ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Engineering a Runtime System for AQL

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en informatique à finalité spécialisée (MA-IRIF)

Andrea Morciano

Directeur
Professeur Stijn Vansummeren

Superviseur
Martin Ugarte

Service
CoDE-WIT

Année académique
2015 - 2016

To Attilio and Magda.

Acknowledgments

I am very grateful to my advisor prof. Stijn Vansummeren, for his constant attention to my work, his willingness and his valuable comments and inputs. I am equally grateful to my supervisor Martin Ugarte, that helped me considerably throughout my efforts by following my progress closely, reasoning with me on every issue that I met and always providing useful suggestions, validating my results and scrupulously reviewing this document.

Finally, I would like to thank Attilio, Magda, Andrea, Sara, Isabella, Cecilia, Nicolas and Thu Minh for their precious support. Without you this thesis would not be possible.

"Your task is not to foresee the future, but to enable it."

Antoine de Saint Exupéry

Abstract

Information Extraction addresses the task of extracting information from unstructured text automatically. Traditionally, IE systems focus on the accuracy of extraction tasks. However, the problem of the *scalability* of these tasks has recently began to receive attention. SystemT is an IE system that successfully addresses this concern. This success stimulated many studies related to SystemT. One of the most prominent introduces a formal model that delineates a new approach to extraction tasks, considerably different from that of SystemT. According to this formal model, an extraction task is executed by first obtaining basic elements from the input text, and then combining these elements by means of algebraic operators to obtain the desired results. In this dissertation, I embark on the task of defining and implementing a new *runtime system* that allows for efficiently obtaining the desired results without the necessity to apply algebraic operators over basic elements. To this end, I first introduce a computational model (namely *eVset-automata*), that is based on a previous model (*vset-automata*), used for extracting basic elements from text, and I show that the two models are equivalent in expressive power. Moreover, I prove that under the proposed model there are *polynomial time constructions* that allow for directly evaluating the algebraic operators, without the need for obtaining the basic elements first. This represents an improvement over the previous approach (vset-automata), under which constructions for the same purpose existed but resulted in exponentially larger computations. Then, I discuss the results of a series of experiments executed on a corpus of real-life blog posts, that compare the performance of the runtime system with that of a subsystem that uses the same approach of SystemT. The experiments show that the runtime system is better than the subsystem in many cases. However, the subsystem is naturally capable of reusing partial results to reduce its workload, which can sometimes be advantageous. Finally, I lay the foundations for further improvements of the runtime system.

Résumé

La discipline nommé *Information Extraction* traite de la tâche d'extraire des informations de texte non structuré automatiquement. Traditionnellement, les systèmes IE mettent l'accent sur la précision des tâches d'extraction. Cependant, le problème de l'évolutivité de ces tâches a récemment commencé à recevoir une attention. SystemT est un système IE qui répond avec succès à cette préoccupation. Ce succès a stimulé de nombreuses études liées à SystemT. L'un des plus importants introduit un modèle formel qui définit une nouvelle approche pour les tâches d'extraction, considérablement différente de celle de SystemT. Selon ce modèle formel, une tâche d'extraction est exécutée en obtenant d'abord les éléments de base du texte d'entrée, puis en combinant ces éléments au moyen d'opérateurs algébriques afin d'obtenir les résultats souhaités. Dans cette thèse, je me embarque sur la tâche de définir et mettre en œuvre un nouveau *système d'exécution* qui permet d'obtenir efficacement les résultats souhaités sans la nécessité d'appliquer les opérateurs algébriques sur les éléments de base. À cette fin, je présente d'abord un modèle de calcul (à savoir *eVset-automata*), qui est basé sur un modèle précédent (*vset-automata*), utilisé pour extraire les éléments de base à partir du texte, et je montre que les deux modèles sont équivalents en puissance expressive. De plus, je montre que selon le modèle proposé il y a des *constructions polynomiaux* qui permettent d'évaluer directement les opérateurs algébriques, sans la nécessité d'obtenir des éléments de base en premier. Cela représente une amélioration par rapport à l'approche précédente (*vset-automata*), dans laquelle des constructions pour le même but existaient mais ils entraînaient des exponentiellement plus grands calculs. Ensuite, je discute les résultats d'une série d'expériences réalisées sur un corpus de blogs réelles, qui comparent les performances du système d'exécution à celui d'un sous-système qui utilise la même approche de SystemT. Les expériences montrent que le système d'exécution est meilleur que le sous-système dans de nombreux cas. Toutefois, le sous-système est naturellement capable de réutiliser les résultats partiels pour réduire sa charge de travail, ce qui peut parfois être avantageux. Enfin, je pose les bases d'autres améliorations du système d'exécution.

Contents

List of Figures

Chapter 1

Introduction

In recent years, companies and people have witnessed a huge increase in the amount of available data. This growth is not likely to stop; instead, it is probably going to proceed at an ever-accelerating pace. Companies are trying to use the datasets that they possess, aided by computer software, to obtain useful insights on their business, on their reference market, etc. Moreover, in many cases, data constitute a fundamental component of the core business of an enterprise, making it even more important to exploit them to the fullest.

A considerable part of the total amount of the available data comes in the form of documents, containing text written (entirely or partially) in natural language. For instance, web pages are a category of such documents. While computer generated data usually have a fixed structure, that is not the case for text documents. In order for the information in a document to be used by a software agent, a way to make its semantic structure explicit must be provided. This is the goal of *Information Extraction*. It is the discipline that deals with the task of obtaining structured information from unstructured or semi-structured documents algorithmically.

Information Extraction (IE) began in the 1970s inside the early Natural Language Processing (NLP) community [Cowie and Wilks, 1996]. Ever since, it has received contributions from a number of other research areas, such as Information Retrieval (IR) and Machine Learning (ML) [Sarawagi, 2008]. Over the years, two prominent approaches to IE were identified: *rule-based* and *statistical*. The main focus of IE systems has historically been the *accuracy* of the extraction, neglecting aspects such as *throughput*, *scalability*, *flexibility*, etc. On the contrary, enterprises, which are nowadays interested in IE as a mean of increasing the added value of their datasets, are concerned with these aspects. To bridge the gap between the traditional systems and the modern requirements, companies have been pursuing a series of research initiatives. At IBM, this effort led to the development of *SystemT*, a system for rule-based Information Extraction. Its first version appeared in 2008 [Reiss et al., 2008]. The main innovation of SystemT lies in its approach to the speci-

fication and execution of extraction tasks: a set of basic extractors are set up, using regular expressions or dictionaries, then they are combined by means of algebraic operators, thus defining two distinct execution phases. The creators of SystemT call this method *the algebraic approach*. The execution of an extraction task is considerably faster with SystemT than with traditional systems, which usually rely on *cascaded formal grammars*, that are very expensive to evaluate [Reiss et al., 2008, Krishnamurthy et al., 2009]. We can say that SystemT tackles the issue of the scalability of extraction tasks with success. Moreover, it supports an iterative and modular development process, thus providing flexibility to its users.

The great results of SystemT drew the attention of the research community on it, and a series of related studies were conducted. One of the most prominent describes a formal model for the core fragment of the language used to express extraction tasks in SystemT, called Annotation Query Language (AQL) [Fagin et al., 2015]. The model introduces the concept of *document spanner*. A document spanner is an entity that extracts a set of tuples containing spans of text from a document, and it generalizes the concept of extractor. The ability of AQL to combine extractors is modeled with the concept of *algebra of spanners*, which is a set of operators that combine the outputs of their input spanners. The most important operators described in [Fagin et al., 2015] are the standard relational operators: *projection, union* and *natural join*. These operators are traditionally used for querying a relational database. The authors propose three classes of formal models to represent document spanners: *regex formulas*, *variable stack automata* (or *vstk-automata*) and *variable set automata* (or *vset-automata*). Regex formulas model regular expressions with capture variables, also used in AQL, while vstk-automata and vset-automata are Nondeterministic Finite State Automata (NFAs) opportunely modified to retain spans of the input strings. A particularly interesting result of the study is that we can simulate the projection, the union and the natural join of spanners represented by vset-automata with another vset-automaton. This fact discloses the opportunity to evaluate AQL queries, at least in part, by using NFA execution engines (modified as needed), whose properties and performances have been extensively studied (see, e.g., [Cox, 2007, Yang et al., 2012, Grathwohl et al., 2016]).

1.1 Contributions

With this thesis, I make the following contributions:

- I introduce a new class of spanner representations: *extended vset-automata* (or *eVset-automata*). EVset-automata are a modified version of vset-automata. I show that the two types of representations are equivalent. Moreover, I define *well-behaved eVset-*

automata, a special kind of eVset-automata that has good execution properties;

- I prove the existence of *polynomial-time* constructions to simulate projection, union and natural join of well-behaved eVset-automata. These constructions represent an improvement over the analogous constructions described in [Fagin et al., 2015] for vset-automata, whose *space* complexity is, in general, exponential. Moreover, the new constructions preserve well-behavedness;

- I have designed and implemented a runtime system for the core of AQL. It is based on the model presented in [Fagin et al., 2015]. It works exclusively with well-behaved eVset-automata, which I show to guarantee the full applicability of the system. The latter has a compilation module that exploits the mentioned constructions to compile an AQL query into a well-behaved eVset-automaton;

- I implemented a subsystem that imitates the way SystemT executes AQL queries, in order to perform a fair comparison between the approach delineated in [Fagin et al., 2015] and the algebraic approach;

- I performed some experiments with the system. I developed a set of queries aiming to find informal movie reviews in a text, to be run on a corpus of blog posts. The queries were executed both with the runtime system and the subsystem imitating SystemT, allowing to compare the two. I also experimented certain transformations of some of the test queries that improved the performance of the runtime system;

In general, I believe that this work could serve as a basis for further research on the performance advantages and issues of the approach presented in [Fagin et al., 2015], and stimulate the development of optimization techniques specific to it.

1.2 Outline

The outline of this dissertation is as follows. Chapter 2 is an overview of Information Extraction. Chapter 3 describes SystemT and its query language AQL, and it discusses the model for the core of AQL proposed in [Fagin et al., 2015]. In Chapter 4, I formally describe the elements of the runtime system, in particular well-behaved eVset-automata. I discuss their expressive power and I introduce the mentioned constructions for simulating projection, union and natural join. Chapter 5 is a description of the implementation of the runtime system. Chapter 6 contains a detailed description of the experiments and a discussion of the results. Finally, in Chapter 7, I summarize the work done, I make some closing considerations and I suggest some interesting research perspectives for the future.

Chapter 2

Information Extraction

In this chapter, I give a generic introduction to Information Extraction. This chapter is a re-elaboration of [Sarawagi, 2008]. Refer to the original source for a more comprehensive and detailed overview. The figures and examples appearing in this chapter are original. The chapter is structured as follows: Section 2.1 contains a general definition of Information Extraction; in Section 2.2, a series of application domains where Information Extraction can be applied are reviewed; Section 2.3 is a discussion of the main challenges that are met in extraction tasks; finally, Section 2.4 and Section 2.5 contain high-level descriptions of the two main subtasks of Information Extraction: Entity Extraction and Relationship Extraction, respectively.

2.1 Definition

Information Extraction (IE) is the discipline that addresses the task of extracting structured information from unstructured sources automatically. Sources usually take the form of text documents. The most explored aspect of IE is the extraction of *named entities.* Another main topic, that has become object of research only in recent years, is the extraction of *relationships* between entities. IE is a field with contributions from many different communities: Machine Learning, Information Retrieval, Database, Web, Document Analysis. The interest towards it is motivated by the constantly increasing amount of data that is generated by our society. IE promises to bring value to many application domains, most notably the enterprise world and the Web.

There are two prominent approaches to IE:

rule-based which defines a set of rules that the output has to respect. The rules can be manually coded or learned from examples;

statistical which seeks to identify a decomposition of unstructured text and to label its

Stationsstraat, 336, 5032, Isnes

Sreet: Herentalsebaan, Nr: 219, ZIP: 3800, City: Brustem

Sreet: Stationsstraat, Nr: 336, ZIP: 5032, City: Isnes

↓

Street	**Nr**	**ZIP**	**City**
Stationsstraat	336	5032	Isnes
Herentalsebaan	219	3800	Brustem
Stationsstraat	336	5032	Isnes

↓

Street	**Nr**	**ZIP**	**City**
Stationsstraat	336	5032	Isnes
Herentalsebaan	219	3800	Brustem

Figure 2.1 – Extraction of structured addresses from flat strings (with deduplication).

components.

2.2 Example Applications

IE can serve for a large variety of tasks. What follows is a review of some of the most common ones.

News Tracking The activity of tracking events in news articles. It can result in a lot of useful services, like automatic creation of multimedia news, linking articles to information pages on the entities found, etc.

Example 2.1: *Google Alerts*[1] *lets the user specify a subject of interest and it offers a series of news articles related to that subject.*

Data Cleaning Extracting structured forms from flat data strings (containing, e.g., addresses). It allows more effective deduplication of information, among the other things.

[1] `https://www.google.com/alerts`

Example 2.2: *Figure 2.1 illustrates an example data cleaning task. A structured table of addresses is obtained from a set of flat address strings. Subsequently, duplicates are removed from the table.*

Citation Databases Articles, conference sites, individual research sites and similar are explored to obtain formatted citations of publications, later stored in publicly accessible databases. The latter are capable to forward references and may provide aggregate statistics and scoring information.

Example 2.3: *The citation database Google Scholar*[2] *automatically maintains, for each paper registered, a list of papers that refer to it. The user can, in turn, consult the citations of these papers too, and so on.*

Relationship Web Search Relationship extraction would be a very useful feature to integrate into web search engines, as the keyword search that they offer at present time is only good for entity identification.

As the reader can see, IE can be used for a variety of diversified data analysis tasks. Nonetheless, the criteria that we use to measure the quality of an IE system are always the same. The next section is a review of the most important issues that a developer of an IE system has take care of, in order to obtain satisfactory results.

2.3 Issues

The main issues that have to be dealt with when performing IE can be divided in two categories: *accuracy* and *running time*.

2.3.1 Accuracy

Only the main metrics for entity extraction are reported here. Those for relationship extraction are similar. We measure the accuracy of an entity extraction task with two quantities:

- *precision*: the percentage of correctly extracted entities among the extracted entities;
- *recall*: the percentage of entities extracted among all the existing entities in the input source.

The main difficulties to achieving a good level of accuracy are:

[2] `https://scholar.google.com/`

- the availability of a great set of clues that might be very different in nature (e. g., orthographic properties, part of speech, typical words, etc.) and that might be difficult to combine;
- the difficulty of identifying missed extractions;
- the fact that with the advancement of research the extracted data structures keep increasing in complexity (for instance, it is becoming more difficult to identify the boundaries of an entity in a text).

While it is possible to reach a very good level of accuracy for entity extraction (90%), relationship extraction is still quite unreliable ($50 - 70\%$ accuracy), mainly due to its intrinsic complexity.

2.3.2 Running Time

Nowadays companies need to process enormous amounts of data on a regular basis. Hence, they have an increasing interest in IE. Since their datasets keep growing in size, *scalability* has become a central issue in the field of IE.

This trend has appeared only in recent years, while, historically, this discipline has always been confined to the domain of theoretical research. As a consequence many IE systems don't address the problem of efficiently carrying out extraction tasks with sufficient attention.

The overall efficiency of an extraction task is influenced most notably by the efficiency of the following tasks:

- filtering documents in order to actually examine only the ones that have good chances to contain the desired information;
- focusing on the parts of a document that contain relevant information;
- processing steps, like database querying, that are typically very expensive and that might have to be performed on the selected pieces of input.

Recently, many solutions have been proposed to target the scalability issue; one of these is SystemT. It is a system for rule-based Information Extraction developed at IBM research laboratories, that is based on an innovative approach to defining and executing extraction task: the so-called "algebraic approach". SystemT is a promising solution, since it has been shown that it can achieve great performance improvements over classical systems (see [Reiss et al., 2008, Krishnamurthy et al., 2009]). SystemT will be discussed in detail in the next chapter.

2.4 Entity Extraction

Named entities are elements of interest in a text. Example of entities are person names, street addresses, institutions, countries, and so on. In this section and in the next one, I assume that the output of an extraction task is a series of labels inserted into an input text document (that becomes *annotated*), although there are other possible output formats.

2.4.1 Rule-based Methods

Rule-based methods employ sets of predicates that are "fired" independently. When a portion of input text satisfies a predicate, the action associated with the predicate is executed. A predicate can be represented in the following generic form:

$$\text{"Contextual Pattern} \longrightarrow \text{Action"} \tag{2.1}$$

Contextual patterns are a way to identify entities by exploiting their properties or the context in which they are usually met. The most common way to express them is the specification of *regular expressions* over tokens of the input. Actions mark the entities that have been identified, and might consist in storing them in a database or adding delimiters directly in the text.

Most systems in this category present a *cascaded structure*: an input document goes through a series of processing phases, and the output of a phase is the input of the next one. A famous example is that of *cascading grammars*: formal grammars are evaluated in sequence on the input.

As mentioned before, a contextual pattern seeks for (groups of) tokens that have certain *features*. In the case of entity extraction, features can be classified in the following categories:

- string representation;
- orthography (e.g., small case, mixed case, number, etc.);
- part of speech;
- list of dictionaries they're contained into;
- annotations obtained in previous phases.

Rules can be hand-coded by experts or learned through learning algorithms. In the second case the goal is to cover each of the entities of interest in the training set with at least one rule. The obtained rules should have good recall and precision on new input. Moreover, when learning a set of rules, we would like to achieve a good level of *generalizability*, that

is: we would like to find the minimum set of rules accounting for the maximum portion of training data, with high precision. There are two main strategies for rule learning: *bottom-up* (start from very specific rules and make them more and more general) and *top-down* (start with rules covering all existing instances, then specialize them).

2.4.1.1 Example Rules

For these examples, I use an abstract syntax, which is the same used in [Sarawagi, 2008].

Example 2.4: *Consider the task of identifying all mentions of ISO standards in a text. A rule for this purpose could be:*

$$\begin{aligned}&(\{\text{String} = \text{"ISO"}\}\,\{\text{String} = \text{"/IEC"}\}\,\{?\}\,\{\text{String} = \text{"/ASTM"}\}\,\{?\}\\&\{\text{Orthography type} = \text{number}\}\,\{\{\text{String} = \text{":"}\}\,\{\text{Orthography type} = \text{number}\}\}\,\{?\})\\&\longrightarrow \text{ISO Standards}\end{aligned} \quad (2.2)$$

This rule matches all strings starting with a substring equal to "ISO"*, followed by an optional (because of the* {?} *modifier) substring equal to* "/IEC"*, then a number identifying the particular standard and finally an optional number, separated by a colon from the previous one. A matched string is added to a set of matches with name* "ISO Standards"*.*

Example 2.5: *Multiple entities can be matched at once. Imagine we need to find mentions of (simple) street addresses in a text consisting of a street name and a street number. A rule that matches the name and number separately could be:*

$$\begin{aligned}&(\{\text{Orthography type} = \text{mixed case word}\}\,\{*\}) : \text{Name}\ (\{\text{String} = \text{","}\})\\&\qquad(\{\text{Orthography type} = \text{number}\}) : \text{Number}\\&\qquad\longrightarrow \text{Street Name} =: \text{Name, Street Number} =: \text{Number}\end{aligned} \quad (2.3)$$

2.4.2 Statistical Methods

Statistical methods aim to decompose the source, assigning a label to each element in the decomposition. We distinguish between three types of statistical models:

- *Token-level models*: they assign a label to each token of the source. Since entities are usually comprised of multiple adjacent tokens, the tags used are of the forms "entity_begin", "entity_continue", "entity_end";

Yesterday I watched a movie called “The Matrix”

i	1	2	3	4	5	6	7	8	9	10
x	Yesterday	I	watched	a	movie	called	“	The	Matrix	“
y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}

Figure 2.2 – Decomposition of a sentence into a sequence of tokens.

- *Segment-level models*: they try to find the best segmentation of the source text;
- *Grammar-based models:* they use formal grammars, outputting parse trees. All the valid parses are considered for an input document, assigning a score to each. The parse with the highest score is retained.

I now give a brief description of Token-level Models and Segment-level Models. In the following the term 'clue' is used as a synonym of 'feature'.

2.4.2.1 Token-Level Models

Features The features that these models exploit are of the form

$$f : (\mathbf{x}, y, i) \longmapsto \mathbb{R} \tag{2.4}$$

where $\mathbf{x}$ is a sequence of tokens, i is a position in $\mathbf{x}$ and y is a candidate label for the token at i. We distinguish between these types of features:

- *word features*;
- *orthographic features;*
- *dictionary lookup features.*

Example 2.6: *Consider the sentence shown in Figure 2.2 and its corresponding decomposition. An example of word feature at position* 9 *is*

$$f_1\left(y, \mathbf{x}, i\right) = [\![x_i \text{ equals "Matrix"}]\!] \cdot [\![y = \text{Movie}]\!]$$

where $[\![P]\!] = 1$ *if predicate* P *is true, and* 0 *otherwise. An orthographic feature might be*

$$f_2\left(y, \mathbf{x}, i\right) = [\![x_i x_{i+1} \text{ matches INITIAL\_QUOTE CapsWord}]\!] \cdot [\![y = \text{Movie}]\!] .$$

Finally, an example of dictionary lookup feature is

$$f_3\left(y, \mathbf{x}, i\right) = [\![x_i \text{ in Movie\_dictionary}]\!] \cdot [\![y = \text{Movie}]\!] .$$

"The Matrix" by The Wachowski Brothers

i	1 2 3 4	5	6 7 8
x	"The Matrix"	by	The Wachowski Brothers
(l_j, u_j, y_i)	$1, 4, M$	$5, 5, O$	$6, 8, D$

Figure 2.3 – Segmentation of a sentence.

Models for Labeling Tokens The best models are the ones that take into account dependencies between tokens, among which we may find:

- *ordered classifiers*;
- *Hidden Markov models;*
- *Maximum Entropy Taggers;*
- *Conditional Markov Models;*
- *Conditional Random Fields* (the state of the art).

2.4.2.2 Segment-Level Models

Features In these models the label for a segment depends on the properties of its tokens and on the previous segment. We can describe a segment s_j as:

$$s_j = (y_i, l_j, u_j) \tag{2.5}$$

where y_i is the proposed label for s_j and l_j, u_j are the start and end positions of s_j. Therefore, a feature is of the form:

$$f\left(y_i, y_{i-1}, \mathbf{x}, l_j, u_j\right) \tag{2.6}$$

where $\mathbf{x}$ is the input sequence of tokens, y_{i-1} is the proposed label for the previous segment and the other symbols are defined as for Equation 2.5. Besides token-level features (see Subsection 2.4.2.1), we can exploit the following feature types:

- *Similarity to an entity in a database;*
- *Length of the segment.*

Example 2.7: *Consider the sentence in Figure 2.3 and its corresponding segmentation, which identifies a movie and a director. A basic similarity feature for the director could be:*

$$f\left(y_i, y_{i-1}, \mathbf{x}, 6, 8\right) = [\![x_6x_7x_8 \text{ appears in a list of movie directors}]\!] \cdot [\![y_i = \text{Director}]\!] .$$

A more realistic feature would make use of some similarity function, rather than requiring an exact match. An example of length feature is

$$f\left(y_i, y_{i-1}, \mathbf{x}, l, u\right) = [\![u - l = 3]\!] \cdot [\![y_i = \text{Director}]\!] .$$

There exist also global segmentation models, that try to find the best segmentation of a token sequence by maximizing a target function.

2.5 Relationship Extraction

When extracting relationships between entities, we might face three types of specific tasks:

1. given a pair of entities, find the relationship between them;
2. given an entity e and a relationship r, find all the other entities e' such that $(e, e') \in r$;
3. given a big and open-ended input and a relationship r find all pairs of entities e', e'' such that $(e', e'') \in r$.

2.5.1 Predicting the Relationship Between a Pair of Entities

For the first task, we can exploit the following resources:

- *surface tokens*: tokens that are usually placed between entities, which are strong clues;
- *part of speech tags* (the most important being *verbs*);
- *syntactic parse tree*: allows grouping words in phrase types, e.g., noun phrases, propositional phrases, and so on;
- *dependency graph*: it is a less expensive structure to compute than the parse tree and it links a word to those that depend on it.

The main methods available to carry out the task are:

- *Feature-based methods*, that simply transform the clues mentioned above for usage by conventional classifier models;
- *Kernel-based methods*, that use kernel functions to encode the similarity between two graphs;
- *Rule-based methods*, creating rules over structures around pairs of entities.

The second task is a special case of the third, and will not be treated here.

2.5.2 Finding All Possible Entity Pairs Belonging to a Relationship

The third task can be met especially when dealing with the Web. Usually we can exploit the following resources to fulfill it:

- the *types of arguments* of r (that might need specific recognition patterns);
- a *seed database* of pairs of entities belonging to r;
- *manually coded patterns*.

The generic procedure that is used in this case can be described with these steps:

1. Use the seed database to learn the relevant extraction patterns;
2. Use the obtained patterns to define candidate triples of the form (e', e'', r);
3. Retain a subset of the candidate triples, using a statistical test.

There exist also rule-based methods for the task.

In this chapter, a high-level introduction to Information Extraction was given. Examples of applicability were mentioned, the main challenges were highlighted, and the two prominent approaches to extraction tasks were described.

The next chapter focuses on SystemT, a concrete system for rule-based information extraction.

Chapter 3

SystemT and AQL

This chapter contains a description of *SystemT* and its extraction rule language *AQL* (Annotation Query Language), followed by a description of a new formal model capable of capturing the core of AQL. The model is focused on the concept of *document spanner*. A document spanner provides a basis for a mathematical description of an AQL query. Spanners can be associated with formal representations, for which execution models can be defined easily. This new model is the foundation for the runtime system for the core of AQL developed in this thesis. This chapter is a re-elaboration from [Reiss et al., 2008, Krishnamurthy et al., 2009, Fagin et al., 2015]. In particular, the mathematical statements from 3.1 to 3.8 are taken or adapted from [Fagin et al., 2015], except for Remarks 3.1 to 3.3, that come from [Reiss et al., 2008]. Refer to the original sources for more details on the topics that I treat in this chapter. The figures and examples appearing in this chapter are original, except when I explicitly state otherwise. The outline of the chapter is as follows. Section 3.1 is a general description of SystemT: its architecture, data model, execution model, query operators and optimization techniques are presented. Section 3.2 contains a brief overview of AQL. Section 3.3 is an extensive description of the new model: the concept of document spanner is defined, then three primitive spanner representations, originally defined in [Fagin et al., 2015], are presented. These representations are *regex formulas, variable stack automata,* and *variable set automata.* Subsequently, a series of operators for composing spanners is introduced. Finally, some interesting classes of spanners are described. These classes are based on the aforementioned primitive spanner representations. The spanner class that models the core of AQL is identified, along with some convenient classes of spanner representations that can model it.

3.1 SystemT

As mentioned in the previous chapter, scalability is now a main concern in IE. Companies rely on IE for many tasks; a prominent example is *Business Intelligence*. Unfortunately, many systems developed in the past don't address this issue correctly. Traditionally, most rule-based system rely on cascading grammars: formal grammars are executed in sequence on the input, each grammar representing a stage that takes as input the output of the previous one. Rules in such grammars are matched using regular expressions: if a part of the input text satisfies the regular expression associated with a rule, that rule is activated (i.e., "fires") and the corresponding action is executed. Evaluating a grammar on a document tends to be costly, because simply evaluating a single rule might require scanning the whole document. On large datasets, the running time becomes enormous (see, for instance, [Reiss et al., 2008]). Moreover, these kind of systems are not able to fulfill the expressivity requirements of complex extraction tasks.

SystemT was developed at IBM to address these issues. It is based on a new approach to extraction rules: the *algebraic approach*. According to this approach, data manipulations are viewed as operations in a (relational) algebra. Extraction tasks are defined using *annotators* (see Section 2.4) whose rules are conceived as *queries* on input documents, that act as virtual databases. Complex annotators are obtained by combining simpler ones, using *relational operators*. By doing so, all the optimization techniques that are typical of relational databases become available, but new techniques are possible too, due to the characteristics of text documents. Another advantage of SystemT is that it is capable of handling *overlapping rule matchings*, which are difficult to resolve in cascading grammars, where they typically require disambiguation policies that solve the problem only in part.

I now give an overview of the high-level structure of SystemT. Then, I talk about its data model, execution model and algebra. For more information on SystemT and AQL, refer to [Reiss et al., 2008, Krishnamurthy et al., 2009].

3.1.1 Architecture

These are the main components of SystemT:

Development Environment: allows constructing annotators for extraction tasks. Rules are expressed in AQL, and compiled into an algebra. It supports an iterative development process.

Optimizer: seeks for the best query plan for an extraction task, evaluating the most convenient optimization techniques in a cost-based manner.

Runtime Environment: given a query plan, it instantiates the physical operators corresponding to the logical ones in the plan, then it proceeds evaluating the latter on input documents.

Once the development of an annotator is complete, it is *published* to the optimizer and runtime. After optimization, the runtime evaluates it on a continuous stream of documents.

3.1.2 Data Model

SystemT uses an *object-relational* data model for annotations on a text document, that allows applying logical operators over it. Operators can be composed. There is an important assumption to mention before continuing: an extraction task over a set of documents is performed on one document at a time. This means that any relationships between entities in different documents are disregarded. This assumption is crucial to some optimization techniques that SystemT uses. While there exist tasks where considering these relationships would be useful (think of the Web), still a large number of relevant tasks can be carried out this way. In the following, I report the definitions of the basic data types in the data model of SystemT.

Let Σ be a finite alphabet of symbols (characters). Σ^* denotes the set of all strings of finite length over Σ. A document is modeled as one such string.

Definition 3.1: *A document is a string* $\mathbf{s} \in \Sigma^*$.

The most basic data type is the *span*.

Definition 3.2: *Given a string* $\mathbf{s} = \sigma_1 ... \sigma_n$ *where* $\forall i,\ \sigma_i \in \Sigma$, *with length* $|\mathbf{s}| = n$ *and whose characters are indexed in the natural way, a span of* $\mathbf{s}$ *is an interval* $[i, j\rangle$, *where* i, j *are indices of* $\mathbf{s}$ *satisfying* $1 \leq i \leq j \leq n+1$. *The substring of* $\mathbf{s}$ *beginning at* i *and ending at* $j-1$ *is denoted as* $\mathbf{s}_{[i,j\rangle}$.

Spans can be aggregated in *tuples*, which are finite sequences of spans. To formally define a tuple, we need to introduce some additional concepts and notation. The first concept is that of *span variable*. A span variable is simply a variable which can be assigned a span. Now, let us denote by SVars an infinite set of span variables. A tuple is always parametrized by a string $\mathbf{s}$ and a finite subset V of SVars. We stress this fact by calling a tuple a $(V, \mathbf{s})$-tuple. Let $\mathrm{Spans}(\mathbf{s})$ be the set of all possible spans of a string $\mathbf{s}$. A $(V, \mathbf{s})$-tuple is defined as follows:

Definition 3.3: *Given a finite set* $V \subseteq \mathrm{SVars}$, *and a string* $\mathbf{s} \in \Sigma^*$, *a* $(V, \mathbf{s})$*-tuple is a mapping* $\mu : V \longmapsto \mathrm{Spans}(\mathbf{s})$. *When* V *is clear from the context, we might call* μ *simply a* $\mathbf{s}$*-tuple.*

Algorithm 3.1 Annotating all local databases in a global database (taken from [Reiss et al., 2008]).

$E \longleftarrow$ {algebra expression}
for localDB in globalDB do
begin

1. {Read localDB into main memory}
2. $R \longleftarrow E$ (localDB)
3. {Add R to localDB}
4. {Write modified localDB to disk}

In general, a set of tuples is called a *relation.* Here, we focus on *span relations*, which are formally defined as $(V, \mathbf{s})$-relations.

Definition 3.4: *A* $(V, \mathbf{s})$*-relation is a set of* $(V, \mathbf{s})$*-tuples. As before, we can speak of* $\mathbf{s}$*-relation when* V *is clear.*

Each operator in the algebra of SystemT takes one or more span relations as input and outputs a single span relation.

3.1.3 Execution Model

A single document is conceived as a *local annotation database*, to which annotators are applied in order to build *views.* In general, a local database fits into main memory. Local databases are contained in a *global annotation database.* The runtime of SystemT takes a global database and it annotates its local databases. The procedure is described by Algorithm 3.1.

3.1.4 Algebra of Operators

The operators of the algebra of SystemT can be classified in three groups:

- *relational operators;*
- *span extraction operators;*
- *span aggregation operators.*

In addition, there exist some *span selection predicates* that are used for span selection.

3.1.4.1 Relational Operators

Relational operators are the usual operators of relational algebra that appear in classical database query plans. Important examples are:

- *selection* (σ);
- *projection* (π);
- *Cartesian product* ($\times$);
- Join ($\bowtie$);
- *Union* ($\cup$);
- *Intersection* ($\cap$).

Span Extraction Operators

Loosely speaking, span extraction operators take a pattern and a document as input and output a maximal set of spans that match that pattern. There are two main span extraction operators:

- *standard regular expression matcher (ε_{re})*: this operator takes a regular expression r as input and it identifies all non-overlapping matchings of r in the current document, from left to right;
- *dictionary matcher (ε_d)*: this operator outputs all the spans that match some entry in a given dictionary.

Although the dictionary matcher might seem useless since there is a regular expression matcher, it has some advantages over the latter, as the fact that it can find overlapping matchings or that it enforces the semantics of word boundaries.

Span Aggregation Operators

Span aggregation operators are used to aggregate spans in a meaningful way. They are of two types:

- *Consolidation*: they are used to coalesce overlapping spans that were matched using patterns for the same concept. They are:

  containment consolidation (Ω_c): discards a span fully contained into another one.

  overlap consolidation (Ω_o): merges any sequence of overlapping spans into a single span.

- *Block (β)*: it matches a series of spans, each at a distance from its neighbor span(s) that is not superior to a threshold. It is thought to identify regularity, and it is tuned by two parameters: a *distance constraint* to control regularity and a *count constraint* that establishes the minimum number of spans in a block.

3.1.4.2 Span Selection Predicates

Consider two spans s_1, s_2. The main span predicates that may be used for selection are the following:

$s_1 \preceq_d s_2$ when s_1, s_2 do not overlap, s_1 precedes s_2 and there are less than d characters between them;

$s_1 \simeq s_2$ when the two spans overlap;

$s_1 \subset s_2$ when s_1 is strictly contained in s_2;

$s_1 \subseteq s_2$ when s_1 is contained in or equals s_2;

$s_1 = s_2$ when s_1 equals s_2.

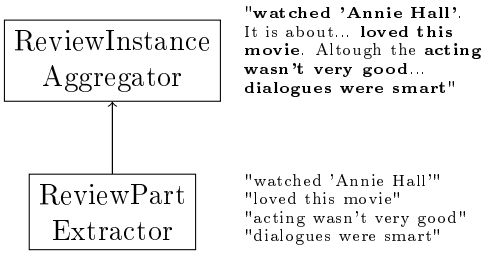
Example 3.1: *Consider the task of extracting informal movie reviews from blog posts. A possible annotator to perform the task is shown in Figure 3.1. Figure 3.1a shows the high-level structure of the annotator. There are two main components: the* ***ReviewPart*** *module, which extracts the text snippets that are identified as parts of a movie review, and the* ***ReviewInstance*** *module that aggregates adjacent snippets into single blocks of text. Figure 3.1b shows the operator tree of the annotator. Review parts are extracted by a series of join operators that combine elements of basic relations, extracted by running span extractors on the text. Subsequently, the block operator aggregates the parts, and review instances that are contained into others are discarded.*

3.1.5 Optimization Techniques

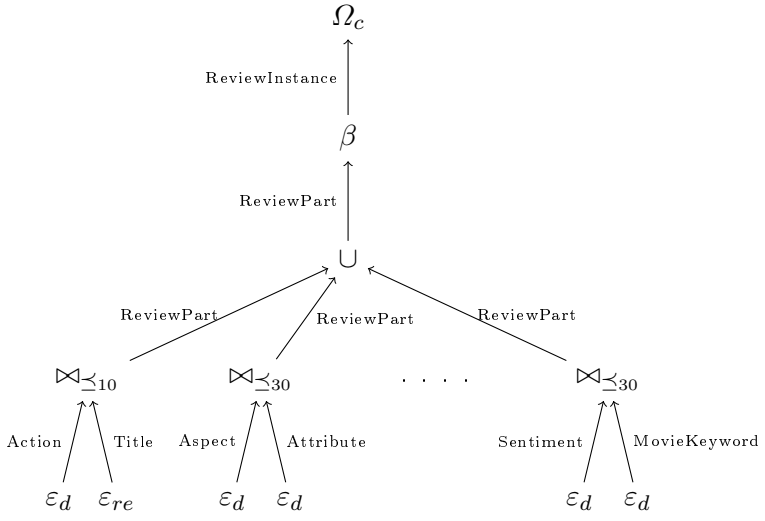
As mentioned in the introduction of this chapter, SystemT can make use of the optimization techniques from traditional database systems, but there exist some peculiar aspects of SystemT and span extraction tasks that enable new optimization methods.

Remark 3.1: *The effect of document-at-a-time processing is that the span relations produced and consumed for a single document by operators are very small in size and often empty.*

"Yesterday I **watched 'Annie Hall'**. It is about the relationship between a TV writer and his girlfriend, who wants to become an actress. I really **loved this movie**. Although the **acting wasn't very good** sometimes, the **dialogues were smart**. I'd recommend it to anyone."



(a) High-level structure.



(b) Operator tree.

Figure 3.1 – Extraction of informal movie reviews.

Remark 3.2: *Evaluating an annotator on a large set of documents is a* CPU-intensive process. *This is because the running time is by far dominated by the execution of the operators ε_{re} and ε_d, that are applied to each document in an input set.*

Remark 3.3: *Spans are nothing but* intervals*, so we can exploit* interval algebra.

With Remark 3.1 and Remark 3.3 in mind, running time can be reduced in a number of ways. Let us look at them briefly.

Regular Expression Strength Reduction Some classes of regular expressions, as defined in the POSIX standard, can be executed by using *specialized engines,* that are able to improve performance. For example, an expression that looks for a finite number of strings in a text is evaluated more efficiently by a string-matching engine.

Shared Dictionary Matching (SDM) Dictionary lookups are usually very expensive, as we need to consult the dictionary thousand of times in a typical setting. Instead of evaluating each ε_d operator in an extraction task independently for each input document, it is evaluated once and for all at the beginning of the process, and the obtained matches are shared among the single documents.

Conditional Evaluation (CE) As we know, an annotator is evaluated independently on each input document. Thus, by employing some heuristic, it can be guessed if a (sub)query will give any matches in a document without loss of generality, and in case it does not, it is not evaluated on it.

Restricted Span Extraction (RSE) It consists in executing expensive operations on selected regions of a document. It is used for *join* operators involving a dictionary matching operator in their arguments. One input operator is evaluated on the whole document, while the others are modified to consider only some neighborhood of the results from the first one. These neighborhoods are established by some ad hoc heuristic.

The techniques described allow the use of a *cost-based plan optimizer*: in a query plan, the subgraphs suitable for optimization are identified, and all the possible optimized plans for them are formulated. In the end, the best ones are retained. Results of sample experiments on a large dataset are described in [Reiss et al., 2008, Krishnamurthy et al., 2009].

3.2 AQL

AQL is the concrete language used by SystemT to express annotators, designed to support the execution model described in Subsection 3.1.3. It is a *declarative language* with a syntax

```
–Define a dictionary of actions
create dictionary Action as ('saw', 'watched', ... );

–Use a regular expression to find movie titles
create view Title as
extract regex /["'][A-Z]\w+(s+\w+){0,3}["']/
   on D.text
   as title
from Document D;

–A single ReviewPart rule. It looks for instances
–of Action followed within 10 characters by a title
create view ReviewPart as
select CombineSpans(A.act, T.title) as part
from
   (extract dictionary 'Action' on D.text as act
    from Document D) A, Title T
where
   Follows(A.act, T.title, 0, 10)
consolidate on CombineSpans(A.act, T.title);
```

Figure 3.2 – Parts of the annotator from Figure 3.1 expressed in AQL.

very similar to that of SQL. It supports all the operators presented in Subsection 3.1.4. When coding an extraction task in AQL, we can build a series of views of a document, of progressively higher abstraction level. Higher-level views are typically based on lower-level views. The content of a view corresponds with annotations in the text. An input document is modeled as a view too, which is provided by default. The main advantages of AQL are:

- it allows formulating complex low-level patterns in a declarative fashion;
- it enables *modularization* and *reuse* of the queries, making development and maintenance of complex high-level structures easier.

Example 3.2: *The AQL code in Figure 3.2 realizes a join combining an instance of an action related to movies with an instance of movie title, which follows the action within 10 characters. It is the first join from the left in Figure 3.1b.*

3.3 A Model for the Core of AQL

In [Fagin et al., 2015], a formal model capturing the core functionality of AQL is described. The authors show a way to represent annotators expressed in AQL by means of modified

finite state automata, namely *variable stack automata* and *variable set automata*. In this section, I present this model and look at the relative expressive power of its main elements.

3.3.1 Basic Definitions

Let us start with the definition of a language.

Definition 3.5: *A language L is a subset of Σ^*.*

An important concept is that of regular expressions, which can be defined by describing their language.

Definition 3.6: *Regular expressions over Σ are the strings that belong to the language γ, defined by the following grammar:*

$$\gamma := \emptyset \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \tag{3.1}$$

where:

- *$\emptyset$ is the expression matching the empty language;*
- *ϵ is the empty string;*
- *$\sigma \in \Sigma$;*
- *$\vee$ is the ordinary disjunction operator;*
- *$\cdot$ is the concatenation operator;*
- *$*$ is the Kleene Star operator.*

Additionally, we might use γ^+ as a shortcut for $\gamma \cdot \gamma^*$, and Σ as an abbreviation of $\sigma_1 \vee ... \vee \sigma_n$. The language $\mathcal{L}(\gamma)$ of a regular expression γ is the set of strings $\mathbf{s}$ over Σ that are matched by that expression. A language L is *regular* if $L = \mathcal{L}(\gamma)$ for some regular expression γ. We also need the definition of a *string relation*.

Definition 3.7: *A n-ary string relation is a subset of $(\Sigma^*)^n$.*

An interesting class of string relations is that of recognizable relations, denoted as REC.

Definition 3.8: *Given a k-ary string relation R, R is recognizable if it is representable by a finite union the form:*

$$\bigcup L_1 \times ... \times L_k \tag{3.2}$$

where each L_i is a regular language.

String s

I	**_**	**w**	**a**	**t**	**c**	**h**	**e**	**d**	**_**	**“**	**T**	**h**	**e**	**_**	**M**	**a**	**t**	**r**	**i**	**x**	**“**	**_**	**t**	**h**	**e**	**n**
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

_	**w**	**e**	**_**	**s**	**a**	**w**	**_**	**“**	**A**	**n**	**n**	**i**	**e**	**_**	**H**	**a**	**l**	**l**	**“**
28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47

$P(\mathbf{s})$

	x	y	z
μ_1	$[3, 10\rangle$	$[11, 23\rangle$	$[3, 23\rangle$
μ_2	$[32, 35\rangle$	$[36, 48\rangle$	$[32, 48\rangle$
μ_3	$[3, 10\rangle$	$[36, 48\rangle$	$[3, 48\rangle$

Figure 3.3 – A string $\mathbf{s}$ and the span relation obtained by applying spanner P to $\mathbf{s}$.

I have already mentioned span relations (Definition 3.4). They allow giving a precise definition of annotators, by introducing the concept of *document spanner*.

Definition 3.9: *Given a string* $\mathbf{s}$*, a document spanner* P *is a function that maps* $\mathbf{s}$ *to a* $(V, \mathbf{s})$*-relation* r*, where* $V := \mathrm{SVars}(P)$*. We say* P *is* n*-ary if* $|V| = n$*.*

Example 3.3: *Consider the string* $\mathbf{s}$*, over the alphabet* $\Sigma = \{A, \ldots, Z, a, \ldots, z, \_, ''\}$*, shown in Figure 3.3. The character '*_*' can be thought as a space. The table shown below* $\mathbf{s}$ *represents the span relation* $P(\mathbf{s})$*, which is the result of applying the ternary spanner* P *to* $\mathbf{s}$*.*

I now present two special types of spanners, which were originally defined in [Fagin et al., 2015] (see Subsection 2.2, ibid.), and that will be useful later on, when we will reason on the relative power of various classes of spanners of interest. The first one is that of *hierarchical spanners*. In order to formulate their definition, we need to say when a $\mathbf{s}$-tuple is hierarchical.

Definition 3.10: *Given a string* $\mathbf{s}$*, a document spanner* P *and a* $\mathbf{s}$*-tuple* $\mu \in P(\mathbf{s})$*,* μ *is hierarchical if, for every* $x, y \in \mathrm{SVars}(P)$*, one of the following conditions holds:*

- $\mu(x) \supseteq \mu(y)$;
- $\mu(x) \subseteq \mu(y)$;
- $\mu(x)$ *is disjoint from* $\mu(y)$*.*

The definition of *hierarchical spanner* follows.

Definition 3.11: *Given a document spanner* P*,* P *is hierarchical if* $\forall \mathbf{s} \in \Sigma^*$*,* $\forall \mu \in P(\mathbf{s})$*,* μ *is hierarchical.*

The class of hierarchical spanners is denoted by **HS**. The second type of spanners considered is that of *universal spanners*. Some accessory definitions are needed.

Definition 3.12: *Given a string* $\mathbf{s}$ *and a document spanner* P*,* P *is total on* $\mathbf{s}$ *if* $P(\mathbf{s})$ *consists of all the possible* $\mathbf{s}$*-tuples on* $\mathrm{SVars}(P)$.

Definition 3.13: *Given a string* $\mathbf{s}$ *and a document spanner* P*,* P *is hierarchically total on* $\mathbf{s}$ *if* $P(\mathbf{s})$ *consists of all the possible hierarchical* $\mathbf{s}$*-tuples on* $\mathrm{SVars}(P)$.

There are two kinds of universal spanners: the *universal spanner* and the *universal hierarchical spanners*.

Definition 3.14: *Given a finite set of span variables* $Y \subseteq \mathrm{SVars}$*, the universal spanner* Υ_Y *over* Y *is the unique document spanner* P *such that* $\mathrm{SVars}(P) = Y$ *and* P *is total on every string* $\mathbf{s} \in \Sigma^*$.

Definition 3.15: *Given a finite set of span variables* $Y \subseteq \mathrm{SVars}$*, the universal hierarchical spanner* Υ_Y^H *over* Y *is the unique document spanner* P *such that* $\mathrm{SVars}(P) = Y$ *and* P *is hierarchically total on every string* $\mathbf{s} \in \Sigma^*$.

The next subsection introduces the basic ways of representing spanners, as described in [Fagin et al., 2015].

3.3.2 Spanner Representations

We saw, in Subsection 3.1.4, that the operations used by SystemT and AQL to extract spans relations from text are regular expression matching and dictionary matching. In the remainder of this dissertation I focus on regular expressions. The regular expressions of AQL can be seen as usual regular expressions enriched with *capture variables*, that are precisely the span variables constituting the span relations of Definition 3.4. In [Fagin et al., 2015], three types of representations are described, that are able to model this kind of modified regular expressions. They are:

- *regex formulas;*
- *variable stack automata;*
- *variable set automata.*

These models are also called *primitive spanner representations*.

3.3.2.1 Regex Formulas

In order to define a regex formula, let us introduce the concept of *variable regex.*

Definition 3.16: *A variable regex is a regular expression with capture variables that extends usual regular expressions in the following way:*

$$\gamma := \emptyset \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \mid x\{\gamma\} \tag{3.3}$$

where:

- $x \in SVars$*;*
- $x\{\gamma\}$ *means that we assign to* x *the span defined by the string matched by the evaluation of* γ *.*

The set of span variables appearing in a variable regex γ is called $\mathrm{SVars}(\gamma)$. Evaluating a variable regex on a string produces a parse tree over the alphabet $\Lambda = \Sigma \cup \mathrm{SVars} \cup \{\epsilon, \vee, \cdot, *\}$. A valid parse tree for a variable regex γ is called a γ-parse. We accept a variable regex expression only if its parses have exactly one occurrence of each of the variables appearing in it, otherwise the variable assignment would remain unclear. Such an expression is referred to as *functional.*

Definition 3.17: *A variable regex* γ *is functional if* $\forall \mathbf{s} \in \Sigma^*$, $\forall \gamma$*-parse* t *for* $\mathbf{s}$, $\forall x \in \mathrm{SVars}(\gamma)$, x *occurs exactly one time in* t*.*

Definition 3.18: *A regex formula is a functional variable regex.*

The spanner represented by a regex formula γ may be denoted as $[\![\gamma]\!]$. We have that $\mathrm{SVars}([\![\gamma]\!]) = \mathrm{SVars}(\gamma)$, and the span relation $[\![\gamma]\!](\mathbf{s})$ is the set $\{\mu^p \mid p \text{ is a } \gamma\text{-parse for } \mathbf{s}\}$, where μ^p is a tuple defined by a γ-parse p. The class of regex formulas is referred to as RGX.

Example 3.4: *Consider the variable regex* γ *defined by:*

$$\Sigma^* \cdot z\left\{x\{\gamma_{\text{action}}\} \cdot \Sigma^+ \cdot y\{\gamma_{\text{title}}\}\right\} \cdot \Sigma^* \tag{3.4}$$

where $\gamma_{\text{action}} = (\mathrm{w} \cdot \mathrm{a} \cdot \mathrm{t} \cdot \mathrm{c} \cdot \mathrm{h} \cdot \mathrm{e} \cdot \mathrm{d}) \vee (\mathrm{s} \cdot \mathrm{a} \cdot \mathrm{w})$ *and* $\gamma_{\text{title}} = \text{"} \cdot (\mathrm{A} \vee \ldots \vee \mathrm{Z}) \cdot (\mathrm{a} \vee \ldots \vee \mathrm{z})^* \cdot (\_ \cdot (\mathrm{A} \vee \ldots \vee \mathrm{Z}) \cdot (\mathrm{a} \vee \ldots \vee \mathrm{z})^*)^* \cdot \text{"}$*. Notice that* $\mathrm{SVars}(\gamma) = \{x, y, z\}$*. Figure 3.4 shows a* γ*-parse* p *for string* $\mathbf{s}$ *of Figure 3.3. In this parse, each variable contained in* $\mathrm{SVars}(\gamma)$ *occurs exactly once. It is easy to verify that this holds for every* γ*-parse. Hence,* γ *is functional. We have that* $\mu^p(x) = [3, 10\rangle$*,* $\mu^p(y) = [11, 23\rangle$ *and* $\mu^p(z) = [3, 23\rangle$*, thus* μ^p *is the* $\mathbf{s}$*-tuple* μ_1 *from Figure 3.3. Finally, considering the span relation* $P(\mathbf{s})$ *(Figure 3.3), we have that* $[\![\gamma]\!](\mathbf{s}) = P(\mathbf{s})$*.*

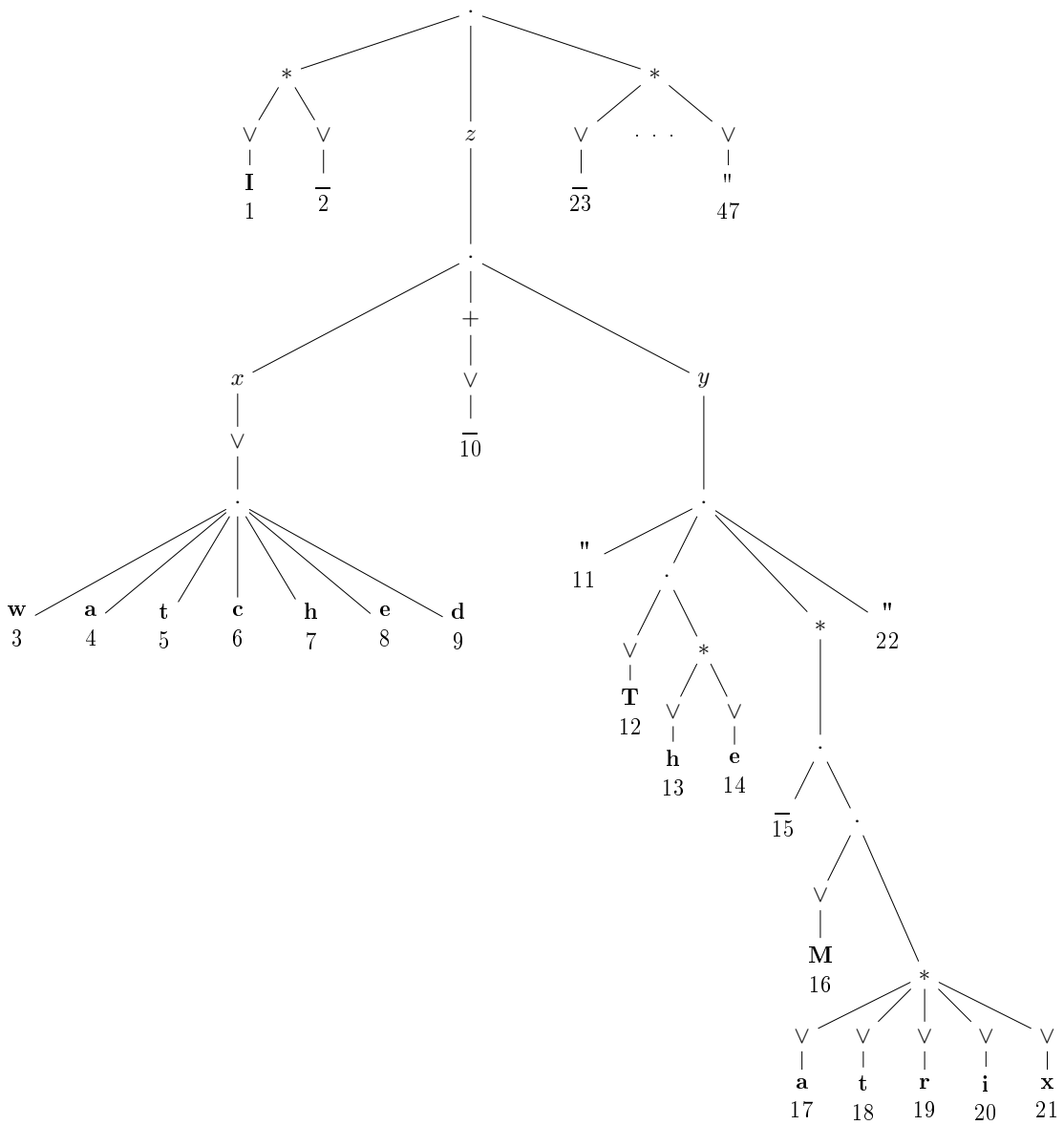


Figure 3.4 – A γ-parse p for string **s** (see Figure 3.3).

We can be more precise on the characterization of a regex formula. In order to do this, let us introduce the notion of *syntactically Y-functional* variable regex, Y being a finite subset of SVars.

Definition 3.19: *Given a variable regex γ, γ is syntactically Y-functional if at least one of following hold:*

- $\gamma = \emptyset$;
- $\gamma = \epsilon$ *or* $\gamma = \sigma$ *with* $\sigma \in \Sigma$ *and* $Y = \emptyset$;
- $\gamma = \gamma_1 \vee \gamma_2$, *where* γ_1, γ_2 *are syntactically Y-functional variable regexes;*
- $\gamma = \gamma_1 \cdot \gamma_2$, *where* γ_1, γ_2 *are variable regexes, and there exists a subset* Y_1 *of* Y *such that* γ_1 *is syntactically* Y_1*-functional and* γ_2 *is syntactically* Y_2*-functional, for* $Y_2 = Y \setminus Y_1$;
- $\gamma = (\gamma_1)^*$, *where* γ_1 *is a variable regex with no variable assignments, and* $Y = \emptyset$;
- $\gamma = x\{\gamma_1\}$, *where* $x \in Y$ *and* γ_1 *is a syntactically* $(Y \setminus \{x\})$*-functional variable regex.*

It is easy to prove, for a variable regex γ, the next proposition (by induction on the structure of γ).

Proposition 3.1: *Given a variable regex γ, γ is functional if and only if it is syntactically* $\text{SVars}(\gamma)$*-functional. Moreover, whether γ is syntactically* $\text{SVars}(\gamma)$*-functional can be tested in polynomial time.*

This characterization of regex formulas will be useful in Chapter 4, where I will present a formal model of a runtime system for the core of AQL, and I will show its applicability.

3.3.2.2 Variable Stack Automata

Variable stack automata (vstk-automata for short) are representations of document spanners by means of modified NFAs. Basically, a NFA is augmented with a *stack of span variables*. A variable is pushed on the stack when its corresponding span is opened, and popped when it is closed. The formal definition of a vstk-automaton follows.

Definition 3.20: *A Variable Stack Automaton is a tuple (Q, q_0, q_f, δ), where:*

- Q *is a finite set of states;*
- $q_0 \in Q$ *is the initial state;*
- $q_f \in Q$ *is the accepting state;*

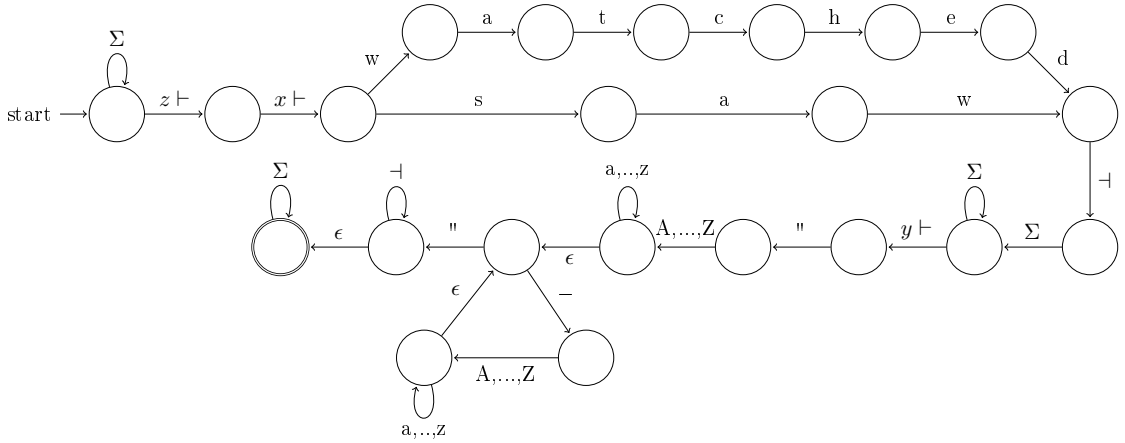


Figure 3.5 – A vstk-automaton A with $[\![A]\!] = [\![\gamma]\!]$, where γ is the regex formula 3.4 (see Example 3.4).

- *δ is a finite transition relation, containing triples of the forms (q, σ, q'), (q, ϵ, q'), $(q, x\vdash, q')$, $(q, \dashv, q')$, where:*
  - $q, q' \in Q$;
  - $\sigma \in \Sigma$;
  - $x \in \text{SVars}$;
  - $\vdash$ *is the push symbol;*
  - $\dashv$ *is the pop symbol.*

Notice that we don't need to specify which variable we want to pop, as it is naturally the last that was pushed onto the stack. Given a vstk-automaton A, the set of variables that appear in its transitions is denoted as $\text{SVars}(A)$.

Example 3.5: *Figure 3.5 shows a vstk-automaton A. Each circle represents a state. The double circle is the accepting state. A label a on an edge from state q to state q' represents the transition $\left(q, a, q'\right)$. A sequence $\sigma_1, \ldots, \sigma_k$ on an edge from q to q' is a shorthand for the k transitions $\left(q, \sigma_1, q'\right), \ldots, \left(q, \sigma_k, q'\right)$. Assuming $\Sigma = \{\sigma_1, \ldots, \sigma_l\}$, the label Σ is used in place of $\sigma_1, \ldots, \sigma_l$.*

Next, I report the definitions of a *configuration* and of a *run* of a vstk-automaton, which define its semantics.

Definition 3.21: *Given a string $\mathbf{s}$ with length $|\mathbf{s}| = n$ and a vstk-automaton A , a configuration of A is a tuple $c = (q, \overrightarrow{v}, Y, i)$, where:*

- $q \in Q$ *is the current state;*
- $\overrightarrow{v}$ *is the current variable stack;*
- $Y \subseteq \mathrm{SVars}(A)$ *is the set of available variables (those not already pushed on the stack);*
- $i \in \{1, ..., n+1\}$ *is the position of the next character to be read in* $\mathbf{s}$.

Here, as for regex formulas, we want the variable assignment to be clear, so once a variable is pushed on the stack, it is removed from the set of available variables, thus it can be pushed only once. For the rest, a run of a vstk-automaton is similar to those of ordinary NFAs.

Definition 3.22: *Given a string* s *and a vstk-automaton* A*, a run* ρ *of* A *on* s *is a sequence of configurations* $c_0, ..., c_m$ *such that:*

- $c_0 = (q_0, \epsilon, \mathrm{SVars}(A), 1)$;
- $\forall j \in \{0, ..., m-1\}$, *for* c_j=$(q_j, \overrightarrow{v_j}, Y_j, i_j)$, c_{j+1}=$(q_{j+1}, \overrightarrow{v_{j+1}}, Y_{j+1}, i_{j+1})$ *one of the following holds:*
  - $\overrightarrow{v_{j+1}} = \overrightarrow{v_j}$, $Y_{j+1} = Y_j$ *and either:*
    - $i_{j+1} = i_j + 1$, $\left(q_j, s_{i_j}, q_{j+1}\right) \in \delta$;
    - $i_{j+1} = i_j$, $(q_j, \epsilon, q_{j+1}) \in \delta$.
  - $i_{j+1} = i_j$ *and for some* $x \in \mathrm{SVars}(A)$ *either:*
    - $\overrightarrow{v_{j+1}} = \overrightarrow{v_j} \cdot x$, $x \in Y_j$, $Y_{j+1} = Y_j \setminus \{x\}$, $(q_j, x\vdash, q_{j+1}) \in \delta$ *(*x *is pushed on the stack);*
    - $\overrightarrow{v_j} = \overrightarrow{v_{j+1}} \cdot x$, $Y_{j+1} = Y_j$, $(q_j, \dashv, q_{j+1}) \in \delta$ *(the variable on top of the stack is popped).*

Definition 3.23: *Given a string* $\mathbf{s}$ *with length* $|\mathbf{s}| = n$ *and a vstk-automaton* A*, a run* $\rho = c_0, ..., c_m$ *of* A *on* $\mathbf{s}$ *is accepting if* $c_m = (q_f, \epsilon, \emptyset, n+1)$.

The set of all possible accepting runs of a vstk-automaton A on a string $\mathbf{s}$ is denoted as $\mathrm{ARuns}(A, \mathbf{s})$. The spanner represented by A may be referred to as $[\![A]\!]$. We have that $\mathrm{SVars}([\![A]\!]) = \mathrm{SVars}(A)$, and the span relation $[\![A]\!](\mathbf{s})$ is the set $\{\mu^\rho \mid \rho \in \mathrm{ARuns}(A, \mathbf{s})\}$, where μ^ρ is a tuple defined by a run ρ. In particular, for every variable $x \in \mathrm{SVars}(A)$, $\mu^\rho(x)$ is the span $[i_b, i_e\rangle$, where:

- c_b=$(q_b, \overrightarrow{v_b}, Y_b, i_b)$ is the unique configuration of ρ where x appears in the stack for the first time;

- c_e=$(q_e, \overrightarrow{v_e}, Y_e, i_e)$ is the unique configuration of ρ where x appears in the stack for the last time.

The class of variable stack automata is called $\mathrm{VA}_{\mathrm{stk}}$.

Example 3.6: *Consider the regex formula γ from Example 3.4 and the vstk-automaton A from Figure 3.5. We have that* $\mathrm{SVars}(A) = \{x, y, z\}$. *The reader can verify that* $[\![\gamma]\!] = [\![A]\!]$.

3.3.2.3 Variable Set Automata

Variable Set Automata (vset-automata for short) are another model for representing document spanners that is based on NFAs. Vset-automata are defined in a very similar way to that of vstk-automata. The main differences are:

- the stack of span variables is replaced by a *set;*
- no order is defined on the variables in the set, so when we want to remove a variable from it, we need to specify which one.

The following is the formal definition of a vset-automaton.

Definition 3.24: *A variable set automaton is a tuple (Q, q_0, q_f, δ), where:*

- *Q, q_0 and q_f are defined as in Definition 3.20;*
- *δ is the same as in Definition 3.20, except that it has triples of the form $(q, \dashv x, q')$, with $x \in \mathrm{SVars}$, instead of those of the form $(q, \dashv, q')$.*

We also need to slightly modify the definitions of configuration and run with respect to those of a vstk-automaton.

Definition 3.25: *Given a string $\mathbf{s}$ with length $|\mathbf{s}| = n$ and a vset-automaton A , a configuration of A is a tuple $c = (q, V, Y, i)$, where:*

- *q, i are defined as in Definition 3.21;*
- *$V \subseteq \mathrm{SVars}(A)$ is the active variable set;*
- *$Y \subseteq \mathrm{SVars}(A)$ is the set of available variables (those not already inserted in the set).*

Definition 3.26: *Given a string $\mathbf{s}$ and a vset-automaton A, a run ρ of A on $\mathbf{s}$ is a sequence of configurations $c_0, ..., c_m$ such that:*

- *$c_0 = (q_0, \emptyset, \mathrm{SVars}(A), 1)$;*

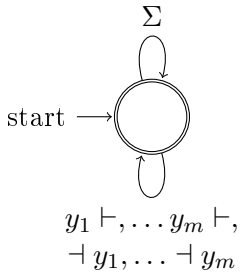


Figure 3.6 – A vset-automaton B with $[\![B]\!] = \Upsilon_Y$, for $Y = \{y_1, \ldots, y_m\}$ (Figure 2b , [Fagin et al., 2015]).

- $\forall j \in \{0, ..., m-1\}$, *for* c_j=(q_j, V_j, Y_j, i_j), c_{j+1}=$(q_{j+1}, V_{j+1}, Y_{j+1}, i_{j+1})$ *one of the following holds:*
  - $V_{j+1} = V_j$, $Y_{j+1} = Y_j$ *and either:*
    - $i_{j+1} = i_j + 1$, $\left(q_j, s_{i_j}, q_{j+1}\right) \in \delta$;
    - $i_{j+1} = i_j$, $(q_j, \epsilon, q_{j+1}) \in \delta$.
  - $i_{j+1} = i_j$ *and for some* $x \in \mathrm{SVars}(A)$ *either:*
    - $x \in Y_j$, $V_{j+1} = V_j \cup \{x\}$, $Y_{j+1} = Y_j \setminus \{x\}$, $(q_j, x\vdash, q_{j+1}) \in \delta$ *(x is inserted into the active set);*
    - $x \in V_j$, $V_{j+1} = V_j \setminus \{x\}$, $Y_{j+1} = Y_j$, $(q_j, \dashv x, q_{j+1}) \in \delta$ *(x is removed from the active set).*

Definition 3.27: *Given a string with length* $|\mathbf{s}| = n$ *and a vset-automaton* A*, a run* $\rho = c_0, ..., c_m$ *of* A *on* $\mathbf{s}$ *is accepting if* $c_m = (q_f, \emptyset, \emptyset, n+1)$.

Given a string $\mathbf{s}$ and a vset-automaton A, $\mathrm{ARuns}(A, \mathbf{s})$ and $[\![A]\!]$ are defined as for vstk-automata. The class of variable set automata is called $\mathrm{VA}_{\mathrm{set}}$.

Example 3.7: *Figure 3.6 shows a vset-automaton* B*, with* $\mathrm{SVars}(B) = Y$*, where* $Y = \{y_1, \ldots, y_m\}$*. We have that* $[\![B]\!] = \Upsilon_Y$*. This example shows that vset-automata can express spanners that regex formulas and vstk-automata cannot. (Example 3.12, [Fagin et al., 2015]).*

The next subsection describes a series of operators with can be used to combine spanners.

3.3.3 Algebras of Spanners

Besides mere span extraction, AQL offers the capability to combine, transform and filter extracted tuples by using a series of operators. Here, I list the algebraic operators for

spanners, described in [Fagin et al., 2015], that are considered to capture the core algebraic operations of AQL. They are:

- *Union* ($\cup$);
- *Projection* (π);
- *Natural Join* ($\bowtie$);
- *String Selection* (ς).

A finite set of spanner operators forms a *spanner algebra*. In the following, I look at the definitions of the listed spanner operators.

3.3.3.1 Union

Before giving the definition of the union operator, we need to introduce the concept of *union-compatible spanners.*

Definition 3.28: *Given two document spanners P_1 and P_2, they are union-compatible if and only if* $\mathrm{SVars}(P_1) = \mathrm{SVars}(P_2)$.

The definition of union of two spanners is as follows.

Definition 3.29: *Given two union-compatible document spanners P_1 and P_2, their union is the spanner denoted as $P_1 \cup P_2$, for which we have that:*

- $\mathrm{SVars}(P_1 \cup P_2) = \mathrm{SVars}(P_1)$;
- *given a string* $\mathbf{s}$, $(P_1 \cup P_2)(\mathbf{s}) = P_1(\mathbf{s}) \cup P_2(\mathbf{s})$.

3.3.3.2 Projection

Definition 3.30: *Given a document spanner P and a set of span variables $Y \subseteq \mathrm{SVars}(P)$, the projection of P over Y is the spanner denoted as $\pi_Y P$, satisfying* $\mathrm{SVars}(\pi_Y P) = Y$, *where, for every string* $\mathbf{s}$, $\pi_Y P(\mathbf{s})$ *is obtained by reducing the domain of each* $\mathbf{s}$*-tuple belonging to* $P(\mathbf{s})$ *to* Y.

3.3.3.3 Natural Join

Definition 3.31: *Given two document spanners P_1 and P_2, their natural join is the spanner denoted as $P_1 \bowtie P_2$, for which we have that:*

- $\mathrm{SVars}(P_1 \bowtie P_2) = \mathrm{SVars}(P_1) \cup \mathrm{SVars}(P_2)$;

- *given a string* $\mathbf{s}$, $(P_1 \bowtie P_2)(\mathbf{s})$ *consists of every* $\mathbf{s}$*-tuple* μ *for which there exists a pair of tuples* $\mu_1 \in P_1(\mathbf{s})$,$\mu_2 \in P_2(\mathbf{s})$, *such that* μ_1 *and* μ_2 *assign the same spans to the span variables they have in common. Note that this implies that* μ_1, μ_2 *agree on variables that are common to* P_1, P_2*:* $\forall x \in \mathrm{SVars}(P_1) \cap \mathrm{SVars}(P_2)$, $\mu_1(x) = \mu_2(x)$.

Example 3.8: *Consider again the regex formula* γ *from Example 3.4. The reader can verify that* γ *can be expressed as:*

$$(\Sigma^* \cdot x\{\gamma_{\text{action}}\} \cdot \Sigma^*) \bowtie (\Sigma^* \cdot y\{\gamma_{\text{title}}\} \cdot \Sigma^*) \bowtie \left(\Sigma^* \cdot z\left\{x\left\{\Sigma^+\right\} \cdot \Sigma^+ \cdot y\left\{\Sigma^+\right\}\right\} \cdot \Sigma^*\right) \quad (3.5)$$

3.3.3.4 String Selection

Definition 3.32: *Given a document spanner* P *and a* k*-ary string relation* R*, the string selection operation according to* R *is denoted as* ς^R*, and is parametrized by* $x_1, ..., x_k \in \mathrm{SVars}(P)$. *We have that, given a string* $\mathbf{s}$ *and a spanner* $P' := \varsigma^R_{x_1,...,x_k} P$, $P'(\mathbf{s})$ *consists of all the* $\mathbf{s}$*-tuples* $\mu \in P(\mathbf{s})$ *such that* $\left(\mathbf{s}_{\mu(x_1)}, ..., \mathbf{s}_{\mu(x_k)}\right) \in R$.

In the remainder of this dissertation, the only string selection operator considered is $\varsigma^=_{x,y}$ which, given a spanner P, restricts $P(\mathbf{s})$ to those $\mathbf{s}$-tuples that satisfy $\mathbf{s}_{\mu(x)} = \mathbf{s}_{\mu(y)}$.

At this point, we have defined three models to represent spanners and a series of operators to combine spanners. The question we might want to ask now is: what is the most convenient way to represent an AQL query containing only core operations? As we will see, this question is not trivial, because if we try and combine the elements we have seen in different ways, we might obtain different classes of spanners. To answer our question, we will formally define the class of spanners that captures the core of AQL, i.e. the *core spanners,* and we will identify those classes of spanner representations that can model it, by reasoning on the relative expressive power of some relevant spanner classes. Before beginning, let us introduce some additional notation. Given a generic class of spanner representations SR, the set of all the spanners that can be represented by SR is denoted as $[\![\mathrm{SR}]\!]$. Formally, we have that $[\![\mathrm{SR}]\!] = \{[\![r]\!] \mid r \in \mathrm{SR}\}$. Let O be a spanner algebra. Let us denote by SR^O the closure of SR under O, that is: the class of spanner representations obtained by applying (compositions of) operators contained in O to the representations in SR. The corresponding set of spanners is referred to as $[\![\mathrm{SR}^O]\!]$. We can now start looking for an answer to our current question. In the following, many mathematical statements will be formulated, without proving them. The reader can refer to [Fagin et al., 2015] for the missing proofs, in particular to Chapter 4.

Proposition 3.2: *The following hold:*

1. *every document spanner represented in the classes* RGX *and* $\mathrm{VA}_{\mathrm{stk}}$ *is hierarchical, that is:* $[\![\mathrm{RGX}]\!], [\![\mathrm{VA}_{\mathrm{stk}}]\!] \subseteq \mathbf{HS}$*;*

2. *there exist some spanners represented in* $\mathrm{VA}_{\mathrm{set}}$ *that are not hierarchical:* $[\![\mathrm{VA}_{set}]\!] \nsubseteq \mathbf{HS}$ *(see, e.g., Example 3.7);*

3. *the operators* $\cup$, π, ς^R *preserve the property of being hierarchical, while* $\bowtie$ *does not, thus we have that:*

   (a) *given a class of spanner representations* SR, $[\![\mathrm{SR}]\!] \subseteq \mathbf{HS} \Rightarrow \left[\!\!\left[\mathrm{SR}^{\{\cup,\pi,\varsigma^R\}}\right]\!\!\right] \subseteq \mathbf{HS}$*;*

   (b) *there exist two hierarchical spanners* P_1, P_2 *such that* $P_1 \bowtie P_2 \notin \mathbf{HS}$.

In the next Subsection, I discuss the class of *regular spanners*, which plays a central role in the construction of the class of core spanners.

3.3.4 Regular Spanners

A regular spanner is defined as follows.

Definition 3.33: *A spanner is regular if it can be defined by a vset-automaton.*

Let us see how regular spanners are related to the other basic spanner classes. A preliminary result is that regex formulas and vstk-automata have the same expressive power.

Theorem 3.1: $[\![\mathrm{RGX}]\!] = [\![\mathrm{VA}_{\mathrm{stk}}]\!]$.

It turns out that the spanners expressed by representations in $\mathrm{VA}_{\mathrm{stk}}$ (RGX) are exactly those that are both regular and hierarchical.

Theorem 3.2: $[\![\mathrm{VA}_{\mathrm{stk}}]\!] = [\![\mathrm{VA}_{\mathrm{set}}]\!] \cap \mathbf{HS}$.

For what concerns the algebraic operators presented in Subsection 3.3.3, it can be shown that union, projection and natural join don't increase the expressive power of regular spanners.

Theorem 3.3: $\left[\!\!\left[\mathrm{VA}_{\mathrm{set}}{}^{\{\cup,\pi,\bowtie\}}\right]\!\!\right] = [\![\mathrm{VA}_{\mathrm{set}}]\!]$.

On the other hand, applying the same operators to the spanner representations in $\mathrm{VA}_{\mathrm{stk}}$ results in a class equivalent to regular spanners.

Theorem 3.4: $\left[\!\!\left[\mathrm{VA}_{\mathrm{stk}}{}^{\{\cup,\pi,\bowtie\}}\right]\!\!\right] = [\![\mathrm{VA}_{\mathrm{set}}]\!]$.

Let us now look at which string relations can be simulated by regular spanners, starting with the concept of *selectable string relation*.

Definition 3.34: *Given a string relation R and a class of spanners C, R is selectable by C if for every document spanner $P \in C$ and for every $\overrightarrow{x} = x_1, ..., x_k$, $x_i \in \mathrm{SVars}(P)$, we have that $\varsigma^R_{\overrightarrow{x}} P \in C$.*

Let us introduce the concept of *restricted universal spanner*.

Definition 3.35: *Given a string relation R and a sequence of variables $\overrightarrow{x} = x_1, ..., x_k$ with their corresponding set $X = \{x_1, ..., x_k\}$, the R-restricted universal spanner over $\overrightarrow{x}$ is $\Upsilon^R_{\overrightarrow{x}} := \varsigma^R_{\overrightarrow{x}} \Upsilon_X$.*

Selectability of a string relation R by a class of spanners C corresponds to the presence in C of all the possible R-restricted universal spanners, under some conditions.

Proposition 3.3: *Given a string relation R and a class of spanners C containing all the possible universal spanners and closed under natural join, R is selectable by C if and only if, for every $\overrightarrow{x} = x_1, ..., x_k \in \mathrm{SVars}^k$, $\Upsilon^R_{\overrightarrow{x}} \in C$.*

In the case of regular spanners, the class of string relations that they can select is exactly REC.

Theorem 3.5: *The class of string relations selectable by $[\![\mathrm{VA}_{\mathrm{set}}]\!]$ is* REC.

The relation "=" is not in REC, thus it is not selectable by regular spanners, nonetheless it is important for selection predicates in AQL. Therefore, regular spanners are incapable of modeling its core. In the following subsection, I describe the class of core spanners, that, as shown in [Fagin et al., 2015], models the core of AQL.

3.3.5 Core Spanners

An expression in the core of AQL belongs to $\mathrm{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}$. Consequently, a core spanner is defined as follows.

Definition 3.36: *A core spanner is a document spanner belonging to $\left[\!\left[\mathrm{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right]$.*

Thanks to Theorems 3.1, 3.3 and 3.4 we can easily state the next theorem.

Theorem 3.6: $\left[\!\left[\mathrm{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] = \left[\!\left[\mathrm{VA}_{\mathrm{st\,k}}{}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] = \left[\!\left[\mathrm{VA}_{\mathrm{set}}{}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right]$.

This shows that core spanners can be reduced to regular spanners extended with the algebra $\{\cup, \pi, \bowtie, \varsigma^=\}$. But the following lemma tells us that an algebra with fewer operators is also sufficient.

Lemma 3.1: $\left[\!\left[\mathrm{VA}_{\mathrm{set}}{}^{\{\cup,\pi,\bowtie,\zeta^{=}\}}\right]\!\right] = \left[\!\left[\mathrm{VA}_{\mathrm{set}}{}^{\{\pi,\zeta^{=}\}}\right]\!\right]$.

Another lemma, known as the *core simplification lemma*, gives an even simpler way of representing core spanners.

Lemma 3.2: (**Core Simplification Lemma**) *Every core spanner can be defined by an expression of the form*

$$\pi_V S A \tag{3.6}$$

where:

- A *is a vset-automaton;*
- $V \subseteq \mathrm{SVars}(A)$*;*
- S *is a sequence of string selections* $\zeta^{=}_{x,y}$*, for* $x, y \in \mathrm{SVars}(A)$.

For what concerns which string relations can be simulated by core spanners, the next definition presents three string relations of relevance.

Definition 3.37: *Given two strings* $\mathbf{s}, \mathbf{t} \in \Sigma^*$*:*

- $\mathbf{s} \sqsubseteq \mathbf{t}$ *if* $\mathbf{s}$ *is a (consecutive) substring of* t *(i.e.* $\mathbf{s} = \mathbf{t}_{[i,j\rangle}$*);*
- $\mathbf{s} \sqsubseteq_{prf} \mathbf{t}$ *if* $\mathbf{s}$ *is a prefix of* t *(i.e.* $\mathbf{s} = \mathbf{t}_{[1,j\rangle}$*);*
- $\mathbf{s} \sqsubseteq_{sfx} \mathbf{t}$*if* $\mathbf{s}$ *is a suffix of* t *(i.e.* $\mathbf{s} = \mathbf{t}_{[i,|\mathbf{t}|+1\rangle}$*).*

Proposition 3.4: *All the string relations in* REC*,* $\sqsubseteq$*,* $\sqsubseteq_{prf}$ *and* $\sqsubseteq_{sfx}$ *are selectable by the core spanners.*

3.3.6 Difference

Besides the operators introduced so far, AQL also supports difference, which is defined as follows.

Definition 3.38: *Given two union compatible document spanners* P_1 *and* P_2*, their difference is the spanner* $P_1 \setminus P_2$*, for which we have:*

- $\mathrm{SVars}(P_1 \setminus P_2) = \mathrm{SVars}(P_1)$*;*
- *given a string* $\mathbf{s}$*,* $(P_1 \setminus P_2)(\mathbf{s}) = (P_1) \setminus (P_2)(\mathbf{s})$.

It can be shown that regular spanners are closed under difference.

Theorem 3.7: $\left[\!\left[\mathrm{VA}_{\mathrm{set}}{}^{\{\setminus\}}\right]\!\right] = [\![\mathrm{VA}_{\mathrm{set}}]\!]$.

Despite the result of of Theorem 3.7, core spanners are not closed under difference. This is why this operator has not been considered in this discussion.

Theorem 3.8: $\left[\!\left[\mathrm{RGX}^{\{\cup,\pi,\bowtie,\varsigma^{=}\}}\right]\!\right] \subsetneq \left[\!\left[\mathrm{RGX}^{\{\cup,\pi,\bowtie,\varsigma^{=},\setminus\}}\right]\!\right]$.

Chapter 4

A Runtime System for the Core of AQL

In this chapter I describe the model of a new runtime system for the core fragment of AQL. It is based on a modified version of vset-automata, the *extended vset-automata (*or *eVset-automata).* In particular, the system works with a special kind of eVset-automata: *well-behaved eVset-automata.* The structure of the chapter is as follows. In Section 4.1 I define eVset-automata. Then, in Section 4.2, I describe well-behaved eVset-automata and their properties, with particular attention to their ability to support the operators presented in Subsection 3.3.3: I show construction methods to simulate projection, union and natural join with well-behaved eVset-automata. [Fagin et al., 2015] contains descriptions of construction methods for the same purpose, but for vset-automata. However, those methods have exponential space complexity, while the methods presented in this chapter have polynomial *time* complexity. Finally, I show how well-behaved eVset-automata can be used in a class of spanner representations that models the class of core spanners, with some simplifications, that are similar to those obtained in Subsection 3.3.5 of the previous chapter.

4.1 Extended Vset-automata

Before I define extended vset-automata formally, I introduce some basic concepts.

Definition 4.1: *Given a set $X \subseteq SVars$, we define:*

- $\text{SVOps}^{\vdash}(X) = \{x \vdash \mid x \in X\}$;
- $\text{SVOps}^{\dashv}(X) = \{\dashv x \mid x \in X\}$;
- $\text{SVOps}(X) = \text{SVOps}^{\vdash}(X) \cup \text{SVOps}^{\dashv}(X)$.

With this definition in mind, we are ready to describe an extended vset-automaton.

Definition 4.2: *An extended* vset-automaton *(or* eVset-automaton*) is a tuple* (Q, q_0, q_f, δ)*, where:*

- Q *is a finite set of states;*
- $q_0 \in Q$ *is the initial state;*
- $q_f \in Q$ *is the accepting state;*
- $\delta = \delta^{\text{char}} \cup \delta^{\text{op}}$ *is a finite transition relation consisting of triples, where:*
  - $\delta^{\text{char}} = \{(q, \sigma, q') \mid q, q' \in Q, \sigma \in \Sigma\}$*, whose elements are called character transitions;*
  - $\delta^{\text{op}} = \{(q, S, q') \mid q, q' \in Q, S \subseteq \text{SVOps}(\text{SVars}), S \text{ finite}\}$*, whose elements are called operation transitions .*

In a transition (q, a, q') with label a from state q to state q', q is called the *source state*, while q' is called the *destination state*. With a slight abuse of notation, I denote by $\text{SVOps}(A)$ the set of variable operations appearing in the transitions of an eVset-automaton A. $\text{SVars}(A)$ is defined as for a usual vset-automaton. In order to make the semantics of an eVset-automaton clear, we need the definitions of a *configuration* and a *run* of an eVset-automaton.

Definition 4.3: *Given a string* $\mathbf{s}$*,* $|\mathbf{s}| = n$*, and an eVset-automaton* $A = (Q, q_0, q_f, \delta)$*, a configuration of* A *is a tuple* $c = (q, V, Y, i)$*, where:*

- $q \in Q$ *is the current state;*
- $V \subseteq \text{SVars}(A)$ *is the active variable set;*
- $Y \subseteq \text{SVars}(A)$ *is the set of available variables;*
- i *is an index belonging to* $\{1, \ldots, n+1\}$*.*

Definition 4.4: *Given a string* $\mathbf{s} = s_1, \ldots, s_n$ *and an eVset-automaton* $A = (Q, q_0, q_f, \delta)$*, a run* ρ *of* A *on* $\mathbf{s}$ *is a sequence* $c_0, \ldots, c_m$ *of configurations, where:*

- $c_0 = (q_0, \emptyset, \text{SVars}(A), 1)$*;*
- *for* $j = 0, \ldots, m-1$ *one of the following holds for* $c_j = (q_j, V_j, Y_j, i_j)$ *and* $c_{j+1} = (q_{j+1}, V_{j+1}, Y_{j+1}, i_{j+1})$*:*

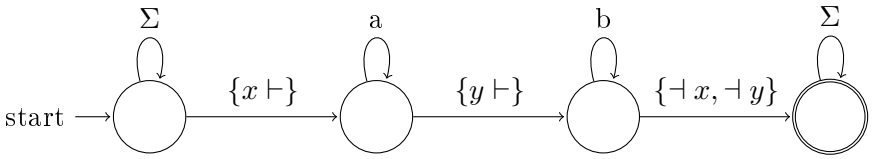


Figure 4.1 – An eVset-automaton A.

- $V_{j+1} = V_j$, $Y_{j+1} = Y_j$, $i_{j+1} = i_j + 1$ *and* $(q_j, s_j, q_{j+1}) \in \delta^{\text{char}}$;
- $i_{j+1} = i_j$, *and for some* $S \subseteq \text{SVOps}(A)$ *we have :*
  - *for each* $x \in \text{SVars}(A)$*:*
    - $x \vdash \in S \Rightarrow x \in Y_j$;
    - $\dashv x \in S \Rightarrow (x \vdash \in S \vee x \in V_j)$;
  - $V_{j+1} = (V_j \cup \{x \mid x \vdash \in S\}) \setminus \{x \mid \dashv x \in S\}$;
  - $Y_{j+1} = Y_j \setminus \{x \mid x \vdash \in S\}$;
  - $(q_j, S, q_{j+1}) \in \delta^{\text{op}}$.

ρ *is accepting if* $c_m = (q_f, \emptyset, \emptyset, n+1)$.

$\text{ARuns}(A, \mathbf{s})$ and $[\![A]\!]$, for an eVset-automaton A and a string $\mathbf{s}$, are defined in similar ways to those of usual vset-automata. This new kind of vset-automata allows to perform an arbitrary number of variable operations in one transition. The operations in a transition are to be performed in a given order. The exact order is not very important, except for the fact that, in a valid run, the insertion and removal of a variable from the active variable set need to happen in the correct order (i.e. insertions first).

Example 4.1: *The automaton A, shown in Figure 4.1, is an extended vset-automaton. The operation transitions are those with a label of the form* $\{o_1, \ldots, o_n\}$*, with* $o_i \in \text{SVOps}$ (SVars).

To understand the motivation for using eVset-automata instead of plain vset-automata (that will be clear in the next section) we need to introduce some additional useful concepts.

Definition 4.5: *Given a transition* $t \in \delta$ *in an eVset-automaton A, and an ordering* φ *on the elements of* $\text{SVOps}(A)$*, we define*

- $\text{Ops}(t)$ *as either:*
  - *if* $t = (q, S, q') \in \delta^{\text{op}}$*, the set S;*
  - *if* $t \in \delta^{\text{char}}$*, the empty set.*

- $\mathrm{LOps}_{\varphi}(t)$ *as either:*
  - *if* $t = (q, S, q') \in \delta^{\mathrm{op}}$*, the list* $o_1, \ldots, o_{|S|}$ *of the operations belonging to* S*, ordered according to* φ*;*
  - *if* $t \in \delta^{\mathrm{char}}$*, the empty list.*

Definition 4.6: *Given an extended vset-automaton* $A = (Q, q_0, q_f, \delta)$ *and a pair of states* $q, q' \in Q$*, a path* p *between* q *and* q' *in* A *is a sequence of transitions* $t_1, \ldots, t_n \in \delta$*, such that:*

- *the source state of* t_1 *is* q*;*
- *the destination state of* t_n *is* q'*;*
- *for every pair* t_i, t_{i+1}*, the destination state of* t_i *equals the source state of* t_{i+1}*.*

We also write $p_q^{q'}$*. We refer to the set of paths in* A *as* $\mathrm{Paths}(A)$*.*

A path in an eVset-automaton is closely related to a run, as formalized by the next definition.

Definition 4.7: *Given a string* $\mathbf{s}$*, an eVset-automaton* $A = (Q, q_0, q_f, \delta)$*, a run* $\rho = c_0, \ldots, c_m$ *of* A *on* $\mathbf{s}$ *and a path* $p = t_0, \ldots, t_{m-1}$ *in* A*, we say that* p *supports* ρ *if, for every pair* c_i*,* c_{i+1} *of configurations,* c_{i+1} *is obtained from* c_i *by applying* t_i*, using the rules given in Definition 4.4.*

Definition 4.8: *Given a path* $p = t_1, \ldots, t_n$ *in an eVset-automaton* A*, and considering an ordering* φ *on the elements of* $\mathrm{SVOps}(A)$*, we define*

- *the set* $\mathrm{Ops}(p)$ *as*
$$\bigcup_{i=1}^{n} \mathrm{Ops}(t_i)$$
- *the list* $\mathrm{LOps}_{\varphi}(p)$ *as*
$$\bigoplus_{i=1}^{n} \mathrm{LOps}_{\varphi}(t_i)$$
  where $\bigoplus$ *is the usual list concatenation operator.*

If every transition in p *belongs to* δ^{op}*, we say that* p *is an operation-only path.*

From now on, without loss of generality, let us consider a fixed order φ, in which insertions of variables (e.g., $x\vdash$) into the active set come before deletions (e.g., $\dashv x$).

As shown in the previous chapter, the class of vset-automata does not get more expressive power when extended with the algebra $O = \{\cup, \pi, \bowtie\}$. Hence, this class is a good candidate to be the base for a class of representations of the core spanners, as the Core Simplification Lemma suggests. Nonetheless, the constructions presented in [Fagin et al., 2015] to simulate the operators of O with vset-automata are inefficient, as in general they result in vset-automata that are exponential in the size of the input. In the following, I introduce a subclass of extended vset-automata*, well-behaved eVset-automata*, and I describe *polynomial-time constructions* to simulate the operators of O for this subclass. Moreover, we will see that an AQL query belonging to the core fragment of the language, as defined in [Fagin et al., 2015], can be converted into a well-behaved eVset-automaton extended with string equality selection (and an external final projection, if necessary), making it possible for the runtime to be based almost entirely on well-behaved eVset-automata.

4.2 Well-Behaved Extended Vset-Automata

In order to be well-behaved, an eVset-automaton has to respect some constraints on its paths that are between its initial and accepting state. I call this kind of paths *complete paths.*

Definition 4.9: *Given an eVset-automaton $A = (Q, q_0, q_f, \delta)$, a path $p = t_1, \ldots, t_n$ in A is complete if it is between q_0 and q_f, that is $p = p_{q_0}^{q_f}$.*

I now formally define a well-behaved eVset-automaton.

Definition 4.10: *An eVset-automaton A is well-behaved if, for every complete path p in A we have:*

1. *for every $o \in \mathrm{SVOps}(A)$, o appears exactly once in $\mathrm{LOps}(p)$;*
2. *for every pair of operations $x\vdash, \dashv x \in \mathrm{SVOps}(A)$, $x\vdash$ appears before $\dashv x$ in $\mathrm{LOps}(p)$.*

According to the definition, well-behaved eVset-automata guarantee that any of their complete paths will support a valid accepting run. This property is desirable, because it allows to execute a well-behaved eVset-automaton A with an engine that does not need to check, for each run ρ on a given string $\mathbf{s}$, that all the operations in $\mathrm{SVOps}(A)$ are performed correctly. This is an advantage over generic eVset-automata (or vset-automata).

Example 4.2: *Consider again the eVset-automaton A from Figure 4.1. It is easy to verify that A is well-behaved.*

Since states that cannot reach the accepting state, or that cannot be reached from the initial state, are not used in complete paths, we might want to consider only well-behaved eVset-automata where these states do not exist.

Definition 4.11: *Given an eVset-automaton A, A is pruned if for every state q in A, there is a path between q_0 and q, and a path between q and q_f in A.*

Example 4.3: *The eVset-automaton A from Figure 4.1 is pruned.*

The next proposition tells us that the we can always prune a well-behaved eVset-automaton, obtaining a new eVset-automaton equivalent to the original.

Proposition 4.1: *Given a well-behaved eVset-automaton A, there exists a pruned well-behaved eVset-automaton A' such that $[\![A']\!] = [\![A]\!]$. Moreover, A' can be produced in polynomial time.*

PROOF: To obtain A', it is sufficient to remove from A those states from which we can't reach the final state, those states that are unreachable from the initial state, and all the transitions that have them as source or destination states. To test each state for removal we can use, e.g., Dijkstra's shortest path algorithm, which ensures A' can be found in polynomial time. Because of our construction, A' is well-behaved, as all its complete paths are also in A. We now show that, for every string $\mathbf{s}$, we have $[\![A']\!](\mathbf{s}) = [\![A]\!](\mathbf{s})$. To see that $[\![A']\!](\mathbf{s}) \subseteq [\![A]\!](\mathbf{s})$, we can notice that each run ρ' of A' on $\mathbf{s}$ that is accepting can be supported only by a complete path p', which is also in A by construction. $[\![A]\!](\mathbf{s}) \subseteq [\![A']\!](\mathbf{s})$ is true as well because we include in A' all the complete paths existing in A so every run p of A on $\mathbf{s}$ is also a run of A'. ∎

If a well-behaved eVset-automaton is pruned, it automatically gets two other properties.

Corollary 4.1: *Given a pruned well-behaved eVset-automaton A, for every path p in A* $\mathrm{LOps}(p)$ *contains no duplicates.*

PROOF: Since A is pruned, p is part of a complete path p'. But A is also well-behaved, so $LOps(p')$ contains no duplicates, which implies that $LOps(p)$ has no duplicates either. ∎

Corollary 4.2: *Given a pruned well-behaved eVset-automaton A, and two states q, q' in A, for every pair of paths p, p' between q and q' in A,* $\mathrm{Ops}(p) = \mathrm{Ops}(p')$.

PROOF: Since A is pruned, there exists a path p^0 between q_0 and q. For the same reason, there exists a path p^f between q' and q_f. So q and q' appear in two complete paths, which we may call p^0pp^f and $p^0p'p^f$, that differ only in the subpaths between q and q'. If $\mathrm{Ops}(p) \neq \mathrm{Ops}(p')$, then one of p^0pp^f and $p^0p'p^f$ would not satisfy the requirement 1 of Definition 4.10, making A not well-behaved, a contradiction. ∎

In the following, I describe the constructions used by the runtime system to simulate algebraic operations with well-behaved eVset-automata. The main advantage of these constructions is that their time complexity is *polynomial* in the size of the input automaton/a. Before we start, a formal definition of the size of an eVset-automaton is necessary, along with some assumptions.

Definition 4.12: *Given an eVset-automaton $A = (Q, q_0, q_f, \delta)$, the size of A is defined as $|A| = |Q| + |\delta|$, where:*

- *$|Q|$ is the cardinality of Q;*
- *$|\delta|$ is defined by the following sum:*

$$\sum_{t \in \delta} |t| \tag{4.1}$$

  where, for every transition $t \in \delta$:

  - *if $t \in \delta^{\text{op}}$, $|t| = 2 + |\text{Ops}(t)|$, $|\text{Ops}(t)|$ being the cardinality of $\text{Ops}(t)$;*
  - *$|t| = 3$ otherwise.*

This definition takes into account the number of states of an eVset-automaton, as well as the number of transitions it contains. Each transition is weighted by the number of elements that define it. In case it is a character transition, we count two states and a character. If it is an operation transition instead, we count the two states and the number of operations it performs. Although this definition is abstract, it is closely related to the sizes of real eVset-automata representations, that are indeed determined by the elements considered, up to a multiplicative constant. Moreover, in the proofs that will follow, I assume that the operations of adding a state to/deleting a state from the state set of an eVset-automaton, or adding a transition to/deleting a transition from its transition function, take *constant time*. These assumptions are reasonable, because the state set and the transition function are implemented as Scala hashsets in the runtime system, and the hashset data structure indeed guarantees this complexity[1]. Another important assumption is that, given an eVset-automaton A, creating a new transition requires $O(|\text{SVars}(A)|)$ time. This complexity is justified by the fact that, given an operation transition t in A, we create $\text{Ops}(t)$ by adding $O(|\text{SVars}(A)|)$ operations to it, and $\text{Ops}(t)$ is implemented as a hashset as well. Moreover, the time required to instantiate a character transition can be considered constant, so it is dominated by $O(|\text{SVars}(A)|)$.

We are ready to discuss the announced constructions, starting with the one for projection.

[1] See `http://docs.scala-lang.org/overviews/collections/performance-characteristics.html`.

Theorem 4.1: *Given a well-behaved eVset-automaton A, the set $X = \mathrm{SVars}(A)$ and a set $Y \subseteq X$, a well behaved eVset-automaton A' can be produced in linear time such that $[\![A']\!] = \pi_Y [\![A]\!]$.*

PROOF: Let us consider $A = (Q, q_0, q_f, \delta)$. We can take $A' = \left(Q', q_0', q_f', \delta'\right)$, where:

- $Q' = Q$;
- $q_0' = q_0$;
- $q_f' = q_f$;
- $\delta' = \left(\delta \setminus \delta^{\text{unprojected}}\right) \cup \delta^{\text{projected}}$, where:
  - $\delta^{\text{unprojected}} = \{(q, S, q') \in \delta \mid S \cap \mathrm{SVOps}(X \setminus Y) \neq \emptyset\}$;
  - $\delta^{\text{projected}} = \left\{(q, S', q') \mid \exists\, (q, S, q') \in \delta^{\text{unprojected}} : S' = S \setminus \mathrm{SVOps}(X \setminus Y)\right\}$.

This construction removes all the occurrences of variable operations concerning variables excluded from the projection. A' is still well-behaved, since we maintain the occurrences of operations concerning the variables on which we project, that continue to appear exactly once on each complete path, and in the right order. To obtain this construction, we can examine all transitions in δ and, for each operation transition t, remove it from δ if it contains variable operations involving variables we need to exclude, then add a new transition t' that is identical to t, except for the fact that it does not contain the unwanted operations. In this procedure, we scan $O(|\delta|)$ transitions and we replace a transition t in $O(|t|)$ time. Hence, the total running time is $O\left(\sum_{t \in \delta} O(|t|)\right) = O(|\delta|) = O(|A|)$. Next, let us show that $[\![A']\!] = \pi_Y [\![A]\!]$. Notice that given a string $\mathbf{s}$, for each accepting run ρ of A on $\mathbf{s}$ producing an $\mathbf{s}$-tuple μ, there is an accepting run ρ' of A' that produces an $\mathbf{s}$-tuple μ', which assigns the same spans as μ to the variables in common with μ', which by construction belong to Y. The path p' supporting ρ' is exactly the path obtained by modifying the path p, that supports ρ, by eliminating the operations on non-projected variables. This shows that $\pi_Y [\![A]\!] \subseteq [\![A']\!]$. It must also be that $[\![A']\!] \subseteq \pi_Y [\![A]\!]$, because no additional complete paths were added to A'. ∎

A similar result holds for the union operation.

Theorem 4.2: *Given two well-behaved eVset-automata A and B that are union-compatible (i.e., $\mathrm{SVars}(A) = \mathrm{SVars}(B)$), a third well-behaved eVset-automaton C can be produced in linear time such that $[\![C]\!] = [\![A]\!] \cup [\![B]\!]$.*

PROOF: Let us consider $A = \left(Q^A, q_0^A, q_f^A, \delta^A\right)$ and $B = \left(Q^B, q_0^B, q_f^B, \delta^B\right)$. We can take $C = \left(Q^C, q_0^C, q_f^C, \delta^C\right)$, where:

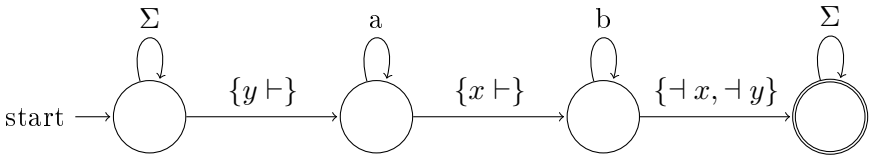


Figure 4.2 – A well-behaved eVset-automaton B.

- $Q^C = Q^A \cup Q^B \cup \left\{q_0^C, q_f^C\right\}$;
- $\delta^C = \delta^A \cup \delta^B \cup \left\{\left(q_0^C, \emptyset, q_0^B\right), \left(q_f^B, \emptyset, q_f^C\right), \left(q_0^C, \emptyset, q_0^A\right), \left(q_f^A, \emptyset, q_f^C\right)\right\}$.

In this construction, we allow to go from the initial state of C to the initial state of either A or B, and to go from the accepting state of A or that of B to the one of C, without any new variable operations. Thus, given a string $\mathbf{s}$, C can span exactly the $\mathbf{s}$-tuples contained in $[\![A]\!](\mathbf{s}) \cup [\![B]\!](\mathbf{s})$. Regarding the complexity of the construction, the operations that we perform here are the union of the state sets and transition functions, and the addition of a fixed number of new transitions and states. Let us consider the size of the input as $n = |A| + |B|$. With the assumptions we made, the time complexity is $O\left(\left|Q^B\right|\right) + O\left(\left|\delta^B\right|\right) = O(n)$. It is trivial to verify that C is well-behaved, thus the details are omitted. ■

In order to show that we can obtain the natural join of two spanners, represented by well-behaved eVset-automata, in polynomial time, we need a few more steps than in the previous cases. The construction that I present is conceptually very similar to that described in [Fagin et al., 2015] for the natural join of plain vset-automata. That construction simulates running the input automata in parallel, making sure that operations on common variables are performed simultaneously. Unfortunately this does not work in general, as the next example shows.

Example 4.4: *Consider the well-behaved eVset-automata A from Figure 4.1, and B from Figure 4.2 and the string $\mathbf{s}$ =*'b'*. $[\![A]\!](\mathbf{s})$ contains the $\mathbf{s}$-tuple μ such that $\mu(x) = [0, 1\rangle$ and $\mu(y) = [0, 1\rangle$. We also have that $\mu \in [\![B]\!](\mathbf{s})$. If we attempt to run A and B in parallel on $\mathbf{s}$, we will not be able to span μ, because the two automata disagree on the order of the operations $x \vdash$ and $y \vdash$. Thus, we don't get $[\![A]\!] \bowtie [\![B]\!](\mathbf{s})$ as a result of the execution.*

For the construction to work, the input automata must be modified in some way. In my system, I convert them into a particular form of eVset-automata, that I call *operation-closed*. The definition of an operation-closed eVset-automaton follows.

Definition 4.13: *Given an eVset-automaton A, A is operation-closed if, for every pair of states q, q' in A, whenever there exists an operation-only path $p{=}t_1, \ldots, t_n$ between q and q', then there exists a transition $t = (q, \bigcup_{i=1}^{n} \mathrm{Ops}(t_i), q')$ in A.*

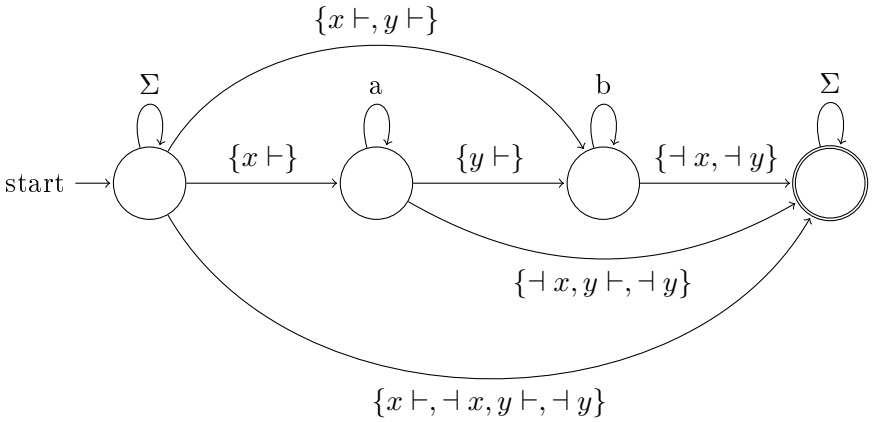


Figure 4.3 – An operation-closed well-behaved eVset-automaton A', with $[\![A']\!] = [\![A]\!]$, where A is the automaton from Figure 4.1.

Example 4.5: *Consider the well-behaved eVset-automaton A' from Figure 4.3. A' is operation-closed. The reader can verify that $[\![A']\!] = [\![A]\!]$, where A is the eVset-automaton from Figure 4.1.*

As the next proposition states, given a well-behaved eVset-automaton, we can always find an equivalent operation-closed well-behaved eVset-automaton. This manipulation of a well-behaved eVset-automaton, along with the fact that it is always possible, ensures the applicability of the join construction.

Proposition 4.2: *Given a well-behaved eVset-automaton A, there exists an operation-closed well-behaved eVset-automaton A' such that $[\![A']\!] = [\![A]\!]$. Moreover, the size of A' is cubic in the size of A.*

Proof: Let us consider $A = (Q, q_0, q_f, \delta)$. Without loss of generality, we can assume that A is pruned. Then we can take $A' = \left(Q', q'_0, q'_f, \delta'\right)$, where:

- $Q' = Q$;
- $q'_0 = q_0$;
- $q'_f = q_f$;
- $$\delta' = \delta \cup \left\{ \left(q, \bigcup_{i=1}^{n} \mathrm{Ops}\left(t_i\right), q' \right) \mid \exists p = t_1, \ldots, t_n \in \mathrm{Paths}\left(A\right) : p = p_q^{q'}, \right.$$
  $$\left. p \text{ is operation-only} \right\}.$$

This construction does nothing but including in A' the transitions that are missing in A to be operation-closed. A' is well-behaved because for each new operation transition t between

$q, q' \in Q$ and each operation-only path $p = p_q^{q'}$ in A we have that $\mathrm{Ops}(t) = \mathrm{Ops}(p)$ (see Corollary 4.2). Let us show that, for every string $\mathbf{s} \in \Sigma^*$, $[\![A']\!](\mathbf{s}) = [\![A]\!](\mathbf{s})$. To see that $[\![A']\!](\mathbf{s}) \subseteq [\![A]\!](\mathbf{s})$, consider an accepting run ρ' of A' on $\mathbf{s}$, that returns an $\mathbf{s}$-tuple μ. This run is supported by a complete path p' in A'. We can always find a complete path p in A that supports a run ρ of A on $\mathbf{s}$, which returns μ as well. To obtain p, we substitute every operation transition t in p' that does not belong to δ with an operation-only path p'' in A such that $\mathrm{Ops}(p'') = \mathrm{Ops}(t)$ and p'' is between the source and destination states of t. This is always possible by construction of A'. It is easy to verify that the run ρ of A on $\mathbf{s}$, supported by p, returns indeed μ, thus the details are omitted. $[\![A]\!](\mathbf{s}) \subseteq [\![A']\!](\mathbf{s})$ is also true, because we include all the transitions belonging to δ in δ'. Because of Corollary 4.2, in the worst case we add a new operation transition of size $O(|\mathrm{SVars}(A)|) = O(|A|)$ between each pair of states $q, q' \in Q$. Hence, the size of A' is cubic in the size of A. ∎

The construction described in the last proposition is the reason why extended vset-automata were used for the runtime system instead of plain vset-automata. When we construct an operation-closed automaton, starting from an automaton that does not have this property, the size of the resulting transition function might explode (although it will stay polynomial in the size of the original automaton). The ability to include multiple variable operations in a single transition allows for a more compact representation, that is also easier to manipulate. Now that we have seen how to transform a well-behaved eVset-automaton for the use of the join construction, we can look at its formal definition.

Definition 4.14: *Given two operation-closed well-behaved eVset-automata* $A = \left(Q_A, q_A^0, q_A^f, \delta_A\right)$ *and* $B = \left(Q_B, q_B^0, q_B^f, \delta_B\right)$*, their product is an eVset-automaton* $C = \left(Q, q^0, q^f, \delta\right)$*, where:*

- $Q = Q_A \times Q_B$;
- $q^0 = \langle q_A^0, q_B^0 \rangle$;
- $q^f = \left\langle q_A^f, q_B^f \right\rangle$;
- δ *has the following transitions:*
  - $(\langle q_A, q_B \rangle, \sigma, \langle q'_A, q'_B \rangle)$ *whenever* $\sigma \in \Sigma$, $(q_A, \sigma, q'_A) \in \delta_A$ *and* $(q_B, \sigma, q'_B) \in \delta_B$;
  - $(\langle q_A, q_B \rangle, S_A \cup S_B, \langle q'_A, q'_B \rangle)$ *whenever* $(q_A, S_A, q'_A) \in \delta_A$, $(q_B, S_B, q'_B) \in \delta_B$ *and* $\mathrm{SVOps}(B) \cap S_A = \mathrm{SVOps}(A) \cap S_B$;
  - $(\langle q_A, q_B \rangle, S_A, \langle q'_A, q_B \rangle)$ *whenever* $(q_A, S_A, q'_A) \in \delta_A$ *and* $\mathrm{SVOps}(B) \cap S_A = \emptyset$;
  - $(\langle q_A, q_B \rangle, S_B, \langle q_A, q'_B \rangle)$ *whenever* $(q_B, S_B, q'_B) \in \delta_B$ *and* $\mathrm{SVOps}(A) \cap S_B = \emptyset$.

We write $C = A \otimes B$.

We can now state the following theorem.

Theorem 4.3: *Given two operation-closed well-behaved eVset-automata* A *and* B*, and given an eVset-automaton* C *such that* $C = A \otimes B$*, then* $[\![C]\!] = [\![A]\!] \bowtie [\![B]\!]$*. Moreover,* C *is well-behaved, and can be obtained in quadratic time.*

PROOF: This proof is similar to the proof for the analogous construction for plain vset-automata described in [Fagin et al., 2015]. To show that $[\![C]\!] \subseteq [\![A]\!] \bowtie [\![B]\!]$, we can decompose a run of C on a string $\mathbf{s}$ into two consistent runs of ρ_A of A and ρ_B of B. Two runs ρ, ρ', with supporting paths p and p' respectively, are consistent with each other if, for every pair of operations o, o' both belonging to $\mathrm{Ops}(p)$ and $\mathrm{Ops}(p')$, o appears before o' in $\mathrm{LOps}(p)$ if and only if the same holds for $\mathrm{LOps}(p')$. Since a run of C represents two parallel runs of A and B by construction, the decomposition aims to isolate the two individual runs of A and B. The details of this decomposition are not difficult to figure out, and are omitted. To show that $[\![A]\!] \bowtie [\![B]\!] \subseteq [\![C]\!]$, let us consider a string $\mathbf{s}$, a $\mathbf{s}$-tuple $\mu_A \in [\![A]\!](\mathbf{s})$ and a $\mathbf{s}$-tuple $\mu_B \in [\![B]\!](\mathbf{s})$ that assigns the same spans as μ_A to the variables they have in common. Given the $\mathbf{s}$-tuple μ that contains all the variable assignments of μ_A and μ_B, we need to find a run ρ of C on $\mathbf{s}$ that returns μ. Let us call $\rho_A \in \mathrm{ARuns}(A, \mathbf{s})$ and $\rho_B \in \mathrm{ARuns}(B, \mathbf{s})$ the runs that return μ_A and μ_B, respectively. We can obtain ρ by combining ρ_A and ρ_B. For this construction to work, ρ_A and ρ_B need to be consistent on the order of the variable operations they perform. Since A and B are well-behaved and operation-closed, ρ_A and ρ_B can always be selected so that they are consistent. Again the details of this construction are straightforward and are not reported. Moreover, it is easy to see that C is well-behaved by construction. To obtain $C = A \otimes B$ algorithmically, the simplest approach is to compare all the possible pairs of transitions $t_A \in \delta_A, t_b \in \delta_B$, and to generate, for each pair, a new transition, according to the rules provided by Definition 4.14. The cost of generating a new transition from t_A and t_B is $O(|t_A| + |t_B|)$, the most costly case being the one involving two operation-transitions. Let us consider the size of the input as $n = |A| + |B|$. The running time for obtaining C is then equal to $\sum_{t_A \in \delta_A, t_B \in \delta_B} O(|t_A| + |t_B|) = \sum_{t_A \in \delta_A, t_B \in \delta_B} O(|t_A|) + \sum_{t_A \in \delta_A, t_B \in \delta_B} O(|t_B|) = \sum_{t_B \in \delta_B} \sum_{t_A \in \delta_A} O(|t_A|) + \sum_{t_A \in \delta_A} \sum_{t_B \in \delta_B} O(|t_B|) = O(|\delta_B| \cdot |\delta_A|) + O(|\delta_A| \cdot |\delta_B|) = O(n^2)$. ∎

In the previous chapter, we saw that the Core Simplification Lemma allows us to represent core spanners in a convenient way, based on vset-automata. What about well-behaved eVset-automata? Does a similar statement hold? The answer is affirmative, and in the following we will prove this claim. First of all, let us reason on the relative expressive power of eVset-automata with respect to plain vset-automata.

Lemma 4.1: *Given an eVset-automaton $A = (Q, q_0, q_f, \delta)$, A can be converted in linear time into a vset-automaton A' such that $[\![A']\!] = [\![A]\!]$, in a well-behavedness preserving manner*[2]*.*

PROOF: Without loss of generality, we consider an ordering of the symbols in $\mathrm{SVOps}(A)$ of the following form:

$$x\vdash, \ldots, y\vdash, \dashv x, \ldots, \dashv y$$

In this ordering, all insertion operations come before the deletion operations. Let us define $o \prec o'$, with $o, o' \in \mathrm{SVOps}(A)$, if o comes before o' (*not* if they are equal) in the chosen ordering. Consider $A' = \left(Q', q_0', q_f', \delta'\right)$, with $\mathrm{SVars}(A') = \mathrm{SVars}(A)$, whose components are defined as follows:

- $Q' = Q \cup Q^{\mathrm{ops}} \cup Q^{\emptyset}$, where:
  - $Q^{\mathrm{ops}} = \left\{q_{q',o,q''} \mid \exists\,(q', S, q'') \in \delta : o \in S\right\}$;
  - $Q^{\emptyset} = \left\{q_{q',\emptyset,q''} \mid \exists\,(q', \emptyset, q'') \in \delta\right\}$;
- $q_0' = q_0$;
- $q_f' = q_f$;
- $\delta' = \left(\delta \setminus \delta^S\right) \cup \delta^{\mathrm{ops}} \cup \delta^{\emptyset} \cup \delta^{\epsilon}$, where:
  - $\delta^S = \{(q, S, q') \in \delta\}$;
  - $\delta^{\mathrm{ops}} = \Big\{\left(q_{q',o,q''}, o, q_{q',o',q''}\right) \mid \exists\left(q', S, q''\right) \in \delta : \big(o, o' \in S \wedge o \prec o' \wedge$ $\forall o'' \in S : o \not\prec o'' \not\prec o'\big)\Big\} \cup \Big\{\left(q_{q',o,q''}, o, q''\right) \mid \exists\left(q', S, q''\right) \in \delta :$ $\big(o \in S \wedge \forall o' \in S : o \not\prec o'\big)\Big\}$;
  - $\delta^{\emptyset} = \left\{\left(q_{q',\emptyset,q''}, \epsilon, q''\right) \mid \exists\,(q', \emptyset, q'') \in \delta\right\}$;
  - $\delta^{\epsilon} = \Big\{\left(q, \epsilon, q'\right) \mid \Big(\big(\exists\left(q', o, q''\right) \in \delta^{\mathrm{ops}} : \forall\left(q', o', q''\right) \in \delta^{\mathrm{ops}} : o' \not\prec o\big) \vee$ $\Big(\exists\left(q', \epsilon, q''\right) \in \delta^{\emptyset}\Big)\Big) \wedge \big(\exists\left(q, S, q'''\right) \in \delta\big)\Big\}$.

This construction expands the transitions of A that are labeled with a set of variable operations into a sequence of transitions performing one operation at a time, taking care of putting the insertion operations before the deletion ones. This construction clearly preserves well-behavedness, because each complete path p' in A' is obtained from a path p in A, preserving the original operations of p and ensuring a correct order of their appearance

[2]Notice that I didn't define well-behavedness in the case of a plain vset-automaton. Nonetheless, the idea underlying the definition for extended vset-automata remains unchanged and the actual definition for a standard vset-automaton is not difficult to figure out.

(if A is well-behaved). Each sequence starts with an ϵ-transition. This is not necessary in principle, but it allows to reduce the complexity of the formulation. The construction also substitutes transitions labeled with the empty set with ordinary ϵ-transitions. To prove the equivalence between A and A' it is sufficient to notice that for every string $\mathbf{s}$, every run belonging to $\mathrm{ARuns}(A, \mathbf{s})$ can be put in correspondence with a run belonging to $\mathrm{ARuns}(A', \mathbf{s})$, and vice versa. Indeed, if we start from $\rho \in \mathrm{ARuns}(A, \mathbf{s})$ we can obtain a run ρ'$\in \mathrm{ARuns}(A', \mathbf{s})$ that spans the same $\mathbf{s}$-tuple μ. It is sufficient to expand configuration pairs in ρ whose current states are linked in A by a transition t, which we expand in A', with a series of configurations that let us perform the set of variable operations in t one at a time. This is always possible by construction of A'. If we start from ρ'$\in \mathrm{ARuns}(A', \mathbf{s})$, we can obtain an equivalent run $\rho \in \mathrm{ARuns}(A, \mathbf{s})$ in the opposite way, by compressing consecutive configurations. The details are omitted.
It remains to show that the construction of A' can be carried in linear time. Let us refer to the size of A as n. With the usual assumptions, expanding a single transition t of A takes $O(|t|)$ time. Expanding every transition that is needed will then take $O\left(\sum_{t \in \delta} O(|t|)\right) = O(n)$ time. ∎

The opposite direction is also true.

Lemma 4.2: *Given a vset-automaton $A = (Q, q_0, q_f, \delta)$, an eVset-automaton A' can be found in linear time such that $[\![A']\!] = [\![A]\!]$, in a well-behavedness preserving manner.*

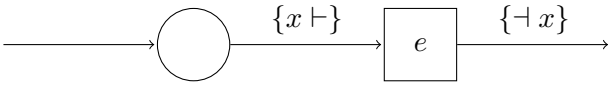
Proof: In this case it is sufficient to replace every transition t performing a variable operation o in A with an operation transition t' in A' such that $\mathrm{Ops}(t') = \{o\}$ and $\mathrm{LOps}(t') = o$. ϵ-transitions can be replaced by operation transitions with an empty set. We need to show that this construction can be carried out in linear time. We do not have a definition of the size of a vset-automaton, but we can easily adapt Definition 4.12: the only difference is that every transition in the transition function weights 3. With this assumption, this construction can be obtained in linear time, as its time complexity has a similar expression to that of the construction discussed in the previous lemma. It is easy to verify equivalence, and that well-behavedness is preserved, thus the details are omitted. ∎

These results are interesting. In particular, Lemma 4.1 ensures that any well-behaved eVset-automaton, opportunely extended with the needed algebraic operators, represents a query belonging to the core fragment of AQL. Is also the inverse true? As mentioned, the runtime system works only with well-behaved eVset-automata. Then, for the system to have full applicability, it must hold that an AQL core query can be represented by a well-behaved eVset-automaton (extended with the needed operators). This is indeed the case, as we are going to see. According to Definition 3.36, an AQL core query is a set

of regex formulas combined by using the operators described in Subsection 3.3.3. The first step is to prove that a regex formula can always be converted into a well-behaved eVset-automaton.

Theorem 4.4: *Given a regex formula γ, a well-behaved eVset-automaton A can be found in polynomial time such that $[\![A]\!] = [\![\gamma]\!]$.*

PROOF: It is well known that there exists a polynomial-time method to generate an NFA corresponding to a regular expression r. This method generates the NFA inductively while parsing r. A description of the method is available in, e.g., [Cox, 2007]. This procedure does not deal with submatch extraction, thus we need to describe how to handle the $x\{\cdot\}$ operator, $x \in \mathrm{SVars}(\gamma)$. We can add the following fragment to A:



where the rectangle with label e is the part of A representing the subexpression of γ contained in $x\{\cdot\}$. The fact that the arrow pointing to the leftmost state comes from nowhere means that the fragment is connected to whichever state came before it, and, similarly, the arrow coming out of the rectangle is going nowhere because we connect it to the state that comes after. We could also merge consecutive operation transitions in A, to fully exploit the fact that it is an eVset-automaton, without altering the runtime complexity. Correctness follows from the correctness of the single fragments generated, which is easy to verify. Finally, A is well-behaved, because of the construction we used, along with the fact that the new fragment for span capturing cannot appear in a loop and that every path in A spans the same variables (because γ is syntactically $\mathrm{SVars}(\gamma)$-functional). ■

Let us call the class of well-behaved eVset-automata $\mathrm{VA}_{\mathrm{WESet}}$. By combining Theorems 4.4, 4.2 and 4.3 and Lemma 4.1, we can state the following theorem.

Theorem 4.5: $\left[\!\left[\mathrm{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] = \left[\!\left[\mathrm{VA}_{\mathrm{WESet}}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] = \left[\!\left[\mathrm{VA}_{\mathrm{WESet}}^{\{\pi,\varsigma^=\}}\right]\!\right]$.

PROOF: Theorem 4.4 tells us that $[\![\mathrm{RGX}]\!] \subseteq [\![\mathrm{VA}_{\mathrm{WESet}}]\!]$. Thus we can state that

$$\left[\!\left[\mathrm{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] \subseteq \left[\!\left[\mathrm{VA}_{\mathrm{WESet}}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] \tag{4.2}$$

Moreover, Lemma 4.1 and Theorem 3.6 imply that

$$\left[\!\left[\mathrm{VA}_{\mathrm{WESet}}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] \subseteq \left[\!\left[\mathrm{VA}_{\mathrm{set}}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] = \left[\!\left[\mathrm{RGX}^{\{\cup,\pi,\bowtie,\varsigma^=\}}\right]\!\right] \tag{4.3}$$

Finally, Theorems 4.2 and 4.3 justify the right-hand equality in the statement of this theorem. ■

We can also formulate a modified version of the Core Simplification Lemma (Lemma 3.2). The proof is very similar to that of the original lemma, which can be found in [Fagin et al., 2015], and is omitted here.

Lemma 4.3: *Every core spanner can be defined by an expression of the form*

$$\pi_V SA \tag{4.4}$$

where:

- *A is a well-behaved eVset-automaton;*
- *$V \subseteq \mathrm{SVars}(A)$;*
- *S is a sequence of string selections $\varsigma^{=}_{x,y}$, for $x, y \in \mathrm{SVars}(A)$.*

In this chapter, eVset-automata were introduced. Then, a particular kind of eVset-automata, well-behaved eVset-automata, that I use in the runtime system, was discussed: we saw that an execution engine for these automata is less complex than one for generic eVset-automata. Then, I proved that we can simulate the operators in the algebra $\{\pi, \cup, \bowtie\}$ by combining the input automata with polynomial-time procedures, that return well-behaved eVset-automata. Finally, I have shown that well-behaved eVset-automata can be used to conveniently represent core spanners. The next chapter is a discussion of the actual implementation of the runtime system.

Chapter 5

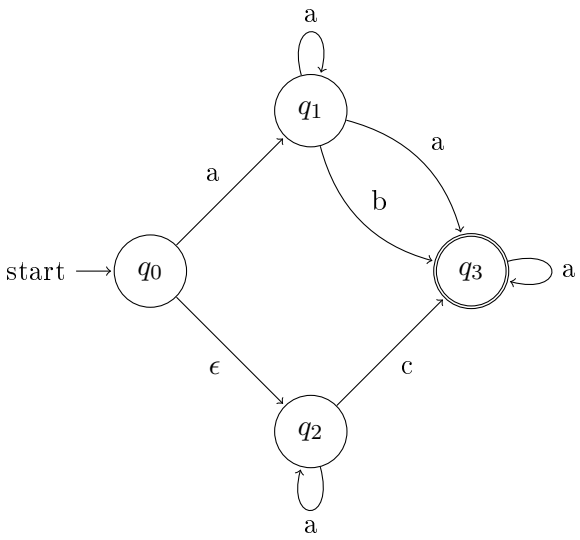
Implementation

This chapter describes the implementation of the runtime system. The outline of the chapter is the following. Section 5.1 describes an idealized engine for the evaluation of an NFA on a string, originally proposed by Ken Thompson. A modified version of this engine is used in the runtime system for evaluating core spanners, by means of representations based on well-behaved eVset-automata. In Section 5.2, I discuss the actual implementation of the runtime system.

5.1 A Method for NFA Execution: The Thompson Approach

Given a regular expression r, its corresponding NFA A and a string $\mathbf{s}$, the Thompson algorithm will try all the feasible runs of A on $\mathbf{s}$ at the same time. More precisely, the method iterates over the characters of $\mathbf{s}$ and, for each iteration, there is a set of current states, each representing the advancement in a feasible run. The following steps are performed in an iteration of the algorithm: the set of outgoing transitions, for each current state, is examined, all ϵ-transitions are fired iteratively and, subsequently, any transition labeled with the character corresponding to the current one is fired. This produces a new set of states and the execution continues as described. $\mathbf{s}$ is matched by A if, at the end of the execution, all the symbols of $\mathbf{s}$ have been consumed and at least one state in the current state set is an accepting state. It can be shown that the time complexity of this approach is $O(mn)$, where m is the size of r and n is the size of $\mathbf{s}$.

Usually, we accept *partial matching*: we don't require that a regular expression reaches the end of the input string to produce a match. In terms of the corresponding automaton, we consider as accepting a run that ends in the final state even if it didn't scan the whole string. This is useful for implementing *unanchored matching,* where we want to obtain not only the matches starting from the beginning of a string, but also those starting from its suffixes. The regular expressions in AQL queries perform unanchored matching transpar-



(a) An ϵ-NFA A.

$$\underset{\{q_0,q_2\}}{\underline{a}aab} \longrightarrow \underset{\{q_1,q_2\}}{a\underline{a}ab} \longrightarrow \underset{\{q_1,q_2,q_3\}}{aa\underline{a}b} \longrightarrow \underset{\{q_1,q_2,q_3\}}{aaa\underline{b}} \longrightarrow \underset{\{q_3\}}{aaab\_}$$

(b) The execution of A on 'aaab' according to the Thompson approach.

Figure 5.1 – Example of the Thompson algorithm.

ently to the user: they automatically scan the input document for matches. Thus, this kind of matching is implicitly adopted by the runtime system too. The regular expressions of AQL retain all the matches they can find. Instead, the traditional Thompson algorithm produces a single match. Thus, for the original procedure, partial matching poses the problem of ambiguous matches. This is why concrete algorithms implement a policy to discriminate among multiple possible matches (e.g., greedy leftmost). For more information on the Thompson approach, see [Cox, 2007].

Example 5.1: *The automaton A of Figure 5.1 accepts the string* **s** ='aaab'. *Figure 5.1 also illustrates the execution of A on* **s**. *Notice that the current state set is not a multiset. Hence, runs that end up in the same state are naturally merged. For instance, if we are in q_1 and we read an* 'a' *we can go to q_3, but we can also read an* 'a' *while in q_3, remaining in that state. Nonetheless, in the cases where both q_1 and q_3 are in the current state set and an* 'a' *is read, q_3 appears only once in the next. This is because the two originally distinct runs are now equivalent. This lets the algorithm keep a reduced list of current states, and justifies the runtime complexity presented.*

One interesting implementation of the Thompson algorithm views an NFA as a program that can be executed by a virtual machine on a string. This method is described in [Cox,

```
0  CHAR a
1  SPLIT 0, 2
2  SPLIT 3, 5
3  CHAR b
4  JUMP 2
5  MATCH
```

(a) A program for the regex a^+b^*.

```
0  SAVE 0
1  CHAR a
2  SPLIT 1, 3
3  SAVE 1
4  SAVE 2
5  SPLIT 6, 8
6  CHAR b
7  JUMP 5
8  SAVE 3
9  MATCH
```

(b) A program for the regex $(a^+)\,(b^*)$.

Figure 5.2 – Example programs.

2009].

5.1.0.1 The Virtual Machine Implementation

In this implementation, an NFA (or a regex) is converted into a program, written in a simple assembly language with few instructions. The basic instructions are character match (CHAR), string match (MATCH), and control flow instructions (SPLIT and JUMP). A virtual machine is provided, which treats each concurrent run on a string **s** as a conceptual thread. The virtual machine advances all the threads in lockstep, in the spirit of the Thompson approach. Different threads that reach the same instruction in a program are merged.

The main advantage of this method is that it is easy to enrich the assembly language with new instructions, in order to support new features. To execute new instructions, modifying the virtual machine is required. For instance, capturing groups can be implemented by equipping each thread with an array of saved pointers that are grouped by two: the first element would point to the beginning of a span of text and the second one would point to its end. Then, a SAVE instruction could be added, that would make the virtual machine record the current position in the input into the pointer specified by the instruction.

Example 5.2: *The program (a) in Figure 5.2 can be used to match the regular expression a^+b^*. SPLIT instructions explicitly divide a thread into two, telling each of the generated threads which position in the program to reach. The program (b) in Figure 5.2 matches the same expression as program (a), but it retains the substrings matching the subexpressions between parentheses too.*

```
0   SPLIT 1, 8
1     CHAR a
2   SPLIT 1, 3
3   SPLIT 4, 6
4     CHAR a
5     JUMP 7
6     CHAR b
7    JUMP 12
8   SPLIT 9, 11
9     CHAR a
10    JUMP 8
11    CHAR c
12 SPLIT 13, 15
13    CHAR a
14   JUMP 12
15    MATCH
```

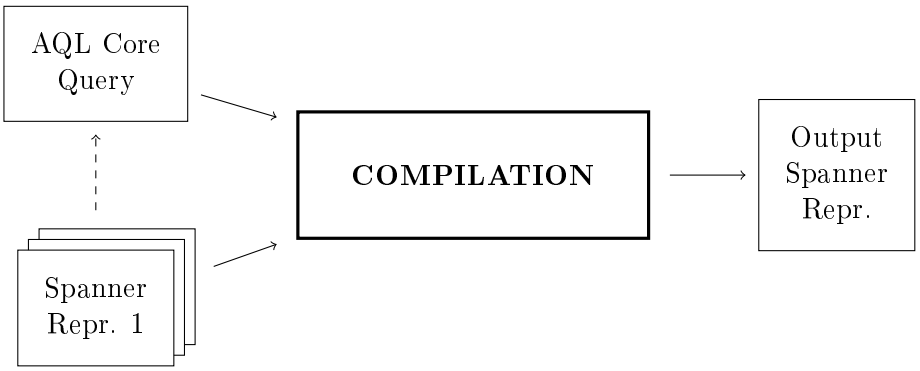
Figure 5.3 – A program for the automaton A of Figure 5.1.

Example 5.3: *The program shown in Figure 5.3 corresponds to the automaton A of Figure 5.1. Notice how the ϵ-transition between q_0 and q_2 is automatically omitted from the program. Blocks of SPLIT instructions correspond to multiple outgoing transitions from a state. JUMP instructions are used to merge execution branches.*

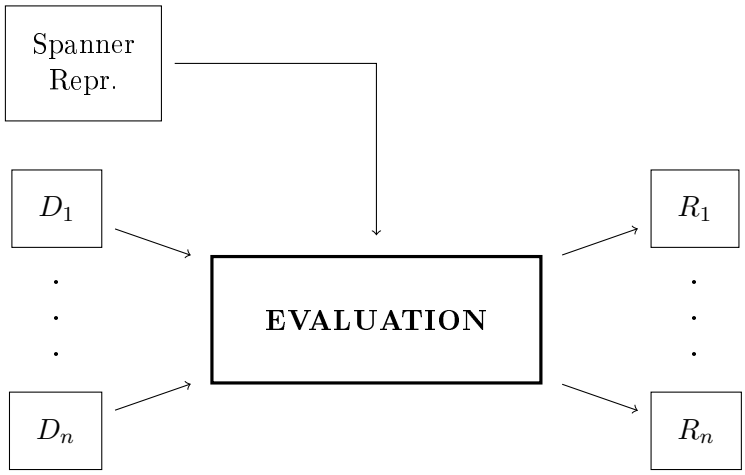
5.2 Implementation

In the system, core spanners are represented by well-behaved eVset-automata, for which a set of string equality constraints can be specified on pairs of their span variables. A final projection, if present, is handled separately. Moreover, the implementation fixes SVars $= \mathbb{N}$. Hence, span variables are identified by nonnegative integers. The execution approach used by the system is radically different from the one of SystemT, because instead of realizing algebraic operators as operations to apply to the outputs of their arguments as SystemT does, they are simulated with the constructions presented in Section 4.2. The system can be used in two modes:

compilation mode: reads an AQL core query and produces an equivalent core spanner representation, using the constructions of Section 4.2;



(a) Compilation mode.



(b) Evaluation mode.

Figure 5.4 – Overview of the system.

evaluation mode: evaluates a core spanner representation on a series of text documents, returning, for each of them, a $(V, \mathbf{s})$-relation as output.

The system is single-threaded.

5.2.1 Compilation Mode

The compilation mode allows to easily compose spanner representations to obtain more complex ones. A high-level description of the composition process is shown in Figure 5.4a. It is performed according to an input AQL query, written in an ad-hoc syntax that linearizes its operator tree. A query specification consists of two parts: an initial part where a set of input spanners is specified and another part containing a series of operations to be performed on spanners from the input set. As mentioned, input spanners are encoded by spanner representations belonging to $\mathrm{VA}_{\mathrm{WSet}}^{\{\varsigma^=\}}$. Any potential final projection involving

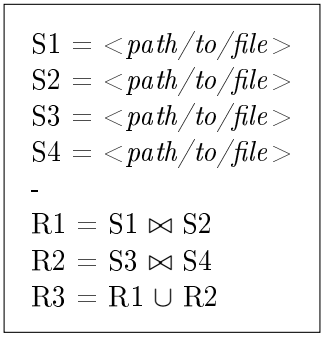


Figure 5.5 – A sample AQL query, written in the syntax used by the system.

variables that are used in one or more string equality selections is handled separately. The next example discusses a sample AQL query, formulated in the syntax expected by the system.

Example 5.4: *Consider the query representation shown in Figure 5.5. In this representation, four spanners, with identifiers S1, S2, S3 and S4, are included by initializing the corresponding variables with the paths of the files containing their representations. Then the separating character "-" is inserted to mark the beginning of the series of operations to be performed on the input spanners. Here we have the following operations: natural join between S1 and S2 (result assigned to variable R1), natural join between S3 and S4 (result assigned to R2), and union between R1 and R2 (result assigned to R3). The operations are compiled in order of appearance. In general, the spanner representation resulting from the last operation is returned as output, while intermediate results are discarded.*

The compilation mode supports all the operators described in Section 3.3.3. For projection, union and natural join, the constructions described in Theorems 4.1, 4.2 and 4.3 are respectively used (keep in mind that input spanners are represented by well-behaved eVset-automata). String equality selection operations are simply reported in the resulting representation, for use of the evaluation engine.

In addition to the operators that we saw so far, a specialized join operator was implemented. It is based on a predicate which may be referred to as "followedBy(**min**, **max**)": according to it, two spans from two different input relations are combined if and only if they are distant from each other at least **min** characters and at most **max** characters. This kind of join is supported by SystemT, and it is often used in AQL queries (see [Reiss et al., 2008]). Given two unary core spanners A and B, with $\mathrm{SVars}(A) = \{x\}$, $\mathrm{SVars}(B) = \{y\}$ and $x \neq y$, we can express the described join construction, with A and B as arguments, by the following core spanner:

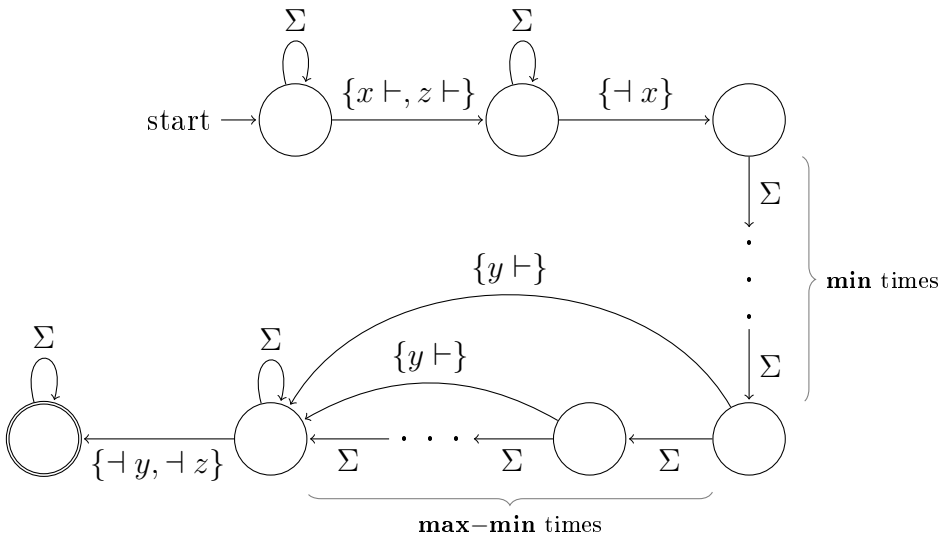


Figure 5.6 – A well-behaved eVset-automaton representing $S^{x,y,z}_{\text{followedBy}(\mathbf{min},\mathbf{max})}$.

$$C = \pi_{\{z\}} \left(A \bowtie S^{x,y,z}_{\text{followedBy}(\mathbf{min},\mathbf{max})} \bowtie B \right) \tag{5.1}$$

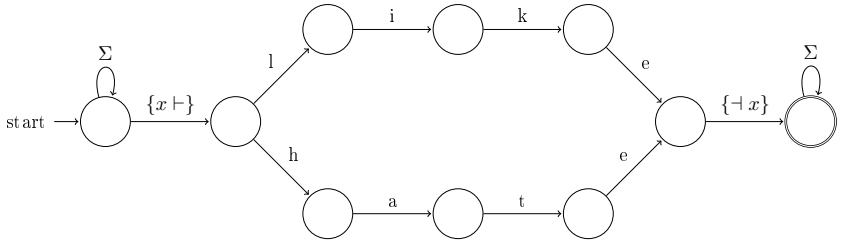
where:

- for a given string $\mathbf{s}$ and any two $\mathbf{s}$-tuples $\mu_A \in A(\mathbf{s}), \mu_B \in B(\mathbf{s})$ such that $\mu_A(x) = [i_b, i_e\rangle$, $\mu_B(y) = [j_b, j_e\rangle$ and $\mathbf{min} \leq j_b - i_e \leq \mathbf{max}$, $C(\mathbf{s})$ contains a tuple μ_C such that $\mu_C(z) = [i_b, j_e\rangle$, and no additional tuples;
- $S^{x,y,z}_{\text{followedBy}(\mathbf{min},\mathbf{max})}$ is the contextual core spanner that realizes the join, parametrized by the input variables x and y, the output variable z, **min** and **max**.

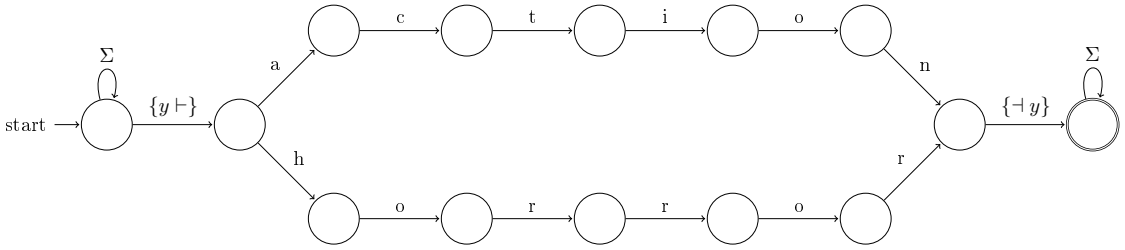
$S^{x,y,z}_{\text{followedBy}(\mathbf{min},\mathbf{max})}$ can be represented by the well-behaved eVset-automaton shown in Figure 5.6. In the system, the join operation that we are discussing is compiled exactly by following formula 5.1. More precisely, we have that:

- A and B are encoded by members of VA_{WSet} (no string equality selections are possible since they are unary);
- $S^{x,y,z}_{\text{followedBy}(\mathbf{min},\mathbf{max})}$ is represented by the automaton of Figure 5.6;
- the operators π and $\bowtie$ are simulated with the constructions described in Section 4.2 of the previous chapter.

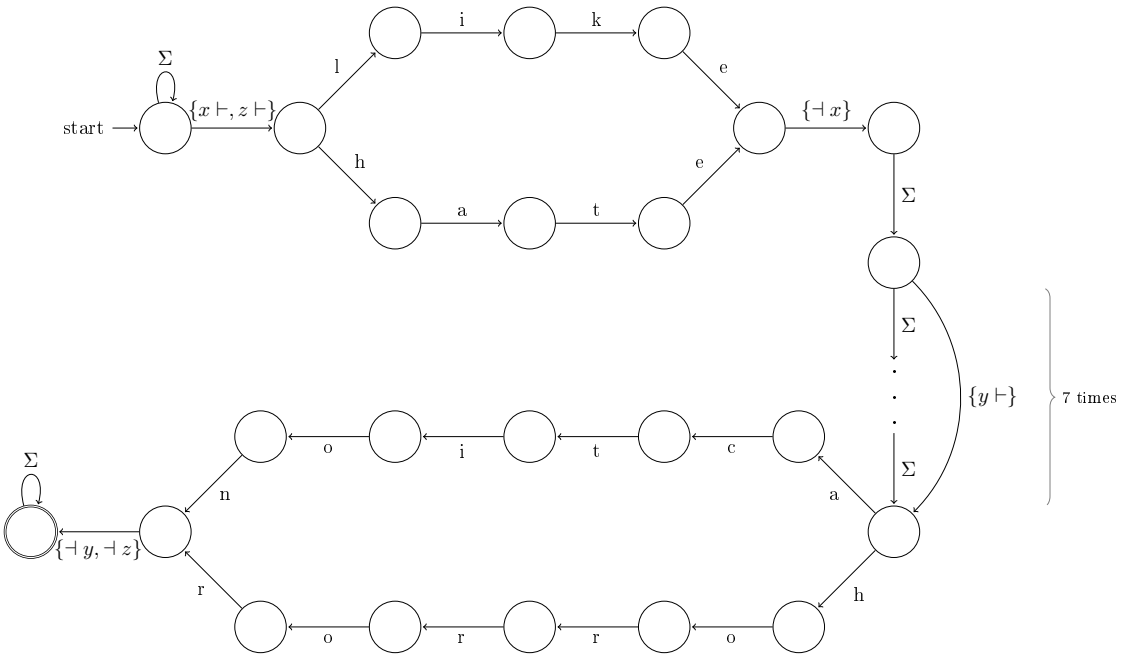
Notice that this construction only works if A and B support unanchored matching, i.e., the corresponding eVset-automata have universal loop transitions on their initial and accepting states. A concrete example of the described join follows.



(a) A well-behaved eVset-automaton A, accepting the words 'like' and 'hate'.



(b) A well-behaved eVset-automaton B, accepting the words 'action' and 'horror'.



(c) A well-behaved eVset-automaton C, with $[\![C]\!] = \pi_{\{z\}} \left([\![A]\!] \bowtie S^{x,y,z}_{\text{followedBy}(1,8)} \bowtie [\![B]\!] \right)$.

Figure 5.7 – The join of two automata A and B based on the predicate followedBy $(1, 8)$.

String s

	I	_	l	i	k	e	_	m	a	n	y	_	a	c	t	i	o	n	_	m	o	v	i	e	s	,	_	b	u	t
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	

	_	I	_	h	a	t	e	_	a	l	l	_	h	o	r	r	o	r	_	f	l	i	c	k	s	.	
31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56			

$[\![A]\!](\mathbf{s})$

	x
μ_1	$[3,7\rangle$
μ_2	$[34,38\rangle$

$[\![B]\!](\mathbf{s})$

	y
μ_3	$[13,19\rangle$
μ_4	$[43,49\rangle$

$[\![C]\!](\mathbf{s})$

	z
μ_5	$[3,19\rangle$
μ_6	$[34,49\rangle$

Figure 5.8 – The results of the execution of the automata A, B and C from Figure 5.7 on a string $\mathbf{s}$.

Example 5.5: *Consider the well-behaved eVset-automata A from Figure 5.7a and B from Figure 5.7b. A accepts the words* 'like' *and* 'hate'*, while B accepts* 'action' *and* 'horror'*. Notice that they both support unanchored matching. The well-behaved eVset-automaton C from Figure 5.7c simulates the join of the spanners represented by A and B, based on the predicate followedBy$(1,8)$. We have that* $[\![C]\!] = \pi_{\{z\}}\left([\![A]\!] \bowtie S^{x,y,z}_{\text{followedBy}(1,8)} \bowtie [\![B]\!]\right)$*. Figure 5.8 shows the results of the execution of the mentioned automata on a string* $\mathbf{s}$*. A spans the tuples μ_1 and μ_2. We have that $\mu_1(x) = [3,7\rangle$, corresponding to the snippet* 'like' *and $\mu_2(x) = [34,38\rangle$, corresponding to* 'hate'*. Similarly, the tuples μ_3 and μ_4, spanned by B, contain the spans corresponding to* 'action' *and* 'horror'*, respectively. C spans μ_5 and μ_6, that correspond, respectively, to the snippets* 'like_many_action' *and* 'hate_all_horror'*. Thus, we have that* $[\![C]\!](\mathbf{s}) = \pi_{\{z\}}\left([\![A]\!](\mathbf{s}) \bowtie S^{x,y,z}_{\text{followedBy}(1,8)}(\mathbf{s}) \bowtie [\![B]\!](\mathbf{s})\right)$*. Notice how the distance constraint in the join of A and B allows to join only the spans that are semantically correlated, avoiding to include in $[\![C]\!](\mathbf{s})$ the tuple μ with $\mu(z) = [3,49\rangle$, corresponding to the snippet* 'like_many... all_horror'*, which would have been present if the join had been realized as in Example 3.8 from Subsection 3.3.3, where the* followedBy (**min**,**max**)*predicate is not used.*

5.2.2 Evaluation Mode

The evaluation mode requires as input a core spanner representation and the path of the directory containing the text documents to span. For each of these documents, an output span relation, resulting from the evaluation of the input spanner, is returned. The process is illustrated in Figure 5.4b. The execution algorithm, represented by Algorithm 5.1, is similar to that of SystemT (see Algorithm 3.1 in Subsection 3.1.3). In particular, Algorithm 5.1 conforms to document-at-a-time processing as well. The main difference with SystemT is that the results are written to a separate file, instead of annotating the input documents.

Algorithm 5.1 The execution algorithm of a core spanner representation on a set of input documents.

$E \longleftarrow$ {core spanner representation}
for document in inputDirectory do
begin

1. {Read document into main memory}
2. $R \longleftarrow E$ (document)
3. {Write R to disk}

end

The engine that evaluates a spanner representation on a document draws inspiration from the Thompson approach, described in Subsection 5.1. The procedure defined by this approach cannot be used as is. The reason is that, when evaluating an eVset-automaton A on a string $\mathbf{s}$, we are interested in all the $\mathbf{s}$-tuples defined by the runs belonging to $\text{ARuns}(A, \mathbf{s})$, instead of a single one. Therefore, the implementation of the engine does not discard any of the feasible runs in $\text{ARuns}(A, \mathbf{s})$.

Initially, the implementation followed the Virtual Machine method: a core spanner representation was converted into an NFA program and executed by a modified implementation of the virtual machine. However this implementation revealed itself to be slow. Thus, the engine was changed to use input eVset-automata directly: in this implementation, a set of current runs is maintained, and each run is advanced by following the outgoing transitions of its current state. All the runs are advanced in lockstep. There's another difference with the original Thompson approach: operation transitions, which may be considered as a special kind of ϵ-transitions, are not followed iteratively, thus different runs may point to different positions of the input string at the same moment.

The main motivation for developing this system was to look for evidence of performance benefits over the algebraic approach of SystemT. Thus, after the system was finished, I developed a set of AQL queries to be executed on a specific text corpus. The details will be given in the next chapter. I wanted to compare the performance of the evaluation approach described in this dissertation with the algebraic approach. Therefore, I developed a subsystem that takes the approach of SystemT. I did not use SystemT directly because I wanted to ensure a genuine comparison, that would provide insight on the differences of the two approaches considered. I believe that my subsystem manages to do so, for two reasons. First, it uses the same evaluation engine of the runtime system to execute basic span extractors in a query. Moreover, it does not include all the optimization techniques that SystemT has. Thus, it eliminates any possible factor that could interfere with a true

comparison. I describe the subsystem in the next subsection.

5.2.2.1 A Subsystem for the Algebraic Approach

The subsystem works by following Algorithm 5.1 as well, but it executes AQL query representations directly, instead of spanner representations. The execution of an AQL query representation is as follows: the input spanners it points to are evaluated with the engine I have just described, then the operations contained in the query are performed. Each supported algebraic operator has an implementation that takes one or more $(V, \mathbf{s})$-relations as input and returns a new $(V, \mathbf{s})$-relation. The details of the implementations are not very interesting, except for two cases: natural join and the special join combining spans based on their relative distance. In the first case, I used a hash-join algorithm, with the hash function assigning values to the tuples of the input $(V, \mathbf{s})$-relations based on the values of the common variables. This choice was made for two reasons: it was particularly easy to realize and hash-join algorithms generally have better performance than more generic join algorithms, as, e.g., nested-loop join algorithms. For the second case, another quite suitable implementation was chosen, inspired by sort-merge join algorithms. Again, this family of algorithms is in general preferable to nested-loop joins. In order to describe the concrete algorithm, let us consider two input unary relations R_1 and R_2, assigning spans to variables x and y, respectively. Let us assume that R_1 is the left argument to the join and R_2 is the right one. Then the operation is performed according to Algorithm 5.2.

This chapter described the details of the implementation of the runtime system. The next chapter describes the experiments that were made with it and the results that were obtained.

Algorithm 5.2 Algorithm to join R_1 and R_2 according to the followedBy(min, max) predicate.

$L_1 \leftarrow$a list containing the elements of R_1 sorted by second index of the spans assigned to x
$L_2 \leftarrow$a list containing the elements of R_2 sorted by first index of the spans assigned to y
$R_3 \leftarrow \emptyset$
while $|L_1| > 0$ **and** $|L_2| > 0$ **do**
  while $|L_2| > 0$ **do**
    $[i_b, i_e\rangle \leftarrow L_1$.head
    $[j_b, j_e\rangle \leftarrow L_2$.head
    if $\min \leq j_b - i_e \leq \max$ **then**
      $R_3 \leftarrow R_3 \cup \{[i_b, j_e\rangle\}$
      $L_2 \leftarrow L_2$.tail
    else
      break
    end if
  end while
  if $|L_1| > 0$ **and** $|L_2| > 0$ **then**
    $[i_b, i_e\rangle \leftarrow L_1$.head
    $[j_b, j_e\rangle \leftarrow L_2$.head
    if $j_b - i_e < \min$ **then**
      $L_2 \leftarrow L_2$.tail
    else
      $L_1 \leftarrow L_1$.tail
    end if
  end if
end while
return R_3

Chapter 6

Experiments

After the development of the system, an experimental validation phase followed. The experiments were designed to test whether the evaluation approach described in this paper brings performance benefits over the approach of SystemT. As mentioned, a subsystem that takes the algebraic approach used in SystemT, but evaluates basic span extractors with the same engine of the runtime system, was developed in order to make a true comparison. In particular, the subsystem does not support the optimization techniques described in [Reiss et al., 2008, Krishnamurthy et al., 2009]. The outline of the chapter follows. Section 6.1 contains a detailed discussion of the experimental setup. In Section 6.2, I discuss the results of the execution of the experiments.

6.1 Setup

I performed the experiments on a document corpus consisting of blog entries, organized in 19230 files, each containing posts from the same author. The corpus was originally proposed in [Schler et al., 2006][1]. Its total size is 806,2 MB. All the queries were run on a machine with a 2,7 GHz Intel i7-2620M processor and 4 GB of RAM. The next subsection describes the queries in detail.

6.1.1 Queries

The queries are about finding informal movie reviews in a text. They are built by combining these basic span extractors:

Action extracts verbs usually associated with movies (e.g., “watch”, “rent”);

Aspect extracts mentions of aspects of a movie (e.g., “cast”, “plot”);

[1] The corpus is available at `http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm`

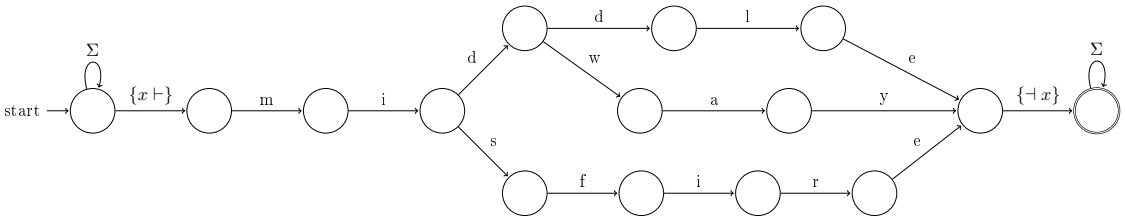


Figure 6.1 – An automaton A, that simulates a dictionary matcher.

Attribute extracts attributes (e.g., "funny", "boring");

Genre extracts mentions of the most common movie genres (e.g., "action", "comedy");

Movie extracts synonyms of the word "movie" (e.g., "flick", but also "movie" itself), and also related words like "dvd";

Name extracts digrams that potentially represent full names (e.g., "Brad Pitt");

PlotClue extracts words that hint to a synopsis (e.g., "story", "begins", "ending");

Role extracts mentions of roles both in the plot of a movie (e.g., "protagonist") and in its realization (e.g., "director");

RoleClue extracts words hinting to a description of a role (e.g., "role", "play");

Sentiment extracts verbs expressing feelings (e.g., "loved", "liked");

Title extracts titles (e.g., "The Matrix", "Annie Hall").

A well-behaved eVset-automaton representing each of these extractors was developed. Each automaton assigns the spans it matches to a single span variable, different from those of the others. The automata for **Name** and **Title** encode regular expressions that try to identify entities of interest by exploiting their usual structure, while all the other automata look for sets of fixed words, thus they simulate dictionary matchers. They are implemented as prefix trees[2]. As an example, given an eVset-automaton A, with $\mathrm{SVars}(A) = \{x\}$, that extracts the words 'middle', 'midway' and 'misfire', the structure of A is shown in Figure 6.1.

I now describe the queries, using the syntax from Subsection 3.1.4. The queries are divided in groups. The first group contains only binary joins, that can be described with the following formulas:

[2]See, e.g., `https://en.wikipedia.org/wiki/Trie`.

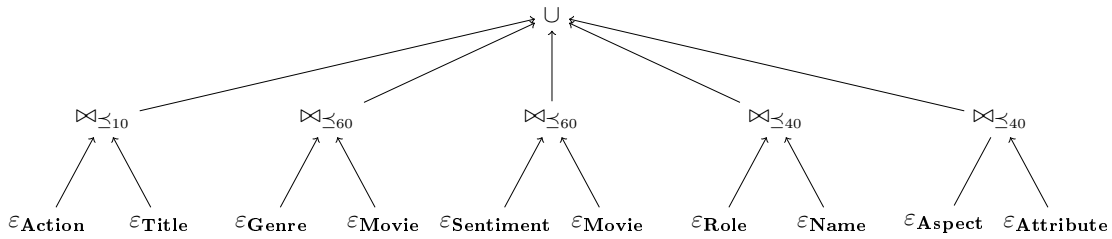


Figure 6.2 – The operator tree of query Q_6.

$$
\begin{aligned}
Q_1 &= \varepsilon_{\mathbf{Action}} \bowtie_{10} \varepsilon_{\mathbf{Title}} \\
Q_2 &= \varepsilon_{\mathbf{Attribute}} \bowtie_{60} \varepsilon_{\mathbf{Movie}} \\
Q_3 &= \varepsilon_{\mathbf{Genre}} \bowtie_{60} \varepsilon_{\mathbf{Movie}} \\
Q_4 &= \varepsilon_{\mathbf{Movie}} \bowtie_{40} \varepsilon_{\mathbf{Title}}
\end{aligned}
$$

where the ε operators were implemented by the corresponding automata and the $\bowtie_d$ operator was realized by the join specialized on the predicate "followedBy($\mathbf{min}, \mathbf{max}$)", with $\mathbf{min}= 0$ and $\mathbf{max}= d$.

The queries of the second group are unions of binary joins:

$$
\begin{aligned}
Q_5 &= Q_1 \cup Q_2 \cup (\varepsilon_{\mathbf{Role}} \bowtie_{40} \varepsilon_{\mathbf{Name}}) \\
Q_6 &= Q_1 \cup Q_3 \cup (\varepsilon_{\mathbf{Sentiment}} \bowtie_{60} \varepsilon_{\mathbf{Movie}}) \cup (\varepsilon_{\mathbf{Role}} \bowtie_{40} \varepsilon_{\mathbf{Name}}) \cup \\
&\quad (\varepsilon_{\mathbf{Aspect}} \bowtie_{40} \varepsilon_{\mathbf{Attribute}}) \\
Q_7 &= Q_6 \cup (\varepsilon_{\mathbf{Title}} \bowtie_{60} \varepsilon_{\mathbf{PlotClue}}) \\
Q_8 &= Q_7 \cup (\varepsilon_{\mathbf{Sentiment}} \bowtie_{60} \varepsilon_{\mathbf{Aspect}}) \cup (\varepsilon_{\mathbf{Name}} \bowtie_{40} \varepsilon_{\mathbf{Attribute}}) \\
Q_9 &= \beta\,(\Omega_o\,(Q_1 \cup Q_2 \cup Q_3 \cup (\varepsilon_{\mathbf{Sentiment}} \bowtie_{60} \varepsilon_{\mathbf{Movie}}) \cup (\varepsilon_{\mathbf{Role}} \bowtie_{40} \varepsilon_{\mathbf{Name}}) \cup \\
&\quad (\varepsilon_{\mathbf{Movie}} \bowtie_{60} \varepsilon_{\mathbf{Action}}) \cup (\varepsilon_{\mathbf{Title}} \bowtie_{40} \varepsilon_{\mathbf{RoleClue}}) \cup (\varepsilon_{\mathbf{Title}} \bowtie_{60} \varepsilon_{\mathbf{Movie}}) \cup \\
&\quad (\varepsilon_{\mathbf{Title}} \bowtie_{60} \varepsilon_{\mathbf{Attribute}}) \cup (\varepsilon_{\mathbf{Name}} \bowtie_{40} \varepsilon_{\mathbf{Role}}) \cup (\varepsilon_{\mathbf{Attribute}} \bowtie_{50} \varepsilon_{\mathbf{Name}}) \cup \\
&\quad (\varepsilon_{\mathbf{Sentiment}} \bowtie_{40} \varepsilon_{\mathbf{Name}}) \cup (\varepsilon_{\mathbf{Name}} \bowtie_{40} \varepsilon_{\mathbf{Attribute}}) \cup (\varepsilon_{\mathbf{Action}} \bowtie_{10} \varepsilon_{\mathbf{Name}}) \cup \\
&\quad (\varepsilon_{\mathbf{Action}} \bowtie_{40} \varepsilon_{\mathbf{Movie}}) \cup (\varepsilon_{\mathbf{Movie}} \bowtie_{60} \varepsilon_{\mathbf{PlotClue}}) \cup (\varepsilon_{\mathbf{Title}} \bowtie_{60} \varepsilon_{\mathbf{PlotClue}}) \cup \\
&\quad (\varepsilon_{\mathbf{Role}} \bowtie_{60} \varepsilon_{\mathbf{Attribute}}) \cup (\varepsilon_{\mathbf{Movie}} \bowtie_{60} \varepsilon_{\mathbf{Name}}) \cup (\varepsilon_{\mathbf{Name}} \bowtie_{60} \varepsilon_{\mathbf{Movie}}) \cup \\
&\quad (\varepsilon_{\mathbf{Attribute}} \bowtie_{60} \varepsilon_{\mathbf{Role}}) \cup (\varepsilon_{\mathbf{Movie}} \bowtie_{60} \varepsilon_{\mathbf{Title}})))
\end{aligned}
$$

The operator tree of query Q_6 is represented in Figure 6.2. As the reader can see,

queries from Q_5 to Q_9 are ordered by increasing size. Many individual joins are reused among different queries. Query Q_9 has so many joins because it was meant as a real-life query, which tries to maximize the output entities. For the same reason, The Ω_o operator, which consolidates overlapping spans, and the β operator, which identifies groups of contiguous spans, were used in the query. The β operator had the following parameters: a maximum distance between two spans of 170 characters, and a minimum number of spans per extracted group of 2. Confronted to query Q_9, all the other queries might seem incomplete, as they don't use any operators to aggregate parts of reviews. Nonetheless, such aggregation operators are not covered by the runtime system, because their relationship with core spanners was not analyzed. In fact, the runtime system borrows the implementations for these operators from the subsystem, so including them in the queries is not interesting for the purpose of comparing the two different approaches.

Another thing worth noticing in the last group of queries is that they contain progressively more duplicate instances of span extractors. We will see, in the discussion of the experimental results in the next section, that this fact affects the relative performance of the two approaches in a very important way. Let us quantify the amount of repeated span extractor instances in a query with a parameter, which we may call the *redundancy* of a query. Given a query Q, we define its redundancy by the following formula:

$$r(Q) = \frac{\#\text{span extractor instances in } Q}{\#\text{individual span extractors in } Q} - 1 \tag{6.1}$$

The higher the value of this parameter, the more a query is redundant. As the reader can verify, for any query Q that is non-redundant, we have that $r(Q) = 0$. For queries from Q_5 to Q_9, r takes these values:

- $r(Q_5) = 0$
- $r(Q_6) = \frac{1}{9} \approx 0.11$
- $r(Q_7) = \frac{2}{10} = 0.2$
- $r(Q_8) = \frac{5}{10} = 0.5$
- $r(Q_9) = \frac{34}{10} = 3.4$

The next set of queries is made of ternary joins. The reader can assume the $\bowtie_d$ operator to be left-associative.

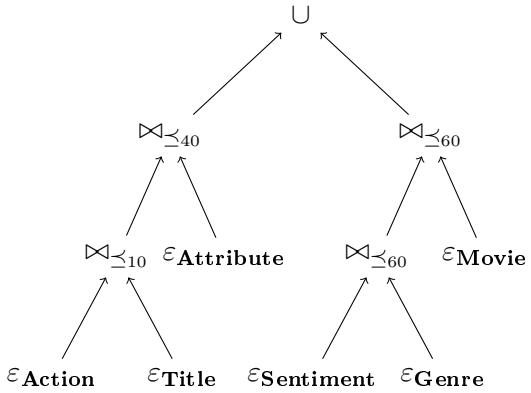


Figure 6.3 – The operator tree of query Q_{13}.

$$
\begin{aligned}
Q_{10} &= \varepsilon_{\mathbf{Action}} \bowtie_{10} \varepsilon_{\mathbf{Title}} \bowtie_{40} \varepsilon_{\mathbf{Attribute}} \\
Q_{11} &= \varepsilon_{\mathbf{Name}} \bowtie_{40} \varepsilon_{\mathbf{RoleClue}} \bowtie_{40} \varepsilon_{\mathbf{Role}} \\
Q_{12} &= \varepsilon_{\mathbf{Sentiment}} \bowtie_{60} \varepsilon_{\mathbf{Genre}} \bowtie_{60} \varepsilon_{\mathbf{Movie}}
\end{aligned}
$$

The last group of queries contains unions of ternary joins, ordered by increasing size and redundancy.

$$
\begin{aligned}
Q_{13} &= Q_{10} \cup Q_{12} \\
Q_{14} &= Q_{13} \cup Q_{11} \\
Q_{15} &= Q_{14} \cup (\varepsilon_{\mathbf{Movie}} \bowtie_{40} \varepsilon_{\mathbf{Attribute}} \bowtie_{40} \varepsilon_{\mathbf{Aspect}}) \\
Q_{16} &= Q_{15} \cup (\varepsilon_{\mathbf{Name}} \bowtie_{30} \varepsilon_{\mathbf{Attribute}} \bowtie_{30} \varepsilon_{\mathbf{Role}})
\end{aligned}
$$

The operator tree of query Q_{13} is represented in Figure 6.3. The values of the r parameter for queries Q_{13} to Q_{16} are the following:

- $r\,(Q_{13}) = 0$
- $r\,(Q_{14}) = 0$
- $r\,(Q_{15}) = 0.2$
- $r\,(Q_{16}) = 0.5$

In the next section, I discuss the results of the execution of the presented queries both with the runtime system and with the subsystem for the algebraic approach.

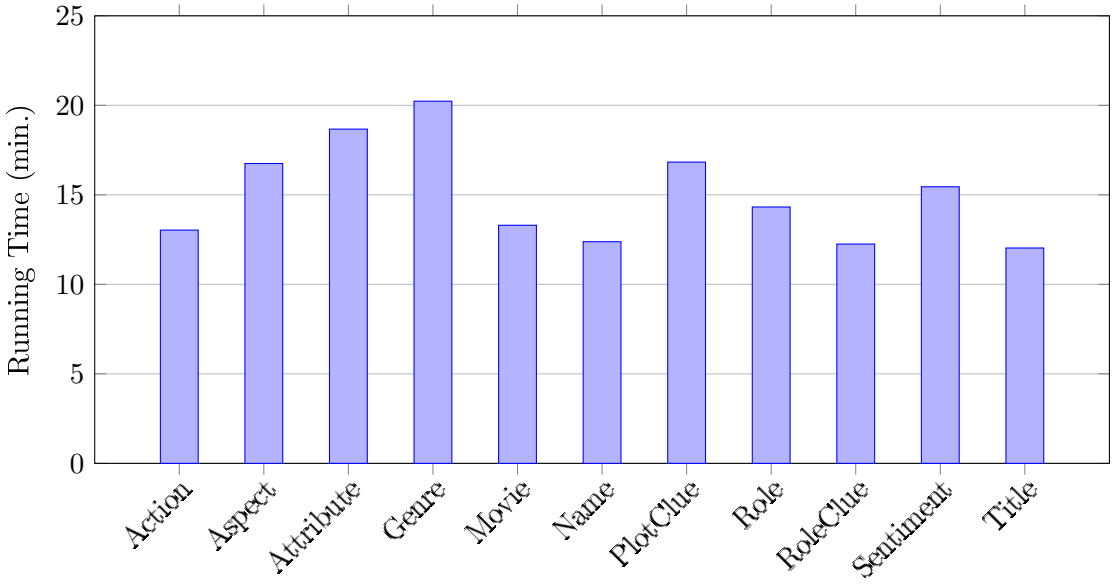


Figure 6.4 – Running times of the span extractors.

6.2 Experimental Results

The running times reported in this sections are not average values. Instead, they result from individual runs of each query. Nonetheless, I reran the queries for which there was a sensible difference between the approach described in this dissertation and the algebraic approach, with a refinement: the time needed to get the text contained in the files of the corpus was excluded. Rerunning the queries confirmed the results appearing in this section, up to the excluded I/O time and small variations due to the machine. To put the results of the experiments in the right perspective, keep in mind that the corpus that was used is composed of 19230 text files, for a total size of 806,2 MB. Before starting with the queries, let us have a look at the running times of the individual span extractors, shown in Figure 6.4. The average running time is about 15 minutes. In the case of the extractors that imitate dictionary matchers, the runtime seems to be related to the number of words they can match. As an example, the three most expensive extractors**, Aspect**, **Attribute** and **Genre**, can match respectively 12, 18, and 23 words, while two of the less expensive ones, **Action** and **RoleClue**, match 3 and 2 words, respectively. More generally, the running time of an extractor seems to be proportional to the number of runs that the corresponding eVset-automaton tries during execution.

Let us now discuss the running times of queries Q_1 to Q_4, shown in Figure 6.5. There are some useful observations we can make. First of all, the running time of a join evaluated with the algebraic subsystem roughly corresponds to the sum of the running times of the individual span extractors. This tells us that the time spent combining spans is negligible,

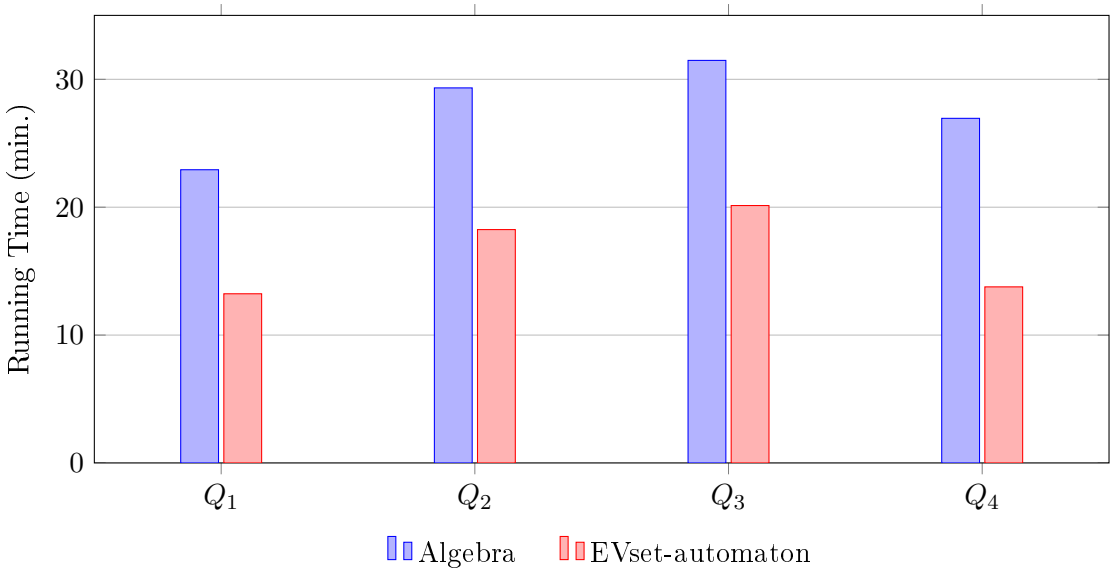


Figure 6.5 – The running times of queries Q_1 to Q_4.

compared to the time needed to evaluate span extractors. This observation is consistent with Remarks 3.1 and 3.2 from Subsection 3.1.5, and with the fact that the sizes of the files containing the snippets resulting from the evaluation of the single span extractors are very small compared to the size of the corpus, as shown in Figure 6.6. Evaluating a single join with an eVset-automaton seems to be always more efficient than using the algebraic approach. Hence, it must be that an eVset-automaton implementing the join tries less runs than the total runs of two eVset-automata corresponding to the span extractors in the join. Consider, for instance, query Q_1. During evaluation, the eVset-automaton corresponding to Q_1 will first look for an instance of an action and only if it finds one it will look for a movie title. Moreover, all the title instances that are not preceded by an action are naturally ignored. Although the finding of an action will cause the generation of 10 new runs, in order to try to match a title between 0 and 10 characters away (see Figure 5.6), this seems to be still less costly than evaluating **Action** and **Title** independently.

Figure 6.7 reports the running times of queries Q_5 to Q_9. Remember that these queries are ordered by increasing size and redundancy. In particular, redundancy plays a central role here. There seems to be a clear trend: the more a query is redundant, the less efficient will be the evaluation of the corresponding eVset-automaton. In the worst case (Q_9), this inefficiency is very marked. The explanation of this phenomenon is two-folded. On the one hand, the cost of evaluating a query with the algebraic subsystem is mostly determined by the cost of the single span extractors. This means that modifying a query with an additional join, without introducing new span extractors, is almost effortless.

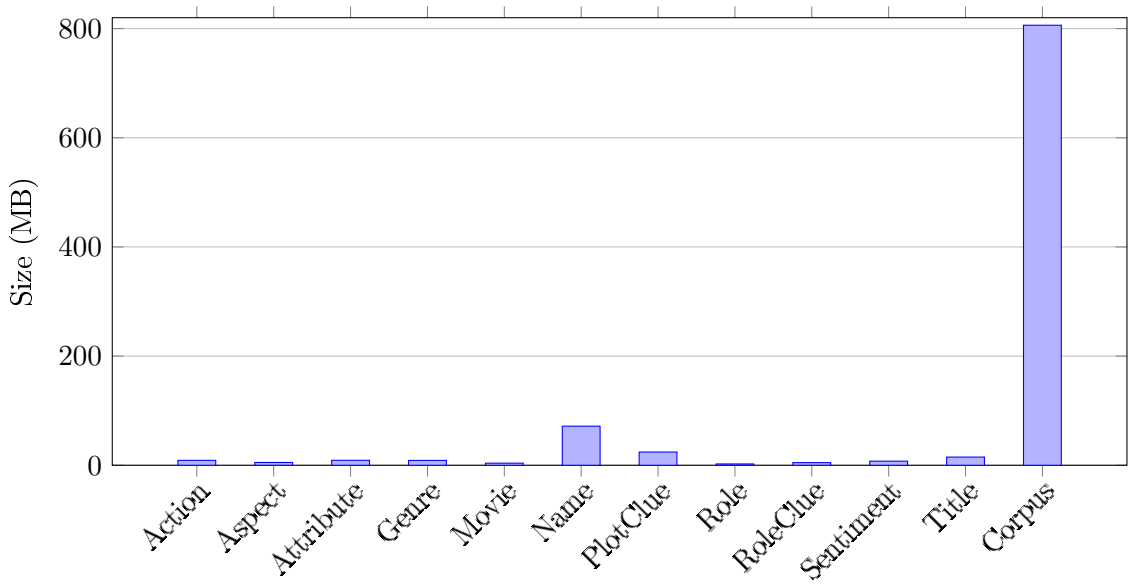


Figure 6.6 – The sizes of the outputs resulting from the execution of the span extractors, compared to the size of the corpus.

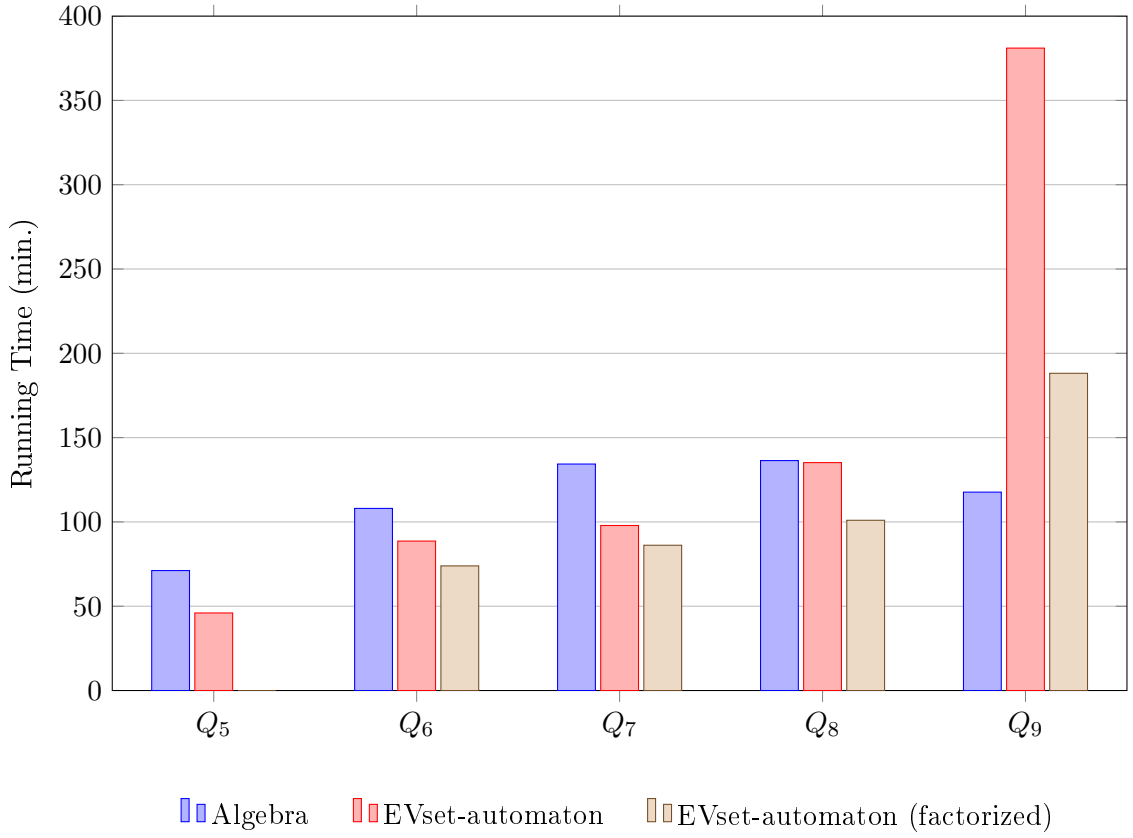


Figure 6.7 – The running times of queries Q_5 to Q_9.

Indeed, Figure 6.7 shows that the running times obtained with the algebraic subsystem benefit from redundancy, and it is not surprising that query Q_9 is not more expensive than the others. On the other hand, an eVset-automaton corresponding to any of these queries does not have the possibility to reuse the results of individual span extractors, because it matches *entire join patterns independently from each other*. Thus, the running time of an eVset-automaton is proportional to the number of joins it contains, as we can see from Figure 6.7. In general, from the point of view of the string evaluation engine, the total number of runs that are performed on an input document seems to be crucial in determining the runtime of a query. As the number of joins in the query increases, the total runs tried by the corresponding automaton increase too. These considerations suggest that one of the key strengths of SystemT is its natural ability to extract basic span relations once and use them in different parts of a query. This feature is unfortunately missing from the runtime system. Nonetheless, in general it is possible to transform a query in such a way that its redundancy is reduced, without altering its semantics. Consider the following modifications of queries Q_6 to Q_9:

$$
\begin{aligned}
Q'_6 &= Q_1 \cup ((\varepsilon_{\mathbf{Genre}} \cup \varepsilon_{\mathbf{Sentiment}}) \bowtie_{60} \varepsilon_{\mathbf{Movie}}) \cup (\varepsilon_{\mathbf{Role}} \bowtie_{40} \varepsilon_{\mathbf{Name}}) \cup \\
&\quad (\varepsilon_{\mathbf{Aspect}} \bowtie_{40} \varepsilon_{\mathbf{Attribute}}) \\
Q'_7 &= Q'_6 \cup (\varepsilon_{\mathbf{Title}} \bowtie_{60} \varepsilon_{\mathbf{PlotClue}}) \\
Q'_8 &= Q_1 \cup ((\varepsilon_{\mathbf{Genre}} \cup \varepsilon_{\mathbf{Sentiment}}) \bowtie_{60} \varepsilon_{\mathbf{Movie}}) \cup (\varepsilon_{\mathbf{Role}} \bowtie_{40} \varepsilon_{\mathbf{Name}}) \cup \\
&\quad ((\varepsilon_{\mathbf{Aspect}} \cup \varepsilon_{\mathbf{Name}}) \bowtie_{40} \varepsilon_{\mathbf{Attribute}}) \cup (\varepsilon_{\mathbf{Title}} \bowtie_{60} \varepsilon_{\mathbf{PlotClue}}) \cup \\
&\quad (\varepsilon_{\mathbf{Sentiment}} \bowtie_{60} \varepsilon_{\mathbf{Aspect}}) \\
Q'_9 &= \beta\,(\Omega_o\,((\varepsilon_{\mathbf{Movie}} \bowtie_{60} (\varepsilon_{\mathbf{Action}} \cup \varepsilon_{\mathbf{Name}} \cup \varepsilon_{\mathbf{PlotClue}} \cup \varepsilon_{\mathbf{Title}})) \cup \\
&\quad (\varepsilon_{\mathbf{Action}} \bowtie_{10} (\varepsilon_{\mathbf{Name}} \cup \varepsilon_{\mathbf{Title}})) \cup \\
&\quad ((\varepsilon_{\mathbf{Attribute}} \cup \varepsilon_{\mathbf{Genre}} \cup \varepsilon_{\mathbf{Name}} \cup \varepsilon_{\mathbf{Sentiment}} \cup \varepsilon_{\mathbf{Title}}) \bowtie_{60} \varepsilon_{\mathbf{Movie}}) \cup \\
&\quad ((\varepsilon_{\mathbf{Role}} \cup \varepsilon_{\mathbf{Title}}) \bowtie_{60} \varepsilon_{\mathbf{Attribute}}) \cup (\varepsilon_{\mathbf{Name}} \bowtie_{40} (\varepsilon_{\mathbf{Attribute}} \cup \varepsilon_{\mathbf{Role}})) \cup \\
&\quad ((\varepsilon_{\mathbf{Role}} \cup \varepsilon_{\mathbf{Sentiment}}) \bowtie_{40} \varepsilon_{\mathbf{Name}}) \cup (\varepsilon_{\mathbf{Title}} \bowtie_{40} \varepsilon_{\mathbf{RoleClue}}) \cup \\
&\quad (\varepsilon_{\mathbf{Attribute}} \bowtie_{50} \varepsilon_{\mathbf{Name}}) \cup (\varepsilon_{\mathbf{Action}} \bowtie_{40} \varepsilon_{\mathbf{Movie}}) \cup (\varepsilon_{\mathbf{Title}} \bowtie_{60} \varepsilon_{\mathbf{PlotClue}}) \cup \\
&\quad (\varepsilon_{\mathbf{Attribute}} \bowtie_{60} \varepsilon_{\mathbf{Role}})\Big)\Big)
\end{aligned}
$$

These modified versions are equivalent to the original queries, but they are less redundant:

- $r\,(Q'_6) = 0 < 0.11 \approx r\,(Q_6)$

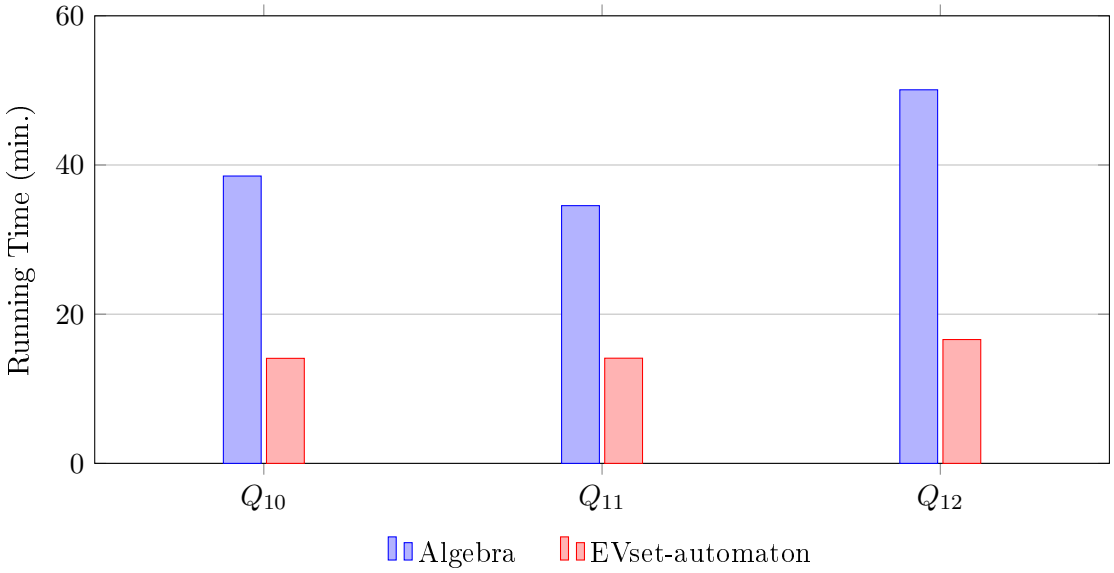


Figure 6.8 – Running times of queries Q_{10} to Q_{12}.

- $r\left(Q'_7\right) = 0.1 < 0.2 = r\left(Q_7\right)$
- $r\left(Q'_8\right) = 0.4 < 0.5 = r\left(Q_8\right)$
- $r\left(Q'_9\right) = 2.3 < 3.4 = r\left(Q_9\right)$

As we can see from Figure 6.7, the transformations have a positive effect on the running times. All they do is *factorize* the common arguments to different joins in a query, with the constraints that the distance parameter of the joins is the same, and that the factorized arguments are in the same position in the joins. Figure 6.7 suggests that this technique can bring modest to huge benefits to the performance of the runtime system, depending on the amount of redundancy that is eliminated. Nonetheless, in a generic query, it probably will not be able to eliminate all the redundancy. It is the case for query Q_9, where the running time of the runtime system becomes comparable with that of the algebraic subsystem, although it is still worse. On average, the speedup does not seem to change significantly when we apply factorization (see Figure 6.11), but the reason might be that the set of queries used to calculate the average are, in general, not very redundant. The introduction of more redundant queries could probably increase the speedup significantly.

The results for queries Q_{10} to Q_{12} are shown in Figure 6.8. Non-redundant ternary joins seem to be faster with the runtime system, and, on average, the speedup with respect to the algebraic subsystem is greater than that obtained for binary joins (see Figure 6.11). This is explained with the observation that a ternary join pattern is more restrictive than a binary one. Hence, on average, an eVset-automaton that implements a ternary join will try

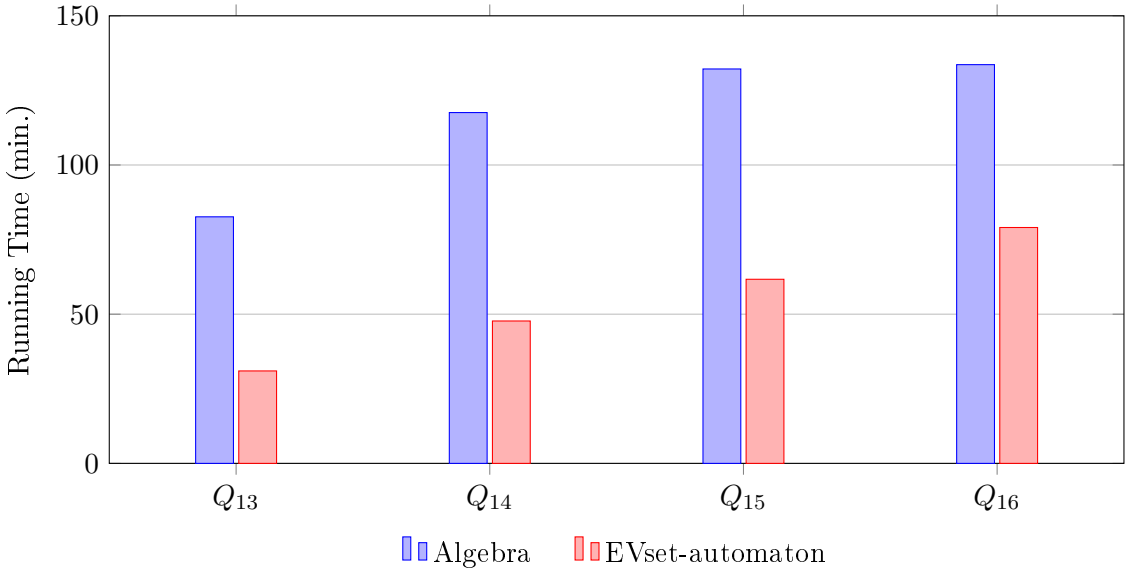


Figure 6.9 – Running times of queries Q_{13} to Q_{16}.

a smaller percentage of the total amount of runs tried by the individual span extractors.

Finally, Figure 6.9 shows the running times Q_{13} to Q_{16} (that are ordered by increasing size and redundancy). Although a query with a great redundancy is missing in this case, the examples reported suggest that the runtime system is less sensitive to redundancy, when the input queries are unions of ternary joins. For instance, queries Q_8 and Q_{16} have approximately the same size: they employ the same amount of span extractors, they contain an almost equal number of $\bowtie_d$ operators (Q_{16} has two more) and $r(Q_8) = r(Q_{16}) = 0.5$, but the speedup with respect to the algebraic approach is greater for Q_{16}.

This situation suggests a trend: the longer the chain of joins, the greater performance gain over the algebraic approach. In spite of this consideration, longer join chains were not tried, because they hardly seem useful in the context of this benchmark: they would probably be effective when the entities we want to find have a complex and, above all, repetitive structure, but this is not the case for informal movie reviews contained in blog posts, whose structure might be complex, but is in general very variable. Already for queries Q_{13} to Q_{16}, the sizes of the results are considerably smaller than, for instance, those of queries Q_5 to Q_9 (see Figure 6.10).

This benchmark highlights the key strengths of the two approaches for running AQL queries belonging to its core fragment, namely the one originally proposed in [Fagin et al., 2015] and the algebraic one ([Reiss et al., 2008, Krishnamurthy et al., 2009]). The results suggest that a runtime system based on well-behaved eVset-automata can efficiently evaluate a join pattern, and its efficiency increases as longer chains of joins are used.

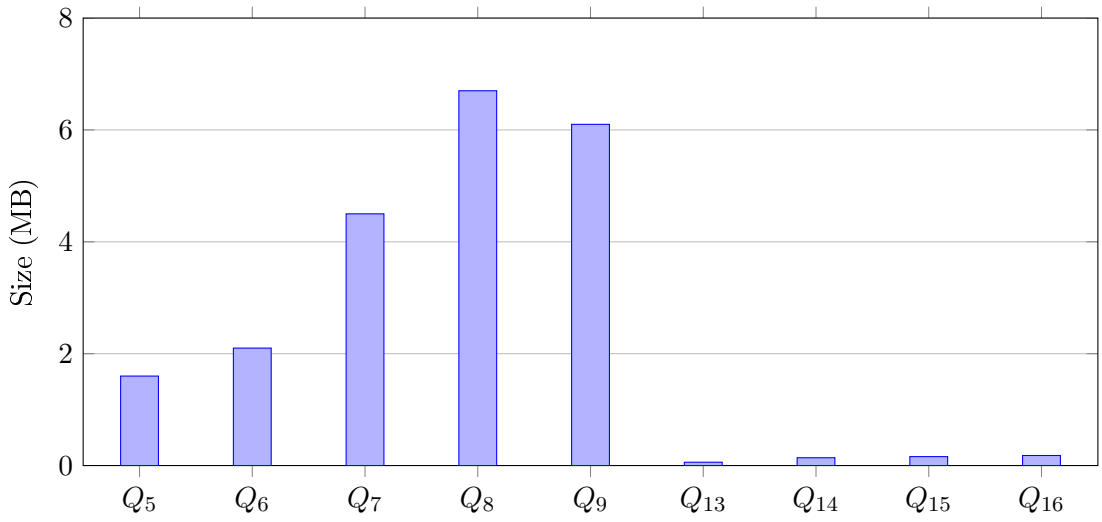


Figure 6.10 – The sizes of the outputs of queries Q_5 to Q_9 and Q_{13} to Q_{16}.

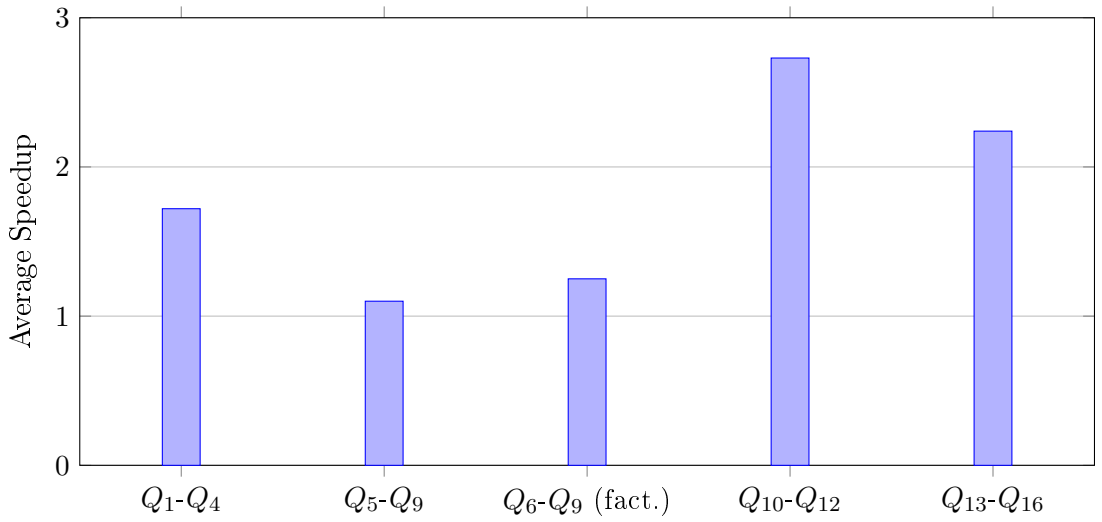


Figure 6.11 – Average speedup of the runtime system w.r.t. the algebraic subsystem for the groups of queries in the benchmark.

Nonetheless, it cannot naturally exploit the redundancy of a query, although we saw that there exist ways to reduce it. On the other hand, the ability to avoid redundant work is a core aspect of the algebraic approach, and it can make a significant difference in large queries. In general, the runtime of a query seems to be determined by the amount of runs tried by the corresponding automaton, in the case of the runtime system, and by the total runs tried by the automata contained in the query, for the algebraic subsystem.

Chapter 7

Summary and Conclusions

7.1 Summary

Drawing upon the formal model for the core fragment of AQL introduced in [Fagin et al., 2015], I have defined and developed a runtime system for AQL queries that belong to this fragment. The system is based on the concept of document spanner. It supports the execution of any core spanner. Spanners are represented into the system by means of well-behaved eVset-automata, a modified version of vset-automata (that were originally proposed in [Fagin et al., 2015]). The difference with vset-automata is that they can do more than one span capturing operation in one transition. The system actually works with a subclass of eVset-automata, which I have shown it can be used as a base for core spanner representations: well-behaved eVset-automata. We saw that this subclass has very useful properties. First of all, it exempts the evaluation engine from any runtime check on the validity of the feasible runs of an automaton. Then, it is closed under the operations of projection, union and natural join, and there exist polynomial-time constructions that allow to simulate the latter with well-behaved eVset-automata. This is an improvement over the constructions proposed in [Fagin et al., 2015], that generate exponentially larger automata. The system has a compilation module that allows to combine well-behaved eVset-automata by exploiting the constructions described in this dissertation.

After the development of the runtime system, a series of experiments were conducted with it. I developed a set of queries and I have run them on a sample text corpus. Moreover, I realized a subsystem that supports the algebraic approach described in [Reiss et al., 2008, Krishnamurthy et al., 2009]. All the queries were run with this subsystem too in order to compare the performances of the two approaches. The results show that the redundancy of a query plays a central role in determining the difference of performance of the two systems. Non redundant queries are more efficiently executed with the runtime system. However, the subsystem loses performance as the redundancy of the queries increases,

while the algebraic subsystem naturally exploits it to reduce its workload. More generally, in both cases the running time depends on the amount of runs tried by the automata evaluated on the input. Another consideration is that the speedup with respect to the algebraic subsystem increases as the matching patterns become more restrictive, as, e.g., in ternary joins.

7.2 Conclusions

One of main objectives of this thesis was to investigate the benefits of the point of view provided in [Fagin et al., 2015] over the one currently adopted by SystemT. The results highlight that the runtime system has its own advantages, although its convenience seems to depend on the context. In fact, the impact of the redundancy of a query on performance was discovered during the experimental phase, and I didn't expect it to be so relevant. A concrete system for the execution of AQL queries would probably need to mix both the execution approaches that were compared in this thesis, opportunely choosing the best one for a given (sub)query. It seems clear that one of main choice criteria would be the redundancy of a query. Another one could be the variability in the structure of the entities targeted by an extraction task. Nonetheless, I invite the reader to interpret these results as preliminary. Indeed, there is a lot of work to be performed on the system yet.

7.3 Future Work

The development of a module that transforms input queries for a more efficient execution with eVset-automata seems promising. Moreover, other benchmarks than the one presented in this paper could be conceived, in order to gain an even better understanding of the system.

Then, I believe that some of the constructions described in Section 4.2 and the specialized join with distance constraint could be further improved. For instance, a BFS (Breadth First Search) algorithm could be used for natural join, instead of a plain conditional product.

Further work could be dedicated to the evaluation engine. The execution algorithm could be optimized. Moreover, other evaluation methods exist, based on approaches that are different from the Thompson one. In particular, two of them were identified as interesting. One is based on Ordered Binary Decision Diagrams (OBDDs). In [Yang et al., 2012], it is shown how to encode NFAs as OBDDs, with support for submatch extraction. Then, a procedure for evaluating a string by using an OBDD representation of a NFA is described. Given a regular expression r of size m and a string of size n, the runtime cost

of this approach is shown to be between $O(m)$ and $O(mn)$. Another interesting engine is called Kleenex [Grathwohl et al., 2016]. It is based on the concept of *transducer*. Its creators show that it has linear-time performance in the worst case, with high throughput.

The focus of this thesis was on the comparative experimental validation, and the specific implementation of the evaluation engine is likely to be orthogonal to this task, because both the approaches that were compared would benefit from a performance increase due to it. Nonetheless, trying to adapt the mentioned engines for the system could be a useful improvement. Another possible amelioration of the engine, which was ultimately not pursued for similar reasons, would be the usage of the Scala library LMS (Lightweight Modular Staging), extensively described in [Rompf, 2012], that uses the principles of generative programming to optimize Scala source code. In particular, it would allow to produce a specialized engine for each input query, ideally boosting performance.

Bibliography

[Cowie and Wilks, 1996] Cowie, J. and Wilks, Y. (1996). Information extraction.

[Cox, 2007] Cox, R. (2007). Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...).

[Cox, 2009] Cox, R. (2009). Regular Expression Matching: the Virtual Machine Approach.

[Cox, 2010] Cox, R. (2010). Regular expression matching in the wild.

[Fagin et al., 2015] Fagin, R., Kimelfeld, B., Reiss, F., and Vansummeren, S. (2015). Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12:1–12:51.

[Grathwohl et al., 2016] Grathwohl, B. B., Henglein, F., Rasmussen, U. T., Søholm, K. A., and Tørholm, S. P. (2016). Kleenex: Compiling nondeterministic transducers to deterministic streaming transducers. *SIGPLAN Not.*, 51(1):284–297.

[Krishnamurthy et al., 2009] Krishnamurthy, R., Li, Y., Raghavan, S., Reiss, F., Vaithyanathan, S., and Zhu, H. (2009). Systemt: A system for declarative information extraction. *SIGMOD Rec.*, 37(4):7–13.

[Reiss et al., 2008] Reiss, F., Raghavan, S., Krishnamurthy, R., Zhu, H., and Vaithyanathan, S. (2008). An algebraic approach to rule-based information extraction. In *2008 IEEE 24th International Conference on Data Engineering*, pages 933–942.

[Rompf, 2012] Rompf, T. (2012). *Lightweight Modular Staging and Embedded Compilers*. PhD thesis, IC, Lausanne.

[Sarawagi, 2008] Sarawagi, S. (2008). Information extraction. *Found. Trends databases*, 1(3):261–377.

[Schler et al., 2006] Schler, J., Koppel, M., Argamon, S., and Pennebaker, J. (2006). *Effects of age and gender on blogging*, volume SS-06-03, pages 191–197.

[Shen et al., 2007] Shen, W., Doan, A., Naughton, J. F., and Ramakrishnan, R. (2007). Declarative information extraction using datalog with embedded extraction predicates. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1033–1044. VLDB Endowment.

[Yang et al., 2012] Yang, L., Manadhata, P., Horne, W., Rao, P., and Ganapathy, V. (2012). Fast submatch extraction using obdds. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 163–174, New York, NY, USA. ACM.