich plan space is relevant. In §7, we compare plans for real queries over real graph data to establish that this is true in practice, as well.

## 6 Optimizer & Enumerator

We present a *cost framework* for WAVEGUIDE search, *search cost factors* that affect the cost (properties of the graph and of resulting pre-paths computed during the evaluation), and *optimization methods* that are enabled by WPs which address the search factors, in turn.

### 6.1 Cost Framework

Recall the three steps in Fig. 4.1 of the search iteration: **crank**, **reduce**, and **cache**. Assume that the search completes in  $n$  iterations. The cost of **crank**,  $\mathbf{C}_{\text{crank}}$ , corresponds to the total number of *edge walks* performed. This *search size* is the sum of sizes of the deltas. The cost of **reduce**, has two components: duplicate removal within a delta ( $\mathbf{C}_{\text{reduce}}^{\Delta}$ ) and for the delta against the search cache ( $\mathbf{C}_{\text{reduce}}^C$ ). The cost of **cache**,  $\mathbf{C}_{\text{cache}}$ , is associated with search cache maintenance procedures (e.g., indexing). The functions  $f_{1-4}$  are placeholders that weigh the costs:

The cost functions  $f_{1-4}$  above are monotone over their parameters; these simply

Step	Generalized Cost
$\mathbf{C}_{\text{crank}}$	$\sum_{i=0}^n f_1( \Delta_i^R ,  G ,  C_i )$
$\mathbf{C}_{\text{reduce}}^\Delta$	$\sum_{i=1}^n f_2( \Delta_i^C )$
$\mathbf{C}_{\text{reduce}}^C$	$\sum_{i=1}^n f_3( \Delta_i^C ,  C_i )$
$\mathbf{C}_{\text{cache}}$	$\sum_{i=1}^n f_4( \Delta_i^R ,  C_i )$

abstract the actual costs as based upon the underlying implementation of WAVEGUIDE’s data structures and algorithms. For example, in a relational-based system, assuming tree indexes on  $C_i$  and  $G$ , functions  $f_{1-4}$  are approximated as:

	Example Implementation	Complexity
$f_1$	Index-nested loop joins: $\Delta_i^R \bowtie_{P_Q} G$ and $\Delta_i^R \bowtie_{P_Q} C_i$ .	$O( \Delta_i^R  \cdot (\log  G  + \log  C_i ))$
$f_2$	Sort $\Delta_i^C$ , remove duplicates.	$O( \Delta_i^C  \cdot \log  \Delta_i^C )$
$f_3$	Scan $\Delta_i^C$ , probe $C_i$ ’s index.	$O( \Delta_i^C  \cdot \log  C_i )$
$f_4$	Scan $\Delta_i^R$ , insert into $C_i$ ’s index.	$O( \Delta_i^R  \cdot \log  C_i )$

In this scenario, **crank** is implemented as joins of  $\Delta_i$  with  $G$  (for graph transitions) and  $C_i$  (for view transitions). The predicates of these joins are dictated by the WP  $P_Q$ . Since typically  $|\Delta| \ll |C|$ ,  $\text{reduce}^\Delta$  is implemented in-memory. On the other hand,  $\text{reduce}^C$  is more expensive since it requires probing  $C$ ’s index. Finally, **cache** involves updating  $C$ ’s index with new  $\Delta$ ’s tuples.

## 6.2 Search Cost Factors

Properties of the graph and of the WP chosen will determine the evaluation cost.

### 6.2.1 Search Sizes

The wavefronts that we choose for the search determine the intermediate  $\Delta_i$  (pairs of nodes connected by valid pre-paths) that we collect in each iteration  $i$ . Just as with different join orders in relational query evaluation, different wavefronts will result in different cardinalities of  $\Delta_i$ . These intermediate cardinalities can vary widely from plan to plan and affect all four costs  $f_{1-4}$ .

### 6.2.2 Solution Redundancy

After much deliberation, the W3C has adopted a *non-counting* semantics for SPARQL property path queries. Each node pair appears only once in the answer, even if there are several paths between the node pair satisfying the given regular expression.

Answer-path redundancy arises from two sources. First, in dense graphs, solutions are re-discovered by following conforming, yet different paths. Second, nodes are revisited by following cycles in the graph. Thus, the same answer pair may be discovered repeatedly during evaluation, which increases both  $|\Delta_i|$  and  $|C|$ . It is critical to detect such duplicate solutions early in order to keep the costs  $f_{1-4}$  low.

### 6.2.3 Sub-path Redundancy

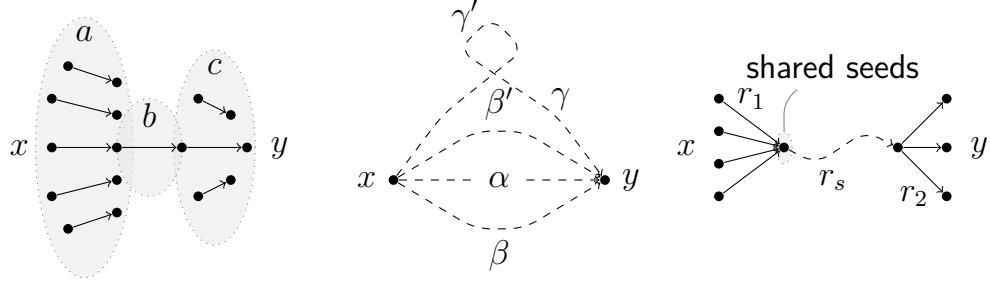
In *solution redundancy*, an answer pair could have multiple paths justifying it. Likewise, the paths justifying multiple answer pairs may *share* significant segments (*sub-paths*) in common. This arises, for instance, in dense graphs and with hierarchical structures (e.g., *isA* and *locatedIn* edge labels). Consider a query “`?p :locatedIn+ Canada`”. Every person located in the neighborhood of the Annex in the city of Toronto qualifies, since the Annex is located in Toronto which is located in Ontario which is located in Canada. The sub-path “`Annex :locatedIn+ Canada`” is shared by the answer path for each Annex resident. Because we keep only node-pairs (plus state) in the search deltas, and not explicitly the paths themselves,<sup>14</sup> we may walk these sub-paths many times, recomputing “`Annex :locatedIn+ Canada`” for each Annex resident, thus unnecessary increasing both  $|\Delta_i|$  and  $|C|$ .

## 6.3 Plan Optimizations

We consider WP-optimization methods in relation to the search cost factors above.

---

<sup>14</sup>Note this design choice in our evaluation strategy is critical for good performance due to solution redundancy!



a) search cardinality b) solution redundancy c) sub-path sharing

Figure 6.1: Types of search cost factors.

### 6.3.1 Choice of Wavefronts

The direction in which we follow edges, and where we start in the graph, with respect to the regular expression will result in different  $|\Delta_i|$  we encounter during the search. Our choice of automata in the WP dictates the wavefront(s). For example, consider query  $Q = (x, abc, y)$  and a fragment of a graph shown in Fig. 6.1a. Since labels  $a$ ,  $b$  and  $c$  have different frequencies, different wavefronts will have different search sizes. Consider two plans  $P_1$  and  $P_2$  that evaluate  $Q$  shown in Fig. 6.2.  $P_1$  has a single wavefront that explores the graph starting from  $a$ , appending  $b$  and then  $c$ . On the other hand,  $P_2$  has a wavefront that starts from the low-frequency label  $b$ , appends  $c$  and then prepends  $a$ . Observe that, in this scenario,  $P_2$  results in fewer edge walks than  $P_1$ .

To reduce overall search size, we need to choose wavefronts that result in fewer

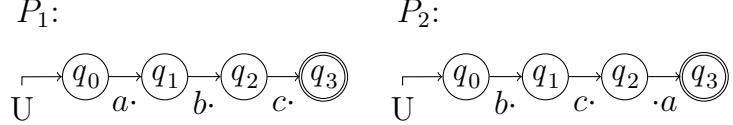


Figure 6.2: Choosing the wavefronts.

edge walks. Wavefronts can be costed to estimate their search sizes based on statistics about the graph, such as various graph label frequencies. (Such graph statistics can be computed offline for this purpose, which we discuss in more depth in §6.5).

### 6.3.2 Reduce

WAVEGUIDE’s evaluation strategy is designed to counter solution redundancy. As shown in Fig. 6.1b, we consider several types of redundant solutions based on a path which was followed to obtain each solution. Path  $\alpha$  is the shortest path. Paths  $\beta$  and  $\beta'$  are of the same length, but go through different nodes. Finally, path  $\gamma'$  shares some nodes with path  $\gamma$ , but it is longer due to a cycle.

In WAVEGUIDE, redundancy of *candidate* solutions is addressed by removal of duplicates against both  $C$  and  $\Delta$  by the `reduce` operation. Assuming a BFS search strategy, duplicate solutions obtained by following paths of the same length ( $\beta, \beta'$ ) are removed within a  $\Delta$ . On the other hand, duplicates obtained by following paths of different lengths ( $\alpha, \beta$ ) are removed when  $\Delta$  is compared against  $C$ . This also includes paths with cycles such as  $\gamma$  and  $\gamma'$  as they also have different lengths.

As a further optimization, once a solution seed-target pair has been discovered, *first-path pruning* (*fpp*) removes the *seed* from further expansion by the search wavefronts. In our example, once path  $\alpha$  has been discovered and solution  $(x, y)$  has been obtained, all longer paths  $(\beta, \beta', \gamma, \gamma')$  are never even materialized.

### 6.3.3 Threading

To counter *sub-path redundancy* requires us to decompose a query into sub-queries. We call this decomposition *threading*, and our WPs accommodate this.

Consider query  $Q = (x, (r_1/r_s/r_2), y)$  where sub-path  $r_s$  is shared among many solutions as shown in Fig. 6.1c. This query can be threaded as follows. First, pre-path  $r_1$  is computed by waveform  $W_{r_1}$ . Then, the portion of the regular expression that will result in sub-paths that will be shared by many answer paths can be computed by a separate waveform  $W_{r_1:r_s}$ . Here,  $W_{r_1}$  seeds waveform  $W_{r_1:r_s}$  which computes a shared path  $r_s$  for each of the partial solutions produced by  $W_{r_1}$ . Finally, the complete path is pieced together by waveform  $W_{\text{join}}$ . Sub-path sharing can be predicted by graph statistics to indicate when threading is useful.

### 6.3.4 Partial Caching

Delta results are cached during the evaluation as we need to check against  $C$  for redundantly computed pairs. For large intermediate  $|\Delta_i|$ , this can be a significant

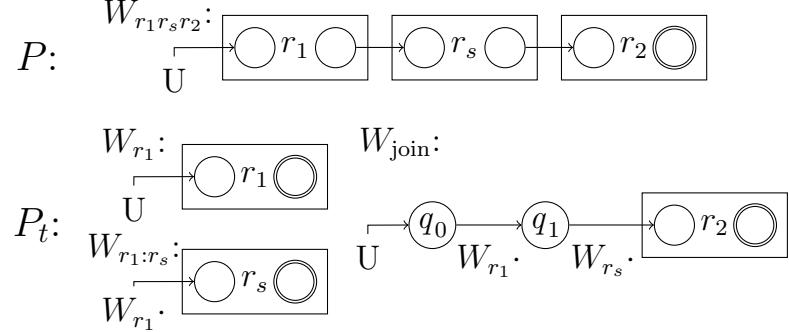


Figure 6.3: Threading a shared sub-path.

cost. However, some of this cost can be eliminated. In particular, not every state in the WP’s automata needs to have its node-pairs cached. Caching is only needed when *unbounded* redundancy is possible, due to cycles in the wavefront automata or in the graph. States without cycles need not be cached.

### 6.3.5 Loop Caching

Transitions over views in wavefront automata allow us to cache and re-use some of the intermediate node pairs we encounter during the search. Such named result sets are useful in reducing unnecessary recomputation by employing an optimization we call *loop caching*.

In transitive query  $Q = (x, (r)+, y)$ , the expression  $r$  is repeatedly evaluated until no new solutions are found. Loop caching rewrites an evaluation plan so that the base  $r$  is cached either fully or partially to speed up the transitive evaluation

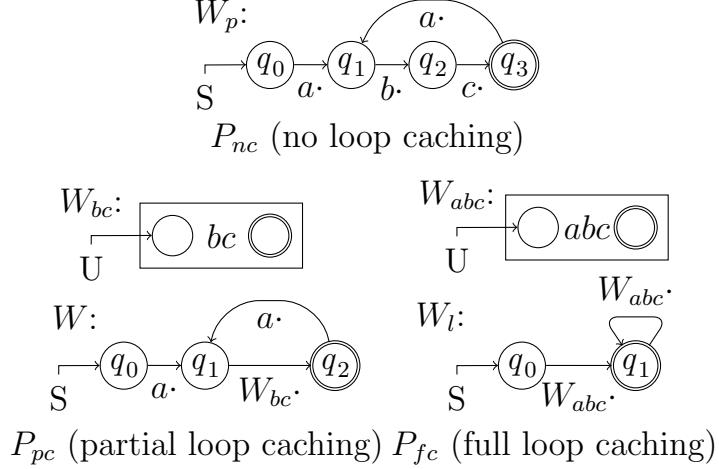


Figure 6.4: Types of loop caching.

of  $(r)+$ .

Consider three plans  $P_{nc}$ ,  $P_{pc}$  and  $P_{fc}$  for query  $Q = (x, (abc)+, y)$  shown in Fig. 6.4. Plan  $P_{nc}$  has no loop caching as it evaluates full expression  $(abc)$  in a pipeline with  $W_p$ . Plan  $P_{pc}$  uses a separate waveform to evaluate  $(bc)$  first, then these results are used in a loop to evaluate transitive  $(abc)+$ . Finally, plan  $P_{fc}$  caches the full base expression  $(abc)$  in  $W_{abc}$ , which is then used in the evaluation of a transitive expression in a loop by  $W_l$ .

## 6.4 Cost Analysis

Consider deltas  $\Delta_i^C$ ,  $\Delta_i^R$  and cache  $C_i$  for all iterations in the evaluation of WP  $P_Q$  over graph  $G$ . These determine the overall cost of  $P_Q$  according to the framework

in §6.1.

We define a *logical* delta  $\delta_s$  over state  $s$  to be a set of delta tuples  $(u, v, s) \in \delta_s$  such that  $\delta_s \subseteq \bigcup_{i=0}^n \Delta_i$ . In other words, from all *physical* deltas  $\Delta_i$  we encountered during the search, we *select* delta tuples which occur in state  $s$ . Logical delta  $\delta_S$  over set of states  $S$  is a union of logical deltas over all states in  $S$ , i.e.  $\bigcup_{s \in S} \delta_s$ . Similarly,  $\delta_W$  over wavefront  $W$  is a union  $\bigcup_{s \in W} \delta_s$ . For example, consider a breakdown of  $\Delta$  into  $\delta$  for an execution of  $P_1$  as shown in Fig. 4.2. Here, for each state  $q \in P_1$ ,  $\delta_q$  is shown by encasing the  $\Delta$  tuples in a dashed, single and double rectangles for  $\delta_{q_0}$ ,  $\delta_{q_1}$  and  $\delta_{q_2}$ , respectively.

#### 6.4.1 Cost of Threading

Given a plan  $P$  with a single wavefront  $W_{r_1 r_s r_2}$  which computes a regular expression of a form  $r = r_1/r_s/r_2$ , threading rewrites it into a plan  $P_t$  with three wavefronts  $W_{r_1}$ ,  $W_{r_1:r_s}$  and  $W_{\text{join}}$  as described in §6.3. We partition states in  $W_{r_1 r_s r_2}$  into three subsets:  $S_{r_1}$ ,  $S_{r_s}$  and  $S_{r_2}$  which are responsible for evaluation of expressions  $r_1$ ,  $r_1 r_s$  and  $r_1 r_s r_2$ , respectively.

To analyze the cost of threading, we partition edge walks into logical deltas for

both  $P$  and  $P_t$  as follows:

$$P : \sum_{i=0}^n |\Delta_i'^C| = |\delta_{W_{r_1 r_s r_2}}^C| = |\delta_{S_{r_1}}^C| + |\delta_{S_{r_s}}^C| + |\delta_{S_{r_2}}^C| \quad (6.1)$$

$$P_t : \sum_{i=0}^{n_t} |\Delta_i''^C| = |\delta_{W_{r_1}}^C| + |\delta_{W_{r_1:r_s}}^C| + |\delta_{W_{\text{join}}}^C| \quad (6.2)$$

From construction (Fig.6.3), we have:

$$|\delta_{W_{r_1}}^C| = |\delta_{S_{r_1}}^C| \quad (6.3)$$

$$|\delta_{W_{\text{join}}}^C| = |\delta_{q_0}^C| + |\delta_{q_1}^C| + |\delta_{S_{r_2}}^C| \quad (6.4)$$

We define a *multiplicity* ratio of an expression  $r_1$  in graph  $G$ , which is computed by analyzing the paths in  $G$ :

$$\mathcal{M}(G, r_1) = \frac{|S_s|}{|S_o|},$$

where  $S_s$  and  $S_o$  is a set of subjects and objects, respectively, connected in  $G$  with paths conforming to  $r_1$ . Then,  $\mathcal{M}(G, r_1) > 1$ , would indicate that, on average, there are many subjects connected to a single object in  $G$ , while  $\mathcal{M}(G, r_1) < 1$  would indicate that the opposite is true. The greater  $\mathcal{M}$  is, the more subjects are connected to the same object by path  $r_1$ , and, hence, more subjects share a path  $r_s$  which originates from this object. Hence, we have:

$$|\delta_{S_{r_s}}^C| = \mathcal{M}(G, r_1) \cdot |\delta_{W_{r_1:r_s}}^C| \quad (6.5)$$

We define a *gain*  $\mathcal{G}_t$  introduced by threading in  $P$  as difference in a number of edge walks performed in  $P$  and  $P_t$ . From (1-5), this gain is:

$$\mathcal{G}_t = \underbrace{(\mathcal{M}(G, r_1) - 1) \cdot |\delta_{W_{r_1:r_s}}^C|}_{\mathbf{A}: \text{savings due to threading}} - \underbrace{(|\delta_{q_0}^C| + |\delta_{q_1}^C|)}_{\mathbf{B}: \text{extra join}}$$

If a shared sub-path  $r_s$  is accurately identified, then the total reduction of number of edge walks in  $P_t$  (**A**) is sufficiently large to offset the cost of the extra join (**B**); i.e.,  $\mathcal{G}_t > 0$ .

To maximize **A**, we need to maximize both  $\mathcal{M}(G, r_1)$  and  $|\delta_{W_{r_1:r_s}}^C|$ . For the latter, a useful metric to consider is an average length  $\mathcal{L}(G, r_s)$  of a path which conforms to  $r_s$  in  $G$ . The longer the shared sub-path  $r_s$  is, the greater the cardinality  $|\delta_{W_{r_1:r_s}}^C|$  is, and, hence, more potential savings in edge walks can be realized by threading split on  $r_s$ .

Thus, given  $\mathcal{M}$  and  $\mathcal{L}$  for sub-expressions of  $r$  in  $G$ , the identification of an efficient threading split  $r = r_1/r_s/r_2$  becomes an  $(\mathcal{M}, \mathcal{L})$  maximization problem.

#### 6.4.2 Cost of Loop Caching

Given a plan with a single wavefront which computes closure  $(r)+$  of a regular expression  $r$ , loop caching rewrites it into a plan in which parts of  $r$  are pre-computed, cached, and then used in an iterative evaluation of a closure. For example, consider the differences in evaluation of query  $Q = (x, (abc)+, y)$  with plans  $P_{nc}$ ,  $P_{pc}$

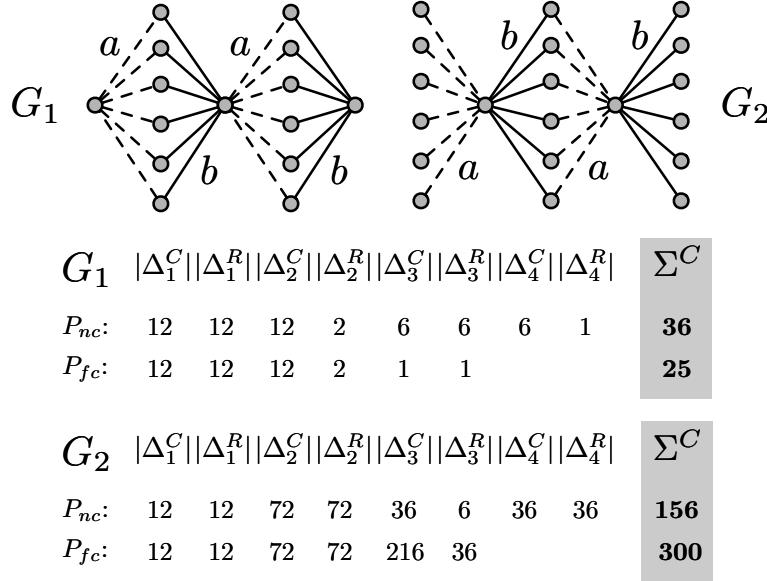


Figure 6.5: Lensing.

and  $P_{fc}$  shown in Fig. 6.4.  $P_{nc}$  defines a single wavefront, which, due to absence of transitions over views, can be executed pipelined. On the other hand,  $P_{pc}$  and  $P_{fc}$  first compute  $(bc)$  and  $(abc)$ , respectively, in separate wavefronts, the results of which are used in a wavefront which computes the final closure.

As previously, to analyze the cost of loop caching, we partition edge walks into logical deltas for both<sup>15</sup>  $P_{nc}$  and  $P_{fc}$ :

$$\begin{aligned}
 P_{nc} : \sum_{i=0}^{n_{nc}} |\Delta_i'^C| &= |\delta_{q_0 \in W_p}^C| + \underbrace{|\delta_{q_1 \in W_p}^C| + |\delta_{q_2 \in W_p}^C| + |\delta_{q_3 \in W_p}^C|}_{\mathbf{C}} \\
 P_{fc} : \sum_{i=0}^{n_{fc}} |\Delta_i''^C| &= \underbrace{|\delta_{W_{abc}}^C|}_{\mathbf{D}} + |\delta_{q_0 \in W_l}^C| + \underbrace{|\delta_{q_1 \in W_l}^C|}_{\mathbf{E}}
 \end{aligned}$$

---

<sup>15</sup>We omit the comparison with  $P_{pc}$  as it follows the same idea.

From construction (Fig.6.4), we have:

$$|\delta_{q_0 \in W_p}^C| = |\delta_{q_0 \in W_l}^C|$$

It follows that a gain  $\mathcal{G}_{lc}$  of loop caching is a difference in a number of edge walks in  $P_{nc}$  and  $P_{fc}$ :

$$\mathcal{G}_{lc} = \mathbf{C} - (\mathbf{D} + \mathbf{E})$$

In order for cached plan  $P_{fc}$  to be faster, sum of edge walks associated with computation of base  $(abc)$  paths (**D**) and looped evaluation of  $(abc) +$  paths (**E**) should be less than a sum of edge walks associated with evaluation of  $(a)$ ,  $(ab)$ ,  $(abc)$ ,  $(abca)$ , ... paths (**C**) in pipelined plan  $P_{nc}$ .

Interestingly, values of terms **C** and **E** can significantly differ depending on the general shape of the graph. For example, consider two basic graphs  $G_1$  and  $G_2$  as presented in Fig. 6.5. Both  $G_1$  and  $G_2$  have the same frequencies of labels  $a$  and  $b$ , but are different in terms of their *shape*.  $G_1$  exhibits *lensing* with focal points on concatenations  $b/a$ , while  $G_2$  has lensing in  $a/b$ . Intermediate cardinalities of  $\Delta^C$  (number of edge walks) of the WAVEGUIDE search are presented for plans with  $(P_{fc})$  and without  $(P_{nc})$  loop caching. Observe that loop caching optimization is beneficial for search in  $G_1$  with 30% fewer edge walks. On the other hand, loop caching performs worse in  $G_2$  with 92% more edge walks. This can be explained by analyzing the edge walks and pruned tuples during the concatenation sequence

$(\dots /a/b)$  which is performed in  $P_{nc}$ , but not in  $P_{fc}$ . In  $G_1$ ,  $(\dots /a/b)$  computes a large number of intermediate tuples most of which are later pruned due to a focal point in  $b/a$ . Meanwhile, in  $G_2$ ,  $(\dots /a/b)$  first prunes many tuples due to a focal point in  $a/b$ , hence reducing the total number of edge walks performed later in the search.

Lastly, we consider queries with constants. In pipelined plan  $P_{nc}$ , this constant can be pushed to seed condition  $S$  of its wavefront, therefore reducing term **C**. On the other hand, plans  $P_{pc}$  and  $P_{fc}$  allow at most partial constant pushdown, since cached relations must be computed (term **D**) with universal seed to ensure completeness of the final closure.

## 6.5 Cardinality Estimator

Recall WAVEGUIDE’s cost framework presented in §6.1. The costs of all three operations **crank**, **reduce**, and **cache** depend on the cardinalities  $|\Delta_i|$ ,  $|G|$ , and  $|C_i|$ . The task of WAVEGUIDE’s cardinality estimator is to approximate these cardinalities at each iteration of the guided graph search in order to identify a cost of a given waveplan.

As we mentioned in §2.5.4, cardinality estimation is a well-established research area within the database community. There are three main methods used in cardinality estimation: catalog table statistics, histograms, and sampling. Here, we rely

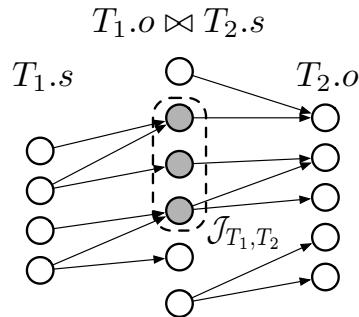


Figure 6.6: Cardinality estimation of a join.

on the simplest method of cardinality estimation which is based on catalog table statistics. We defer exploring histograms and sampling for cardinality estimation in WAVEGUIDE to future work.

In general, a cardinality estimate of join  $T_1 \bowtie_{T_1.o=T_2.s} T_2$  of two tables  $T_1(s, o)$  and  $T_2(s, o)$  can be computed as follows:

$$|T_1 \bowtie_{T_1.o=T_2.s} T_2| = \sum_{j \in \mathcal{J}_{T_1, T_2}} |\sigma_{T_1.o=j}(T_1)| \cdot |\sigma_{T_2.s=j}(T_2)| \quad (6.6)$$

Here, *join set*  $\mathcal{J}_{T_1, T_2}$  (shown in Fig. 6.6) denotes the set of nodes which match the join predicate  $T_1.o = T_2.s$ . For each such node  $j \in \mathcal{J}_{T_1, T_2}$ , the number of paths which go through  $j$  from  $T_1.s$  to  $T_2.o$  is the product  $|\sigma_{T_1.o=j}(T_1)| \cdot |\sigma_{T_2.s=j}(T_2)|$ . Hence, a total cardinality of a join is computed as a sum of cardinalities (numbers of paths) over all  $j \in \mathcal{J}_{T_1, T_2}$ .

Hence, in order to estimate the cardinality of a join of two arbitrary tables  $T_1$  and  $T_2$ , statistics about corresponding join set  $J_{T_1, T_2}$  should be kept. Further,

cardinalities  $|\sigma_{T_1.o=j}(T_1)|$  and  $|\sigma_{T_2.s=j}(T_2)|$  should be maintained for each node  $j \in J_{T_1,T_2}$ . In large databases, maintaining accurate statistics for these is often not feasible. Hence, in most RDBMS today, the following three assumptions are made.

1. **Uniformity.** All nodes  $j$  in a join set  $J_{T_1,T_2}$  have the same number of tuples associated with them in both  $T_1$  and  $T_2$ .
2. **Independence.** Predicates on attributes (in the same table or from joined tables) are independent.
3. **Inclusion.** The domains of join keys overlap such that *all* keys from the smaller domain have matches in the larger domain.

Let  $d(x, T)$  denote a column cardinality of attribute  $x$  in table  $T$ , i.e. a number of distinct values of  $x$  in  $T$ . Then, from **uniformity** we have for all nodes  $j \in J_{T_1,T_2}$ :

$$|\sigma_{T_1.o=j}(T_1)| = \frac{|T_1|}{d(o, T_1)} \quad \text{and} \quad |\sigma_{T_2.s=j}(T_2)| = \frac{|T_2|}{d(s, T_2)} \quad (6.7)$$

Hence, formula (6.6) becomes:

$$|T_1 \bowtie_{T_1.o=T_2.s} T_2| = |J_{T_1,T_2}| \cdot \frac{|T_1|}{d(o, T_1)} \cdot \frac{|T_2|}{d(s, T_2)} \quad (6.8)$$

Further, from **inclusion**, we derive a classical estimation formula which is still widely used in many commercial RDBMS for estimating the cardinality of the join of two tables  $T_1(s, o)$  and  $T_2(s, o)$ :

$$|T_1 \bowtie_{T_1.o=T_2.s} T_2| = \min(d(o, T_1), d(s, T_2)) \cdot \frac{|T_1|}{d(o, T_1)} \cdot \frac{|T_2|}{d(s, T_2)} \quad (6.9)$$

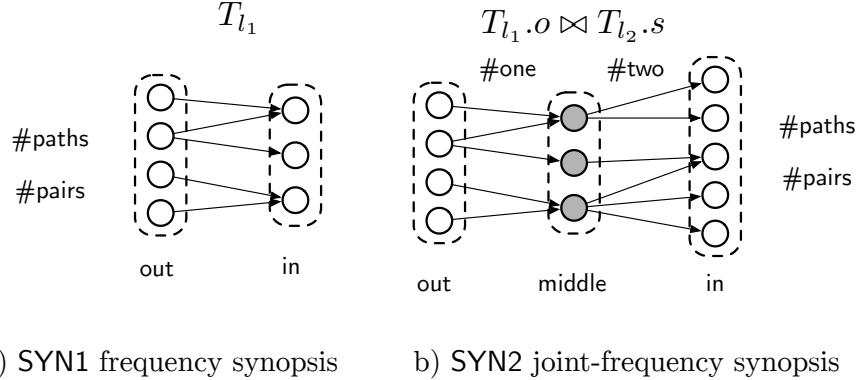


Figure 6.7: Synopsis statistics for graph label frequencies.

### 6.5.1 Synopsis Statistics

In queries with joins on foreign keys (like those used in TPC-H benchmark [16]), formula (6.9) provides acceptable cardinality estimates [40]. However, in WAVEGUIDE, we deal with graph traversals in which intermediate path endpoints  $T_1(s, o)$  and  $T_2(s, o)$  are joined to produce longer paths during `crank` operation. In this scenario, the **inclusion** assumption will almost always significantly overestimate the cardinality due to join set  $\mathcal{J}_{T_1, T_2}$  being significantly smaller than both column cardinalities  $d(o, T_1)$  and  $d(s, T_2)$ .

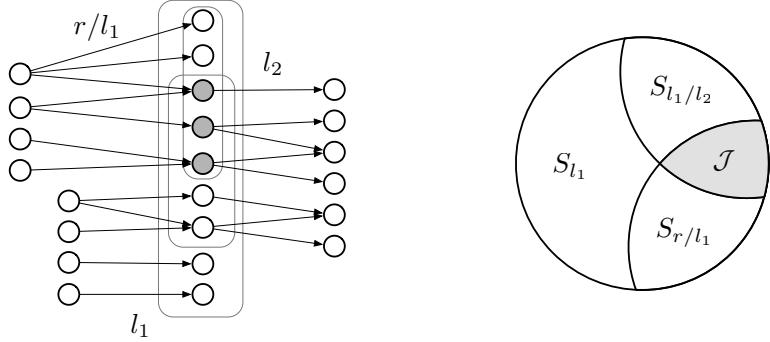
In WAVEGUIDE, in order to provide a better estimation of the size of a join set during `crank` operation, we utilize a compact collection of graph label statistics which we call *synopsis*. As shown in Fig. 6.7a, for each edge label  $l$  in graph  $G$ , synopsis **SYN1** stores the following.

- $|\text{out}|$ : a number of nodes in  $G$  which have outgoing edge labeled with  $l$ ,
- $|\text{in}|$ : a number of nodes in  $G$  which have incoming edge labeled with  $l$ ,
- $\#\text{paths}$ : a number of paths in  $G$  labeled with  $l$ , and
- $\#\text{pairs}$ : a number of distinct node pairs connected with paths labeled with  $l$ .

Similarly as shown in Fig. 6.7b, synopsis **SYN2** stores join-frequency statistics for paths labeled  $l_1/l_2$  in  $G$  each pair of labels  $l_1, l_2$ .

- $|\text{out}|$ : a number of nodes in  $G$  which have outgoing path labeled with  $l_1/l_2$ ,
- $|\text{in}|$ : a number of nodes in  $G$  which have incoming path labeled with  $l_1/l_2$ ,
- $|\text{middle}|$ : a number of nodes in  $G$  which have incoming edge labeled  $l_1$  and outgoing edge labeled  $l_2$ ,
- $\#\text{paths}$ : a number of paths in  $G$  labeled with  $l_1/l_2$ ,
- $\#\text{pairs}$ : a number of distinct node pairs connected with paths labeled with  $l_1/l_2$ ,
- $\#\text{one}$ : a number of paths labeled  $l_1$  from nodes in **out** to nodes in **middle**, and
- $\#\text{two}$ : a number of paths labeled  $l_2$  from nodes in **middle** to nodes in **in**.

Let  $T_{r/l_1}$  denote a table of node pairs  $(s, o)$  such that path between  $s$  and  $o$  in  $G$  conforms to regular expression  $r/l_1$ . Similarly, let  $T_{l_2}$  denote all node pairs in  $G$  such that a path between them conforms to  $l_2$ . Here,  $r$  is an arbitrary regular expression, and  $l_1$  and  $l_2$  are edge labels in  $G$ . Then join  $T_{r/l_1} \bowtie_{T_{r/l_1}.o=T_{l_2}.s} T_{l_2}$  would effectively produce all node pairs in  $G$  such that a path between them conforms to



a) Join  $T_{r/l_1}.o \bowtie T_{l_2}.s$       b) Partition of join candidate nodes

Figure 6.8: Estimating join cardinality using synopsis.

concatenation  $r/l_1/l_2$ .

Using both **SYN1** and **SYN2**, we can estimate a cardinality of concatenation  $T_{r/l_1/l_2}$  as follows. As shown in Fig. 6.8a, let  $S_{l_1}$  denote a set of all nodes with incoming  $l_1$  edges in  $G$ . Then, a subset  $S_{r/l_1} \subseteq S_{l_1}$  will have incoming  $r/l_1$  paths. Another subset of  $S_{l_1/l_2} \subseteq S_{l_1}$  are nodes with incoming  $l_1$  edges and outgoing  $l_2$  edges. An intersection of  $S_{r/l_1}$  and  $S_{l_1/l_2}$  are those nodes which have incoming  $r/l_1$  paths and outgoing  $l_2$  edges. Therefore, a join set  $\mathcal{J}$  is exactly this intersection  $S_{r/l_1} \cap S_{l_1/l_2}$ .

Consider node  $x$  in set  $S_{l_1}$ . Assuming **independence**, the probability that  $x$  is

in the join set  $\mathcal{J}$  is:

$$P[x \in \mathcal{J}] = P[x \in S_{r/l_1} \cap S_{l_1/l_2}] = P[x \in S_{r/l_1}] \cdot P[x \in S_{l_1/l_2}] = \frac{|S_{r/l_1}|}{|S_{l_1}|} \cdot \frac{|S_{l_1/l_2}|}{|S_{l_1}|} \quad (6.10)$$

Consider a Bernoulli trial in which success means node  $x$  is in the join set, and failure otherwise. Then, a probability distribution of a cardinality of a join set  $|J|$  follows binomial distribution  $B(n, p)$  with parameters  $n = |S_{l_1}|$  and  $p = P[x \in \mathcal{J}]$ . Therefore, we can derive an *expected* value of a cardinality of a join set:

$$E[|\mathcal{J}|] = n \cdot p = |S_{l_1}| \cdot \frac{|S_{r/l_1}|}{|S_{l_1}|} \cdot \frac{|S_{l_1/l_2}|}{|S_{l_1}|} = \frac{|S_{r/l_1}| \cdot |S_{l_1/l_2}|}{|S_{l_1}|} \quad (6.11)$$

Observe that  $|S_{r/l_1}|$  is a column cardinality  $d(o, T_{r/l_1})$  of objects  $o$  in table  $T_{r/l_1}$ . Further, from synopsis **SYN2**, we obtain  $|S_{l_1/l_2}| = l_1/l_2.\text{middle}$  and, from synopsis **SYN1**, we obtain  $|S_{l_1}| = l_1.\text{in}$ . Hence, (6.11) becomes:

$$|J| \approx E[|\mathcal{J}|] = \frac{d(o, T_{r/l_1}) \cdot l_1/l_2.\text{middle}}{l_1.\text{in}} \quad (6.12)$$

Above, we approximate cardinality  $|\mathcal{J}|$  of a join set by its expected value  $E[|\mathcal{J}|]$ . If we keep the **independence** and **uniformity** assumptions, but *not inclusion* assumptions, we can estimate a cardinality of a join of two tables  $T_{r/l_1} \bowtie T_{l_2}$  by using (6.8, 6.12) and synopses **SYN1**, **SYN2** as follows:

$$\begin{aligned} |T_{r/l_1/l_2}| &= \frac{d(o, T_{r/l_1}) \cdot l_1/l_2.\text{middle}}{l_1.\text{in}} \cdot \frac{|T_{r/l_1}|}{d(o, T_{r/l_1})} \cdot \frac{|T_{l_1/l_2}|}{d(s, T_{l_1/l_2})} \\ &= |T_{r/l_1}| \cdot \frac{l_1/l_2.\#\text{two}}{l_1.\text{in}} \end{aligned} \quad (6.13)$$

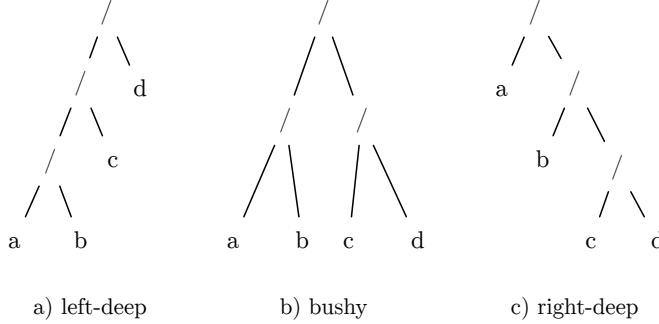


Figure 6.9: Different ways of estimating the cardinality of  $r = a/b/c/d$ .

Further, from **uniformity**, we can estimate column cardinalities of both subjects and objects of the join:

$$d(s, T_{r/l_1/l_2}) = d(s, T_{r/l_1}) \cdot \frac{l_1/l_2.\text{middle}}{l_1.\text{in}} \quad \text{and} \quad d(o, T_{r/l_1/l_2}) = d(o, T_{r/l_1}) \cdot \frac{l_1/l_2.\text{in}}{l_1.\text{in}} \quad (6.14)$$

### 6.5.2 Consistent Estimation

By using formulas (6.13) and (6.14), we can estimate cardinality of any regular expression  $r$  which consists of  $n$  graph labels  $a_i$  and  $n - 1$  concatenations (from (6.15)) by using a bottom-up approach. Given a parse tree of  $r$ , we estimate  $r$ 's cardinality starting from single graph labels in  $r$  and repeatedly combining sub-expressions of  $r$  by going up the parse tree. Since there are many parse trees for any given expression  $r$  (e.g., left-deep, bushy, and right-deep as shown in Fig. 6.9), the estimation would depend on our choice of a parse tree.

In order to be consistent with our estimates in WAVEGUIDE, we fix parse trees for  $r$  to be left-deep. Left- or right-deep trees are considered to be the best choice for cardinality estimation due to the fact that one side of concatenation is always a graph label for which we have accurate statistics in our synopses. Hence, using left- or right-deep trees would typically produce better estimates when compared to bushy (or mixed) trees.

## 6.6 Plan Enumerator

For each regular path query, there are many waveplans which evaluate it. In §5.3, we defined the space of waveplans  $\mathcal{P}_{WP}$  and showed that it properly subsumes  $\mathcal{P}_{FA}$  and  $\mathcal{P}_{\alpha\text{-RA}}$ , the plan spaces of both FA and  $\alpha$ -RA approaches. We need to find a way to *enumerate* through  $\mathcal{P}_{WP}$  in order to select the least expensive (with respect to cost estimation) plan to be executed. Designing an enumerator for *complete*  $\mathcal{P}_{WP}$ , however, is extremely challenging since  $\mathcal{P}_{WP}$  subsumes all FA plans. This means that an FA plan constructed from *any* automaton  $A$  which recognizes a given regular expression should be examined by the enumerator. There exists a multitude of methods on how automaton  $A$  can be obtained by *construction*, *conversion*, and *minimization*. Here, we mention just a few of them:

1. *Thompson's construction* [61] and its optimizations, e.g. the *follow automata* [31],

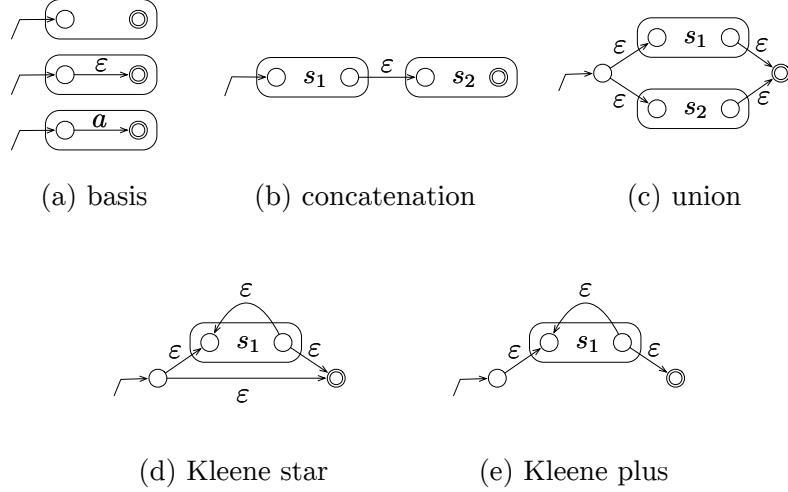


Figure 6.10: Thompson Construction.

2. *Glushkov automaton* [26] or *position automaton* [47] construction, and
  3. *derivative automaton* [6] computation.

We briefly demonstrate how Thompson's construction algorithm produces an NFA for a given regular expression  $r$ . The algorithm works recursively based on the parse tree of  $r$ . We split into cases based on the type of the operator  $op$  in the parse tree and corresponding subexpressions of  $r$ . If  $s$  is a leaf node, then the automaton is constructed as shown in Fig. 6.10a based on whether  $s$  is a graph label  $a$ , an  $\varepsilon$  label, or an empty expression  $\emptyset$ . If  $op$  is a concatenation and  $s_1$  and  $s_2$  are the concatenated subexpressions, then the automaton is constructed as shown in Fig. 6.10b. Fig. 6.10c shows the construction for the union. Figs. 6.10d and 6.10d show the construction for Kleene star and plus, respectively. At any point during

the construction, we can prove by structural induction that the following invariant holds. The resulting automaton  $A$  has: (1) exactly one accepting state, (2) no arcs into the initial state, and (3) no arcs out of the accepting state.

Further methods of obtaining automata from given regular expressions can be found in [8, 10, 12, 24, 30, 62]. Given the multitude of methods of obtaining an FA, and, consequently, an FA plan, the only way to design an enumerator which will be *complete* with respect to  $\mathcal{P}_{WP}$  is to use an *oracle* (or, a black box) which will produce all possible FAs behind-the-scenes for us.

In this work, we are interested in a concrete implementation of the enumerator for waveplans, hence we restrict  $\mathcal{P}_{WP}$  in a way that such enumeration algorithm would exist, it would have an acceptable complexity, and the plan space it would enumerate is still rich enough to be interesting.

In the following subsections, we describe how we accomplish this goal. Specifically, we define  $\mathcal{P}_{SWP}$ , the plan space of *standard* waveplans which are defined by the recursive construction procedure based on a parse tree of a given regular expression. We define  $\mathcal{P}_{TFA}$  as a subset of  $\mathcal{P}_{FA}$  which contains the plans obtained from automata constructed by Thompson's algorithm. We show how  $\mathcal{P}_{SWP}$  compares to  $\mathcal{P}_{WP}$ ,  $\mathcal{P}_{FA}$ ,  $\mathcal{P}_{TFA}$ , and  $\mathcal{P}_{\alpha\text{-RA}}$ . Then, we analyze the size of  $\mathcal{P}_{SWP}$  plan space and show that it is exponential in the size of the regular expression. Finally, we discuss how to design an efficient enumeration of standard waveplans in WAVEGUIDE. A

refinement of this is to enumerate by building up query plans from sub-query plans.

If we can show that this construction procedure exhibits *optimal substructure*, i.e., the least expensive plan for a query can be constructed from least expensive plans for its sub-queries, then dynamic programming approach can be used to reduce the complexity of enumeration. For example, System R [11] is such an approach for queries in relational databases.

### 6.6.1 Standard Plan Space

**Definition 6.1** Given regular path query  $Q = (x, r, y)$ , we define a standard wave-plan  $P$  for  $Q$  as a recursive data structure based on the parse tree of expression  $r$ .

Depending on the type of a node in a parse tree, the construction templates  $T_{1-10}$  are presented in Fig. 6.11. The cases are as follows:

1. If  $r$  is graph label (atom)  $a$ , then  $P_r$  is constructed by creating a single wave-front  $W_a$  by following template  $T_5$  or  $T_6$ .
2. If  $r_1$  and  $r_2$  are subexpressions and  $r = r_1|r_2$ , then  $P_r$  is constructed by following template  $T_{10}$ .
3. If  $r_1$  and  $r_2$  are subexpressions and  $r = r_1/r_2$ , then  $P_r$  is constructed by following templates  $T_{1-4}$  depending on the sizes of subexpressions  $r_1$  and  $r_2$ . Also, if  $r_2$  is a compound expression which has operators other than concatenations, restricted pipeline template  $T_{11}$  is used. Similarly, if  $r_1$  is a compound

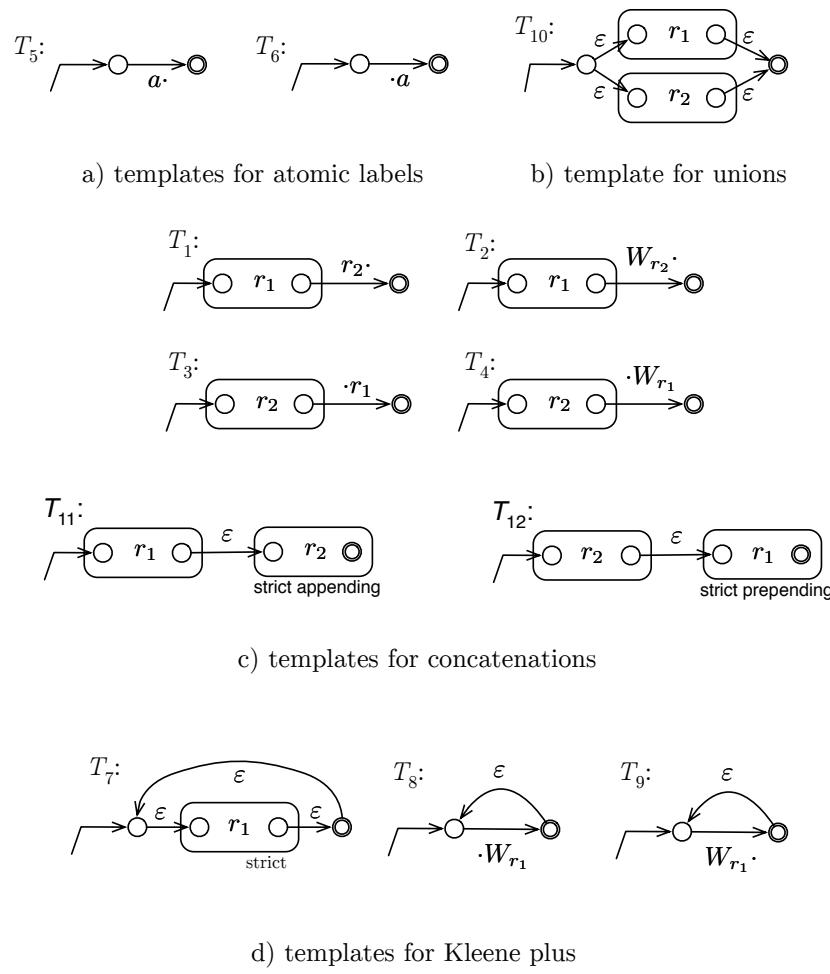


Figure 6.11: Recursive templates for a standard waveplan.

expression which has operators other than concatenations, then  $T_{12}$  is used.

4. If  $r_1$  is a subexpression and  $r = r_1 +^{16}$ , then  $P_r$  is constructed by following templates  $T_{7-9}$ .

Further, standard waveplan  $P_Q$  for query  $Q = (x, r, y)$  has the following invariant which must hold during the recursive definition:  $P_Q$  is legal with respect to the regular expression  $r$  it is designed to evaluate.

**Definition 6.2** We define a space  $\mathcal{P}_{SWP}$  of standard plans for a given query  $Q = (x, r, y)$  such that it contains all legal standard waveplans constructed for all parse trees of  $r$ .

**Definition 6.3** We define a space  $\mathcal{P}_{TFA}$  of FA plans for a given query  $Q = (x, r, y)$  such that the corresponding automaton was constructed by Thompson's algorithm.

**Lemma 6.4** Plan space of standard waveplans  $\mathcal{P}_{SWP}$  properly subsumes the union of  $\mathcal{P}_{\alpha\text{-RA}}$  and  $\mathcal{P}_{TFA}$  ( $\mathcal{P}_{SWP} \supsetneq \mathcal{P}_{TFA} \cup \mathcal{P}_{\alpha\text{-RA}}$ ).

**Proof:** By analyzing the recursive construction of standard waveplans, Thompson's automata, and  $\alpha$ -RA plans. First, we show that SWP extends the TFA model. Consider templates  $T_5$ ,  $T_{10}$ ,  $T_1$ ,  $T_{11}$ , and  $T_7$  in the recursive definition of the standard waveplan. These templates use only appending transitions and do not

---

<sup>16</sup>We only discuss Kleene + operator. Discussion for Kleene \* follows the same ideas.

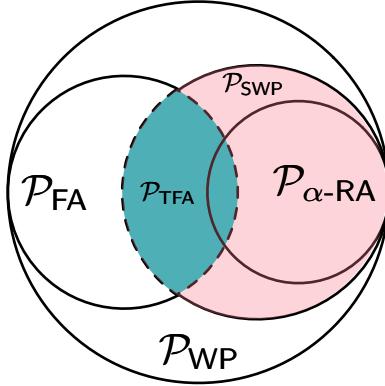
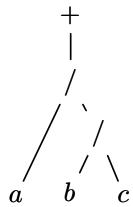


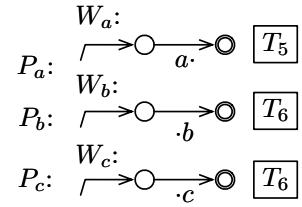
Figure 6.12: Plan space classes.

use views. Observe that by using these templates exclusively during the waveplan construction, we will effectively generate only those plans which correspond to Thompson's automata. Hence,  $\mathcal{P}_{\text{SWP}}$  subsumes  $\mathcal{P}_{\text{TFA}}$ . Second, we show that  $\mathcal{P}_{\text{SWP}}$  subsumes  $\mathcal{P}_{\alpha\text{-RA}}$ . Similarly, consider templates  $T_{1-6}$  and  $T_{8-10}$  which are used in the construction of standard waveplans for all possible parse trees of  $r$ . All parse trees of  $r$  combined with append and prepend transitions correspond to all possible relational algebra (RA) tree plans. Operator  $\alpha$  is captured by transitive closures in  $T_{8,9}$  which use view-labeled transitions over arbitrary relations. Hence,  $\mathcal{P}_{\text{SWP}}$  subsumes  $\mathcal{P}_{\alpha\text{-RA}}$ .

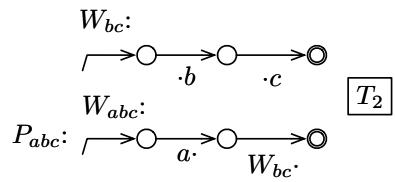
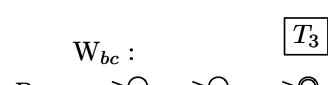
Finally,  $\mathcal{P}_{\text{SWP}}$  properly subsumes the union of  $\mathcal{P}_{\text{TFA}}$  and  $\mathcal{P}_{\alpha\text{-RA}}$  because there exists a standard waveplan that corresponds to neither an TFA plan nor an  $\alpha$ -RA plan. The example of such plan we gave in §5.7 is, in fact, a standard waveplan, e.g., in Fig. 6.4 below,  $P_2$  with partial loop-caching. The trace of generation of a



a) a parse tree of  $(abc)^+$

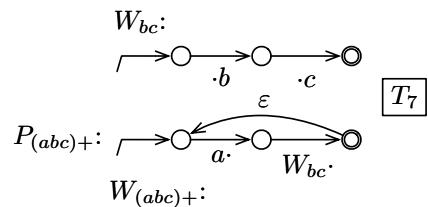


b) processing leafs



c) processing concatenation  $(b/c)$

d) processing concatenation  $(a/b/c)$



e) processing Kleene plus and removing  $\epsilon$  transitions

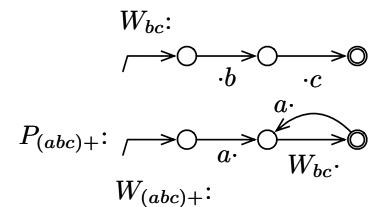


Figure 6.13: Example of SWP generation.

plan like  $P_2$  is shown in Fig. 6.13. All generation steps are shown as a bottom-up traversal of a parse tree for  $(abc)^+$ . At each step, templates which were used are shown. Plans like  $P_2$  are not in  $\mathcal{P}_{\text{TFA}}$  (and, even, not in  $\mathcal{P}_{\text{FA}}$ ) nor they are in  $\mathcal{P}_{\alpha\text{-RA}}$ .

□

**Lemma 6.5**  $\mathcal{P}_{\text{SWP}}$  properly subsumes all plans which are common to FA and  $\alpha$ -RA approaches ( $\mathcal{P}_{\text{SWP}} \supsetneq \mathcal{P}_{\text{FA}} \cap \mathcal{P}_{\alpha\text{-RA}}$ ).

**Proof:** The plans which are common to FA and  $\alpha$ -RA approaches are direct appending pipelines with Kleene closures over single graph labels. These are standard waveplans by construction. □

The total number of plans in  $\mathcal{P}_{\text{SWP}}$  depends on a given query  $Q = (x, r, y)$ . There are many different parse trees for a given regular expression  $r$ . Further, there are many different plans for a fixed parse tree for  $r$  since different templates might be used in the construction of a waveplan. We estimate the asymptotic bound on the size  $|\mathcal{P}_{\text{SWP}}|$  by fixing the concatenation (/), union () and Kleene (\*, +) operators used in the regular expression  $r$  in query  $Q$ .

**Definition 6.6** Let  $|r|$  denote the size of regular expression  $r$ , the number of alphabetical symbols (i.e., atomic graph labels) in  $r$ . Depending on the operator, we define  $|r|$  recursively as follows:

<i>atom a</i>	$r_1/r_2$	$r_1 r_2$	$r_1^*$	$r_1+$
$ r $	1	$ r_1  +  r_2 $	$ r_1  +  r_2 $	$ r_1 $

First, consider regular expression  $r$  of the form:

$$r = a_1/a_2/\dots/a_n \quad (6.15)$$

Here,  $r$  has  $n$  graph labels  $a_i$  and  $n - 1$  concatenations. According to Def. 6.6, size  $|r| = n$ . We want to determine how many different waveplans can evaluate expression  $r$ . Each one of the  $n - 1$  concatenations in  $r$  can be represented as a concatenation of two subexpressions  $r_1$  and  $r_2$  of  $r$ . Given waveplans  $W_{r_1}$  and  $W_{r_2}$  which evaluate  $r_1$  and  $r_2$ , respectively, a waveplan for concatenation  $r$  follows templates  $T_{1-4}$  shown in Fig. 6.11c.<sup>17</sup> In templates  $T_{1,2}$ , paths in the graph conforming to  $r_2$  are *appended* to the paths which conform to  $r_1$ . If size  $|r_2| = 1$  (i.e.,  $r_2$  is a single graph label), then concatenation  $r_1/r_2$  is obtained in  $T_1$  by adding a transition over the graph label  $(r_2 \cdot)$  to the wavefront  $W_{r_1}$  which evaluates  $r_1$ . Otherwise, a transition over the view  $(W_{r_2} \cdot)$  is added in  $T_2$ . Similarly, *prepend*ing  $r_1$  paths to  $r_2$  paths in the graph in templates  $T_{3,4}$  also produces concatenation  $r_1/r_2$ .

Given a split  $r_1/r_2$  and assuming independence of plans for  $r_1$  and  $r_2$ , we obtain a number of plans for templates  $T_2$  and  $T_4$ :

$$\mathcal{P}_{r_1,r_2}^{T_2} = \mathcal{P}_{r_1,r_2}^{T_4} = \mathcal{P}_{r_1} \cdot \mathcal{P}_{r_2} \quad (6.16)$$

---

<sup>17</sup>Templates  $T_{11}$  and  $T_{12}$  are not applicable here as  $r$  does not contain Kleene or union operators.

Similarly, we obtain a number of plans for templates  $T_1$  and  $T_3$ :

$$\mathcal{P}_{r_1,r_2}^{T_1} = \mathcal{P}_{r_1} \quad (6.17)$$

$$\mathcal{P}_{r_1,r_2}^{T_3} = \mathcal{P}_{r_2} \quad (6.18)$$

In order to obtain a total number of plans  $\mathcal{P}_r$ , we analyze all possible splits of  $r$  into  $r_1$  and  $r_2$ , and partition by template:

$$\mathcal{P}_r = \sum_{r_1,r_2} \mathcal{P}_{r_1,r_2}^{T_1} + \sum_{r_1,r_2} \mathcal{P}_{r_1,r_2}^{T_2} + \sum_{r_1,r_2} \mathcal{P}_{r_1,r_2}^{T_3} + \sum_{r_1,r_2} \mathcal{P}_{r_1,r_2}^{T_4} \quad (6.19)$$

We parametrize  $\mathcal{P}(|r|)$  on the size  $|r|$  of  $r$ . In  $T_1$ ,  $r_2$  is fixed at  $|r_2| = 1$ , then the only possible split is when  $|r_1| = |r| - 1$ . From this and (6.17) we have:

$$\sum_{r_1,r_2} \mathcal{P}_{r_1,r_2}^{T_1} = \sum_{|r_1|=|r|-1} \mathcal{P}_{r_1} = \mathcal{P}(|r| - 1) \quad (6.20)$$

Similarly, in  $T_3$ ,  $r_1$  is fixed at  $|r_1| = 1$ , then the only possible split is when  $|r_2| = |r| - 1$ . From this and (6.18) we have:

$$\sum_{r_1,r_2} \mathcal{P}_{r_1,r_2}^{T_3} = \sum_{|r_2|=|r|-1} \mathcal{P}_{r_2} = \mathcal{P}(|r| - 1) \quad (6.21)$$

In  $T_2$ ,  $|r_2| > 1$  and no restrictions are placed on  $r_1$ , hence from (6.16) we have:

$$\sum_{r_1,r_2} \mathcal{P}_{r_1,r_2}^{T_2} = \sum_{r_1,r_2} \mathcal{P}_{r_1} \cdot \mathcal{P}_{r_2} = \sum_{k=1}^{|r|-2} \mathcal{P}(k) \cdot \mathcal{P}(|r| - k) \quad (6.22)$$

Similarly, in  $T_4$ ,  $|r_1| > 1$  and no restrictions are placed on  $r_2$ , hence from (6.16) we have:

$$\sum_{r_1,r_2} \mathcal{P}_{r_1,r_2}^{T_4} = \sum_{r_1,r_2} \mathcal{P}_{r_1} \cdot \mathcal{P}_{r_2} = \sum_{k=2}^{|r|-1} \mathcal{P}(k) \cdot \mathcal{P}(|r| - k) \quad (6.23)$$

From (6.19), (6.20), (6.21), (6.22), (6.23) and after re-arranging the terms, we obtain a recurrence formula for  $\mathcal{P}$ :

$$\mathcal{P}(|r|) = 2 \cdot \left[ \mathcal{P}(|r|-1) + \sum_{k=1}^{|r|-2} \mathcal{P}(k) \cdot \mathcal{P}(|r|-k) \right] \quad (6.24)$$

Fig. 6.11a depicts the possible plans when regular expression  $r$  consists of a single path label  $a$ . In this scenario, plans would consist of either append or prepend transitions over  $a$ . Hence, we obtain a base case for  $\mathcal{P}(|r|)$ :

$$\mathcal{P}(1) = 2 \quad (6.25)$$

From (6.24) and (6.25) we can compute first few values for  $\mathcal{P}(|r|)$ :

$ r $	$\mathcal{P}( r )$
1	2
2	4
3	24
4	176
5	1440
6	12608
7	115584
8	1095424

As shown above, the number of waveplans for  $r$  grows quickly with the size of  $r$ . In order to analyze the complexity of  $\mathcal{P}(|r|)$ , we need to obtain the *closed form* of the recurrence in (6.24). We can obtain this form by using the technique known as *generating functions*.

We begin by defining a power series  $f(z)$  which contain  $\mathcal{P}(|r|)$  (written as  $\mathcal{P}_{|r|}$  for the sake of compactness) for all values of  $|r|$ :

$$f(z) = \mathcal{P}_1 + \mathcal{P}_2 \cdot z + \mathcal{P}_3 \cdot z^2 + \mathcal{P}_4 \cdot z^3 + \dots = \sum_{i=1}^{\infty} \mathcal{P}_i \cdot z^{i-1} \quad (6.26)$$

Now, consider the partial expansion of expression  $2z(f^2(z) + f(z))$ :

$$2z(f^2(z) + f(z)) = 2 \cdot \left( (\mathcal{P}_1 + \mathcal{P}_1^2)z + (\mathcal{P}_2 + 2\mathcal{P}_1\mathcal{P}_2)z^2 + (\mathcal{P}_3 + 2\mathcal{P}_1\mathcal{P}_3 + \mathcal{P}_2^2)z^3 + \right. \\ \left. + (\mathcal{P}_4 + 2\mathcal{P}_1\mathcal{P}_4 + 2\mathcal{P}_2\mathcal{P}_3)z^4 + (\mathcal{P}_5 + 2\mathcal{P}_1\mathcal{P}_5 + \mathcal{P}_3^2 + 2\mathcal{P}_2\mathcal{P}_4)z^5 + \dots \right)$$

After further expansion and rearrangement of terms, we obtain:

$$2z(f^2(z) + f(z)) = 2\mathcal{P}_1z + 2\mathcal{P}_1z \underbrace{(\mathcal{P}_1 + \mathcal{P}_2z + \mathcal{P}_3z^2 + \dots)}_{f(z))} + \left( \underbrace{2(\mathcal{P}_2 + \mathcal{P}_1\mathcal{P}_2)}_{\mathcal{P}_3} z^2 + \right. \\ \left. + \underbrace{2(\mathcal{P}_3 + \mathcal{P}_1\mathcal{P}_3 + \mathcal{P}_2^2)}_{\mathcal{P}_4} z^3 + \underbrace{2(\mathcal{P}_4 + \mathcal{P}_1\mathcal{P}_4 + 2\mathcal{P}_2\mathcal{P}_3)}_{\mathcal{P}_5} z^4 + \dots \right)$$

After replacing the power series with definitions for  $f(z)$ , we get:

$$2z(f^2(z) + f(z)) = 2\mathcal{P}_1z + 2\mathcal{P}_1z \cdot f(z) + (f(z) - \mathcal{P}_2z - p_1) \quad (6.27)$$

Rewrite (6.27) in canonical quadratic equation for  $f(z)$ :

$$(2z)f^2(z) + (2z - 2\mathcal{P}_1z - 1) \cdot f(z) + \mathcal{P}_2z - 2\mathcal{P}_1z + \mathcal{P}_1 \quad (6.28)$$

Simplify (6.28) by plugging the initial conditions  $P_1 = 2$  and  $P_2 = 4$  from 6.25:

$$(2z) \cdot f^2(z) + (-2z - 1) \cdot f(z) + 2 = 0 \quad (6.29)$$

Solve (6.29) for  $f(z)$ :

$$f(z) = \frac{(2z + 1) \pm \sqrt{(-2z - 1)^2 - 4 \cdot 2z \cdot 2}}{2 \cdot 2z} = \frac{1}{2} + \frac{1 \pm \sqrt{4z^2 - 12z + 1}}{4z} \quad (6.30)$$

We can rewrite the square root in (6.30) as a binomial expansion according to the formula:

$$(a+b)^n = a^n + \frac{n}{1}a^{n-1}b^1 + \frac{n(n-1)}{2 \cdot 1}a^{n-2}b^2 + \frac{n(n-1)(n-2)}{3 \cdot 2 \cdot 1}a^{n-3}b^3 + \dots \quad (6.31)$$

Which results in:

$$\begin{aligned} (1 + (4z^2 - 12z))^{\frac{1}{2}} &= 1 + \frac{1}{2}(4z^2 - 12z) + \frac{-\frac{1}{2} \cdot \frac{1}{2}}{2 \cdot 1}(4z^2 - 12z)^2 + \\ &+ \frac{\frac{1}{2} \cdot (-\frac{1}{2})(-\frac{3}{2})}{3 \cdot 2 \cdot 1}(4z^2 - 12z)^3 + \dots = \sum_{n=0}^{\infty} \prod_{j=1}^n \frac{(\frac{3}{2} - j)}{n!}(4z^2 - 12z)^n \end{aligned} \quad (6.32)$$

Next, we expand  $(4z^2 - 12z)^n$  for all  $n$  by using the binomial formula, again:

$$(4z^2 - 12z)^n = \sum_{k=0}^n \frac{n!}{k!(n-k)!} \cdot (-12z)^{n-k} \cdot (4z^2)^k \quad (6.33)$$

Hence, the formula (6.32) becomes:

$$\begin{aligned} (1 + (4z^2 - 12z))^{\frac{1}{2}} &= 1 + \sum_{n=1}^{\infty} \prod_{j=1}^n \frac{(\frac{3}{2} - j)}{n!} \sum_{k=0}^n \frac{n!}{k!(n-k)!} \cdot (-12z)^{n-k} \cdot (4z^2)^k \\ & \end{aligned} \quad (6.34)$$

We substitute the square root in (6.30) by the series obtained in (6.34). Here, our goal is to obtain the formula for coefficients appearing next to each  $z^n$  in the resulting expression since these are  $(\mathcal{P}_0 \dots \mathcal{P}_n)$  by the definition of  $f(z)$ . It is, however, not trivial since each coefficient comes from expansions of  $(4z^2 - 12z)^n$  for different  $n$  (due to a nested sum in (6.34)). Finally, we unwind the nested sum in (6.34) to obtain the closed-form for  $\mathcal{P}_n$ :

$$\mathcal{P}(n) = -\frac{1}{4} \cdot \left( \sum_{k=\lceil \frac{n+1}{2} \rceil}^{n+1} \frac{\prod_{j=1}^k (\frac{3}{2} - j) \cdot 4^k \cdot (-3)^{2k-n-1}}{(2k-n-1)!(n+1-k)!} \right) = \Theta((4 \cdot 3)^n) \quad (6.35)$$

Hence, we have shown that  $\mathcal{P}(|r|)$  grows exponentially in size of  $r$ :

$$\mathcal{P}(|r|) = \Theta(c^{|r|}) \quad (6.36)$$

For a given constant  $c$ .

Next, consider regular expression  $r$  of the form:

$$r = r_1 + \quad (6.37)$$

Here,  $r$  has a Kleene plus which produces one or more concatenations of  $r_1$  with itself. Again, we want to determine how many different waveplans can evaluate expression  $r$ . Fig. 6.11d shows waveplan templates  $T_{7-9}$  for Kleene plus in  $r$ . In  $T_8$  and  $T_9$ , the transitive closure is computed by a fully loop cached plan; i.e.,  $r_1$  is evaluated first by wavefront  $W_{r_1}$  and then its results are used in a  $\varepsilon$ -pipeline loop to compute the results of  $r$  by either appending or prepending transitions over the view  $W_{r_1}$ . Alternatively, in  $T_7$ , the transitive closure is computed by repeatedly pipelining the results of  $W_{r_1}$  into itself. Depending on the structure of the wavefront  $W_{r_1}$ , plans in  $T_7$  have either partial or no loop caching (i.e., they are fully pipelined).

As before, we partition the plans for  $r$  by templates:

$$\mathcal{P}_r = \mathcal{P}_r^{T_7} + \mathcal{P}_r^{T_8} + \mathcal{P}_r^{T_9} \quad (6.38)$$

First, we consider fully cached wavefronts in templates  $T_7$  and  $T_8$ . Here, any wave-

front for  $r_1$  can be used, hence, the number of plans for  $r$  is the same as for  $r_1$ :

$$\mathcal{P}_r^{T_7} = \mathcal{P}_r^{T_8} = \mathcal{P}_{r_1} \quad (6.39)$$

Now, consider the wavefronts in template  $T_9$ . Here, a wavefront for  $r_1$  is pipelined into itself. In this scenario, not all wavefronts for  $r_1$  would produce a *legal* plan for  $r$ .

**Lemma 6.7** *There exists regular expression  $r_1$  such that wavefront  $W_r$  constructed by following template  $T_9$  would produce a plan which is illegal with respect to expression  $r = r_1 +$ .*

**Proof:** We provide an example of such regular expression  $r_1$ . Consider Ex. 6 in which wavefronts  $W_{r_1}$  are constructed for expression  $r_1 = abc$  as shown in Fig.5.2. Fig.5.3 depicts wavefronts  $W_r$  which were constructed by following template  $T_9$  given  $W_{r_1}$ . Observe that two out of four wavefronts  $W_r$  are illegal with respect to  $r$ . □

It follows that if we use *any* wavefront  $W_{r_1}$  in  $T_9$ , we are at risk of producing an illegal plan for  $r$ . Hence, set of plans  $W_{r_1}$  needs to be restricted to guarantee legal plans for  $r$ .

**Definition 6.8** *We call a wavefront strict if it has either only appending or only prepending transitions. Strict wavefronts expand in a single direction, i.e. they are unidirectional.*

**Definition 6.9** *Wavefronts which are not strict expand in multiple directions; i.e. we call these omnidirectional.*

**Lemma 6.10** *For all regular expressions  $r_1$  if wavefront  $W_{r_1}$  is strict, then wavefront  $W_r$  constructed by following template  $T_9$  would produce a plan which is legal with respect to expression  $r = r_1 +$ .*

**Proof:** The proof proceeds by showing that path label string produced by repeatedly appending *or* repeatedly prepending  $r_1$  to it, in fact, produces a path label string which matches expression  $r = r_1 +$ .  $\square$

Consider  $r$  to be a chain of  $n - 1$  concatenations of graph labels (as in (6.15)), hence  $|r| = n$ . Earlier, we have identified the number of waveplans  $\mathcal{P}_r$  for this query. Now, we want to see how many of these waveplans has strict  $W_r$ , we denote as  $\mathcal{S}_r$ . First, we partition  $\mathcal{S}_r$  into plans in which  $W_r$  is only appending and those which  $W_r$  is only prepending:

$$\mathcal{S}_r = \overleftarrow{\mathcal{S}}_r + \overrightarrow{\mathcal{S}}_r \quad (6.40)$$

Again, we consider a split of  $r$  into  $r_1/r_2$ . Observe that for strictly appending wavefronts only templates  $T_1$  and  $T_2$  are used. Similarly, templates  $T_3$  and  $T_4$  are

used for strictly prepending wavefronts:

$$\vec{\mathcal{S}}_r = \sum_{r_1, r_2} \vec{\mathcal{S}}_{r_1, r_2}^{T_1} + \sum_{r_1, r_2} \vec{\mathcal{S}}_{r_1, r_2}^{T_2} \quad (6.41)$$

$$\overleftarrow{\mathcal{S}}_r = \sum_{r_1, r_2} \overleftarrow{\mathcal{S}}_{r_1, r_2}^{T_3} + \sum_{r_1, r_2} \overleftarrow{\mathcal{S}}_{r_1, r_2}^{T_4} \quad (6.42)$$

Given a split  $r_1/r_2$ , we assume independence of plans for  $r_1$  and  $r_2$ . Observe that for  $W_r$  to be strict, wavefront for  $r_1$ ,  $W_{r_1}$ , also needs to be strict. On the other hand,  $r_2$  is evaluated over the view in  $T_2$ , hence no restrictions are necessary for  $W_{r_2}$ . Therefore, we use unrestricted  $\mathcal{P}_{r_2}^{T_2}$  to denote the number of waveplans for  $r_2$ . Finally, we obtain a number of waveplans with strict wavefront for  $r$  in  $T_2$ :

$$\vec{\mathcal{S}}_{r_1, r_2}^{T_2} = \vec{\mathcal{S}}_{r_1}^{T_2} \cdot \mathcal{P}_{r_2}^{T_2} \quad (6.43)$$

Similarly, we obtain the number of waveplans for template  $T_4$ :

$$\overleftarrow{\mathcal{S}}_{r_1, r_2}^{T_4} = \overleftarrow{\mathcal{S}}_{r_1}^{T_4} \cdot \mathcal{P}_{r_2}^{T_4} \quad (6.44)$$

From (6.43) and (6.44), we obtain  $\mathcal{S}$  for  $T_1$  and  $T_3$ :

$$\vec{\mathcal{S}}_{r_1, r_2}^{T_1} = \vec{\mathcal{S}}_{r_1}^{T_1} \cdot 1 = \vec{\mathcal{S}}_{r_1}^{T_1} \quad (6.45)$$

$$\overleftarrow{\mathcal{S}}_{r_1, r_2}^{T_3} = \overleftarrow{\mathcal{S}}_{r_1}^{T_3} \cdot 1 = \overleftarrow{\mathcal{S}}_{r_1}^{T_3} \quad (6.46)$$

As we did for  $\mathcal{P}(|r|)$ , we parametrize  $\mathcal{S}(|r|)$  on the size of  $r$ . From (6.41), (6.43), and (6.45), we obtain a recurrence formula for  $\vec{\mathcal{S}}(|r|)$  for all splits  $r_1/r_2$ :

$$\vec{\mathcal{S}}(|r|) = \vec{\mathcal{S}}(|r|-1) + \sum_{k=1}^{|r|-2} \vec{\mathcal{S}}(k) \cdot \mathcal{P}(|r|-k) \quad (6.47)$$

Similarly, we have

$$\overleftarrow{\mathcal{S}}(|r|) = \overleftarrow{\mathcal{S}}(|r|-1) + \sum_{k=1}^{|r|-2} \overleftarrow{\mathcal{S}}(k) \cdot \mathcal{P}(|r|-k) \quad (6.48)$$

Since only append or only prepend transitions can be used over atom graph label, we obtain base cases:

$$\overleftarrow{\mathcal{S}}(1) = 1 \quad (6.49)$$

$$\overrightarrow{\mathcal{S}}(1) = 1 \quad (6.50)$$

From (6.24), (6.40), (6.47-6.50) we can compute first few values for  $\mathcal{S}(n)$ :

$ r $	$\mathcal{P}( r )$	$\mathcal{S}( r )$
1	2	2
2	4	2
3	24	10
4	176	66
5	1440	506
6	12608	4242
7	115584	37706
8	1095424	349218

As expected,  $\mathcal{S}(|r|)$  grows more slowly than  $\mathcal{P}(|r|)$  since plans with a strict wavefront represent a subset of all plans. However, this restriction applies only to the wavefront  $W_{r_1}$  which evaluates  $r_1$  and not the wavefronts which evaluate sub-expressions of  $r_1$  through views in  $W_{r_1}$ . Hence,  $\mathcal{S}(|r|)$  asymptotically grows the same as  $\mathcal{P}(|r|)$ , exponential in the size of regular expression  $|r|$ .

We conclude that Kleene operator in  $r$  does not asymptotically increase the number of plans for  $r_1$ . In fact, it restricts the plans which can be used in  $r_1$

when template  $T_9$  is used. Similarly, direct and inverse pipelines which are used to concatenate compound expressions in templates  $T_{11}$  and  $T_{12}$  use strict wavefronts and do not asymptotically increase the number of plans.

Finally, we consider the regular expression of the form:

$$r = r_1 | r_2 | \cdots | r_n \quad (6.51)$$

Here,  $r$  is a union of  $n$  regular expressions  $r_1, \dots, r_n$ . Given waveplans  $W_{r_1}, \dots, W_{r_n}$  for each subexpression of  $r$ , a waveplan for a union  $W_r$  follows template  $T_{10}$  shown in Fig. 6.11b. In  $T_{10}$ , a union for  $r$  is evaluated by using transitions over views for subexpressions  $r_1, \dots, r_n$ . Effectively, this means that each subexpression is considered independently of the others. Hence, the number of plans for union  $r$  is:

$$\mathcal{P}_r = \prod_{i=1}^n \mathcal{P}_{r_i} \quad (6.52)$$

From (6.36) each  $\mathcal{P}_{r_i}$  grows exponentially in the size  $|r_i|$  of  $r_i$ . Also, by Def. 6.6, size of  $r$  is the sum of sizes of subexpressions  $r_i$ . It follows:

$$\mathcal{P}_r = \prod_{i=1}^n \mathcal{P}_{r_i} = \Theta(c^{|r_1| + \dots + |r_n|}) = \Theta(c^{|r|}) \quad (6.53)$$

**Theorem 6.11** *Given regular expression  $r$ , a number of legal standard waveplans which evaluate it is exponential in the size  $|r|$  of this regular expression.*

**Proof:** Follows directly from structural induction on all parse trees of  $r$  in which, during the inductive step, concatenation, union, and Kleene operators are examined. As we have shown above, these produce the number of standard waveplans exponential in the size of an arbitrary underlying regular expression.  $\square$

In §6.3.1, we discussed the choice of wavefronts. A wavefront as discussed above is a *pipelined* fragment of a plan. Our plan space covers all legal orderings (sequences of appends and prepends) that can constitute a wavefront. In §6.3.3, we discussed the concept of *threading*. Accommodating threading in plans is complex; we will discuss this further at the end of the next section. In §6.3.5, we introduced the optimization of *loop caching*. This is equivalent to view fragments in plans for subexpressions in Kleene closures.<sup>18</sup>

### 6.6.2 Enumeration

As per Theorem 6.11, the number of standard waveplans which evaluate a regular path query is *exponential* in the size of this query. We propose a plan enumerator for WAVEGUIDE plans which uses a dynamic programming approach which allows it to work in time *polynomial* in the size of the given query.

Dynamic programming generates the solutions for a larger problem by combining solutions for smaller problems in a bottom-up fashion. In this way, we can

---

<sup>18</sup>In §6.3.2 and §6.3.4, we introduced the optimizations of *reduce* and *partial caching*. These are runtime optimizations which are not part of the plan space.

construct an optimal plan for the regular expression of size  $l$  by *combining* optimal plans for regular expressions of sizes  $k$  and  $l - k$ , respectively. Enumeration algorithm **WGEnum** is presented in Fig. 6.14. It takes as an input an RPQ  $Q$ , which is a triple  $(x, r, y)$  with  $r$  being a regular expression and  $x, y$  are either variables or constants. Table *bestPlan* is used for memoization of the lowest cost waveplan for a given expression and seed. The algorithm starts by computing the best plans for expressions  $s \subseteq r$  of size one; i.e., for regular expressions with a single graph label. After that, it constructs plans for expressions of increasing size  $l$ , by combining plans for expressions of smaller sizes  $l_1$  and  $l_2$ , such that  $l = l_1 + l_2$ . Since there are many such sub-expressions, the algorithm loops over all sub-expressions of  $r$ :  $s_1$  and  $s_2$  with sizes  $l_1$  and  $l_2$ , respectively. The loop finishes when  $l = |r|$ , and the best plan for  $r$  can be extracted.

```

memoizeMin ( $P, r$ ) :
  1  for each  $p \in P$ :
  2    if  $\text{cost}(p) < \text{cost}(\text{bestPlan}(r, p.\text{seed}))$  :
  3       $\text{bestPlan}(r, p.\text{seed}) \leftarrow p$ ;

```

Figure 6.15: WAVEGUIDE's memoization sub-routine.

```

WGEnum ( $Q = (x, r, y)$ ) :

1   for all  $s \subseteq r : |s| = 1$ :
2     seedConst ( $s, Q$ ) ;
3     seedKleene ( $s, Q$ ) ;
4     if ( $s = a$ ) :
5        $P \leftarrow \text{getPlans} (a, \text{null}, \text{null}, \text{null})$  ;
6       memoizeMin ( $P, a$ ) ;
7     if ( $s_i = a+$ ) :
8        $P \leftarrow \text{getPlans} (a, \text{null}, +, \text{null})$  ;
9       memoizeMin ( $P, a+$ ) ;
10    for  $1 < l \leq |r|$ :
11      for  $1 \leq l_1 < l$ :
12         $l_2 = l - l_1$ ;
13        for each  $s_1 \subseteq r : |s_1| = l_1$  and
14           $s_2 \subseteq r : |s_2| = l_2$ :
15          if ( $s_1 / s_2 \subseteq r$ ) :
16             $P \leftarrow \text{getPlans} (s_1, s_2, /, \text{null})$  ;
17            memoizeMin ( $P, s_1 / s_2$ ) ;
18          if ( $s_1 | s_2 \subseteq r$ ) :
19             $P \leftarrow \text{getPlans} (s_1, s_2, |, \text{null})$  ;
20            memoizeMin ( $P, s_1 | s_2$ ) ;
21        for each  $s \subseteq r : |s| = l$ :
22          seedConst ( $s, Q$ ) ;
23          seedKleene ( $s, Q$ ) ;
24          if ( $s = s_1+ \subseteq r$ ) :
25             $P \leftarrow \text{getPlans} (s_1, \text{null}, +, \text{null})$  ;
26            memoizeMin ( $P, s_1+$ ) ;
27  return bestPlan ( $r, Q$ ) ;

```

Figure 6.14: WAVEGUIDE's enumeration procedure.

```

getPlans ( $s_1, s_2, op, seed$ ) :

1 Plans  $ps$ ;
2 for each  $d_1 \in \text{getSeeds}(s_1)$  and  $d_2 \in \text{getSeeds}(s_2)$  :
3    $p_1 \leftarrow \text{bestPlan}(s_1, d_1)$  ;
4    $p_2 \leftarrow \text{bestPlan}(s_2, d_2)$  ;
5   for each  $\text{rule} \in ER$ :
6     if ( $\text{rule}.\text{precondition}(s_1, s_2, op, seed)$  ) :
7       Plan  $p \leftarrow \text{rule}.\text{genPlan}(p_1, p_2, seed)$  ;
8        $ps.\text{add}(p)$  ;
9   return  $ps$ ;

```

Figure 6.16: WAVEGUIDE's plan generation sub-routine.

```

seedConst ( $s, Q = (x, r, y)$ ) :

1 if ( $x = c$  and  $s \subseteq_{pre} r$  ) :
2    $P \leftarrow \text{getPlans}(s, \text{null}, \text{null}, (c, \varepsilon, y))$  ;
3 else if ( $y = c$  and  $s \subseteq_{suf} r$  ) :
4    $P \leftarrow \text{getPlans}(\text{null}, s, \text{null}, (x, \varepsilon, c))$  ;
5    $\text{memoizeMin}(P, s)$  ;

```

Figure 6.17: WAVEGUIDE's constant seed passing sub-routine.

```

seedKleene ( $s, Q = (x, r, y)$ ) :

1 if ( $s \subseteq_{pre} s_1$  and  $s_1+ \subseteq r$  ) :
2   for all  $d \leftarrow \text{preSeed}(s_1+, Q)$  :
3      $P \leftarrow \text{getPlans}(s, \text{null}, \text{null}, d/s_1+)$  ;
4      $\text{memoizeMin}(P, s)$  ;
5 if ( $s \subseteq_{suf} s_1$  and  $s_1+ \subseteq r$  ) :
6   for all  $d \leftarrow \text{postSeed}(s_1+, Q)$  :
7      $P \leftarrow \text{getPlans}(\text{null}, s, \text{null}, s_1+/d)$  ;
8      $\text{memoizeMin}(P, s)$  ;

```

Figure 6.18: WAVEGUIDE's Kleene seed passing sub-routine.

rule		waveplan	precondition			
<i>id</i>	<i>description</i>		<i>s</i> <sub>1</sub>	<i>s</i> <sub>2</sub>	<i>op</i>	<i>seed</i>
AA	atom append	$p : \begin{array}{c} \nearrow \circlearrowleft \\ U \end{array} \xrightarrow{s_1 \cdot} \circlearrowright$	$ s_1  = 1$	null	null	null
AP	atom prepend	$p : \begin{array}{c} \nearrow \circlearrowleft \\ U \end{array} \xleftarrow{\cdot s_1} \circlearrowright$	$ s_1  = 1$	null	null	null

Figure 6.19: Enumeration rules for graph label expressions.

Several helper sub-routines are used in `WGEnum()`. These are `memoizeMin()`, `getPlans()`, `seedConst()`, and `seedKleene()`. In `memoizeMin()`, shown in Fig. 6.15, if the given set of plans  $P$  contains plan  $p$  which is cheaper than one stored in  $bestPlan$  (with the same seed), then  $bestPlan$  is updated by replacing previous cheapest plan for  $r$  with  $p$ . Routine `getPlans()` is presented in Fig. 6.16. It *combines* the best plans for given subexpressions  $s_1$  and  $s_2$  based on the operator  $op$  between  $s_1$  and  $s_2$  in  $r$ . The way how best plans for  $s_1$  and  $s_2$  are combined is dictated by operator  $op$ , given seed  $seed$ , and a set of *enumeration rules*  $ER$  shown in Figs. 6.19-6.23. Each enumeration rule constructs a waveplan  $p$  based on the validity of its *precondition*. The `precondition()` routine checks for presence and size of regular expressions  $s_1$  and  $s_2$ , for the type of operator  $op$ , and a given  $seed$ .

Consider enumeration rule **KP** for Kleene star operator presented in Fig. 6.22. Blind application of **KP** to any given waveplan for  $s_1$  might produce an illegal plan for  $s_1+$ . Recall that according to Lemma 6.10, Kleene operator in this scenario

<i>rule id</i>	<i>rule description</i>	<i>waveplan</i>	<i>precondition</i>			
			<i>s</i> <sub>1</sub>	<i>s</i> <sub>2</sub>	<i>op</i>	<i>seed</i>
CC	concat compound	$p : \begin{array}{c} \xrightarrow{\quad} \circlearrowleft s_1 \circlearrowright \xrightarrow{\quad} \\ \text{---} \end{array} \cdot W_{s_2} \cdot \circlearrowleft \circlearrowright$	$ s_1  > 1$	$ s_2  > 1$	$d_2 = U$	/
CCF	concat compound flip	$p : \begin{array}{c} \xrightarrow{\quad} \circlearrowleft s_2 \circlearrowright \xrightarrow{\quad} \\ \text{---} \end{array} \cdot W_{s_1} \cdot \circlearrowleft \circlearrowright$	$d_1 = U$	$ s_2  > 1$	/	null
CP	concat pipe	$p : \begin{array}{c} \xrightarrow{\quad} \circlearrowleft s_1 \circlearrowright \xrightarrow{s_2 \cdot} \\ \text{---} \end{array} \circlearrowleft \circlearrowright$	$ s_1  > 0$	$ s_2  = 1$	/	null
CPF	concat pipe flip	$p : \begin{array}{c} \xrightarrow{\quad} \circlearrowleft s_2 \circlearrowright \xrightarrow{\cdot s_1} \\ \text{---} \end{array} \circlearrowleft \circlearrowright$	$ s_1  = 1$	$ s_2  > 0$	/	null
DP	direct pipeline	$p : \begin{array}{c} \xrightarrow{\quad} \circlearrowleft s_1 \circlearrowright \xrightarrow{\varepsilon} \circlearrowleft \circlearrowright \xrightarrow{\varepsilon} \circlearrowleft s_2 \circlearrowright \xrightarrow{\quad} \\ \text{---} \end{array} \circlearrowleft \circlearrowright$	$ s_1  > 0$	$d_2 = s_1$	/	null
DP	inverse pipeline	$p : \begin{array}{c} \xrightarrow{\quad} \circlearrowleft s_2 \circlearrowright \xrightarrow{\varepsilon} \circlearrowleft \circlearrowright \xrightarrow{\varepsilon} \circlearrowleft s_1 \circlearrowright \xrightarrow{\quad} \\ \text{---} \end{array} \circlearrowleft \circlearrowright$	$d_1 = s_2$	$ s_2  > 0$	/	null

Figure 6.20: Enumeration rules for the concatenation operator.

<i>rule id</i>	<i>rule description</i>	<i>waveplan</i>	<i>precondition</i>			
			<i>s</i> <sub>1</sub>	<i>s</i> <sub>2</sub>	<i>op</i>	<i>seed</i>
U	union	$p : \begin{array}{c} \xrightarrow{\quad} \circlearrowleft \circlearrowright \xrightarrow{\varepsilon} \circlearrowleft \circlearrowright \xrightarrow{\varepsilon} \circlearrowleft \circlearrowright \xrightarrow{\varepsilon} \circlearrowleft \circlearrowright \\ \text{---} \end{array} \circlearrowleft \circlearrowright$	$ s_1  > 0$	$ s_2  > 0$	/	null

Figure 6.21: Enumeration rule for the union operator.

rule		waveplan	precondition			
<i>id</i>	<i>description</i>		<i>s</i> <sub>1</sub>	<i>s</i> <sub>2</sub>	<i>op</i>	<i>seed</i>
KP	kleene plus	$p : \begin{array}{c} \xrightarrow{d} \circ \xrightarrow{\varepsilon} \text{---} \xrightarrow{\varepsilon} \circ \\ s_1 \end{array}$	$d_1 = d/(s_1) +$ $d_1 = (s_1) + / d$	null	+	null
KS	kleene star	$p : \begin{array}{c} \xrightarrow{d} \circ \xrightarrow{\varepsilon} \text{---} \xrightarrow{\varepsilon} \circ \\ s_1 \end{array}$	$d_1 = d/(s_1)^*$ $d_1 = (s_1)^* / d$	null	*	null

Figure 6.22: Enumeration rules for Kleene closures.

rule		waveplan	precondition			
<i>id</i>	<i>description</i>		<i>s</i> <sub>1</sub>	<i>s</i> <sub>2</sub>	<i>op</i>	<i>seed</i>
ASDP	absorb seed direct pipe	$p : \begin{array}{c} \xrightarrow{d} \circ \xrightarrow{s_1} \circ \\ \cdot \end{array}$	$ s_1  = 1$	null	null	$d$
ASIP	absorb seed inverse pipe	$p : \begin{array}{c} \xrightarrow{d} \circ \xrightarrow{\cdot s_2} \circ \\ \cdot \end{array}$	null	$ s_2  = 1$	null	$d$
ASDC	absorb seed direct compound	$p : \begin{array}{c} \xrightarrow{d} \circ \xrightarrow{W_{s_1}} \circ \\ \cdot \end{array}$	$ s_1  > 1$ $d_1 = U$	null	null	$d$
ASIC	absorb seed inverse compound	$p : \begin{array}{c} \xrightarrow{d} \circ \xrightarrow{\cdot W_{s_2}} \circ \\ \cdot \end{array}$	null	$ s_2  > 1$ $d_2 = U$	null	$d$

Figure 6.23: Enumeration rules for *seed passing*.

requires the wavefront for  $s_1$  to be strict. A naive way to prevent generation of such illegal plans is to check whether a given wavefront for  $s_1$  is strict. If it is not, then make **KP** simply discard  $s_1$  without producing a plan. This would work if *all* plans for  $s_1$  were generated. However, WAVEGUIDE's enumerator keeps only a *single*<sup>19</sup> plan for each sub-expression and seed pair, the plan with a lowest estimated cost. Consequently, if this lowest cost plan for  $s_1$  is not strict, then the bottom-up enumeration will break without producing a legal plan in either of the Kleene rules. Hence, we need an approach which ensures that during the bottom-up enumeration single best plans are generated and stored for all subexpressions of  $s_1$  would eventually produce a strict waveplan for  $s_1$ . In WAVEGUIDE, we accomplish this by using a special mechanism which we call *seed passing*. The main idea behind the seed passing is that all wavefronts which have been seeded will be generated strict during the enumeration. Hence, if we can *pass* seeds during enumeration for Kleene subexpressions and subexpressions bound by constants, we can guarantee that the eventual plans generated for these subexpressions will be legal. Before we discuss how seed passing mechanism operates in WAVEGUIDE, we set up some helpful definitions.

**Definition 6.12** *We define a prefix of regular expression  $r$  to be a subexpression*

---

<sup>19</sup>Keeping a single plan per seed-expression pair is key for achieving low complexity of enumeration.

$s \subseteq_{pre} r$  such that  $s \subseteq r$  and  $s$  generates path strings which are prefixes of path strings generated by  $r$ .

**Definition 6.13** Similarly, we define a suffix of regular expression  $r$  to be a subexpression  $s \subseteq_{suf} r$  such that  $s \subseteq r$  and  $s$  generates path strings which are suffixes of path strings generated by  $r$ .

**Example 7** Consider regular expression  $r = abc$ . Then, for  $r$ , expressions  $a, ab, abc$  are prefixes, and  $c, bc, abc$  are suffixes.

**Definition 6.14** Given query  $Q = (x, r, y)$  and subexpression  $s$  of  $r$ , we define prepath seed  $d_{pre}$  of  $s$  in  $Q$  as an RPQ which encodes constraining prepaths of  $s$  in  $r$ .

**Example 8** Consider query  $Q = (k, abc(def)+, y)$  and subexpression  $s = (def) +$ . Here, variable  $x$  is bound to constant  $k$ . Then, prepath seeds of  $s$  in  $Q$  are the following RPQs:  $(x, \varepsilon, y) = U, (x, c, y), (x, bc, y), (k, abc, y)$ .

**Definition 6.15** Given query  $Q = (x, r, y)$  and subexpression  $s$  of  $r$ , we define postpath seed  $d_{post}$  of  $s$  in  $Q$  as an RPQ which encodes constraining postpaths of  $s$  in  $r$ .

**Example 9** Consider query  $Q = (x, (def)+abc, k)$  and subexpression  $s = (def) +$ . Here, variable  $y$  is bound to constant  $k$ . Then, postpath seeds of  $s$  in  $Q$  are the following RPQs:  $(x, \varepsilon, y) = U, (x, a, y), (x, ab, y), (x, abc, k)$ .

**Definition 6.16** A seed defined by an RPQ can be concatenated with any regular expression as follows. Given seed  $d = (x, r, y)$  and regular expression  $s$ , we define the result of concatenation  $d/s$  as seed  $(x, r/s, y)$ . Similarly, concatenation  $s/d$  yields to seed  $(x, s/r, y)$ .

Seed passing is handled by subroutines `seedConst()` and `seedKleene()`. The seed is passed as a parameter to plan generation routine `getPlans()`. Pseudocode for `seedConst()` is presented in Fig. 6.17. Given expression  $s$  and query  $Q = (x, r, y)$ , `seedConst()` passes the seed which is created by the binding of variables  $x$  and  $y$  to the constants in  $Q$ . The binding of  $x$  to a constant creates a seed which is passed to all prefixes of  $r$ . Similarly, a seed defined by the binding of  $y$  to a constant is passed to all suffixes of  $r$ .

**Example 10** Consider query  $Q_1 = (c, abc, y)$  in which variable  $x$  is bound to a constant  $c$ . Then, during enumeration, `seedConst(c, Q)` is called for subexpressions  $s$  of  $r$  with increasing sizes starting from  $|s| = 1$  and finishing with  $l = |s| = |r|$ . Hence, a constant seed is passed to plans for all prefixes  $a, ab, abc$  of  $abc$ . Indeed, only plans for prefixes of  $abc$  are affected by a binding of  $x$  to a constant. Plans for all other subexpressions of  $abc$  are not directly affected by this binding, and hence do not require seed passing. The same logic applies to seed passing to plans for suffixes of  $abc$  in the case when variable  $y$  is bound to a constant.

Pseudocode for `seedKleene()` is presented in Fig. 6.18. Given expression  $s$  and query  $Q = (x, r, y)$ , `seedKleene()` passes a specific seed into plans for subexpressions  $s$  of  $r$  which satisfy the following condition:  $s$  should be a prefix or a suffix of a Kleene subexpression  $s_1+$  in  $r$ . The seed which is passed is constructed as follows. *Prepath* seeds and *postpath* seeds of  $s_1+$  in  $r$  are identified by subroutines `preSeed()` and `postSeed()`, respectively. Here, we omit the description of these subroutines, but, in short, they work by analyzing the automata constructed from a given regular expression  $r$  and have the complexity linear in the size of  $r$ . Both prepath and postpath seeds are concatenated with the Kleene expression itself, and then the resulting seeds are passed to the plan generation routine for  $s$ .

Consider an example run of enumeration algorithm `WGEnum` for query  $Q = (x, (abc)+, y)$  shown in Fig. 6.24. During the execution of the algorithm, Fig. 6.24 demonstrates the generated plans, their seeds, and corresponding enumeration rules which generated the plans. The enumeration starts by considering subexpressions  $s$  of  $(abc)+'$  of size  $l = 1$ . There are no constants in  $Q$ , so `seedConst()` does not execute. There exists a Kleene subexpression in  $Q$ , hence `seedKleene()` executes. There are no prepaths or postpaths of  $(abc)+$  in  $Q$ , hence Kleene plans are seeded with the Kleene expression itself. A plan for prefix  $a$  ( $|a| = 1$ ) seeded with  $(abc)+$  is generated by rule **ASDP**. Note that this plan is strictly appending. Also, a strictly prepending plan for suffix  $c$  seeded with  $(abc)+$  is generated by rule **ASIP**. Then,

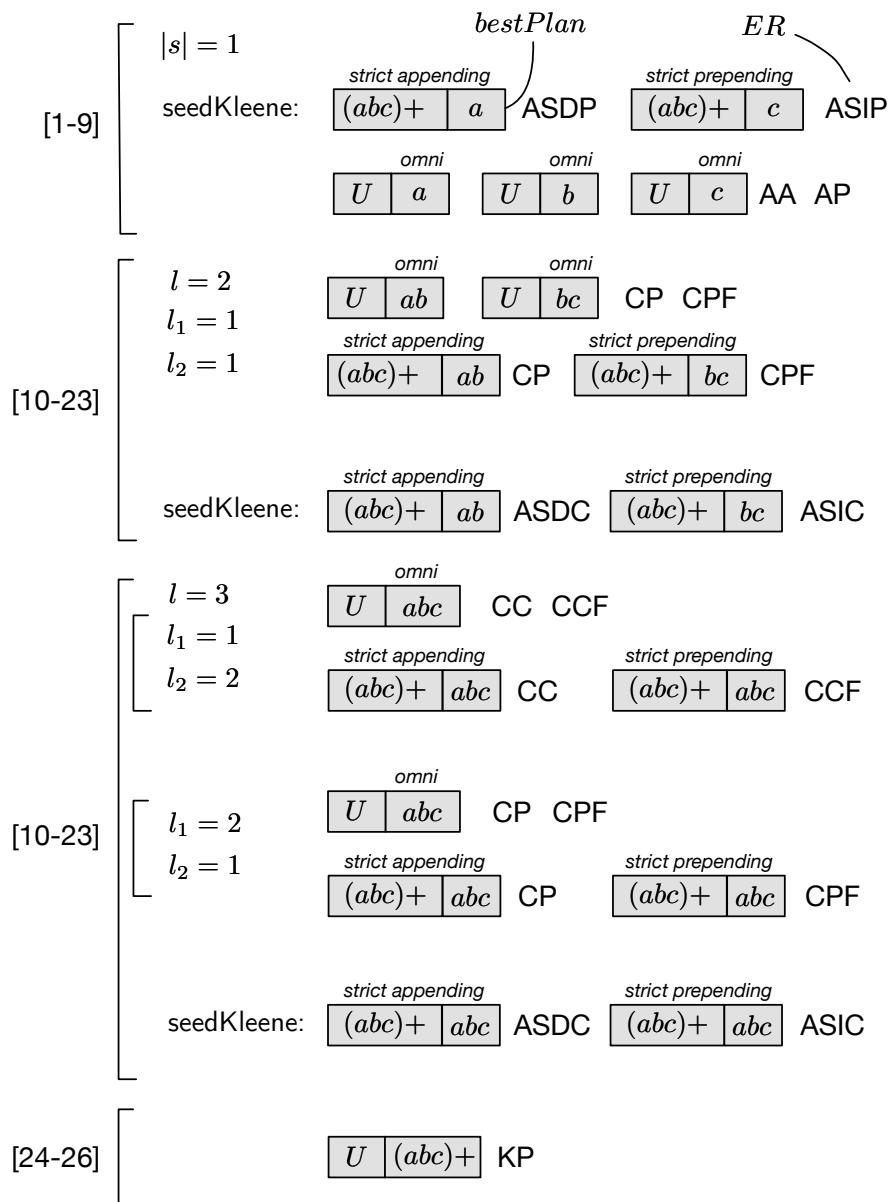


Figure 6.24: A run of the enumeration algorithm for  $Q = (x, (abc)^+, y)$ .

plans with universal seeds are generated for all subexpressions  $a, b, c$  of size one by rules **AA** and **AP**.

The enumeration continues by constructing plans for subexpressions of size  $l = 2$  by combining plans for subexpressions of size  $l = 1$ . First, omnidirectional wavefronts are generated for  $ab$  and  $bc$  by rules **CP** and **CPF**. Then, **CP** is used to generate a seeded wavefront for  $ab$ . Note, that this wavefront continues to be strictly appending. Similarly, **CPF** is used to generate a seeded strictly prepending wavefront for  $bc$ . Again, **seedKleene()** routine is used to generate strict wavefronts for both  $ab$  and  $bc$ . Here, rules **ASDC** and **ASIC** are using omnidirectional wavefronts computed earlier for  $ab$  and  $bc$  to produce strict wavefronts by transitioning over the corresponding views for  $ab$  and  $bc$ .

The generation of plans proceeds in the similar fashion for  $l = 3$ . Here, expressions of size  $l = 2$  are combined with expressions of size  $l = 1$  and vice versa. Now, we have a seeded plan for full expression  $abc$  used in the Kleene closure in  $Q$ . So, rules **KP** and **KS** are used to produce a plan for the closure. Note that all seeded plans were generated strict by the enumeration rules. Since both **KP** and **KS** use a seeded plan for a subexpression in Kleene, the generated plan for the Kleene closure is legal with respect to the given query.

**Theorem 6.17** *Given query  $Q = (x, r, y)$ , WAVEGUIDE’s bottom-up enumerator **WGEnum** generates a plan for  $Q$  in time polynomial in the size of  $r$ .*

**Proof:** By analysis of the running time of all subroutines used in `WGEnum`. Routine `memoizeMin()` updates *bestPlan* in  $O(|P|)$  time where  $|P|$  is the number of plans given to it. Set  $|P|$  is always generated by the `getPlans()` routine. Each rule in  $ER$  generates exactly one plan per seed-subexpression pair. There are  $O(|r|)$  seeds and  $|ER|$  rules, hence  $|P| = O(|ER| \cdot |r|^2)$ . Note that  $O(|ER|) = O(1)$  since constant  $|ER|$  is predefined at compile time. Hence, `memoizeMin()` runs in  $O(|r|^2)$  time.

Routine `getPlans()` generates plans for a given operator and subexpression pair. Again, there are  $O(|r|)$  seeds and  $|ER|$  rules, hence `getPlans()` runs in  $O(|ER| \cdot |r|^2) = O(|r|^2)$  time. Routine `seedConst()` passes a seed if a given subexpression is a suffix or a prefix of  $r$ . Hence, `seedConst()` runs in  $O(|r|^2)$  time. In `seedKleene()` routine seeds are generated based on prepaths and postpaths of a given expression-query pair. There are at most  $O(|r|)$  of (pre/post)paths, hence `seedKleene()` runs in  $O(|r|^3)$  time.

We then analyze how many times loops are executed in `WGEnum`. First loop  $L[1\text{-}9]$  is executed  $|r|$  times for each subexpression  $s \in r$  of size 1. Each operation inside  $L[1\text{-}9]$  takes at most  $O(|r|^3)$  time. Hence, loop  $L[1\text{-}9]$  runs in  $O(|r|^4)$  time. We then consider loop  $L[10\text{-}26]$ . It contains two inner loops  $L[11\text{-}20]$  and  $L[21\text{-}26]$ .  $L[11\text{-}20]$  loops over pairs of subexpressions  $s_1, s_2$  such that  $|s_1| + |s_2| = l$ . If [15] condition would hold only if concatenation  $s_1/s_2 \in r$ . Hence, for a subexpression

of size  $l$ , at most  $l - 1$  splits with concatenation (/) into  $s_1$  and  $s_2$  is possible.

Similarly, for `If[18]`, at most  $l - 1$  splits with union (|) into  $s_1$  and  $s_2$  is possible.

Each operation inside `If[15]` and `If[18]` takes at most  $O(|r|^2)$  time. Similarly, each operation in loop `L[23-29]` also takes at most  $O(|r|^3)$  time.

There are at most  $|r| - l + 1$  such subexpressions of length  $l$  in  $r$  for each of which both `L[11-20]` and `L[21-26]` execute. Hence, `L[10-26]` takes time:

$$\sum_{l=2}^{|r|} (|r| - l + 1)(l - 1) \times O(|r|^3) = \frac{|r|^3 - 3}{6} \times O(|r|^3) = O(|r|^6) \quad (6.54)$$

Since `L[1-9]` runs in  $O(|r|^4)$  time and `L[10-26]` runs in  $O(|r|^6)$  time, the whole `WGEnum` runs in  $O(|r|^6)$  time which is polynomial in the size of given regular expression  $r$ . □

**Theorem 6.18** *Given RPQ  $Q = (x, r, y)$  with regular expression  $r$ , WAVEGUIDE's bottom-up enumerator `WGEnum` generates a plan for  $Q$  which is legal with respect to  $r$ .*

**Proof:** By splitting into cases based on the type of the query  $Q = (x, r, y)$  given. First, suppose that both variables  $x$  and  $y$  are *free*; i.e., not bound to any constants. Further, suppose  $r$  does not have any Kleene closures; i.e., operators  $*$  and  $+$  are not used. In this case, all seeds used are universal and strictness of wavefronts is not required; hence, we only need to show that enumeration rules in  $ER$  produce legal plans when used with concatenation (/) and union (|) operators. This is shown

by using structural induction on the parse tree of  $r$ . Specifically, we assume that plans  $p_1$  and  $p_2$  generated for subexpressions  $s_1$  and  $s_2$  are legal. Given that, we show that the application of each enumeration rule would produce a legal plan for a combined expression.

Second, suppose that one of the variables  $x$  and  $y$  is bound to a constant and  $r$  is still free of Kleene expressions. Without loss of generality, assume that  $x$  is bound to a constant  $k$ . Since  $x$  is bound to  $k$ , the generated plan for expression  $r$  should be strictly appending to be legal. Hence, we need to show that all rules which led to the generation of a plan for  $r$  would produce a strictly appending plan.

We split into cases based on subexpressions  $s_1$  and  $s_2$  of  $r$  and the operator  $op$ . Suppose  $op$  is a concatenation operator ( $/$ ), such that  $r = s_1/s_2$ . In this case,  $s_1$  is a prefix of  $r$  by Def. 6.12; i.e.,  $s_1 \subseteq_{pre} r$ . During enumeration, `seedConst()` passes the constant seed for all such prefixes  $s_1$ . Hence, depending on the size  $|s_1|$ , only rules **ASDP** and **ASDC** are used for generating a plan for  $s_1$ . Both of these generate strictly appending plans  $p_1$  for prefixes  $s_1$ . Since, none of the plans for  $s_2$  are seeded, rules **CC** and **CP** are used to produce the plan for concatenation  $r = s_1/s_2$ . In both **CC** and **CP**, the plan for  $s_1$  is copied into the wavefront for  $r$ , and then the appending transition is used for subexpression  $s_2$ . Since we have shown that  $p_1$  is strictly appending, it follows that all plans generated for  $r$  are also strictly appending.

Next, suppose  $op$  between  $s_1$  and  $s_2$  is a union ( $\mid$ ) operator. Regardless of the size of subexpressions  $s_1$  and  $s_2$ , plans  $p_1$  and  $p_2$  are copied in the rule **U** to produce a plan for union  $s_1 \mid s_2$ . Both  $s_1$  and  $s_2$  are bound by the same constant as  $r$ , hence we can reduce recursively this problem into showing that both  $p_1$  and  $p_2$  are strict. We continue splitting recursively  $s_1$  and  $s_2$  into subexpressions until we either reach concatenation or when the expression is an atomic label. If  $op$  is a concatenation ( $/$ ), then we are done (as we have proven this case already). If the expression is an atomic label, then it is a prefix of itself. Hence, it would have been seeded by `seedConst()`. Then, rule **ASDP** is used for generating a plan for this subexpression. By construction, this rule produces a strictly appending plan.

Finally, we consider that one of the variables is bound to a constant and  $r$  includes Kleene subexpressions. Here, we are left to prove that plans for subexpressions of Kleene closures are generated strict. Indeed, `seedKleene()` seeds all prefixes and suffixes of its subexpressions similar to `seedConst()`. So, using the same proof logic we show that all subexpressions of Kleene closures the enumeration rules will generate strict plans. By Def. 6.10, this is sufficient to show that generated plan is legal with respect to the Kleene closure.  $\square$

**Theorem 6.19** *Given RPQ  $Q = (x, r, y)$  with regular expression  $r$ , WAVEGUIDE's bottom-up dynamic programming enumerator **WGEnum** generates a plan for  $Q$  which is optimal with respect to WAVEGUIDE's cost model and plan space.*

**Proof:** To show the optimality of the plan generated by WAVEGUIDE’s enumeration dynamic programming procedure, we need to show that the enumeration procedure exhibits the *optimal substructure* property. This property requires the ability to construct an optimal solution of a *problem* from optimal solutions of its *subproblems*.

In our enumeration, we define a solution of a *problem*  $P$  as the generation of an *optimal* plan (a plan with the lowest estimated cost) for a given regular expression  $r$ . Then, optimal solutions to *subproblems*  $S_i$  are the lowest cost plans for subexpressions  $s_i$  of  $r$ . Recall that there are many ways to split  $P$  into subproblems depending on how expression  $r$  can be constructed from its subexpressions  $s_i$ ; i.e., different parse trees of  $r$  need to be considered. Hence, to prove optimality of our enumeration, we need to show the following:

1. a construction of a solution for  $P$  from optimal solutions for  $S_i$  is optimal;  
and
2. during the enumeration, that we explore all possible splits of problem  $P$  into subproblems  $S_i$ .

To show (1), we perform a structural induction on a fixed parse tree of  $r$  and split into cases based on the operator in the parse tree. First, we consider the concatenation operator (/). Suppose, according to the parse tree, expression  $r$  is a concatenation  $r = s_1/s_2$ . Let  $p_1$  and  $p_2$  denote optimal plans for expressions  $s_1$  and

$s_2$ , respectively. Then, having  $p_1$  and  $p_2$ , we need to show that we can construct an optimal plan for  $r$  with respect to this fixed parse tree, and, hence, with respect to this particular split of  $r$  into  $s_1$  and  $s_2$ . The optimality of a solution for  $r$  is dictated by its total cost which is a sum of costs of  $p_1$ ,  $p_2$ , and concatenation  $s_1/s_2$ . Both  $p_1$  and  $p_2$  produce an answer set (a set of variable bindings in the graph) for their respective subexpressions. The cost of a concatenation (a join) of these answer sets would depend only on the sets themselves and not on *how* these sets were obtained.<sup>20</sup> Hence, an optimal solution for the concatenation  $r = s_1/s_2$  can be constructed by considering only optimal solutions for  $s_1$  and  $s_2$ .

Second, we consider the union operator ( $\mid$ ). Suppose, according to the parse tree, expression  $r$  is a union  $r = s_1 \mid s_2$ . Here, the evaluations of  $s_1$  and  $s_2$  are *independent* of each other. Hence, by having optimal solutions for  $s_1$  and  $s_2$ , we can construct an optimal solution for their union  $s_1 \mid s_2$ .

Third, we consider the Kleene closure operators (+ and \*). Suppose that, without loss of generality, according to the parse tree, expression  $r$  is a Kleene plus closure of its subexpression  $s_1$ . Then, by having an optimal plan for  $s_1$ , can we

---

<sup>20</sup>A nuance of this problem is when we consider the *order* of the answer sets. For example, it might be more expensive to produce ordered answer sets for  $s_1$  and  $s_2$ , but then the cost of concatenation  $s_1/s_2$  might be lower due to use of a merge join. In this case, *suboptimal* (from an estimated cost perspective) solutions of subproblems might produce an optimal solution to the problem itself. This nuance can be handled by additionally keeping interesting orders for sub-problems, without asymptotic increase in the overall complexity. In WAVEGUIDE, we are not using merge joins in our implementation, and, hence, we are not keeping additional orders in the memoization table.

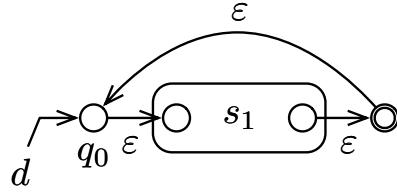


Figure 6.25: A plan template for Kleene plus  $Q = (x, s_1+, y)$ .

construct an optimal plan for the whole closure? In general, the answer is no — having an optimal plan for the subexpression does not guarantee an optimal plan for the whole closure. This is caused by the *repeated evaluation* nature of the Kleene closure. The optimal plan for  $s_1$  guarantees us the lowest cost of evaluation of only the first loop in the Kleene closure which finds paths  $s_1$ . However, the plan for  $s_1$  might not be optimal for the second loop in the closure which finds paths  $s_1/s_1$ , and so forth. In general, different plans can be used for each subsequent loops in the Kleene closure (see *unrolling* in §8.3). However, by Def. 6.1, we restrict the same plan to be used for the entire closure (we defer *unrolling* to future work). Hence, we need to ensure that the plan we choose for  $s_1$  is optimal for the whole closure.

The reason the same plan for  $s_1$  might be optimal for one loop in the Kleene closure and suboptimal for another is the fact that its *seed* changes from one loop to another. For example, assume that  $d$  represents the seed for the whole closure  $s_1+$  (see Fig. 6.25). Then,  $d_0 = d$  is seeding the first iteration when  $d/s_1$  paths are computed.  $d_1 = d/s_1$  is seeding the second iteration when  $d/s_1/s_1$  paths are

computed, and so forth. So, in order to pick the single plan for  $s_1$ , which is best for the whole closure, we need to take into account seeds  $d_0, d_1, \dots, d_n$  during all  $n$  iterations of the closure. Observe that a union of all seeds is the closure itself:  $\bigcup_{i=0}^n d_i = d/s_1+$ . Hence, to guarantee optimality of the whole closure, we just need to seed the subplan for  $s_1$  with  $d_+ = d/s_1+$ , which is what is being done in enumeration rule **KP** (and **KS** for Kleene \*). This shows (1) for all different operators in the parse tree.

Now, to show (2), we analyze if all possible splits into subproblems are considered by our enumeration presented in Fig. 6.14. Again, split into cases based on the operator  $op$ . He we only need to consider *binary* operators:<sup>21</sup> the concatenation (/) and union (|). Observe that, indeed, all possible splits of  $r$  of size  $|r| = l$  into  $s_1$  and  $s_2$  are considered in **WGEnum** by iterating over all sizes  $|s_1|$  and  $|s_2|$  such that  $|s_1| + |s_2| = l$ . This means that by considering all *local* optimal solutions for splits of  $r$  into  $s_1$  and  $s_2$ , we find the *global* optimal solution for  $r$ , in general.

□

In §6.3.3, we discussed the optimization of *threading*. We accommodate this in-part in the enumeration above. For Kleene closures in the query, these can be potentially seeded by other parts of the plan. Constants define seeds for subplans. Both of these are covered by our enumerator **WGEnum**. Another threading opti-

---

<sup>21</sup>Kleene closures are *unary* operators, hence no splits are possible.

mization is for subplans for evaluating views which can be restricted to a smaller graph walk by the results of other subplans. If such restrictions were applied at runtime, the evaluation may be more efficient. To choose the best plan, with respect to this view threading optimization, we would need to accommodate this consideration in the cost estimation and caching of subplans for views during the enumeration. **WGEnum** presently does not do this. This is a point of future work.

## 7 Implementation & Benchmarking

### 7.1 Implementation

#### 7.1.1 Software: The System

We have prototyped a system that implements the methodology from §3 in order to benchmark waveguide plans to study their performance. We illustrate how WAVEGUIDE can be implemented effectively on a modern relational database system. We use POSTGRESQL due to that it is open-source and has a high-performance procedural SQL implementation. However, any RDBMS with good procedural SQL support could be used. Further, many native graph databases with support for procedural languages equivalent to procedural SQL can also be used. In this WAVEGUIDE system, resource-intensive tasks are delegated to POSTGRESQL via SQL and *procedural* SQL routines.

Fig. 7.1 shows the architecture. It consists of two layers: *application* and *RDBMS*. The application layer provides a user front-end, preprocessing the graph

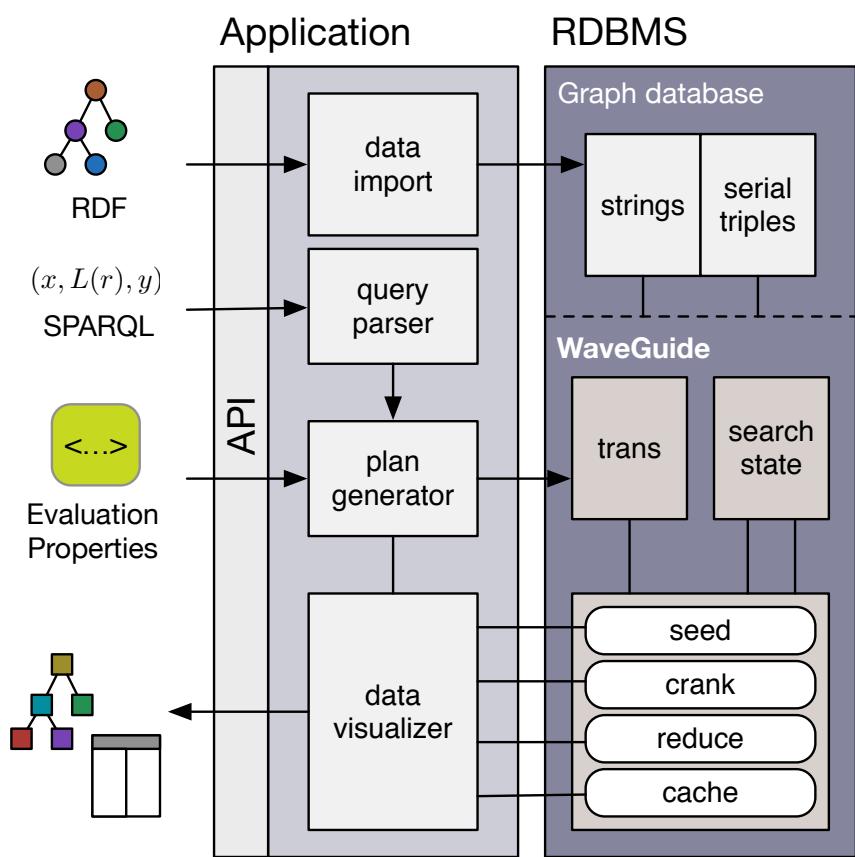


Figure 7.1: Overview of the prototype system.

data, parsing user queries, generating WPs, and visualizing key steps during the search. The RDBMS layer provides postprocessing of the graph data and performing the iterative WAVEGUIDE graph search for the given WP. The RDBMS acts both as the graph store and as the execution platform for WAVEGUIDE’s iterative algorithms.

**Graph database.** We represent a graph database in a single logical triples table, which is decomposed into two physical tables—`strings` and a surrogate `serial-Triples`—to reduce storage space and improve performance due to IRI compression as discussed in §2.2.1.1. The surrogate table is indexed in all six ways—`spo`, `sop`, `pos`, `pso`, `osp`, and `ops`—to accommodate the guided search. In is a subject of future work for us if other relational graph storage techniques such as *property tables* (§2.2.1.2) and *vertical partitioning* (§2.2.1.3) would be a benefit for WAVEGUIDE approach.

**Guided search.** The *graph walk* framework discussed in §3.2 is implemented completely in procedural SQL routine in POSTGRESQL. Through query hints, we force POSTGRESQL to use WAVEGUIDE’s primitive graph search operations (`crank`, `reduce`, and `cache`) such that they match the example implementations ( $f_1$ - $f_4$ ) given in §6.1. The WP that guides the search process is encoded in the `trans` table. To improve the performance, the cache is stored in an unlogged, ephemeral `searchCache` table. This is indexed to cover the access paths used by the iterative

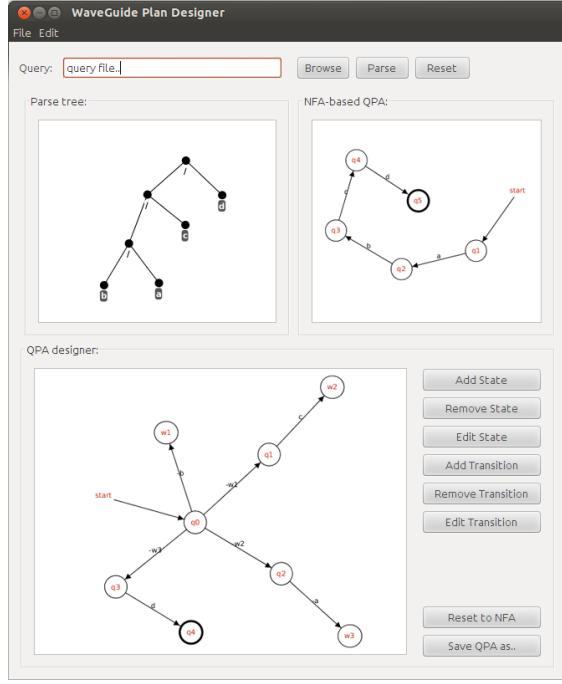


Figure 7.2: Query plan designer.

search. We implement profiling functions here to feed evaluation statistics to the data visualizer.

**Data importer.** This validates RDF data encoded in common formats (e.g., N-triples and Turtle.). It converts these to a tab-separated value format for bulk loading in the RDBMS.

**Query parser.** We use the Apache ANTLR open-source framework to parse SPARQL 1.1 property path query strings into an internal tree representation.

**Plan generator.** Given the query parse tree, we produce a base WP from an NFA that recognizes the regular expression of the query. We then employ our cost-based

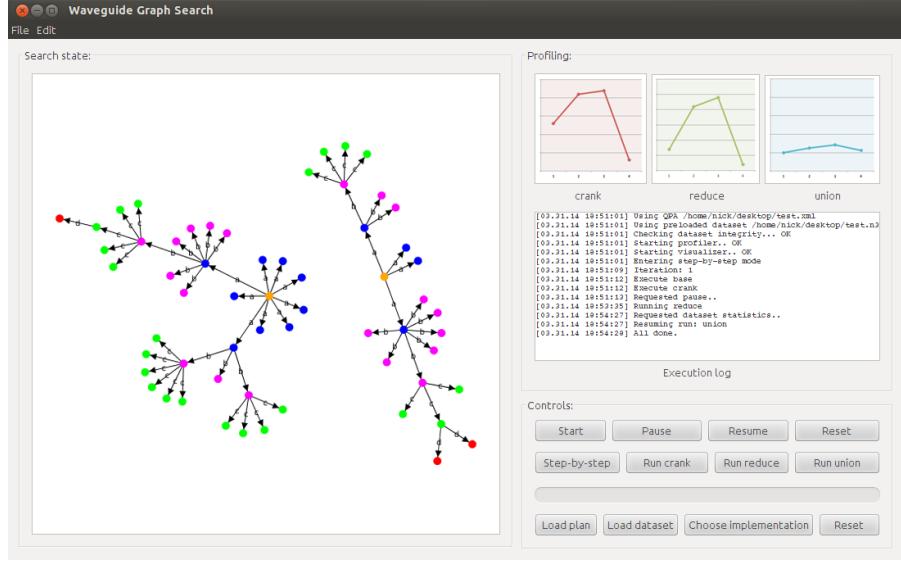


Figure 7.3: Runtime visualizer and profiler.

enumeration algorithm to obtain a waveplan with the lowest estimated cost. The end user can manually tune the produced WP via *graphical evaluation plan designer* (shown in Fig. 7.2).

**Data visualizer.** We employ GRAPHSTREAM open-source library [20] to perform the graph visualization in our system. This allows us to visualize dynamically the key steps involved in the WAVEGUIDE search process. We interface with the RDBMS to visualize the search cache at each iteration of the `crank`, `reduce`, and `cache` steps. To provide technical insight to the WAVEGUIDE process, we display a number of relevant evaluation parameters and statistics (shown in Fig. 7.3).

### **7.1.2 Hardware: Runtime**

Our benchmark was executed on a host 2xXeon E5-2640v2 CPU server with 10 7200RPM HDDs running CentOS 6. PostgreSQL, OpenLink Virtuoso, and Jena TDB were executed in their own virtual machines which were given 4 CPU cores, 4GB of RAM and 1TB of HDD space.

### **7.1.3 Software: Runtime**

WAVEGUIDE prototype planner was implemented in Java 1.8. WAVEGUIDE database execution engine was based on PostgreSQL 9.3. VIRTUOSO was built against open source distribution of version 7.2.1 (latest at the time of submission of this dissertation) which is maintained by OpenLink on GitHub. We have modified the source code to increase the transitive memory limit of VIRTUOSO’s underlying RDBMS. We obtained the latest version (2.13.0) of JENA TDB for our tests. Both DBpedia and YAGO2s were fully indexed on all systems to ensure indexed access paths for all queries used in benchmarks.

## **7.2 Methodology**

We test our implementation of WAVEGUIDE by running a collection of realistic path queries over real-world datasets YAGO2s [64] and DBpedia [18]. The datasets

were preprocessed by removing invalid and duplicate triples. After preprocessing, YAGO2s had 242M triples and DBpedia had 463M triples, with 104 and 65K distinct predicates, respectively. A large number of predicates makes these datasets well suited for benchmarking of path queries.

At the time of this dissertation, we could not find any available benchmarks for SPARQL property path queries.<sup>22</sup> We, therefore, generate path queries based on data patterns we identified in real-world graphs. The goal of these experiments is to verify the gains offered by WAVEGUIDE optimizations, and show that they correspond to the cost framework (§6.1) and analysis (§6.4).

## 7.3 The Optimizations

### 7.3.1 Threading

We benchmark the threading optimization by executing a query of the following template:

$$\text{?x } p/p_1+/p_2+ \text{ ?y} \quad (\mathcal{Q}_{7.1})$$

over the YAGO2s dataset, with “ $p$ ”, “ $p_1$ ” and “ $p_2$ ” as variable predicates. We chose this template because it contains the concatenation of two transitive closures, which makes it difficult to predict the average length of the paths in the answer.

---

<sup>22</sup>The first benchmark to include RPQs, *gMark* [9], was in development at that time.

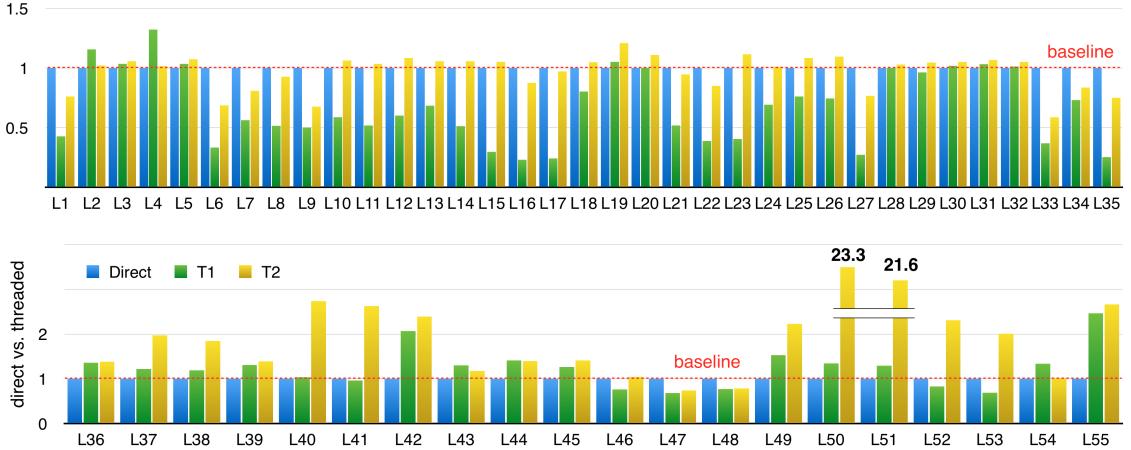


Figure 7.4: Benchmarking Threading.

We group  $p$  candidates in two sets: the first (queries **L1–L35**) having an  $\mathcal{M}$  value greater than 10; and the second (queries **L36–L55**) having an  $\mathcal{M}$  value less than 1. Each of the queries is executed with three different plans:  $D$ , a direct evaluation with a single waveform with no threading;  $T_1$  threads on shared path ( $p_1+/p_2+$ ); and  $T_2$  threads on ( $p_2+$ ) to illustrate the effect of potentially lower  $\mathcal{L}$  than in  $T_1$ . Given  $p$ , predicates  $p_1$  and  $p_2$  are chosen such that the running time of  $\mathcal{Q}_{7.1}$  is between 0.5 and 5 minutes.

The relative running times for queries **L1–L55** executed with plans  $D$  (the baseline),  $T_1$ , and  $T_2$  are presented in Fig. 7.4. As anticipated, the evaluation of queries in the first group is significantly (up to 75%) faster threaded than direct, and with  $T_1$  being faster than  $T_2$ . This can be attributed to the fact that the length of the shared path  $\mathcal{L}$  is shorter in  $T_2$  due to a “later” threading split in the query

expression. Also as anticipated, queries in the second group show bad results for  $T_1$ . Indeed, picking a predicate with  $\mathcal{M} < 1$  for a threading split will generally be bad due to few shared paths. Further, some queries (**L50-L51**) are evaluated significantly slower (up to 23 times!) when threaded in  $T_2$ . This illustrates the importance of choosing the correct threading split.

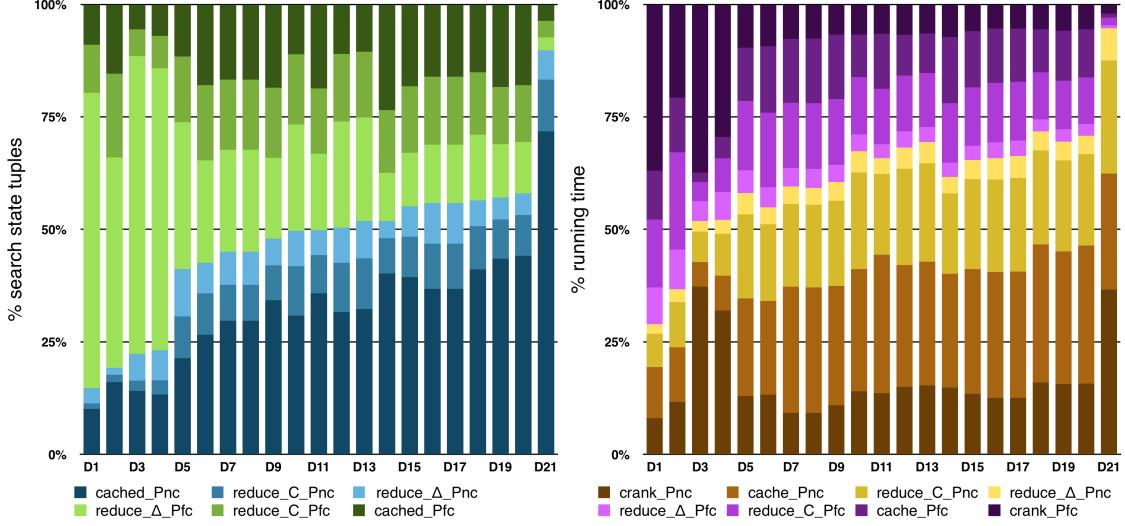
### 7.3.2 Loop Caching

We benchmark the loop-caching optimization by executing a collection of queries of the simple template  $Q_{(ab)+} = (x, (ab)+, y)$  (**Q<sub>3.1</sub>**) with two WPs  $P_{nc}$  and  $P_{fc}$ , which specify executions of  $Q$  with no loop caching and with full loop caching, respectively.

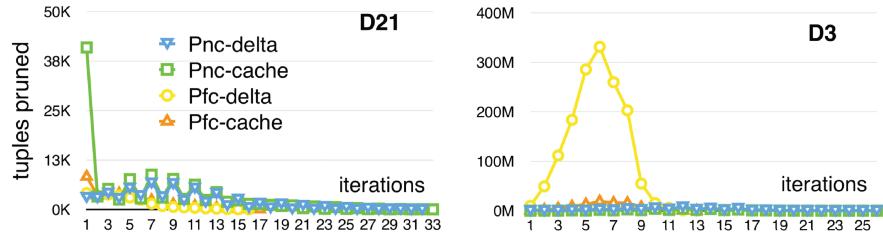
Values for  $a$  and  $b$  were chosen by iterative pruning of predicates appearing in the DBpedia dataset. First, we excluded predicates with very high (more than 25M) and low (less than 75K) cardinalities, to keep query running times reasonable. Then, we ran query  $Q_{abab} = (x, (abab), y)$  and recorded those  $(a, b)$  predicate pairs for which the result of  $Q_{abab}$  was not empty. DBpedia had 1171 such pairs, which indicates a high number of  $(ab)+$  paths in this dataset. For each of these pairs, we ran the full closure query  $Q_{(ab)+}$  to obtain its *expansion ratio*,

$$r_{\text{exp}} = \frac{|Q_{(ab)+}|}{|Q_{ab}|} \quad (\mathcal{E}_{7.1})$$

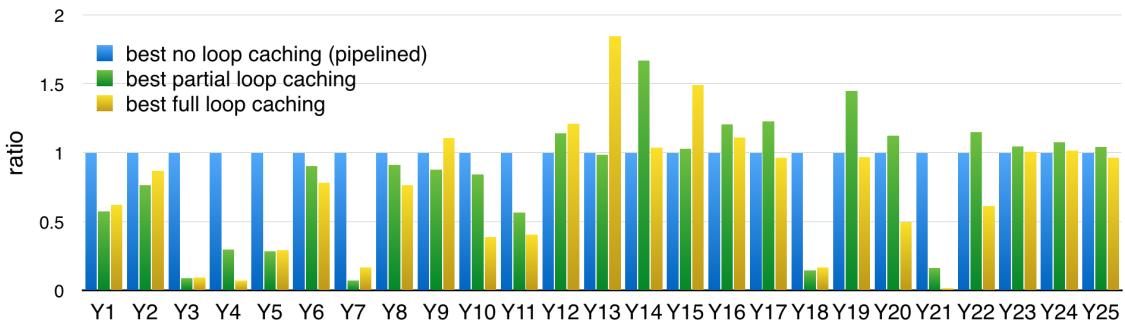
where  $|Q|$  denotes the cardinality of a query result.



(a) Edge walks vs. runtime in plans w/ & w/o loop caching in DBpedia.



(b) Tuples pruned in  $P_{fc}$  vs.  $P_{nc}$ .



(c) Best pipelined, partially and fully cached plans for queries over YAGO2s.

Figure 7.5: Benchmarking Loop Caching.

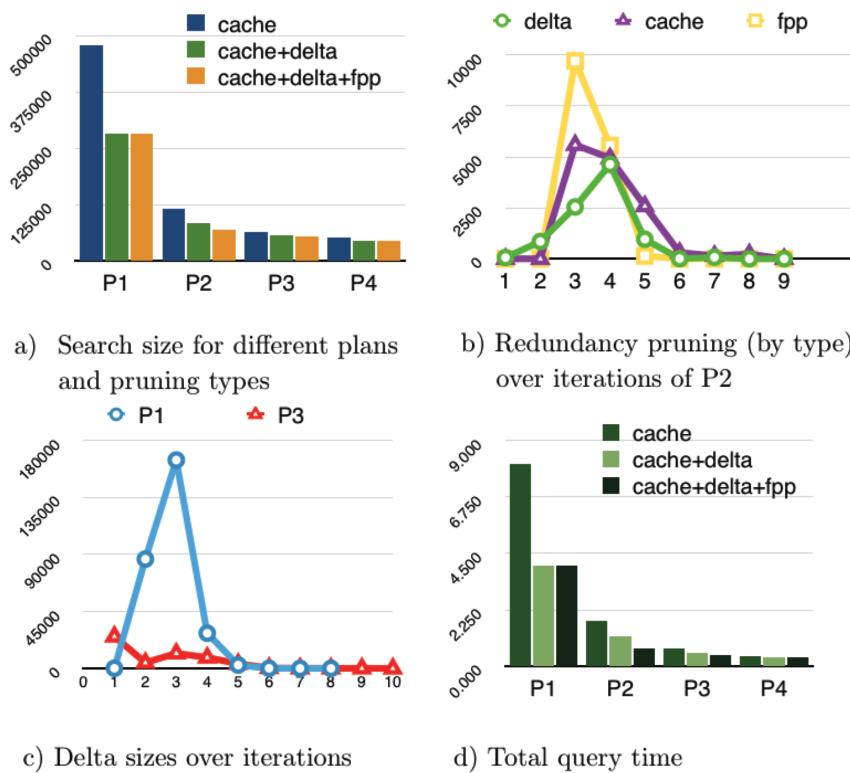


Figure 7.6: Effect of plans on query evaluation.

Recall that both  $P_{nc}$  and  $P_{fc}$  initially evaluate  $(ab)$  paths in the same way, while the rest of the closure  $(ab)^+$  is computed differently. Hence, in order to show the differences between these plans, we chose predicate pairs with  $r_{\text{exp}} > 2$ , so that the computation of the rest of the closure constitutes at least 50% of the plan execution time. We identified 21 such queries by analyzing graph patterns in DBpedia.

We evaluated each one of these queries with  $P_{nc}$  and  $P_{fc}$  plans and recorded the *running time*, *edge walks* and *pruning statistics*. Due to widely varying absolute

values for these statistics across queries, we present their relative percentage breakdowns in Fig. 7.5a, as follows. Each query is represented by a multi-colored bar, which shows the percentage breakdown of statistics values between  $P_{nc}$  and  $P_{fc}$  executions. In this way, we present edge walks (in the left chart) and running-time execution (in the right chart). We enumerate the queries from **D1** to **D21** according the ascending sorting of the percentage of edge walks performed in the  $P_{nc}$  execution relative to the  $P_{fc}$  execution. Hence, in query **D1**,  $P_{nc}$  execution resulted in significantly fewer edge walks relative to  $P_{fc}$  execution, with the opposite true for query **D21**. For edge walks, we perform a further breakdown, for each query, of the total number of edge walks into the number of tuples which were cached, were reduced against the cache, or were reduced against the delta. This breakdown is represented by different shades of the color associated with  $P_{nc}$  or  $P_{fc}$  executions, respectively. Similarly, for each query, we break down its running time into total times of `crank`, `cache`, `reduceC` and `reduceΔ` steps to illustrate the costs of each in relation to the framework presented in §6.1.

Our first observation is that, in general, the loop caching optimization can both significantly increase *or* decrease the total number of edge walks performed by the search. In our benchmark, loop caching resulted in fewer edge walks in 10 queries, with almost an order of magnitude reduction, in the best case. On the other hand, in 11 queries, loop caching resulted in more edge walks, with a more than 5X

increase, in the worst case.

Our second observation is that the real cost of WAVEGUIDE search corresponds to the approximations of costs  $f_1-f_4$  given in §6.1. Consider queries **D1-D11** in which loop caching resulted in a higher total number of edge walks. However, the majority of these queries still ran faster than their pipelined counterparts due to the fact that, the bulk of edge walks produced duplicate tuples, which were removed by `reduce $^\Delta$` . Indeed, such in-memory removals are inexpensive according to  $f_2$ . Further, we observe that expensive ( $f_1, f_3$  and  $f_4$ ) disk-based operations `crank`, `reduce $^C$`  and `cache` which require probing or maintaining a tree index contribute significantly to the overall cost.

Finally, we study the effect of lensing by analyzing the degree of `delta` and `cache` pruning. Fig. 7.5b plots pruning over iterations for the queries which exhibit lensing: **D3** and **D21**. Query **D3** has  $\mathcal{M}(G, a) = 10.58$  and  $\mathcal{M}(G, b) = 0.33$ , which suggests lensing with focal point on the concatenation  $a/b$ . As discussed in §6.4, this can significantly increase the amount of pruning for loop caching, which is indeed what we observe. On the other hand, **D21** has  $\mathcal{M}(G, a) = 0.07$  and  $\mathcal{M}(G, b) = 5.34$ , which suggests lensing with a focal point on the concatenation  $b/a$ . This lensing benefits loop caching by decreasing the amount of pruning over iterations, which is what we observe.

### 7.3.3 Partial Loop Caching

In a previous section, we have shown experimentally that plans from both FA and  $\alpha$ -RA spaces need to be considered in query evaluation. In this section, we aim to show that plans which are not in FA or  $\alpha$ -RA often also require consideration.

We design a collection of queries over YAGO2s dataset which are based on the template below:

$$?x (a/b/c)+ ?y \quad (\mathcal{Q}_{7.2})$$

We pick values for  $a$ ,  $b$  and  $c$  by iterative pruning of predicates appearing in the YAGO2s dataset similar to the process discussed in §7.3.2. As a result, we came up with 25 queries **Y1–Y25** which follow template  $\mathcal{Q}_{7.2}$ .

The results of the benchmark are presented in Fig.7.5c. We group plans into three categories: no loop caching (pipelined), partial loop caching and full loop caching. For each category, we pick the best plan as judged by a total number of edge walks performed. Due to wide differences in absolute values, we present the best pipelined plan as a baseline and present the other two groups in relation to it.

As expected, pipelined plans are usually not the best choice in the evaluation of open-ended queries like  $\mathcal{Q}_{7.2}$ . Here, pipelined plans win in 5 out 25 queries. This is explained by the strict evaluation order imposed by such plans. For a query like  $\mathcal{Q}_{7.2}$  there are only two legal pipelined plans, direct and inverse pipelines. Hence,

often these plans lose to other plans which allow more flexibility in evaluation order. Fully loop cached plans which allow the most flexibility in evaluation order (we generate 20 such plans for  $\mathcal{Q}_{7.2}$ ), are fastest in 11 queries out of 25, sometimes by an order of magnitude. Finally, in 9 queries out of 25, partially loop cached plans win, sometimes by a large margin. Hence, these plans, which are exclusive to WAVEGUIDE, need to be considered as well.

#### 7.3.4 Combined Optimizations

We illustrate the impact of combining WAVEGUIDE optimizations over the example query

$$\text{?p :marriedTo/:diedIn/:locatedIn+/:dealsWith+ USA} \quad (\mathcal{Q}_{7.3})$$

over the YAGO2s dataset. We instantiate  $p$  as follows.

$P_1$ : single wavefront  $\text{USA} \rightarrow \text{?p}$ .

$P_2$ : single wavefront  $\text{?p} \rightarrow \text{USA}$ .

$P_3$ : two wavefronts

$$\text{?p} \rightarrow \text{:locatedIn+/:dealsWith} \leftarrow \text{USA}.$$

$P_4$ :  $P_2$  but with a threaded sub-path

$$\text{:locatedIn+/:dealsWith+ USA}.$$

Fig. 7.6a shows the effect of wavefront choice on search cardinality. Note the order of magnitude difference between the best,  $P_4$ , versus the worst,  $P_1$ . The three

types of redundancy pruning—`cache`, `delta`, and `fpp`—are illustrated for each plan. Fig. 7.6b plots search size across iterations for  $P_2$  with pruning; over 40% of tuples are pruned! Fig. 7.6c plots delta sizes over iterations for  $P_1$  and  $P_3$ . Note how the selective search of  $P_3$  is better behaved than the rapid expansion of  $P_1$ . In Fig. 7.6d, the total execution time for each plan is presented. This demonstrates the significant improvement in performance achievable by careful design of the WP.

## 7.4 Comparison with Other Systems

We analyze the performance of the WAVEGUIDE prototype in comparison with two RDF stores: VIRTUOSO and JENA TDB. Both VIRTUOSO and JENA TDB are well known and widely used today to store and query RDF data. VIRTUOSO is an example of an RDF store which is backed by a relational database, which makes it similar to WAVEGUIDE prototype. On the other hand, JENA TDB is a native solution developed specifically to handle graph data. We should point out that a fair and comprehensive comparison with VIRTUOSO is impossible because it does not handle full open-ended property path queries which are a part of SPARQL 1.1 standard specification. WAVEGUIDE does not have this limitation.

We design three experiments which aim to benchmark two main aspects of property path evaluation: 1) transitive closure computation and 2) query planning.

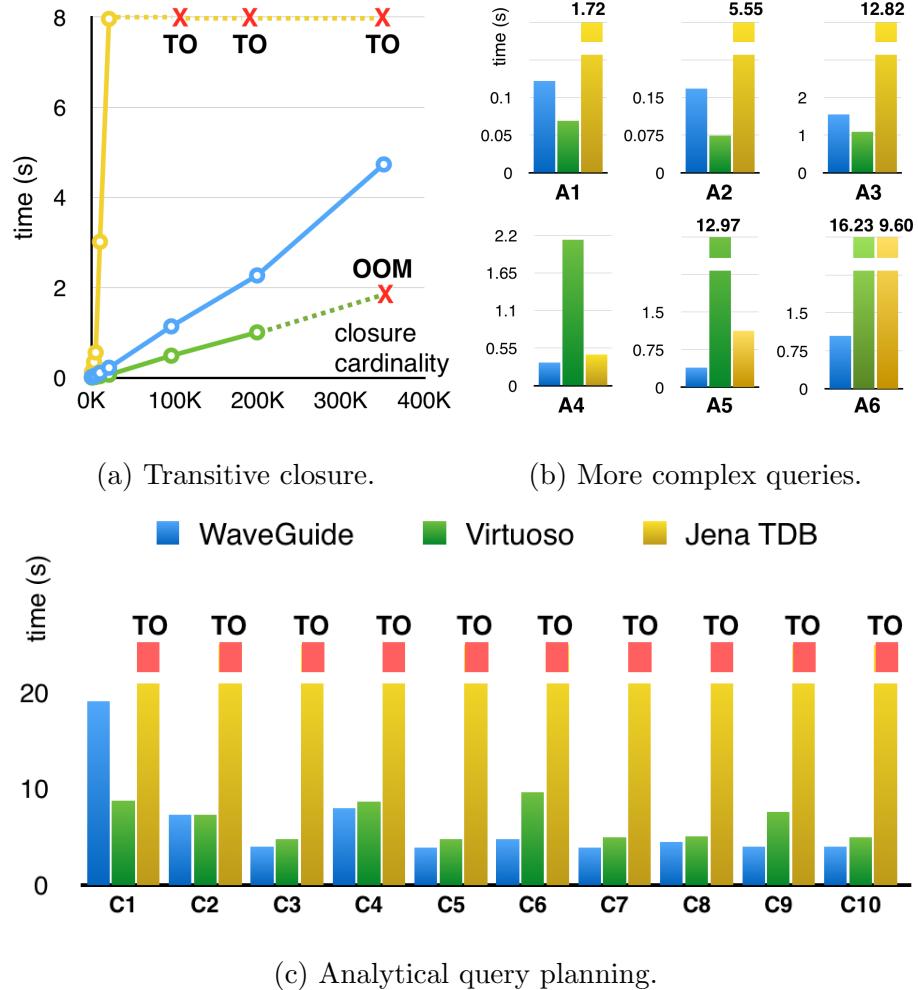


Figure 7.7: Benchmarking vs. state of the art.

#### 7.4.1 Transitive Closure

Computation of transitive closure is one of the main operations which are encountered during evaluation of property paths. Hence, it is important to handle this operation efficiently in order to be able to answer property path queries quickly.

Our benchmark consists of a number of realistic, simple transitive closure queries over YAGO2s dataset which follow the following template:

$$\text{?entity :locatedIn+ } place \quad (\mathcal{Q}_{7.4})$$

$\mathcal{Q}_{7.4}$  finds all entities transitively located in a certain place, which is given as a query constant. By varying this constant, we design transitive closure queries with increasing answer cardinalities, which allow us to benchmark the efficiency of a transitive closure computation in a given system.

The benchmark results are presented in Fig.7.7a. VIRTUOSO has the fastest transitive closure computation engine out of all systems. This is expected as VIRTUOSO is based on high-performance relational database written in C. Moreover, all its transitive closure computations are performed completely in main memory without spilling to disk. (Incidentally, this can be serious performance bottleneck for large cardinality property path queries.) WAVEGUIDE prototype is less than two times slower than VIRTUOSO. While our prototype is also based on high-performance relational database such as POSTGRESQL, we utilize stored procedures for our computation which are far less efficient than inline SQL or in-memory computations used by VIRTUOSO. Surprisingly, JENA TDB did not scale at all for transitive closures. While JENA TDB is competitive for general SPARQL queries, we conclude that the algorithms it uses for transitive closure computation need more work.

### 7.4.2 Query Planning

This experiment aims to highlight the importance of planning in the evaluation of property path queries. We design a number of queries over YAGO2s dataset which are based on the following template:

```
select ?x ?y {  
    ?x :locatedIn+ place .      [ $\mathcal{Q}_{7.4}$ ]  
    ?x (a/b/c)+ ?y . }        [ $\mathcal{Q}_{7.2}$ ]
```

Observe that  $\mathcal{Q}_{7.5}$  is a conjunction of templates  $\mathcal{Q}_{7.4}$  and  $\mathcal{Q}_{7.2}$ . We could not use just  $\mathcal{Q}_{7.2}$  because of VIRTUOSO limitation which requires property path queries to be bounded on at least one variable. In  $\mathcal{Q}_{7.5}$ , that binding is done by  $\mathcal{Q}_{7.4}$ .

First, we set *place* in  $\mathcal{Q}_{7.4}$  to :Earth, to simulate an *analytical* workload by placing a *loose* binding on  $?x$  in property path  $\mathcal{Q}_{7.2}$ . Second, we set values of *a*, *b* and *c* to those used in queries **Y1–Y10** in §7.3.3 to produce queries **C1–C10**. The benchmark results are presented in Fig.7.7c. We observe that JENA TDB did not finish in allotted time frame for any of the queries. We attribute this to the poor implementation of transitive closure computation as discussed in §7.4.2. On the other hand, despite having faster transitive closure computation engine, VIRTUOSO is slower than WAVEGUIDE in most of the queries, sometimes significantly. We attribute this to the limited plan space which is considered by VIRTUOSO. Due to

the limitation which requires transitive queries to be bound by one of the variables, VIRTUOSO was always choosing to evaluate  $\mathcal{Q}_{7.4}$  first, and then *pipeline* the results to  $\mathcal{Q}_{7.2}$ . As discussed in §7.3.3, such pipelined plans often perform badly for loosely bound queries. Hence, this limitation led to poor VIRTUOSO performance for **C2–C10**.

#### 7.4.3 Query Planning vs. Transitive Closure

The goal of our third experiment is to understand the capacity in which efficiency of transitive closure and query planning affect query evaluation. We design a set of six real-world queries over YAGO2s dataset which we split into two groups. Queries in the first group (**A1–A3**) are simple as they contain one or two non-transitive and one transitive predicate. Therefore, efficient transitive closure computation plays a bigger role than query planning in this group. Queries in the second group are slightly more complex as they either contain one transitive (**A5–6**) or a single transitive and three non-transitive predicates (**A4**). Hence, a good evaluation plan in this group is more important than efficient transitive closure.

The benchmark results are presented in Fig. 7.7b. Due to simplicity of the queries in the first group, all three engines used identical plans to evaluate them. Hence, the engine with the fastest transitive closure (in this case, VIRTUOSO) had the lowest evaluation time. On the other hand, for queries in the second group,

WAVEGUIDE was the fastest engine in all of the queries. Further, a slight increase in query complexity from one (**A4**) to two transitive predicates (**A5-6**) caused JENA TDB to fail to evaluate both **A5-6** and VIRTUOSO’s fail on **A6**. This highlights the importance of a large plan space considered by WAVEGUIDE over fast transitive closure computation in VIRTUOSO in all RPQs besides the simplest ones.

## 8 Conclusions

Regular path queries (RPQs) offer a succinct, declarative mechanism to query graphs in which the exact paths between nodes are unknown. RPQs are useful in many application domains such as social networks, life sciences, transportation and others. RPQs as a concept have been known and studied in the database community for quite some time now, first appearing during the initial spike of interest in graph databases in the 1980s. The first methods of evaluation of RPQs appeared during this time. These methods were based on finite state automata (FA). FA methods provided a convenient foundation for proving the complexity of evaluation of RPQs under various semantical configurations. Yet, FA methods were not practical in a real-world setting as query evaluation; FA methods are too rigid from the query planning perspective, and, hence, would often fail under many query workloads. The problem of RPQ evaluation has been revisited by the database community recently during the resurgence of interest in graph databases over the last couple of years due to the rise of the Semantic Web. The standards such as SPARQL 1.1

which define RPQs as a part of the query language have started coming into place. In this iteration, researchers proposed to deal with the evaluation of RPQs by using methods borrowed from the relational databases. Once the relational algebra is extended with transitive computations ( $\alpha$ -RA), cost-based query planning becomes available to make query evaluation more practical.

In this dissertation, we have made the observation that the effective plan spaces of FA and  $\alpha$ -RA approaches are, in fact, incomparable. This means that query evaluation techniques which utilize either one of FA and  $\alpha$ -RA plan spaces *will* be missing plans which might be the best (have the lowest cost) for a given query. This motivated us to come up with WAVEGUIDE, a *hybrid* RPQ evaluation approach, which combines FA and  $\alpha$ -RA plans spaces and, in fact, extends well beyond them.

## 8.1 Contributions

WAVEGUIDE models a rich space of plans for path queries which encompass powerful optimization techniques. In this dissertation, we have made the following contributions. First, we have *summarized* the state of the art for evaluation of RPQs and SPARQL property paths. We *established* why none suffices. We *designed* a novel hybrid approach of RPQ evaluation which we call WAVEGUIDE. We *provided* an evaluation framework based on iterative search with guided *wavefronts* which explore the graph. We systematically *defined* *waveplans* as structures

which guide the search. Further, we *demonstrated* that plan space in WAVEGUIDE approach *subsumes* state of the art and extends well beyond it. We *built* a full-fledged cost-based query *optimizer* for RPQs based on WAVEGUIDE. Specifically, we *modeled* the cost factors that determine the efficiency of the plans, and *presented* powerful optimizations offered by waveplans. We *devised* a concrete cost model for waveplans and *determined* an array of *statistics* which can be used in conjunction with the cost model. We *analyzed* the waveplan space and *designed* an efficient enumeration algorithm to walk it dynamically. Finally, we *implemented* a prototype of a WAVEGUIDE system and benchmarked it on realistic path queries on large real-world graph datasets. We *substantiated* the optimizations offered by our approach and *justified* the necessity of rich waveplan space. Further, we *tested* our prototype against state-of-the-art RPQ evaluation systems and demonstrated the significant performance gains offered by our approach.

## 8.2 Future Work

In this work, we have designed a system called WAVEGUIDE which aims to provide viable cost-based path query optimization and evaluation for SPARQL over RDF stores.

We now discuss several immediate directions in which this work can be extended. First, we discuss the extension of the WAVEGUIDE framework to handle multiple

RPQs (MRPQs) and conjunctive RPQs (CRPQs). Next, we discuss the improvements to the cardinality estimation that can be achieved in WAVEGUIDE. Finally, we present an extended way of handling Kleene subexpressions in WAVEGUIDE’s plan enumerator which will allow it to explore an even richer waveplan space.

### 8.2.1 Multiple & Conjunctive RPQs

In WAVEGUIDE, we focus on single-path, property path queries, essentially the RPQ fragment of SPARQL 1.1. An immediate extension of WAVEGUIDE framework is to handle *multiple* RPQs in a batch (MRPQs) and *conjunctive* RPQs (CRPQs).

Here, a basic optimization observation is that queries running in a batch may have common subexpressions among them. Instead of running each query individually, one could benefit from sharing the results of common subexpressions. We can think of evaluation of MRPQs as a two-step procedure. First, we identify common subexpressions among submitted queries. Second, we search for the global optimal plan such that its cost is less than combined cost of local optimal plans in a batch. We propose SWARMGUIDE, a generic optimization framework for MRPQs in graph databases.

In SWARMGUIDE, we identify a pipeline of four steps needed to optimize MRPQs: query clustering, finding common sub-plans, query rewriting, and global optimization. Query clustering is a preprocessing step which goal is to find the commonalities

among the RPQs in the batch. The commonalities can be detected by finding the isomorphic subgraphs of their corresponding finite automata (FAs). This process is known to be NP-hard, in general [39]. Therefore, we can use heuristics to group only those queries that can have common sub-automata, and then do the hard task of identifying the common sub-automata within each group.

Finding common sub-automata resembles finding the maximum common subgraph among graphs. The problem becomes more difficult here as we need to find the largest common subgraphs (sub-automata) for multiple graphs (FAs). Most existing solutions only consider non-labeled edges and nodes in undirected graphs. We adopt the solution from the maximal common-edge subgraph (MCES) problem [55] to detect common sub-automata. This has three steps: transforming labeled-graphs into the equivalent line-graphs; producing a product graph from the line-graphs; and detecting the maximal cliques in the product graph, which corresponds to MCESs (therefore, common sub-automata).

A local optimal plan for a given query may not be the best plan when optimizing a batch of queries. The goal of the global optimization is then to search local plans of queries to find a global plan by choosing one local plan per query in a cluster of RPQs. The cost of this global plan should be less than, or equal to, the total cost of local optimal plans of queries in the batch. In [58], the authors propose cost-based heuristic algorithms for this; we likewise adapt their algorithms for here.

A single waveplan is often decomposed into several wavefronts as views. When planning globally, one has to check if views from other queries plans can be shared. Intermediate node-pairs along with their states in FA are cached to avoid unbounded computation over cyclic graphs. When evaluating a global plan in MRPQs, these local caches need to be maintained globally so subplans shared among queries can be leveraged. The global cache also can be used to obtain useful statistics about the graph to obtain a more accurate global optimal plan and choosing the initial nodes for search exploration.

Conjunctive RPQs can be considered as an extension of MRPQs in which the end-points of one RPQ can be *joined* with end-points of another RPQ. In addition to challenges which we described for MRPQs, the end-point bindings specified by conjuncts in a CRPQ introduce more opportunities for *seeding* in which results of one RPQ are seeded into another RPQ. Even further, CRPQs evaluation opens opportunities for *sideways information passing* (SIP) which allows for RPQs to be evaluated *in parallel* while still influencing each other's execution at runtime. The seeding framework we introduced in WAVEGUIDE would need to be extended to handle these cases.

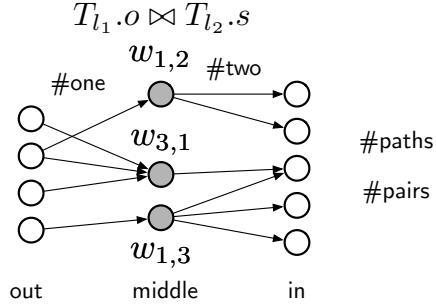


Figure 8.1: Extended Synopsis.

### 8.2.2 Better Cardinality Estimation

In this work, we have defined a graph search framework which is based on an iterative graph exploration guided by a waveplan. As discussed in 6.1, we have proposed an associated cost model based on the number of edge walks ( $\Delta_i$ ), cardinalities of subgraphs of  $G$ , and the sizes of search cache  $|C_i|$  at each iteration  $i$  of the search.

Cardinality estimation model described in §6.5 utilizes *synopsis*: a catalog of table statistics which stores single and joint label frequencies collected from given graph  $G$ . In WAVEGUIDE, a transition from one state to another in a wavefront automaton is a primitive operation (`crank`) which drives the search in a graph. In order to estimate the number of edge walks during the crank, we can use a classical formula used in the estimation of join sizes of two tables. Recall that this formula is based on three assumptions: *uniformity*, *independence*, and *inclusion*. While these assumptions greatly simplify the cardinality estimation, they also introduce

errors in estimates. We argue that inclusion assumption almost never holds in a case of path queries and, hence, might introduce catastrophic errors in cardinality estimation. We counter this in WAVEGUIDE by using joint-frequency synopsis which allows us to relax the inclusion assumption. However, we are still left with uniformity and independence assumptions. Here, we propose how to deal with both of them in WAVEGUIDE.

Recall that uniformity assumption states that the tuples are distributed uniformly across the join tables  $T_1$  and  $T_2$  such that each node in a join set  $\mathcal{J}_{T_1, T_2}$  would have the same number of tuples. This assumption, of course, does not usually hold in a case of path queries. In order to lift the uniformity assumption, we need to keep some sort of statistics for tuple cardinalities in all possible join sets. In order to achieve that, we propose an *extended* synopsis which stores additional statistics related to tuple cardinalities in a join set as follows. We partition nodes in a join set into set of bins  $\mathcal{W}$  such that each node in a bin  $w_{i,j} \in \mathcal{W}$  has exactly  $i$  incoming edges and  $j$  outgoing edges as shown in Fig. 8.1. Hence, nodes with high  $i$  and  $j$  are *heavy-hitters* as due to *multiplicity* they would produce  $i \cdot j$  paths. On the other hand, nodes in bin  $w_{1,1}$  would create only a single path as a result of a join. Moreover, just from having statistical estimates of binning  $\mathcal{W}$ , we can

produce other estimates of joint-frequency synopsis as follows:

$$\sum_{i=0}^{\#\text{out}} \sum_{j=0}^{\#\text{in}} i \cdot w_{i,j} = \#\text{one} \quad (8.1)$$

$$\sum_{i=0}^{\#\text{out}} \sum_{j=0}^{\#\text{in}} j \cdot w_{i,j} = \#\text{two} \quad (8.2)$$

$$\sum_{i=0}^{\#\text{out}} \sum_{j=0}^{\#\text{in}} i \cdot j \cdot w_{i,j} = \#\text{paths} \quad (8.3)$$

$$\sum_{i=0}^{\#\text{out}} \sum_{j=0}^{\#\text{in}} i \cdot w_{i,j} = \#\text{middle} \quad (8.4)$$

Of course, storing exact binning  $\mathcal{W}_{T_1, T_2}$  for every join set of tables  $T_1$  and  $T_2$  would be prohibitive. Thus, we would like to know how to produce an estimate of  $\mathcal{W}_{T_1, T_2}$  such that it would be useful in overcoming the uniformity assumption. Further, can equations (8.1-8.4) be used in conjunction with some form of an end-biased two-dimensional histogram to provide such an estimate.

### 8.2.3 A Richer Enumerator: Beyond Standard Waveplans

As we have shown in §5.3, the space of waveplans  $\mathcal{P}_{WP}$  is quite rich as it subsumes the plan spaces which correspond to both FA and  $\alpha$ -RA approaches. In §6.6.1, we have analyzed a subspace of  $\mathcal{P}_{WP}$ , a space of *standard* waveplans. We show that while  $\mathcal{P}_{WP}$  is exponential, it is possible to design an enumerator for it which has polynomial time complexity.

While enumerable  $\mathcal{P}_{SWP}$  is subsumed by  $\mathcal{P}_{WP}$ , we show that it still has many

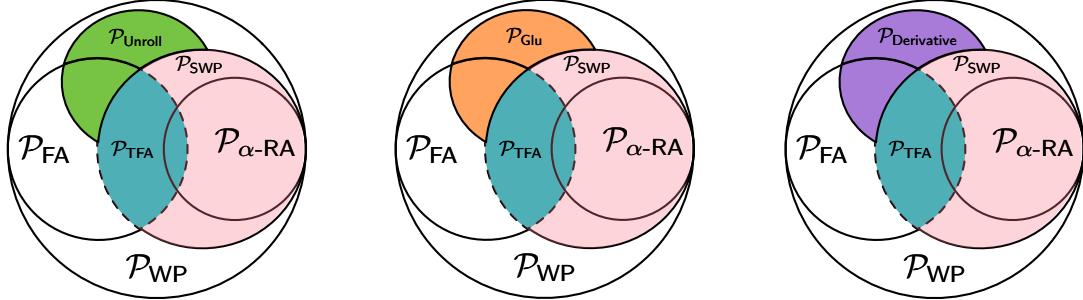


Figure 8.2: A richer enumerator.

interesting plans. Specifically,  $\mathcal{P}_{\text{SWP}}$  subsumes all plans which correspond to FAs generated by Thompson's construction algorithm,  $\mathcal{P}_{\text{TFA}}$ .  $\mathcal{P}_{\text{SWP}}$  also subsumes all plans in  $\alpha\text{-RA}$ ,  $\mathcal{P}_{\alpha\text{-RA}}$ . Further,  $\mathcal{P}_{\text{SWP}}$  has many interesting plans which are not found in  $\mathcal{P}_{\text{FA}}$ ,  $\mathcal{P}_{\alpha\text{-RA}}$ , and  $\mathcal{P}_{\text{TFA}}$ . One direction of future work in this project is to design an enumerator for several *extensions* to  $\mathcal{P}_{\text{SWP}}$ . We discuss some of them here (Fig. 8.2). We focus on three spaces specifically.  $\mathcal{P}_{\text{Unroll}}$  contains plans obtained by the *k-unrolling* procedure for Kleene expressions.  $\mathcal{P}_{\text{Glu}}$  has plans which use Glushkov automata in some or all of the wavefronts. Similarly, plans in  $\mathcal{P}_{\text{Derivative}}$  use derivative automata.

First, we tackle  $\mathcal{P}_{\text{Unroll}}$ . During the enumeration for standard waveplans, we generate a plan for Kleene expression  $r^+$  by following a fixed template  $W_{r^+}^1$  as shown in Fig. 8.3. We argue that by setting seed in this template to a full closure  $|r^+|$  in sub-plan  $p_r$  we are guaranteed to generate plan optimal for Kleene expression

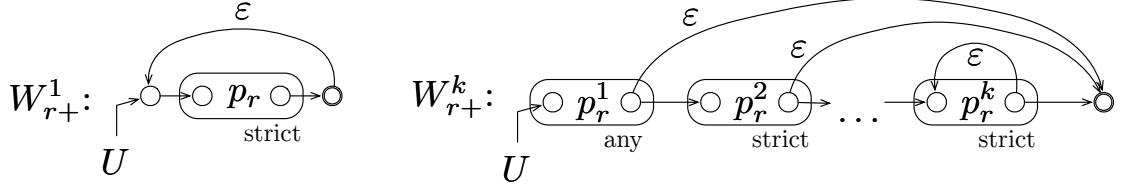


Figure 8.3: Example of  $k$ -unrolling.

$r+$ .

In template  $W_{r+}^1$  we are forcing the *same* plan  $p_r$  to be used for all “loops” in the plan for  $r+$ . It might be beneficial to lift this restriction and use *different* plans for  $p_r$  during the evaluation of  $r+$ . However, one needs to be careful in doing so due to unbounded nature of loops in  $W_{r+}^1$ . Technically, such *unrolling* may lead to an infinite number of waveplans for  $r+$ . Fortunately, we can easily show that for any graph  $G$  and any RPQ  $Q$  there exists a parameter  $k$  such that unrolling the Kleene plan beyond  $k$  different sub-plans would not make any difference in the query evaluation. The intuition beyond this is that all non-cyclic paths in any graph  $G$  are of bounded length. Hence, if you unroll a plan to match the length of the longest path satisfying  $r+$ , then unrolling further would not make any difference as nothing would match.

Plans in  $\mathcal{P}_{\text{Glushkov}}$  and  $\mathcal{P}_{\text{Derivative}}$  utilize Glushkov and derivative automata. These automata might have a different number of states and transitions when compared to Thompson’s automata in  $\mathcal{P}_{\text{TFA}}$ .

Some of the plans in  $\mathcal{P}_{\text{Unrolling}}$ ,  $\mathcal{P}_{\text{Glushkov}}$ , and  $\mathcal{P}_{\text{Derivative}}$  will be in  $\mathcal{P}_{\text{FA}}$ s, in some cases, constructs used in these plans can be mapped to automate. However, we envision that most interesting plans in these spaces will be outside  $\mathcal{P}_{\text{FA}}$ . These plans will be using the extended machinery offered by WAVEGUIDE such as multiple wavefronts, append and prepend transitions, and views. It is interesting to see if using these *extended* plans can lead to better performance when compared to standard waveplans.

### 8.3 In Summary

Just as new data models necessitate new query languages, these new query languages necessitate new approaches if we are to evaluate their queries efficiently and effectively. The second rise of graph databases has necessitated new, powerful query languages so that we can make use of them. But we are only beginning to understand how we can deal effectively with these types of queries.

In this dissertation, we have devised a rich domain of evaluation plans for property path type queries in SPARQL, and have shown it extends significantly over the state of the art. We have demonstrated that choice of a plan can make orders of magnitude difference in performance. We have illustrated the cost factors behind these plans' performance and the types of optimizations that can be achieved. We have shown which plans are effective depends on the underlying graph database,

which means a cost-based means of choosing plans is required.

The rise of graph data is well underway. And as we learned in the past to do the “impossible” for relational data, for semi-structured, for unstructured search, we too will meet this challenge.

## Bibliography

- [1] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley, 1995.
- [3] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE TSE*, 14(7):879–885, 1988.
- [4] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. Concepts for graph-oriented object manipulation. In *Advances in Database Technology*, pages 21–38. Springer, 1992.
- [5] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.
- [6] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
- [7] M. Arenas, S. Conca, and J. Pérez. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st international conference on World Wide Web*, pages 629–638. ACM, 2012.
- [8] A. Asperti, C. S. Coen, and E. Tassi. Regular expressions, au point. *arXiv preprint arXiv:1010.2604*, 2010.
- [9] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. Generating flexible workloads for graph databases. *PVLDB*, 9(13):1447–1460, 2016.

- [10] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.
- [11] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, et al. Support for repetitive transactions and ad hoc queries in system r. *ACM Transactions on Database Systems (TODS)*, 6(1):70–94, 1981.
- [12] C.-H. Chang and R. Paige. From regular expressions to DFA's using compressed NFA's. In *Annual Symposium on Combinatorial Pattern Matching*, pages 90–110. Springer, 1992.
- [13] S. Christodoulakis, U. of Waterloo. Dept. of Computer Science, and U. of Waterloo. Faculty of Mathematics. *On the estimation and use of selectivities in database performance evaluation*. Department of Computer Science, University of Waterloo, 1989.
- [14] M. P. Consens, A. O. Mendelzon, D. Vista, and P. T. Wood. Constant Propagation Versus Join Reordering in Datalog. In *Rules in Database Systems*, pages 245–259. Springer, 1995.
- [15] G. Copeland and S. Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.
- [16] T. P. P. Council. TPC-H benchmark specification. *Published at <http://www.tpc.org/hspec.html>*, 2008.
- [17] Cypher Query Language. <http://docs.neo4j.org/chunked/stable/cypher-query-lang.html>.
- [18] The DBpedia Knowledge Base. <http://dbpedia.org/>.
- [19] S. Dey, V. Cuevas-Vicentín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher. On implementing provenance-aware regular path queries with relational query engines. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 214–223. ACM, 2013.
- [20] A. Dutot, F. Guinand, D. Olivier, Y. Pigné, et al. GraphStream: A tool for bridging the gap between complex systems and dynamic graphs. In *Emergent Properties in Natural and Artificial Complex Systems (Satellite Conference within ECCS)*, 2007.
- [21] O. Erling and I. Mikhailov. Virtuoso: RDF Support in Native RDBMS. *Semantic Web Information Management*, 1:501, 2010.

- [22] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and robust pruning for top-down join enumeration algorithms. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 414–425. IEEE, 2012.
- [23] G. H. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. *Proceedings of the 19th International Conference on Extending Database Technology*, 2016.
- [24] P. García, D. López, J. Ruiz, and G. I. Álvarez. From regular expressions to smaller NFAs. *Theoretical Computer Science*, 412(41):5802–5807, 2011.
- [25] M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. GOOD: a graph-oriented object database system. In *ACM SIGMOD Record*, volume 22, pages 505–510. ACM, 1993.
- [26] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1, 1961.
- [27] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling Kleene: Fast Property Paths in RDF-3X. In *Workshop on Graph Data Management Experiences and Systems*, pages 14–20. ACM, 2013.
- [28] R. Guha and B. McBride. RDF Vocabulary Description Language 1.0: RDF Schema. 2004.
- [29] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C working draft. <http://www.w3.org/TR/sparql11-query/>, November 2012.
- [30] J. Hromkovič, S. Seibert, and T. Wilke. Translating regular expressions into small  $\epsilon$ -free nondeterministic finite automata. *Journal of Computer and System Sciences*, 62(4):565–588, 2001.
- [31] L. Ilie and S. Yu. Follow automata. *Information and computation*, 186(1):140–162, 2003.
- [32] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.
- [33] Y. E. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Transactions on Database Systems (TODS)*, 18(4):709–748, 1993.
- [34] Apache Jena. <https://jena.apache.org/>, 2013.

- [35] K. J. Kochut and M. Janik. SPARQLeR: Extended SPARQL for semantic association discovery. In *The Semantic Web: Research and Applications*, pages 145–159. Springer, 2007.
- [36] R. P. Kooi. The optimization of queries in relational databases. 1980.
- [37] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Scientific and Statistical Database Management*, pages 177–194. Springer Berlin Heidelberg, 2012.
- [38] G. M. Kuper and M. Y. Vardi. The logical data model. *ACM Transactions on Database Systems (TODS)*, 18(3):379–413, 1993.
- [39] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 666–677. IEEE, 2012.
- [40] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.
- [41] M. Levene and G. Loizou. A graph-based data model and its ramifications. *Knowledge and Data Engineering, IEEE Transactions on*, 7(5):809–823, 1995.
- [42] M. Levene and A. Poulovassilis. The hypernode model and its associated query language. In *Information Technology, 1990. 'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No. 90TH0326-9)*, pages 520–530. IEEE, 1990.
- [43] M. Levene and A. Poulovassilis. An object-oriented data model formalised through hypergraphs. *Data & Knowledge Engineering*, 6(3):205–224, 1991.
- [44] K. Losemann and W. Martens. The complexity of evaluating path expressions in SPARQL. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 101–112. ACM, 2012.
- [45] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys (CSUR)*, 20(3):191–221, 1988.
- [46] D. L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.
- [47] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 1(EC-9):39–47, 1960.

- [48] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.
- [49] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 539–552. ACM, 2008.
- [50] M. Muralikrishna and D. J. DeWitt. Equi-depth multidimensional histograms. In *ACM SIGMOD Record*, volume 17, pages 28–36. ACM, 1988.
- [51] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
- [52] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB*, volume 86, pages 25–28, 1986.
- [53] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):255–270, 2010.
- [54] E. Prud’Hommeaux and A. Seaborne. SPARQL query language for RDF. *W3C working draft*, 4(January), 2008.
- [55] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 16(7):521–533, 2002.
- [56] W3C: Resource Description Framework (RDF). <http://www.w3.org/TR/rdf-concepts/>, 2004.
- [57] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.
- [58] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.
- [59] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

- [60] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1009–1020. IEEE, 2013.
- [61] K. Thompson. Regular expression search algorithm. *Comm. ACM*, 11(6):419–422, 1968.
- [62] B. W. Watson et al. *Taxonomies and toolkits of regular language algorithms*. Eindhoven University of Technology, Department of Mathematics and Computing Science, 1995.
- [63] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, et al. Efficient RDF storage and retrieval in Jena2. In *Proceedings of SWDB*, volume 3, pages 131–150, 2003.
- [64] YAGO2s: A High-Quality Knowledge Base. <http://yago-knowledge.org/resource/>. Max Planck Institut Informatik.
- [65] N. Yakovets, P. Godfrey, and J. Gryz. Evaluation of SPARQL property paths via recursive SQL. In L. Bravo and M. Lenzerini, editors, *AMW*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org, May 2013.
- [66] H. Zauner, B. Linse, T. Furche, and F. Bry. A RPL through RDF: expressive navigation in RDF graphs. In *Web Reasoning and Rule Systems*, pages 251–257. Springer, 2010.

# A Appendix

## A.1 Nomenclature

$\alpha$ -RA relational algebra extended with  $\alpha$  operator, page 5

$\mathcal{P}_{\alpha\text{-RA}}$  space of relational algebra extended with  $\alpha$  plans, page 75

$\mathcal{P}_{\text{FA}}$  space of finite automata plans, page 75

$\mathcal{P}_{\text{SWP}}$  plan space of standard waveplans, page 101

$\mathcal{P}_{\text{TFA}}$  plan space of automata constructed by Thompson's algorithm, page 101

$\mathcal{P}_{\text{WP}}$  space of waveguide plans, page 75

SQL structured query language, page 141

SWP standard waveplan, page 104

fpp first-path pruning, page 83

TFA Thompson's automaton, page 104

ANTLR another tool for language recognition, page 144

BFS breadth-first search, page 82

Cypher Cypher query language, page 30

EDB extensional database, page 55

FA finite automata, page 5

FERRARI FERRARI reachability index, page 47

GMOD graph-oriented object modification, page 14

GOOD graph-oriented object database, page 14

IRI international resource identifier, page 15

LDM logical data model, page 13

LOD linked open data, page 3

OWL Web Ontology Language, page 3

RDBMS relational database management system, page 61

RDF resource description framework, page 1

RDFS resource description framework schema, page 17

RPQ regular path query, page 2

SPARQL SPARQL Protocol and RDF Query Language, page 1

SPJRU select-project-join-rename-union relational algebra, page 44

W3C world wide web consortium, page 15

WP waveplan, page 6

WWW world wide web, page 12

## A.2 Queries

### A.2.1 Threading

L	<i>p</i>	<i>p</i> <sub>1</sub>	<i>p</i> <sub>2</sub>
1	byTransport	rdf:type	owl:disjointWith
2	happenedIn	hasCapital	created
3	happenedIn	hasCapital	owns
4	happenedIn	hasCapital	rdf:type
5	happenedIn	hasCapital	participatedIn
6	happenedIn	dealsWith	rdf:type
7	happenedIn	dealsWith	hasCapital
8	happenedIn	rdf:type	owl:disjointWith
9	hasWonPrize	rdf:type	owl:disjointWith
10	hasWordnetDomain	rdf:type	owl:disjointWith
11	isCitizenOf	hasCapital	created
12	isCitizenOf	hasCapital	owns
13	isCitizenOf	hasCapital	rdf:type
14	isCitizenOf	hasCapital	participatedIn
15	isCitizenOf	dealsWith	rdf:type
16	isCitizenOf	dealsWith	hasCapital
17	isCitizenOf	rdf:type	owl:disjointWith
18	isLocatedIn	hasCapital	created
19	isLocatedIn	hasCapital	owns
20	isLocatedIn	hasCapital	rdf:type
21	isLocatedIn	dealsWith	hasCapital
22	isLocatedIn	rdf:type	owl:disjointWith
23	isPoliticianOf	hasCapital	created
24	isPoliticianOf	hasCapital	owns
25	isPoliticianOf	hasCapital	rdf:type
26	isPoliticianOf	hasCapital	participatedIn
27	isPoliticianOf	rdf:type	owl:disjointWith
28	rdfs:subClassOf	rdf:type	owl:disjointWith
29	wasBornIn	hasCapital	created
30	wasBornIn	hasCapital	owns
31	wasBornIn	hasCapital	rdf:type
32	wasBornIn	hasCapital	participatedIn
33	wasBornIn	dealsWith	rdf:type
34	wasBornIn	dealsWith	hasCapital
35	wasBornIn	rdf:type	owl:disjointWith
36	created	hasChild	isMarriedTo
37	created	influences	isMarriedTo
38	created	influences	hasChild
39	created	isMarriedTo	hasChild
40	hasChild	influences	isMarriedTo
41	hasChild	influences	hasChild
42	hasChild	isMarriedTo	hasChild
43	hasChild	hasChild	isMarriedTo

L	p	$p_1$	$p_2$
44	influences	hasChild	isMarriedTo
45	influences	isMarriedTo	hasChild
46	influences	influences	isMarriedTo
47	influences	influences	hasChild
48	isKnownFor	hasChild	isMarriedTo
49	isKnownFor	isMarriedTo	hasChild
50	isKnownFor	influences	isMarriedTo
51	isKnownFor	influences	hasChild
52	isMarriedTo	influences	isMarriedTo
53	isMarriedTo	influences	hasChild
54	isMarriedTo	hasChild	isMarriedTo
55	isMarriedTo	isMarriedTo	hasChild

### A.2.2 Loop Caching

D	a	b
1	formerTeam	name
2	region	candidate
3	clubs	currentMember
4	starring	title
5	successor	predecessor
6	successor	before
7	associatedMusicalArtist	currentMembers
8	associatedBand	currentMembers
9	associatedActs	currentMembers
10	after	predecessor
11	successor	successor
12	before	successor
13	before	successor
14	successor	after
15	currentMembers	associatedBand
16	before	predecessor
17	after	successor
18	imageCaption	ordo
19	imageCaption	order
20	label	subdivisionName
21	candidate	region

### A.2.3 Partial Loop Caching & State of the art Planning

Y/C	a	b	c
1	isLocatedIn	hasCapital	owns
2	isConnectedTo	isLocatedIn	owns
3	dealsWith	participatedIn	isLocatedIn
4	isLocatedIn	owns	created
5	participatedIn	isLocatedIn	dealsWith

Y/C	a	b	c
6	hasCapital	isLocatedIn	dealsWith
7	happenedIn	hasCapital	participatedIn
8	owns	isConnectedTo	isLocatedIn
9	dealsWith	hasCapital	isLocatedIn
10	participatedIn	happenedIn	hasCapital
11	owns	isLocatedIn	dealsWith
12	hasCapital	participatedIn	happenedIn
13	isLeaderOf	dealsWith	participatedIn
14	dealsWith	owns	isLocatedIn
15	created	isLocatedIn	owns
16	hasCapital	owns	isLocatedIn
17	owns	isLocatedIn	hasCapital
18	hasAcademicAdvisor	isInterestedIn	influences
19	isInterestedIn	influences	hasAcademicAdvisor
20	isMarriedTo	hasChild	influences
21	influences	hasAcademicAdvisor	isInterestedIn
22	isMarriedTo	influences	hasChild
23	hasChild	influences	isMarriedTo
24	influences	isMarriedTo	hasChild
25	owns	created	isLocatedIn

#### A.2.4 State of the art Planning

A	query
1	select distinct ?p {?p :diedIn/:isLocatedIn+ :France .}
2	select distinct ?p {?p :wasBornIn/:isLocatedIn+ :France .}
3	select distinct ?p {?p :wasBornIn/:isLocatedIn+/:type :Europe .}
4	select distinct ?p {?p :isMarriedTo/:diedIn/:isLocatedIn+/dealsWith :United_States .}
5	select distinct ?p {?p :isMarriedTo/:diedIn/:isLocatedIn+/dealsWith+ :United_States .}
6	select distinct ?p {?p :connectedTo+/:isLocatedIn+ :United_States .}