Maastricht University

universiteit hasselt

KNOWLEDGE IN ACTION

Faculty of Sciences

Topics in Data Mining: Pattern Enumeration, XML Key Inference and Big Data Query Optimization

Doctoral dissertation submitted to obtain the degree of
Doctor of Science: Information Technology, to be defended by

Jonny Daenen

Promoter: Prof. Dr Frank Neven
Co-promoters: Prof. Dr Tony Tan
Prof. Dr Jan Van den Bussche

September 16th, 2016

D/2016/2451/49

"*To infinity … and beyond!*"

— Buzz Lightyear

Acknowledgments

During the past seven years, I had the opportunity to explore one of the most interesting fields in computer science: data mining. I was given the chance to participate in academic research, discovered my passion for teaching and picked up a lot of extra's along the way: organizing events, teaching people how to program, building robots with LEGO, explaining my research to a broad audience, etc. All of this could not have been accomplished without the support I received from numerous people.

First and foremost, I would like to thank my advisor Frank Neven, for being my guide on this journey and for the general discussions related to "productivity" tools and Apple products. I thank my co-advisors Jan Van den Bussche, whose enthusiasm got me excited about theoretical computer science, and Tony Tan, for allowing me to take part in distributed systems research, which sparked (pun intended) my own enthusiasm. Special thanks go to Stijn Vansummeren and to Martín Ugarte for the enriching collaborations that resulted in several important insights.

I would like to extend my gratitude to my colleagues and all people I have cooperated with, especially those of the InfoLab (DBTI) research group, who helped at making this an unforgettable experience—especially during the travels. Extra credits go to (in order of appearance) Joris Gillis, Robert Brijder, Jelle Hellings and Bas 'The Boss' Ketsman, who helped me figuring out results, supplied writing tips and found various ways of distracting me; and to Jeroen Dello, who was always thrilled to talk about anything.

I'm especially thankful towards my parents Benny Daenen and Justine Vandewauw for allowing me to make my own decisions and supporting me in all possible ways, and towards my whole family as they motivated me and were always ready to help. Towards my friends for being there, for providing moments of relaxation and distraction, and for showing interest in whatever strange projects I was doing. Mark Fontaine, my former guitar teacher, deserves a special mention here, as he planted the seed from which my interest in computer science has grown.

Finally, I would like to thank my girlfriend Michelle Gybels for her patience and understanding, cheering me up when necessary, acting as a sounding board for my thoughts, and most of all for being there.

There is not enough space here to encode the entire dataset of those who have supported me. Hence, to everyone who has contributed in a direct or indirect way to this work: Bedankt!

Diepenbeek, September 2016

Abstract

In this work, we identify three challenging subtopics in regard to optimizing Big data mining workflows. First, we focus on pattern mining and investigate the problem of enumerating string patterns described by a context-free language. We derive guarantees on the delay between generated items when using a naive algorithm. Our results contribute to the foundational aspects of computer science and provide a basis for obtaining similar guarantees in more complex enumeration problems. The second topic remains in the domain of pattern mining: we study the pattern mining problem applied to XML keys. We discuss the complexity of several important decision problems and devise an algorithm for discovering XML keys from a given set of XML data. The presented algorithm leverages previous results from search space exploration and relational key mining and is experimentally validated. For our final topic, we shift our attention to Big data mining, where query engines answer questions about data that exceeds the capacity of traditional relational database systems. To construct answers within a reasonable amount of time, we focus on parallel evaluation. We present a two-tiered strategy for optimizing query plans for a collection of strictly guarded fragment queries. The nature of these queries allows for a low-cost MapReduce evaluation (in terms of total and net time) that takes up to two rounds per subquery. We provide an implementation in our system called Gumbo and extensively compare it to existing systems.

Samenvatting

Tijdens de laatste decennia merken we een enorme toename op van de hoeveelheid data die verzameld, bijgehouden en verwerkt moet worden. De lage prijzen van zowel sensoren, rekenkracht en opslagcapaciteit, alsook de toename van gebruikersgegenereerde content hebben hier mede voor gezorgd. Deze grote hoeveelheid beschikbare data kan gebruikt worden om nieuwe inzichten te verkrijgen die bijvoorbeeld belangrijke beslissingen kunnen ondersteunen. Het ontdekken en extraheren van deze informatie of inzichten noemt men *data mining* en situeert zich binnen het domein van *knowledge discovery.*

Omdat het handmatig verwerken van grote hoeveelheden informatie vaak niet haalbaar is, wegens te tijdrovend, worden computers ingezet om data aan hoge snelheden te verwerken. Echter zijn zowel de processorsnelheid en leessnelheid van harde schijven onvoldoende meegeëvolueerd met de opslagcapaciteit. Daarom moeten we op zoek naar nieuwe technologieën en algoritmen die snelle data-analyses mogelijk maken. Niet alleen moeten we de eigenschappen en limieten van dergelijke technologieën en algoritmen goed begrijpen, we dienen hierbij ook rekening te houden met de karakteristieken van moderne data. Deze zogenaamde "Big data" wordt vaak omschreven met termen als velocity (snelheid), variety (diversiteit) en volume (grootte), al wordt tegenwoordig veracity (accuraatheid) als vierde "V" aangeduid [67, 104]. Elk van deze karakteristieken komt samen met nieuwe uitdagingen m.b.t. data management en analyse.

In dit werk bestuderen we enkele kernproblemen binnen het minen van data. De eerste twee problemen situeren zich in de context van pattern mining, terwijl het laatste onderwerp de nadruk legt op query evaluatie en optimalisatie binnen het gebied van Big data.

Enumeratie van Patronen

Een interessante data mining techniek is *pattern mining*, waar het de bedoeling is om interessante patronen te ontdekken in een berg data. Eén van de meest eenvoudige manieren om dit probleem op te lossen bestaat eruit alle mogelijke

patronen één voor één op te sommen en voor elk van hen te testen of het voorkomt in de data. Dit is een vrij naïeve techniek, maar wel één die in de meeste gevallen gebruikt kan worden. Een centraal probleem binnen deze aanpak is het opsommen van patronen. In deze thesis gaan we na hoe zulke patronen opgesomd kunnen worden indien deze voorhanden zijn in de vorm van een contextvrije grammatica. Een contextvrije grammatica beschrijft een (mogelijk oneindige) verzameling strings en kan dienen om de voorkeur voor bepaalde patronen uit te drukken.

We tonen aan dat een naïef algoritme voor het enumereren van de strings in een contextvrije taal voldoet aan de *incremental polynomial time* (IPT) eigenschap. Dit houdt in dat de tijd tussen een gegenereerd patroon n en $n+1$ begrensd wordt door $p(n)$, waarbij p een veelterm is die afhankelijk is van de grammatica. Dit is een vrij verrassend resultaat aangezien we geen beperkingen leggen op de grammatica zelf. Een tweede resultaat bestaat uit het bewijs dat bestaande algoritmes voor het opsommen van strings met een gegeven lengte (zoals dat van Dömösi [72]) eenvoudig omgevormd kunnen worden naar een IPT-algoritme voor de gehele taal.

De bekomen resultaten zijn enerzijds van een didactisch nut, maar de gebruikte aanpak en bewijstechnieken dienen ook als basis voor het afleiden van gelijkaardige eigenschappen voor meer complexe talen. Hierbij denken we aan boom- of graaftalen [61], dewelke ingezet kunnen worden voor het beschrijven van meer complexe patronen.

XML Key Mining

Het tweede onderwerp gaat dieper in op pattern mining, maar dit keer met als doel het ontdekken van alle XML-keys in gegeven XML-documenten. XML-documenten zijn interessant omdat ze een flexibeler datamodel hanteren (semi-gestructureerd) dan hetgeen we terugvinden in relationele databases (gestructureerd). Dit houdt in dat de structuur van XML in de data zelf aanwezig is, in plaats van dat deze vooraf opgelegd wordt. Deze flexibiliteit heeft van XML een veelgebruikt datatype gemaakt voor zowel het uitwisselen als het opslaan van data.

Omdat structuur in sommige situaties toch vereist is (optimalisaties [51], verificatie en validatie [38]) zijn er schematalen zoals *XML Schema Definitions* (XSDs) [161] in het leven geroepen. Deze laten toe om beperkingen te leggen op de toegelaten structuur van XML-documenten. Een weinig gebruikt mechanisme dat reeds aanwezig is in XSD's zijn XML-keys. In tegenstelling tot in het relationele model is er slechts weinig onderzoek verricht naar XML-keys in de aanwezigheid van een schema. Ons werk brengt hier verandering in.

XML-keys laten toe om in een XML-document aan te geven welke nodes uniek geïdentificeerd moeten worden. Beschouw bijvoorbeeld een XML-document dat bestellingen van boeken bevat. We kunnen dan uitdrukken dat ieder boek binnen een bestelling uniek geïdentificeerd kan worden door de combinatie van zijn titel- en jaar-veld. Merk op dat we niets zeggen over de aanwezigheid van hetzelfde boek in *verschillende* bestellingen. Het is hierbij wel belangrijk dat deze laatste velden (titel en jaar) gegarandeerd aanwezig zijn, en maximaal éénmaal voorkomen.[1]

We geven een algoritme dat alle XML-keys kan afleiden uit een gegeven document en tonen aan dat dit bruikbaar is in de praktijk. Verder bestuderen we enkele belangrijke beslissingsproblemen die de kwaliteit van de gevonden keys moet verhogen, daar de kwaliteit van resultaten een typisch probleem is binnen data mining. Het belangrijkste probleem dat we onderzoeken is dat van consistentie (*consistency*): zal een key op elk document dat voldoet aan het schema altijd de juiste velden selecteren? We tonen aan dat dit probleem zich in de complexiteitsklasse PTIME bevindt.

Ons inferentiealgoritme steunt op reeds bestaande algoritmen die toelaten om de zoekruimte efficiënt af te lopen en om de geldigheid van relationele keys na te gaan. Het kan verder uitgebreid worden naar keys die *bijna* gelden (zogenaamde *approximate keys*) en kan mogelijk dienen als basis voor keys in graafdata, een vorm van data die de laatste jaren steeds populairder geworden is door o.a. de opkomst van sociale netwerken.

Big Data Query Optimalisatie

In dit laatste deel bestuderen we de parallelle evaluatie van *strictly guarded fragment queries*. We maken hiervoor gebruik van het MapReduce programmeermodel, dat eenvoudige parallellisatie toelaat op een cluster van alledaagse machines.

Er zijn reeds heel wat systemen op de markt die trachten om te gaan met de karakteristieken van moderne data. Een belangrijke techniek is het parallelliseren van berekeningen zodat de kost gedeeld kan worden over verschillende machines en er eenvoudig meer rekenkracht toegevoegd kan worden. Vaak worden hiervoor MapReduce systemen zoals Hadoop gebruikt, met daar bovenop een extra query interface voor de gebruiker. De queries die worden ingegeven worden typisch vertaald naar het onderliggende MapReduce systeem. Echter, voor queries die join-operaties bevatten, of queries waarbij de data niet mooi verdeeld kan worden over de machines levert dit niet altijd snelle uitvoeringstijden omdat sommige machines meer werk moeten verrichten.

Wij bestuderen een variant op de relationele algebra (SQL) waarbij we joins in essentie vervangen door semi-joins: *strictly guarded fragment queries*

[1]Bijgevolg dienen de velden *precies* één keer voor te komen voor ieder boek.

of SGF-queries. Een semi-join $R \ltimes S$ heeft als resultaat de R-tupels die voorkomen in de join $R \bowtie S$. De reden voor deze keuze is dat veel queries nog steeds uitgedrukt kunnen worden in deze beperkte taal en dat join-queries in bepaalde gevallen versneld kunnen worden door eerst een semi-join query uit te voeren. SGF-queries laten toe om parallele queryplannen te construeren en om in bepaalde gevallen de totale resource-kost van de queries (hetgeen evenredig is met de prijs van een query) significant te verlagen. Tezamen zorgt dit voor snelle resultaten aan een lage kostprijs in een vrij flexibele querytaal.

We presenteren een vernieuwd kostmodel voor MapReduce, dat ons in staat stelt om de kost van een berekening in te schatten en gebaseerd hierop ontwikkelen we een tweedelig algoritme dat een optimaal MapReduce query plan berekent. We tonen aan dat het optimalisatieprobleem NP-compleet is en geven de nodige heuristieken. Uit onze experimentele studie blijkt dat onze heuristieken goed werken, dat onze optimalisaties hun dienst bewijzen en dat ons systeem (Gumbo) efficiënter is dan de bestaande systemen Pig en Hive wanneer het gaat om SGF queries.

Gumbo kan uitgebreid worden op verschillende vlakken waaronder ondersteuning voor specifieke query elementen, maar kan ook dienen als subsysteem voor het evalueren van SGF queries in bestaande systemen. De optimalisaties die we voorstellen kunnen nog uitgebreid geëvalueerd worden en kunnen mogelijk dienen voor het optimaliseren van andere MapReduce programma's, zelfs buiten de context van dit werk.

Discussie

De deelonderwerpen behandeld in dit werk kaderen allemaal binnen data mining. We hebben zowel theoretische als praktische resultaten gegeven, alsook bruikbare implementaties hiervan. We zijn van mening dat dergelijke bijdragen zeer nuttig zijn in het domein van informatica en zeker binnen Big data, aangezien ze enerzijds meer inzicht geven in de beschouwde problemen, maar anderzijds ook oplossingen aanreiken die gebruikt kunnen worden in een praktische context.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

As we are recording millions of petabytes every day [63], storing and analysing this information has become a major challenge. While the data itself is broadly available, we need technology that facilitates dealing with it on a daily basis and methodologies and algorithms that can extract the most valuable parts [104]. In this work, we identify several important challenges in knowledge discovery and provide theoretical and practical results that contribute to both the algorithmic and the technological aspect. This results into usable data mining tools of which the capabilities are fully understood and which can be readily plugged into modern Big data mining workflows.

1.1 Knowledge Discovery & Data Mining

Maletic and Marcus [124] give the following clear description of knowledge discovery:

> *Knowledge Discovery in Databases (KDD) is an automatic, exploratory analysis and modeling of large data repositories. KDD is the organized process of identifying valid, novel, useful, and understandable patterns from large and complex data sets.*

Table 1.1 shows the major steps that are involved in this KDD process. Each of the steps in this pipeline comes with its own challenges, but we focus on the selection and employment of the data mining algorithm, which is a crucial part of this pipeline [124]:

Step	Description
1	Developing an understanding of the application domain
2	Selecting and creating a data set
3	Preprocessing and cleansing
4	Data transformation
5	Choosing the appropriate Data Mining task
6	Choosing the Data Mining algorithm
7	Employing the Data Mining algorithm
8	Evaluation
9	Using the discovered knowledge

Table 1.1: *Nine steps of the KDD process [124].*

Data Mining (DM) is the core of the KDD process, involving the inferring of algorithms that explore the data, develop the model and discover previously unknown patterns. The model is used for understanding phenomena from the data, analysis and prediction.

Data Mining tasks can be subdivided into two major categories: verification and discovery. The goal of the first is mainly to confirm a hypothesis, where the latter aims at finding *new* insights, i.e., to actually construct a hypothesis from the data. There are many different techniques available and each of them aims to discover other concepts in/of the data. For example, there are classification algorithms that aim to learn how to label new data items, there is association rule mining where connections between different features are exposed, there is clustering, where groups of similar data items are grouped together, there is numeric prediction that allows us to estimate numeric values, etc.

When a specific data mining task is chosen, a suitable algorithm and implementation needs to be selected to perform the actual analysis. It is paramount to be aware of the properties of the algorithm at hand in order to understand its output and to estimate the time needed for the analysis. The latter is of major importance in situations where time is a crucial factor. This illustrates the need for a thorough understanding of the limitations of the algorithms and the guarantees that they provide.

1.1.1 A Pattern Mining Workflow

A certain class of data mining algorithms involves finding interesting *patterns* in a dataset that can be used to understand or describe it. The term 'pattern' should be considered in a very broad sense, as it can mean a diversity

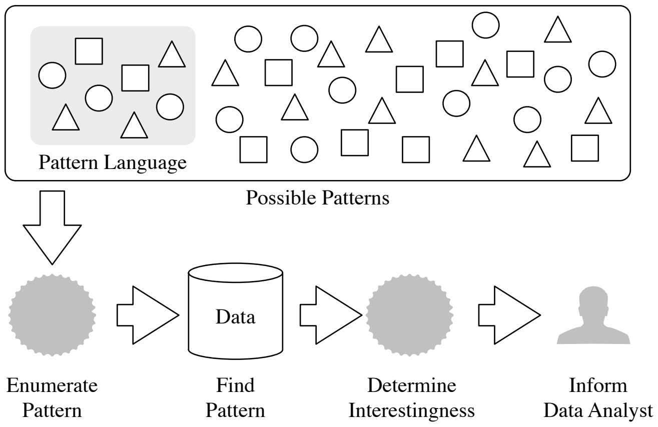


Figure 1.1: *A basic workflow for pattern mining problems.*

of things. In general, a pattern is a potential feature of a dataset; some examples are: an association rule expressing that two products are often bought together [13], a subnetwork on Facebook that matches the structure of a criminal organization [77] or a decision tree that classifies job candidates. We focus on the abstract problem of pattern mining, a problem that involves generating candidate patterns, and testing whether they appear in a given dataset. We zoom in on a selection of interesting theoretical and practical challenges. The following is a very basic 'generate-and-test' workflow that tackles the problem of finding patterns in a dataset:

1. generate candidate pattern;
2. check whether the pattern appears in the dataset;
3. if the pattern appears interesting, output it;
4. repeat.

This workflow is visualized in Figure 1.1. We can already identify the important components of this framework:

- defining and generating the pattern space;
- finding a method to test whether the pattern is present; and,
- deciding when a pattern should be considered interesting.

We study the following different subproblems, where each of them provides insight in one or more of the components of the generic framework presented above:

- generating candidate patterns with speed guarantees from a user-defined pattern language; and
- identifying high-quality, unique substructures in semi-structured data.

In the remainder of this section, we provide a top-level view of these two problems and provide a motivation on why they are interesting.

1.1.2 Pattern Enumeration Speed

In the generate-and-test method described above, the search space is defined by the user, and all potentially interesting patterns must be enumerated. As exploring the entire search space is often not practically feasible, pruning optimizations are often required to speed up the discovery process. An example of this approach can be seen in the well-known Apriori algorithm [13]. In other, more data-driven methods, the data itself is used to construct a search space, as is done in the FP-growth algorithm [97].

In Chapter 2, we study the problem of generating a set of candidate patterns based on directives from the user, and characterize the speed of the enumeration process. We limit our attention to string pattern languages that are described by a context-free grammar and study a naive enumeration algorithm, which iteratively constructs new strings from smaller, intermediate strings. We show that even the naive algorithm exhibits the incremental polynomial time (IPT) property, which guarantees that the time between two generated patterns is bounded by a polynomial in the number of already generated strings. Surprisingly, as we pose only a few restrictions on the form of the grammar, the result even holds for ambiguous grammars where patterns may have different internal representations.

This result on strings is a fundamental one and can be used to derive bounds for other pattern description formalisms that describe more complex properties. For example, when mining graph data, a pattern language can be given in the form of a *graph language* that describes a set of interesting graphs. Indeed, Costa Florêncio et al. [61] give an example of how a context-free language of graph expressions can be used to enumerate classes of graphs corresponding to, e.g., cycles and trees. The results can also be directly applied to data mining scenarios such as sequence mining, where context-free or regular languages can be used to improve the result set quality [17, 112]. However, enumeration might not be the best strategy here, as more data-driven methods are used in the referenced work.

1.1.3 Data-driven Enumeration and Interestingness

We continue the study of pattern enumeration, but this time in the context of *semi-structured* data. This particular form of data is very interesting as it diverges from the more standard relational model, where data is represented using fixed-size tuples. Indeed, since the advent of the Internet, several new data models have been used that facilitate the exchange of information in machine-machine and human-machine communication. The semi-structured data corresponds to a data model where the schema is part of the data itself; it is hence considered to be *self-describing* [50]. In the *No-SQL* community, which aims to provide alternatives for relational database management systems (RDBMSs), we find that several solutions operate on semi-structured data to offer a flexible data model (see, e.g., [20, 135]). In No-SQL solutions, the typical requirements for an RDBMS are weakened in favor of other properties such as scalability.

The *eXtensible Markup Language*, or XML, is a specific type of semi-structured data. Figure 1.2 shows an example XML document that consists of tags (between angular brackets) and actual data. XML data exhibits a tree-like structure. For the XML document in Figure 1.2, the corresponding tree is shown in Figure 1.3. Although XML data and other semi-structured data types are self-describing, using a schema offers several advantages. This can be seen in the opportunities that arise for query decomposition and optimization [51] and, for example, in regard to validation and verification of incoming data in webservices [38]. Hence, some form of structural rules could be beneficial. For XML, two very popular methods for constraining the structure of the allowed documents are XML Document Type Definitions [160] (DTDs) and XML Schema Definitions [161] (XSDs).[2] Both provide a means to define a grammar that describes the language of allowed documents.

While XSDs are more powerful than DTDs and allow for enforcing structural constraints, e.g., "every book must have an ISBN number", or "a book has at least one author" they are also able to enforce more complex data-oriented constraints in the form of, e.g., XML keys, a feature that is rarely used. The concept of keys arises from the relational model, where a key is a set of fields that enables us to identify each tuple by using only its values for the key fields. To illustrate the intuition behind XML keys, consider the book order example from Figure 1.2, where it seems only logical to disallow an order that contains multiple items with the same ISBN number while still allowing ISBN repetition across different orders. Hence, we want to be able to express the following:

> *In every set of orders, all ISBN numbers that appear inside one order must be unique.*

This constraint can easily be expressed by an XML key, as we will now illustrate. Informally, an XML key has three major components: (i) a scope

[2]Other formalisms are available such as DSD, Relax NG and Schematron [58, 71, 149], but these are outside the scope of this work.

```
<orders>
    <order>
        <userid>08051986182</userid>
        <item>
            <isbn>978-0425228227</isbn>
            <amount>3</amount>
        </item>
        <item>
            <isbn>978-1847371263</isbn>
            <amount>1</amount>
        </item>
    </order>
    <order>
        <userid>0805198641</userid>
        <item>
            <isbn>978-1847371263</isbn>
            <amount>54</amount>
        </item>
    <order>
<orders>
```

Figure 1.2: *Example XML document containing book orders of two different users.*

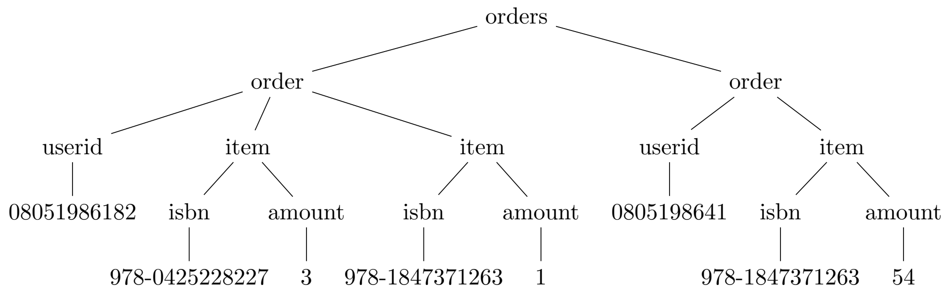


Figure 1.3: *Tree representation of the XML document in Figure 1.2.*

in the XML document, (*ii*) a selection of nodes within the scope, and (*iii*) a method that constructs a 'label' for each node. A key holds in an XML document when inside each scope, all selected nodes are uniquely identified by their label. For the desired constraint above, we could have the following key components: (*i*) the scope is limited to each order element, (*ii*) the nodes that should be unique are the items, (*iii*) which should be identifiable by their ISBN number.

Although this type of constraint can be embedded in current XSDs in the form of an *XML key*, as defined in the W3C specification [161], they are often omitted. What is even more worrisome is that a lot of XML documents refer to low quality schemas or even omit the XSD reference entirely: Grijzenhout and Marx [92] studied a corpus of 180.640 online XML documents in 2013 and found that in 86.7% of the cases no schema information was available. But, there exist methods for automatic schema extraction [38], which can be used to gain understanding about the XML data at hand.

In Chapter 3, we aim to complement the previous work by describing an inference method for XML keys which leverages several existing techniques. The discovered keys can then be used to improve the quality of the existing schema.

The desired inference method is clearly a data mining task that can be described using the pattern mining workflow defined above. Indeed, the patterns we are looking for are XML keys that hold in the considered document(s). As the number of potential keys may become extremely large, we need an algorithm for exploring the search space in an efficient way.

As a lot of keys might hold in a document, identifying good quality results becomes an important problem.[3] We therefore propose a set of key quality criteria that allows us to decide whether a key is (un)interesting. Some examples of uninteresting keys are those that

- can never be satisfied (satisfiability);
- always hold (universality);
- can identify only a bounded amount of elements (boundedness);
- will only sometimes yield 'labels' (data values) for elements (consistency); or
- are equivalent to an already discovered key (equivalence).

As the quality criteria should be embedded in the mining algorithm, it is important to understand the implications of activating them. We therefore

[3]In fact, this is one of the major difficulties in the field of data mining.

characterize the complexity of the decision problems associated with verifying each quality criterion.

Unfortunately, and in strong contrast to, for instance, the relational model, the automatic discovery of XML keys has been left largely unexplored. A major obstacle here is the unavailability of a theory on reasoning about XML keys in the presence of XML schemas, which is needed to validate the quality of candidate keys. Chapter 3 embarks on a fundamental study of such a theory and classifies the complexity of the crucial quality measures mentioned above in the presence of an XSD. As it turns out, some of the proposed measures are tractable, while others are not. We develop a mining algorithm within the framework of levelwise search. The algorithm leverages known discovery algorithms for functional dependencies in the relational model, but incorporates the properties mentioned to assess and refine the quality of derived keys. An thorough experimental study on an extensive body of real world XML data evaluating the effectiveness of the proposed algorithm is provided.

1.2 Big Data

Variety, velocity and volume, also known as the “3 Vs”, are often seen as the major characteristics of “Big data”, a term that has become very popular over the last years [104].[4] The rise of Big data is caused by numerous factors that are all connected to the characteristics mentioned above.

Since the advent of digital storage in the 1980’s, the cost of storing data has become significantly cheaper. Today, storing 1GB of data on Google Cloud, for example, only costs around $0.02 per month[5]. Together with the ubiquity of sensors in everyday devices such as smartphones, thermometers, smart watches, GPS systems, etc., it has led to the generation of vast volumes of data each day. Especially since most devices are now connected to the Internet (Internet of Things, or IoT devices), collecting data has never been easier. This data comes in different types and varieties. Think of click logs from websites, Facebook posts, sensor readings, security cameras, etc. All of these have different sizes, are collected at different rates, are stored in different formats, have special security and privacy requirements, etc. If one desires to aggregate data from different sources, they face great challenges. Finally, there is velocity; an indicator of the data generation speed. The low hardware and energy cost of sensors and data storage, and the availability of a fast networking infrastructure have enabled the generation and ingestion of data at a high frequency.

In different fields, it has become important to extract information from the

[4]In practice, additional characteristics such as veracity are often used [67].

[5]See, e.g., `https://cloud.google.com/storage/pricing`.

available data with the purpose of understanding it better and making justified decisions. When information needs to be extracted from Big data, however, the typical characteristics described above may become a major obstacle:

- when the data is too large to store on one normal device, we are confronted with infrastructural challenges;
- when multiple data types need to be combined we need to think carefully about coping with their different properties; or
- when data arrives at high speeds we may need to examine it in a *streaming* fashion, i.e., process it and never look at it again, as it may not be possible to store it on disk.

These examples illustrate the possible challenges that arise when we are confronted with Big data and they illustrate the need for tools and techniques to manage and process data that allow us to extract new knowledge. This knowledge can take a lot of different forms, a few examples:

- product recommendations on online shopping websites such as Amazon [120];
- trending topics on social networking websites such as Twitter [101, 132];
- discovering drug trafficking in social networks [77]; or
- detecting fraudulent bank transactions [143].

As questions about the data may take different forms, the need of a query engine that provides an understandable query interface and calculates the answers in the most optimal way is clear. Relational databases have been mostly using SQL as the standard query language, which is translated to an actual query plan based on statistical information about the data. When we need to deal with Big data, these traditional systems do not always scale well. Therefore, as we will see next, distributed computing has been used to tackle this issue.

1.2.1 Parallel Query Evaluation

We study the problem of query evaluation in the context of Big data. It should be obvious that, whenever the input size grows, algorithms that process the entire dataset will operate slower. This is no different for data mining algorithms. As time is often important in a knowledge discovery context, e.g., for decision-makers, it is paramount to develop query methods that are scalable so that they can be used with actual Big data that has the characteristics presented at the beginning of this chapter.

To a certain extent, scalability can be obtained by adding faster computing hardware, but this has its limits since the cost involved quickly becomes very large. Furthermore, there are limits to the speed of computers [129, 162]. Another way to speed up an algorithm is to apply a divide and conquer mechanism and parallelize the computations that need to be performed. The idea behind parallelization is to split up a problem into subproblems that can be solved independently. In practice, this corresponds to parallel calculations that do not need to exchange data and can hence be executed in isolation of one another. Afterwards, the results need to be combined into a final form. With the rise of multiprocessor and multi-core systems, which enable the actual parallel execution of different tasks, the idea of splitting up tasks suddenly becomes very interesting. Indeed, the number of cores on computer chips – which can execute tasks concurrently – has drastically increased in the last decade, while the clock frequency has not [88]. Even more parallelization can be obtained by utilizing a computer cluster consisting of commodity machines interconnected through a high-speed network. As it turns out, these set-ups are especially suited for dealing with Big data workflows.[6]

The ultimate goal of parallelization is that when we have n calculations and m processors, we can divide the work among the processors and obtain a factor m speedup. In an ideal situation, this would mean that if the calculations take up n time when executed sequentially, the parallel version would only take up n/m time. Sadly, parallelization always incurs some overhead due to start-up and coordination, and not all tasks or algorithms can be entirely parallelized. Therefore, a factor m speedup is infeasible in practice. Amdahl's law [15] provides a model for determining the amount of speedup that can be obtained through parallelization for a particular problem.

We make use of a programming model called MapReduce for describing parallel computations. The framework is relatively simple and has gained in popularity since it was introduced by Dean and Ghemawat [69] and implemented in Hadoop [94], an open-source system for processing Big data. The core idea behind MapReduce is simple: users write a map function that maps data values to key-value pairs, and a reduce function that maps key-valueset pairs to values. The framework applies the map function to the input tuples, groups the output tuples by their key and then applies the reduce function to it to produce output values. Figure 1.4 shows an abstract version of MapReduce that indicates its most important components.

Example 1.1 (MapReduce program). Suppose the manager of a restaurant wants to analyze the sales data, which is stored in a database having the following schema:

[6]There are other hardware infrastructures that support parallel computations, we refer the reader to Hager and Wellein [95] for more information on this topic.

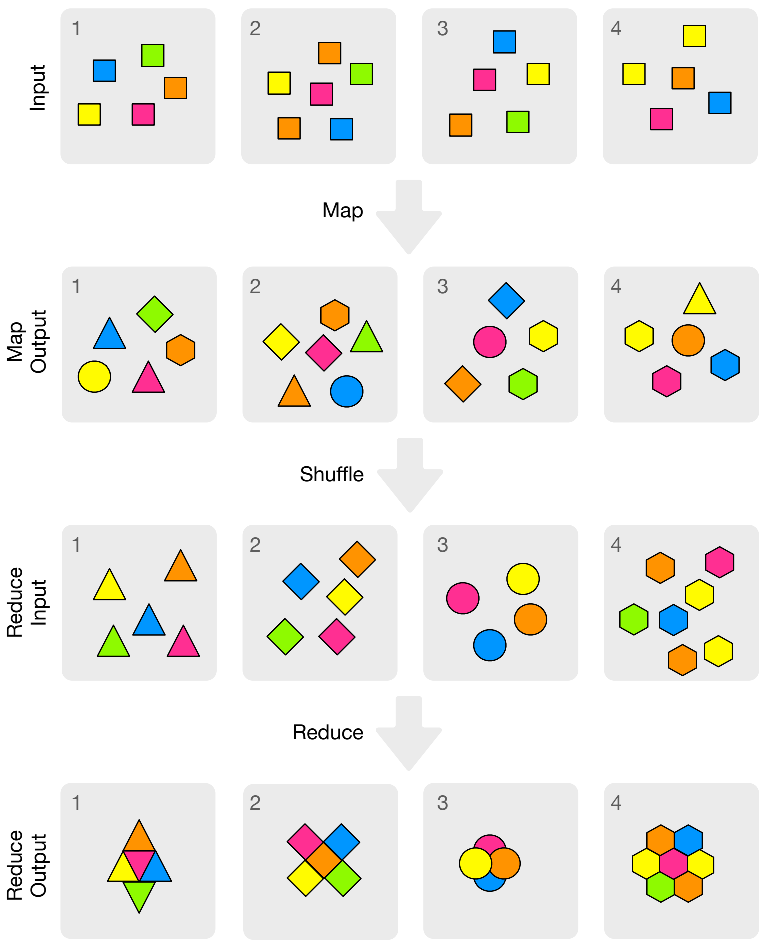


Figure 1.4: *Abstract depiction of the MapReduce pipeline. Here, four machines are used to each process one input file containing data of different colors. The map function extracts certain features of this data in order to determine its shape, which functions as the key. The objects are then shuffled and passed on to four reduce tasks, which each process one type of shape (or key) and assemble the final solution.*

Users(name, id)
Orders(userid, cost, description)

Some basic examples of useful information are the money spent per user, the top 10 best-selling items, which items have the highest chance of being bought together, or which people dine together frequently. Suppose we want to calculate how much money each user has spent; this requires determining the sum of the orders for each user. A naive way of doing this in a relational database is to join a table of customer ids with the table of orders and aggregate the result. In SQL, this would resemble the following query:

```
SELECT SUM(cost), userid
FROM users, orders
WHERE users.id = orders.id
GROUP BY userid
```

In MapReduce, we can calculate the total spending per user using Algorithm 1. For the database in Table 1.2, the result of the map function, the subsequential grouping of its output (shuffling) and the result of the reduce function are shown in Table 1.3. Clearly, both the map and reduce function can be applied to each of their inputs in parallel, as the results are independent of the other inputs. △

Algorithm 1 MapReduce program that calculates the total expenditure per user.

```
1: function MAP1(FACT $f$)
2:     if $f$ is a user then
3:         emit $\langle f.id : f.name \rangle$
4:     if $f$ is an order then
5:         emit $\langle f.userid : f.cost \rangle$

6: function REDUCE1($\langle k : V \rangle$)
7:     name ← extract name from $V$
8:     cost ← 0
9:     for all costs $c$ in $V$ do
10:        cost ← cost + c
11:    output name, cost
```

The MapReduce programming model promises to be an ideal solution for calculating queries related to Big data mining, as it allows for easy scaling of *embarrassingly parallel* computations, which can be accompished by adding

Users.id	Users.name
1	Bas
2	Jelle
3	Joris
4	Dimitri
5	Tom

Orders.userid	Orders.cost	Orders.descr
1	30	rabbit
1	15	wine
2	30	rabbit
5	10	fries

Table 1.2: *Example relational database instance containing users and meal orders.*

$\xrightarrow{\text{map}}$

key	value
1	Bas
1	30
1	15
2	Jelle
2	30
3	Joris
4	Dimitri
5	Tom
5	10

$\xrightarrow{\text{shuffle}}$

key	valueset
1	{Bas, 30, 15}
2	{Jelle, 30}
3	{Joris}
4	{Dimitri}
5	{Tom, 10}

$\xrightarrow{\text{reduce}}$

key	values
Bas	45
Jelle	30
Joris	0
Dimitri	0
Tom	10

Table 1.3: *Transition from map output to reduce output for the example data in Table 1.2.*

more compute nodes to a system (also called *scale-out*). Especially since the availability of cloud platforms such as Google Cloud Platform (GCP), or Amazon AWS, we can easily activate thousands of compute nodes on demand, and rent them for a very reasonable price. But, while the availability of these services makes computing power abundantly available, adding more computing nodes can incur high costs in, for instance, pay-as-you-go plans while not always significantly improving the net running time (or wall-clock time) of queries. Indeed, computing query results using a sequential query plan may not always result in a decrease in net time when more nodes are added, as the calculations in each step of the sequence only require a limited amount of resources. This is why we focus on parallel query plans, where more calculations can be performed in parallel, making scale-outs more useful. Furthermore, we aim to minimize the resources that are necessary for these parallel plans without significantly affecting the net time. This is accomplished by exploiting overlap between different queries and/or query parts. For example, when queries contain similar subqueries that have (partially) identical input relations, we can combine these operations to avoid excessive input reads.

The (extended) relational algebra [59, 85] can be seen as the core query language for relational database systems. Its operators can be expressed easily in the MapReduce framework [167]. We provide a simplified overview:

selection map function that checks if a given predicate holds;

projection map function that eliminates a field of the tuple;

join map function that projects the tuples on the join key. The output consists of this projection (key) and the original tuple (value). The reducer receives all tuples that have the same value for the join key and calculates the cross product;

group by map function that sends a tuple to the reducer that corresponds to the reduce key;

having a reducer that is applied on the grouped tuples.

This means that queries over relational databases can be easily evaluated using MapReduce. Hence, it seems an ideal solution for querying Big data. Indeed, well-known systems such Hive [21, 156] use an SQL-like language that is translated to MapReduce operations internally.[7] A lot of work has been put into the study of join algorithms and accompanying skew (see, e.g., [117]), which is a main source of problems in MapReduce systems. Skew is the problem where the load of different tasks in the system is unbalanced. This may cause

[7]Others systems, such as HBase [100], lack an easy-to-use query language and may require an additional query engine.

underutilization of cluster resources or long wait times. Join calculations are very susceptible to skew as it the number of tuples from a relation that will match with a given tuple from another relation may fluctuate.

1.2.2 SGF Queries

In Chapter 4, we study a slightly different class of queries: strictly guarded fragment (SGF) queries. These can be seen as the relational algebra, where we replace the join operator with the semi-join operator. The reason for this is that SGF's are a class of queries for which we can find efficient evaluation mechanisms, as they only allow a restricted form of joins. They are, on the other hand, more expressive than acyclic conjunctive queries as they may involve disjunction and negation as well as arbitrary levels of nesting. An important aspect of SGF queries is that they can specify all semi-join reducers, which can be used to reduce the communication in join calculations [36, 37, 119].

We focus on finding the right balance between timing constraints (net time) and resource costs (total time), which are important problems that arise when querying Big data. We study MapReduce query execution optimization for SGF queries, i.e., we provide algorithms for parallel evaluation of SGF queries in MapReduce that optimize total time, while retaining low net time. Since SGF queries can be seen as Boolean combinations of (potentially nested) semi-joins, we introduce a novel multi-semi-join (MSJ) MapReduce operator that enables the evaluation of a set of semi-joins in one job. We use this operator to obtain parallel query plans for SGF queries that outperform sequential plans w.r.t. net time and provide additional optimizations aimed at minimizing total time without severely affecting net time. Even though the latter optimizations are NP-hard, we present effective greedy algorithms and several optimizations that are not restricted to our setting.

Chapter 5 gives the implementation details of these algorithms in our open-source tool called Gumbo and presents the results of our experimental validation. The experiments confirm the usefulness of parallel query plans and the effectiveness and scalability of the optimizations present in Gumbo, all with a significant improvement over Pig [22, 142] and Hive [21, 156]. Finally, the optimizations that make it possible to achieve a high performance are fully documented as they might benefit other MapReduce programs.

1.3 Outline

Each chapter provides a thorough introduction to the specific subtopic, gives a more detailed overview of the specific contributions and defines the necessary preliminaries. Table 1.4 shows on overview of the publications that are

Chapter	Articles
Chapter 2	Costa Florêncio et al. [60, 61]
Chapter 3	Arenas et al. [25, 26]
Chapter 4	Daenen et al. [65]
Chapter 5	Daenen et al. [64, 65] and Daenen and Tan [66]

Table 1.4: *Overview of chapters and the associated articles published in the context of this thesis.*

associated with each chapter.

In Chapter 2 the IPT enumeration problem for context-free languages is studied. The main results of this chapter were published in [60], the final goal is outlined in [61]. The chapter expands on the paper as it contains more detailed proofs and adds more details on semi-naive evaluation and unambiguity.

Chapter 3 discusses the key inference and interestingness (quality) problem for XML data. The results were published in [25, 26]. My contributions are centered around the development of the key inference algorithm and accompanying experimental study. The chapter therefore omits results that are not directly related to these contributions.

Chapter 4 studies the optimization of SGF query evaluation in a MapReduce setting, while Chapter 5 discusses the actual implementation of the proposed techniques and the experimental validation thereof. The results in Chapter 4 were published in [65] and expand on the paper in several ways: a more detailed explanation of MapReduce and the associated cost model as well as the addition of NP-hardness proof of a related problem. The results in Chapter 5 were published in [64–66]. The chapter contains a more detailed version of the final algorithm, elaborates on all significant optimizations and includes new experimental results.

In Chapter 6, we conclude and also indicate possible directions for future work.

1.4 Contributions

The contributions of this work can be summarized as follows.

- Regarding context-free language enumeration:
  - We show that the infinite enumeration problem for context-free languages can be solved in incremental polynomial time with a naive enumeration algorithm.

  - We establish a relation between enumeration of given-length strings of a context-free language in polynomial delay and infinite enumeration enumeration.
  - We provide basic foundations and supply methods that are useful for obtaining similar results for regular tree languages and graph languages.

- Regarding XML key quality/interestingness measures and mining:
  - We propose several measures to determine the quality/interestingness of XML keys: consistency, boundedness, satisfiability, universality, and implication of XML keys, as well as equivalence of target paths.
  - We characterize the complexity of the consistency problem for XML keys w.r.t. an XSD for different classes of target and key paths.
  - We develop a novel key mining algorithm leveraging on algorithms for the discovery of relational functional dependencies and on the framework of levelwise search by employing an optimal one-step specialization relation for which the search relation can be computed, if not completely, then at least partly on a prefix tree representation of the document.
  - We experimentally assess the effectiveness of the proposed algorithm on an extensive body of real world XML data and derive suitable values for several parameters for controlling the search space of the different components.

- Regarding MapReduce in general:
  - We provide an updated cost model that can be used to estimate the cost of MapReduce jobs.
  - We show that the general problem of minimizing input reads for a set of interdependent MapReduce jobs is NP-complete.

- Regarding parallel SGF query evaluation in MapReduce:
  - We introduce the multi-semi-join operator $\ltimes\!\cdot(\mathcal{S})$ to evaluate a set $\mathcal{S}$ of semi-joins and present a corresponding MapReduce implementation MSJ$(\mathcal{S})$.
  - We present 2-round MapReduce query plans for basic, that is, unnested, SGF queries. As computing the optimal plan for a given basic SGF query is NP-hard, we utilize an existing greedy heuristic which we refer to as GREEDY-BSGF.

- We show that the evaluation of (possibly nested) SGF queries can be reduced to the evaluation of a set of basic SGF queries in an order consistent with the dependencies induced by the former. In this way, computing an optimal plan for a given SGF query (which is NP-hard as well) can be tackled by a two-tier strategy, where first an optimal ordering of the basic SGF subqueries is determined, followed by an optimal evaluation of each of these basic subqueries. Greedy algorithms are provided for the NP-hard problems.
- We describe a list of optimizations that allow our algorithm to be efficient in practice. Furthermore, these optimizations can be useful for other applications as well.
- We give a high-level description of our open-source system called Gumbo, which can be used on top of an existing Hadoop 2.x cluster.
- We experimentally assess the effectiveness of GREEDY-BSGF and GREEDY-SGF and obtain that backed by an updated cost model these algorithms successfully manage to bring down total times of parallel evaluation, especially in the presence of commonalities among the atoms of queries and outperform Pig and Hive in all aspects when it comes to parallel evaluation of SGF queries.

2

Context-free Language Enumeration in IPT

In the generic pattern mining framework presented in Chapter 1, the traversal of the pattern space is an important component. Furthermore, guarantees on the time between each candidate are desirable when generating candidate patterns one by one. In this chapter, we show that the naive bottom-up concatenation scheme for a context-free string language satisfies the incremental polynomial time property. Next, we show that a polynomial delay solution for a sublanguage containing strings of a given length can easily be transformed into an incremental polynomial time solution for the entire (infinite) language.

2.1 Introduction

Let G be a context-free grammar that is arbitrary but fixed, i.e., G is not considered as part of the input. Hence, we may suppose G is in a convenient normal form, in particular Chomsky Normal Form. We define two basic enumeration problems concerning the language $\mathcal{L}(G)$ generated by G:

1. **Given-length enumeration with polynomial delay.** Given a natural number n, output all strings of length n belonging to $\mathcal{L}(G)$, without duplicates, with *polynomial delay*. By "polynomial delay" we mean that the first output, and every next output, is produced within $p(n)$ time, for some fixed polynomial p. Technically, the output is ended by an "end of output" (EOO) message, and the time spent between the last output

string and EOO should also be bounded by $p(n)$. Moreover, if there are no strings of length n in $\mathcal{L}(G)$, then the algorithm should output an EOO right away, again in time bounded by $p(n)$.

2. **Infinite enumeration in incremental polynomial time.** Output *all* of the strings in $\mathcal{L}(G)$, without duplicates, in incremental polynomial time (IPT), meaning that the time spent between the mth and the $(m+1)$th output is bounded by $p(m)$ for some fixed polynomial p. Here, m is not directly related to string length, but is simply a count of the number of strings that have been output so far. For finite languages, the problem becomes trivial. Hence, we consider only infinite languages and refer to this problem as *infinite enumeration.* In principle, an algorithm for infinite enumeration runs forever, (although one may of course abort it at any time) but the incremental polynomial-time bound guarantees that the time for every next output grows only polynomially.

The notions of polynomial delay and incremental polynomial time were originally introduced (in a setting unrelated to context-free languages) by Johnson et al. [111].

Basic as the above two problems are, the literature on them is relatively scarce. Given-length enumeration was first discussed by Mäkinen [123], but not solved completely; then Dömösi [72] presented a polynomial-delay solution to the same problem by a modification of the well-known CYK parsing algorithm (see, e.g., [108]). Notably, for the special case of regular languages, very efficient algorithms are available [4]. The solutions of both Mäkinen and Dömösi have the additional benefit of enumerating the strings in lexicographic order. Later, Dong [73] reported linear-time improvements to Dömösi's algorithm. A related problem which has received quite some attention in the literature is the efficient generation of a true random sample of a context-free language [29, 83, 89].

Hence, efficient algorithms for given-length enumeration are already available. In this chapter, we consider infinite enumeration. We will show, perhaps unsurprisingly, that any algorithm for given-length enumeration with polynomial delay can be adapted to perform infinite enumeration in incremental polynomial time.

The main topic of this chapter, however, is the naive, bottom-up concatenation scheme that enumerates strings not by length, but by depth of their parse tree. While this scheme is not as efficient as the above algorithms, it is still important because it is basic and natural. Indeed, it is a natural question to ask: does the naive bottom-up concatenation scheme already have the IPT property? We answer this question affirmatively. We believe this result is interesting from a theoretical perspective, as it adds to our fundamental understanding of enumerating context-free languages. The proof of our main result is elementary and is based on detailed pumping-lemma-like arguments. An important property is that the gap in lengths between two consecutive strings in a context-free language is bounded by a constant (which depends on the grammar). Our proofs bound important parameters that govern the amount of work done in one iteration of the concatenation scheme in terms of the number of unique strings generated up to the previous iteration. In particular, ambiguous grammars do not pose a problem.

Infinite enumeration may have practical applications in software testing (see, e.g., Sommerville [154]), where a language of test-inputs can be described by a context-free grammar [29, 74, 134]. In this situation, exhaustive testing of the software on all inputs of the language (e.g., up to a certain length, or until the time budget for testing is exhausted) can be driven by infinite enumeration of a tailor-made context-free language.

Conversely, infinite enumeration may also have applications in verification of context-free languages. While this task is decidable for some properties [31], it is undecidable for many other ones, e.g., containment of one context-free language in another is undecidable [108]. In such cases, infinite enumeration may be useful to detect counterexamples to conjectured properties, or, when no counterexample is found after a sufficiently long time, it may provide confidence in the conjecture, after which the verifier may start an attempt to find a proof by other methods.

Also, there has been interest in tools for testing and debugging the grammars themselves [118, 145, 166], where again infinite enumeration may be helpful.

Outline. This chapter is organized as follows. In Section 2.2 we first give the necessary background on context-free grammars, followed by a formal specification of the naive algorithm in Section 2.3. Regarding this algorithm, we present five important results in Section 2.4 which are then used in Section 2.5 to show that the IPT property holds. In Section 2.6 a general method is given for transforming a given-length enumeration algorithm with polynomial delay to an algorithm for infinite enumeration in incremental polynomial time. Section 2.7 introduces an alternative semi-naive evaluation scheme that has practical benefits. We conclude in Section 2.8.

2.2 Preliminaries

A *context-free grammar* G is a tuple $(\mathcal{N}, \Sigma, \mathcal{P}, S)$, where

- $\mathcal{N}$ is a finite set of *non-terminals*;
- Σ is a finite set of *terminals*, disjoint from $\mathcal{N}$;

- $\mathcal{P}$ is a set of *productions* of the form $X \to \alpha$ with $X \in \mathcal{N}$ and $\alpha \in (\Sigma \cup \mathcal{N})^*$;
- $S \in \mathcal{N}$ is the *start symbol*.

In the remainder of this chapter, we assume that all grammars are in *Chomsky Normal Form* (CNF) [108] without ε-productions, i.e., all productions are of the following form:

- $A \to BC$, a *non-terminal production* or
- $A \to a$, with $a \in \Sigma$, a *terminal production*

where A, B and C are non-terminals. As we mentioned, the empty string ε cannot be used. Importantly, this implies that we will only deal with nonempty strings.

We say a non-terminal A *derives* a string s, written as $A \Rightarrow^* s$, if one of the following holds:

- $s \in \Sigma$ and $A \to s \in \mathcal{P}$ (one-step derivation); or
- $\exists B, C \in \mathcal{N} : u, v \in \Sigma^* : A \to BC \in \mathcal{P} \wedge B \Rightarrow^* u \wedge C \Rightarrow^* v \wedge s = uv$.

The *language of a non-terminal* A belonging to a CGF G, is defined by $\mathcal{L}(G_A) = \{s \mid A \Rightarrow^* s\}$. The language of the start symbol S is also called the *language of* G and is defined by $\mathcal{L}(G) = \mathcal{L}(G_S)$. The set of strings of some length n belonging to a language is also known as a "cross-section" of that language [4].

The *dependency graph* of a context-free grammar (in CNF) is a directed graph having $\mathcal{N}$ as its set of nodes. There is an edge from A to B if there exists a production of the form $A \to BC$ or $A \to CB$, for some non-terminal C. Note that it is possible for the dependency graph to contain self-loops. When we speak of *reachability* in a directed graph, we always mean reachability by a directed path. A *directed path* in a graph is a list of nodes such that there exists a directed edge between two successive nodes. The length of a path π is equal to the number of edges it contains and is denoted by $l(\pi)$. We classify the nodes/non-terminals in the dependency graph as follows:

recursive a node belonging to a directed cycle;

leeching a node that can reach a recursive node, but is not recursive itself;

restricted a node that is neither recursive nor leeching.

We denote the set of recursive non-terminals by $\mathcal{N}_{rec}$, the set of leeching non-terminals by $\mathcal{N}_{leech}$ and the set of restricted non-terminals by $\mathcal{N}_{res}$. Observe that, in order to obtain an infinite language, the start symbol must be either recursive or leeching. Indeed, when the start symbol is restricted, the context-free language described by the grammar is finite and the enumeration problem becomes trivial. See Example 2.1 for an example of an infinite language.

Furthermore, without loss of generality, we consider *proper* grammars only, i.e., we assume that all nodes in the dependency graph are reachable from S, and that all non-terminals are productive[8] (see [108] for an algorithm to clean up a grammar). Next, we need the concept of a parse tree. A *parse tree* is a tree representation that indicates the derivation process for a string from the start symbol. The root is labeled with the start symbol S, and the children of a node A are labeled with B and C iff a production $A \to AB$ was used, or with a terminal symbol a iff a terminal production $A \to a$ was used. This means that the leafs of a parse tree are terminal symbols. See Example 2.1 below for an illustration of this concept. For each string $s \in \mathcal{L}(G_A)$, where A is a non-terminal of grammar G, there exists at least one parse tree that yields s and in which the root of the parse tree is labeled with A. The *depth* of a parse tree τ equals the length of a longest path (number of edges) from the root to a leaf and is denoted by $d(\tau)$. Note that we do not restrict the grammar in terms of ambiguity: ambiguous grammars are allowed, which means that each string may have multiple parse trees. This leads to the notion of a *minimal parse tree* of a string: a parse tree of minimum depth. Note that a string may have more than one minimal parse tree rooted at a non-terminal A. A parse tree is called *non-recursive* if no path contains two nodes labeled with the same non-terminal.

Example 2.1 (Dependency graph). Consider the following context-free grammar G_1:

$$\begin{aligned}
S &\to AG & C &\to c \\
A &\to BD & D &\to d \\
B &\to CE & E &\to e \\
C &\to AF & F &\to f \\
& & G &\to g.
\end{aligned}$$

The corresponding dependency graph is shown in Figure 2.2. An example parse tree for the string `cedg` is shown in Figure 2.1. We observe the following classification of the non-terminals:

- $\mathcal{N}_{rec} = \{A, B, C\}$;
- $\mathcal{N}_{leech} = \{S\}$;

[8]A non-terminal is called *productive* when its language is nonempty, i.e., when a string can be derived from it.

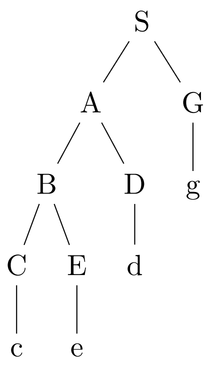


Figure 2.1: *The parse tree for the string* `cedg` *w.r.t. grammar* G_1*from Example 2.1.*

$$\begin{tikzcd}
 & & S \arrow[r, bend left] \arrow[d] & G \\
 & & A \arrow[r, bend left] \arrow[dr] & D \\
F & C \arrow[l] \arrow[ur] & & B \arrow[ll, bend left] \arrow[d] \\
 & & & E
\end{tikzcd}$$

Figure 2.2: *The dependency graph of context-free grammar* G_1 *from Example 2.1.*

- $\mathcal{N}_{res} = \{D, E, F, G\}$.

Note that all non-terminals are reachable and productive, and that $\mathcal{L}(G_1)$ is infinite. It is intuitively clear that $\mathcal{L}(G)$ is infinite: let's start at non-terminal C, for which the string `c` can be generated directly; using this string, we can generate `ce` for B and then `ced` for A. Now, using the strings from A and F, we can generate `cedf` for C. This whole process of passing strings through the cycle C, B, A can be repeated an arbitrary number of times, where each iteration generates longer – and therefore new – strings for the recursive non-terminals. △

2.3 Naive Enumeration Algorithm

We now present a basic iterative algorithm that generates the language de-

Algorithm 2 Naive concatenation scheme for context-free grammars.

$$\begin{aligned}
A^0 &= \{a \in \Sigma \mid A \to a \in \mathcal{P}\};\\
A^{i+1} &= A^i \cup \{u \cdot v \mid \exists B, C \in \mathcal{N} : A \to BC \in \mathcal{P} \wedge u \in B^i \wedge v \in C^i\};\\
\Delta A^0 &= A^0;\\
\Delta A^{i+1} &= A^{i+1} \setminus A^i.
\end{aligned}$$

scribed by a given, fixed grammar G. The iterations are computed according to the standard inductive *concatenation scheme* shown in Algorithm 2.

During the execution of this algorithm, every non-terminal is associated with a set of *terminal strings.* For a non-terminal A, A^i denotes the set of all terminal strings generated in iterations 0 to i. It is easy to see that $A^i \subseteq A^{i+1}$ for each $A \in \mathcal{N}$ and all $i \in \mathbb{N}$. The strings in S^i, where S is the start symbol, are called the *output strings* and make up the language $\mathcal{L}(G)$. Again for a non-terminal A, the set ΔA^i denotes the set of all terminal strings generated in iteration i. It contains all strings that can be obtained by combining previously generated strings, according to the production(s) associated with A, except for those that have already been generated. Note that we are working with sets: duplicates are removed, but it is still possible that in the same iteration, or in two different iterations, two identical strings are generated for a non-terminal A (see Example 2.3 below). Finally, note that the terminal productions are only used in iteration 0 and the non-terminal productions are only used in the subsequent iterations.

Remark 2.2 (Optimized scheme). In the second rule of the scheme, the use of A^0 instead of A^i would yield equivalent definitions. $\diamond$

We denote the length of a string s by $|s|$ and the maximal length of a string in A^i by ω_A^i. Note that it is possible for A^i to be empty when $i < |\mathcal{N}| - 1$ (this will be shown in Section 2.4.2), in which case ω_A^i is undefined. Clearly, $\omega_A^{i+1} \geq \omega_A^i$ holds for $i \geq |\mathcal{N}| - 1$.

For convenient reasoning about all strings that are produces in one iteration, we define

$$\begin{aligned}
\mathcal{T}^i &= \bigcup_{A \in \mathcal{N}} A^i, \text{ and}\\
\Delta\mathcal{T}^i &= \bigcup_{A \in \mathcal{N}} \Delta A^i.
\end{aligned}$$

In addition to the output strings, these sets also contain the strings that are only used as building blocks for the output strings, and are not output strings themselves. These are called the *intermediate strings*. The maximal length

Symbol	Meaning
$\|\mathcal{N}\|$	number of non-terminals
$\|\mathcal{P}\|$	number of productions
$d(\tau)$	depth of tree τ
$\|s\|$	length of string s
$\|X\|$	number of elements in set X
$\mathcal{N}_{rec}$	set of recursive non-terminals
$\mathcal{N}_{leech}$	set of leeching non-terminals
$\mathcal{N}_{res}$	set of restricted non-terminals
A^i	strings for non-terminal A generated up through iteration i
ΔA^i	strings for non-terminal A first generated in iteration i
$\mathcal{T}^i$	intermediate strings generated up to and including iteration i
$\Delta\mathcal{T}^i$	intermediate strings first generated in iteration i
ω^i_A	maximal length of a string in A^i
$\omega^i_{\mathcal{T}}$	maximal length of a string in $\mathcal{T}^i$

Table 2.1: *Overview of symbols used in this chapter.*

of a string in $\mathcal{T}^i$ is denoted by $\omega^i_{\mathcal{T}}$. Note that the same string might be generated for multiple non-terminals, i.e., the union $\bigcup_{A\in\mathcal{N}} A^i$ that defines $\mathcal{T}^i$ is generally not a disjoint union. For completeness, Table 2.1 gives an overview of the symbols used in this chapter.

Example 2.3 (Concatenation scheme). Consider the following grammar G_2:

$$\begin{aligned} S &\to AB & A &\to a \\ S &\to CB & B &\to b \\ A &\to AB & C &\to a. \end{aligned}$$

Table 2.2 shows the results of the concatenation scheme applied on G_2 for the first few iterations. Observe that the string `ab` is generated both by $S \to CB$ and by $S \to AB$ in two different iterations. In iteration 2 the string is already present in S^1, hence it is not in ΔS^2, even though it is in S^2. The output strings shown in the table (S^3) are `ab`, `abb` and `abbb`. The intermediate strings shown in the table are `a`, `b`, `ab`, `abb` and `abbb` (this set equals $\mathcal{T}^3$). △

Remark 2.4 (Semi-naive scheme). An equivalent, but more efficient inductive concatenation scheme that avoids duplicate concatenations is the well-known "semi-naive" scheme [54], which we will discuss in Section 2.7. Although this semi-naive scheme can give practical improvements in terms of performance,

Non-terminal N	ΔN^0	ΔN^1	ΔN^2	ΔN^3	...
S	{}	{ab}	{abb}	{abbb}	...
A	{a}	{ab}	{abb}	{abbb}	...
B	{b}	{}	{}	{}	...
C	{a}	{}	{}	{}	...

Table 2.2: *Generated output and intermediate strings in the first iterations of applying the naive concatenation scheme to the context-free grammar G_2.*

e.g., in applications to databases [32], the theoretical worst-case complexity is of the same order as that of the standard scheme. This is why, in this chapter, we focus on the standard scheme and prove that it runs in polynomial incremental time. $\diamond$

2.4 Upper and Lower Bounds on the Number and Length of Generated Strings

Our main result is that the naive algorithm satisfies the IPT property. In order to prove this (Section 2.5), we obtain five important results in this section:

- a bound on the length of strings that can be generated in a certain iteration i (Section 2.4.1);
- a formalization of the start-up phase (Section 2.4.2);
- a lower bound on the number of intermediate strings, expressed in the iteration number (Section 2.4.3);
- an upper bound on the size of the maximum string, expressed in terms of the number of intermediate strings (Section 2.4.4); and,
- an upper bound on the number of intermediate strings in terms of the number of output strings (Section 2.4.5).

2.4.1 String Properties

Lemma 2.5. *For any non-terminal A, the set ΔA^i consists precisely of the strings that can be derived from A and have a minimal parse tree depth of $i+1$.*

Proof. We prove the lemma by induction on i.

Base For $i = 0$, ΔA^0 contains all strings that can be derived from A in one step. It is obvious that all these strings have a parse tree of depth 1 and no smaller parse tree exists. Clearly, no other strings can be derived from A with a parse tree of depth 1.

Induction Step For $i > 0$, suppose the lemma holds for all values smaller than i. Consider a string $s = u \cdot v \in \Delta A^i$ with $u \in B^{i-1}$, $v \in C^{i-1}$ and $A \to BC \in \mathcal{P}$ for some $B, C \in \mathcal{N}$. By induction u and v have minimal parse trees τ_u and τ_v of depth at most i. Note that these parse trees cannot both have a depth smaller than i, because then we could create a parse tree for s of depth smaller than $i+1$; this would imply (by induction) that $s \in A^{i-1}$, which contradicts $s \in \Delta A^i$. We thus obtain that s has a minimal parse tree of depth $i + 1$.

It remains to show that all strings with a minimal parse tree of depth $i+1$, that can be derived from A, belong to ΔA^i. Thereto, consider such a string $s \in \mathcal{L}(G_A)$ that has a minimal parse tree τ of depth $i + 1$.

We first show that $s \in A^i$. Since $i > 0$, τ has the form of an A-root with two children τ_B and τ_C and $A \to BC \in \mathcal{P}$ for some $B, C \in \mathcal{N}$. Let u and v be the strings yielded by τ_B and τ_C respectively, so $s = u \cdot v$. Since τ has depth $i+1$, we have $d(\tau_B) \leq i$ and $d(\tau_C) \leq i$. By induction, $u \in \Delta B^j$ and $v \in \Delta C^k$, for some $j, k < i$. In particular, $u \in B^{i-1}$ and $v \in C^{i-1}$. It is now obvious from the definition of A^i that $s = u \cdot v \in A^i$.

Finally, we show that $s \in \Delta A^i = A^i \setminus A^{i-1}$ by proving that $s \notin A^{i-1}$. Suppose that $s \in A^{i-1}$. By induction, s has a minimal parse tree of depth smaller than or equal to i, which contradicts our assumption. □

Knowledge about the iteration in which a string is generated gives us information about the length of the string. Because the yield of a parse tree of depth $i + 1$ has length at least $i + 1$ and at most 2^i, the lemma above implies the following:

Lemma 2.6. $\forall i \in \mathbb{N} : \forall s \in \Delta A^i : i + 1 \leq |s| \leq 2^i$.

Proof. Lemma 2.5 shows that the minimal parse tree of a string $s \in \Delta A^i$ has depth $i + 1$. The yield of a parse tree of depth $i + 1$ has length at least $i + 1$ and at most 2^i as the Chomsky Normal Form implies a maximum branching factor of 2. □

2.4.2 Start-up Phase

We look at the first $|\mathcal{N}|$ iterations of the algorithm: the *start-up phase*. After this initial start-up, we show that all non-terminals will have generated at least one (intermediate) string. Thereto, we consider the following definitions:

$$\begin{aligned}
\mathcal{N}^0 &= \{A \in \mathcal{N} \mid \exists a : (A \to a) \in \mathcal{P}\};\\
\mathcal{N}^{i+1} &= \mathcal{N}^i \cup \{A \in \mathcal{N} \mid \exists B, C \in \mathcal{N}^i : (A \to BC) \in \mathcal{P}\};\\
\Delta\mathcal{N}^0 &= \mathcal{N}^0;\\
\Delta\mathcal{N}^{i+1} &= \mathcal{N}^{i+1} \setminus \mathcal{N}^i.
\end{aligned}$$

Note that $\mathcal{N}^i \subseteq \mathcal{N}^{i+1}$ for all i. Intuitively, a non-terminal A is in $\Delta\mathcal{N}^i$ iff it generates its first string in iteration i.

Lemma 2.7. $\forall i \geq 0 : \forall A \in \mathcal{N} : A \in \mathcal{N}^i \Leftrightarrow A^i \neq \emptyset$.

Proof. We prove the lemma by induction on i.

Base Let $i = 0$. If $A \in \mathcal{N}^0$, there is a production $A \to a \in \mathcal{P}$, and by definition $a \in A^0$. When $A^0 \neq \emptyset$, we know (from the concatenation scheme) there must be a $A \to a \in \mathcal{P}$. Now, by definition, $A \in \mathcal{N}^0$.

Induction Step When $i > 0$, assume the lemma holds for all smaller values of i. Suppose $A \in \mathcal{N}^i$ and consider the following two cases:

1) If $A \in \mathcal{N}^{i-1}$, by induction we know that $A^{i-1} \neq \emptyset$. It follows that $A^i \neq \emptyset$, because $A^{i-1} \subseteq A^i$.
2) Otherwise, there exists a production $A \to BC \in \mathcal{P}$, for some $B, C \in \mathcal{N}^{i-1}$. Then by induction $B^{i-1} \neq \emptyset$ and $C^{i-1} \neq \emptyset$. Consider two strings $u \in B^{i-1}$ and $v \in C^{i-1}$. By definition $u \cdot v \in A^i$, which shows $A^i \neq \emptyset$.

Now suppose $A^i \neq \emptyset$. Consider again two cases:

1) If $A^{i-1} \neq \emptyset$, by induction we know that $A \in \mathcal{N}^{i-1} \subseteq \mathcal{N}^i$.
2) Otherwise, there exists a production $A \to BC \in \mathcal{P}$, for some $B, C \in \mathcal{N}$. There exist strings $u \in B^{i-1}$ and $v \in C^{i-1}$ and therefore $B^{i-1} \neq \emptyset$ and $C^{i-1} \neq \emptyset$. By induction $B, C \in \mathcal{N}^{i-1}$ and therefore $A \in \mathcal{N}^i$, by definition of $\mathcal{N}^i$. □

As these definitions capture the dependency structure of the different non-terminals, they can be used to show that each non-terminal will be included after a finite amount of iterations, as the following lemma shows.

Lemma 2.8. $\mathcal{N}^{|\mathcal{N}|-1} = \mathcal{N}$.

Proof. Consider a non-terminal A and a non-recursive parse tree τ rooted at A. We know τ exists because for each non-terminal B there exists a non-recursive parse tree rooted at B (recall the assumption that no non-terminal is useless). As it is non-recursive, the depth of τ is at most $|\mathcal{N}|$. Therefore, $\tau \in A^{i-1}$ and by Lemma 2.7, it follows that $A \in \mathcal{N}^{|\mathcal{N}|-1}$. Hence, all non-terminals are contained in $\mathcal{N}^{|\mathcal{N}|-1}$. The reverse containment is immediate. □

The following corollary shows that every non-terminal contains at least one string in iteration $|\mathcal{N}| - 1$.

Corollary 2.9 (Start-up phase bound). $\forall A \in \mathcal{N} : A^{|\mathcal{N}|-1} \neq \emptyset$.

Proof. From Lemma 2.8 we know $A \in \mathcal{N}^{|\mathcal{N}|-1}$, for each $A \in \mathcal{N}$. After applying Lemma 2.7 we obtain $A^{|\mathcal{N}|-1} \neq \emptyset$. □

For completeness, we mention the following two properties that hold for the naive concatenation scheme. The proofs are immediate.

Property 2.10. *If $A \in \Delta\mathcal{N}^i$, then the first string of A is generated in iteration i, i.e.,*

$$\forall A \in \mathcal{N} : A \in \Delta\mathcal{N}^i \Leftrightarrow \Delta A^i = A^i \neq \emptyset.$$

Property 2.11. *After iteration $|\mathcal{N}| - 1$, restricted non-terminals do not generate any new strings:*

$$A \in \mathcal{N}_{res}, j \geq |\mathcal{N}| : \Delta A^j = \emptyset.$$

2.4.3 Generation Pace

In this section we discuss the “speed” or *generation pace* at which strings are generated when using the naive concatenation scheme in order to establish a lower bound on the number of intermediate strings that are generated in one iteration.

Recursive Non-terminals

The following definition is illustrated in Figure 2.3.

Definition 2.12 (Filled context). Let $A \in \mathcal{N}_{rec}$. A *filled context* τ for A is a parse tree rooted at A with the following properties:

1. A occurs at least twice (note that the root node is already labeled with A);

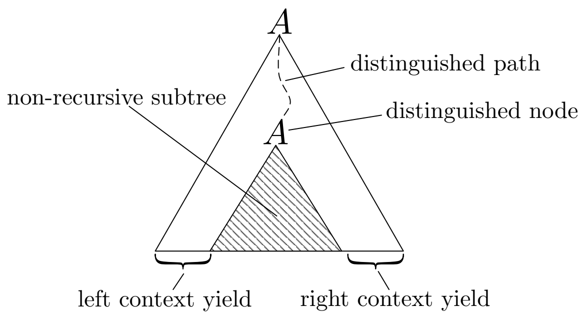


Figure 2.3: *A filled context for a recursive non-terminal A.*

2. some non-root A-node is called the *distinguished node*. The path from the root to the distinguished node is called the *distinguished path*. No non-terminal occurs more than once on the distinguished path, except for A, which appears exactly twice on the distinguished path;

3. for any non-terminal node x not lying on the distinguished path, the subtree rooted at x is non-recursive; and

4. the subtree rooted at the distinguished node is non-recursive.

The *context yield* of τ is the yield of τ without the yield of the distinguished node. The *left context yield* (resp. *right context yield*) is the yield of τ before (resp. after) the yield of the distinguished node. The *context length* is the length of the context yield.

Remark 2.13 (Context depth). Every filled context τ for a recursive non-terminal has a depth of at most $2|\mathcal{N}|$. This can be easily seen: Consider a path π in τ from root to leaf. There are now two options:

1. An initial segment of π *equals* the distinguished path, followed by a path below the distinguished node. By definition, the distinguished path contains at most $|\mathcal{N}|$ edges. The path continues in a non-recursive subtree, and therefore has an additional length of at most $|\mathcal{N}|$. Hence, in this case, the length of π is at most $2|\mathcal{N}|$.

2. The path π diverges from the distinguished path. This means π has at most $|\mathcal{N}| - 1$ edges in common with the distinguished path, after which it follows 1 edge to a non-recursive subtree of depth at most $|\mathcal{N}|$. Hence, in this case, the length of π is at most $|\mathcal{N}| - 1 + 1 + |\mathcal{N}| = 2|\mathcal{N}|$.

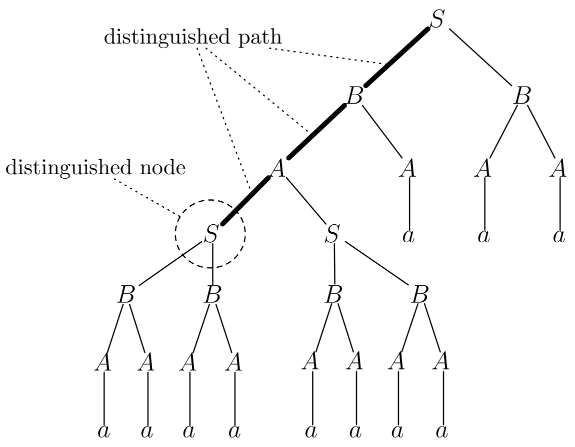


Figure 2.4: *A minimal depth filled context for a recursive non-terminal S.*

To see that this bound can actually be reached, consider the following grammar:

$$
\begin{aligned}
S &\to BB \\
B &\to AA \\
A &\to SS \\
A &\to a.
\end{aligned}
$$

In this case, $|\mathcal{N}| = 3$. Hence, a filled context has depth at most 6. In Figure 2.4, a filled context of minimal depth for S that reaches this bound is depicted. $\diamond$

Lemma 2.14. *For each $A \in \mathcal{N}_{rec}$ there exists a filled context.*

Proof. Consider a simple path π from A to itself in the dependency graph. The length of π is at most $|\mathcal{N}|$ (only A may appear twice). We now show that we can expand π to a filled context for A.

Denote the second occurrence of A on π by x_A. Consider a node x_B on π labeled with a node B that is followed directly by a node labeled with C (also on π). There must exist a production $B \to CD$ or $B \to DC$ for some non-terminal D. We know (see Lemma 2.8) that D has a non-recursive parse tree τ_D of depth at most $|\mathcal{N}|$. Add τ_D as a child of x_B (left or right as indicated by the production). Apply this construction to all nodes on π except for x_A. Replace x_A with a non-recursive parse tree for A. Denote the resulting tree with τ.

It is clear that τ is a parse tree for A and that it contains A at least twice. Because π is simple it contains no repetition (except for A), it can serve as the distinguished path with x_A as the distinguished node. By construction all non-terminal nodes not on the distinguished path are the root of a non-recursive subtree. Finally, again by construction, the subtree rooted at the distinguished node x_A is non-recursive.

It is now clear that τ can serve as a filled context for A, with π playing the role of the distinguished path. ☐

Lemma 2.15. *Let $A \in \mathcal{N}_{rec}$, let τ be a filled context for A of depth δ and distinguished path length ζ. For all $i \geq 0$, there exists a parse tree τ_i for A, such that*

$$d(\tau_i) = \delta + i \cdot \zeta.$$

Furthermore,

$$|s_i| = \gamma + i \cdot \rho,$$

where s_i is the yield of τ_i, γ is the length of the yield of τ and ρ is the context length of τ.

Proof. We first prove the existence of the parse trees τ_i by induction on i.

Base For τ_0, we can take τ itself. By definition this parse tree has depth δ.

Induction Step Now let $i > 0$ and suppose τ_{i-1} exists: τ_{i-1} is a parse tree rooted at A of depth $\delta + (i-1) \cdot \zeta$. Now replace the distinguished node of τ, which has label A, with τ_{i-1} to obtain a parse tree τ_i. We write this as $\tau_i = \tau[\tau_{i-1}]$. This is clearly a parse tree for A. Consider a path π from root to leaf. There are two possibilities:

1) The path lies fully in τ. This means the length is at most δ.
2) The path goes through the distinguished node and the τ_{i-1}-subtree. This means the first part of the path equals the distinguished path, and the second part has length at most the depth of τ_{i-1}, which equals $\delta+(i-1)\cdot\zeta$. Hence, the length of π is at most $\zeta+\delta+(i-1)\cdot\zeta = \delta+i\cdot\zeta$.

Note that there always is a path of length $\delta + i \cdot \zeta$, because τ_{i-1} has depth $\delta + (i-1) \cdot \zeta$. Hence $d(\tau_i) = \delta + i \cdot \zeta$.

It only remains to show that $|s_i| = \gamma + i \cdot \rho$. We show this again by induction.

Base When $i = 0$, obviously $|s_0| = \gamma$, because $\tau_0 = \tau$.

Induction Step When $i > 0$, suppose the lemma holds for $i - 1$. Denote the left and right context yields of τ by u_l and u_r, and their respective lengths by ρ_l and ρ_r. Because $\tau_i = \tau[\tau_{i-1}]$, it holds that

$$s_i = u_l s_{i-1} u_r,$$

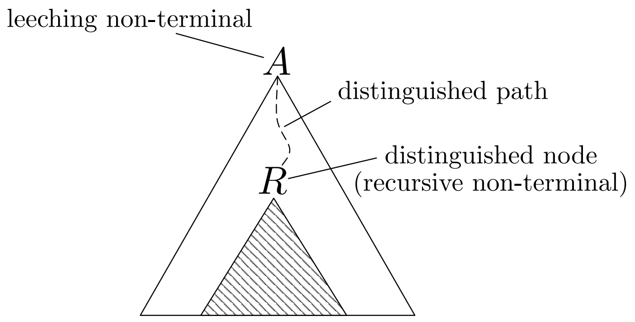


Figure 2.5: *An R-filled context for a leeching non-terminal A.*

where s_{i-1} is the yield of τ_{i-1}. Hence:

$$\begin{aligned}
|s_i| &= |u_l| + |s_{i-1}| + |u_r| \\
&= \rho_l + (\gamma + (i-1)\cdot\rho) + \rho_r && \text{(by induction hypothesis)} \\
&= \gamma + (i-1)\cdot\rho + \rho && (\rho_l + \rho_r = \rho) \\
&= \gamma + i\cdot\rho.
\end{aligned}$$

□

We conclude with the following lower bound on the number of strings generated by recursive non-terminals.

Lemma 2.16. *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{rec}, \forall i \geq 0 : |A^{c+|\mathcal{N}|\cdot i}| > i.$$

Proof. Let τ be a filled context for A (by Lemma 2.14) of depth δ with a distinguished path length of ζ. By Lemma 2.15, for each $i \geq 0$, we have a different string s_i with a parse tree τ_i of depth $\delta + i\cdot\zeta$, where $\delta \leq 2|\mathcal{N}|$ and $\zeta \leq |\mathcal{N}|$. Hence, each τ_i yields a new string lastly in iteration $2|\mathcal{N}| + i\cdot|\mathcal{N}|$. Therefore, $|A^{2|\mathcal{N}|+i\cdot|\mathcal{N}|}| - 1 \geq i$, or $|A^{2|\mathcal{N}|+i\cdot|\mathcal{N}|}| > i$, for $i \geq 0$. □

Leeching Non-terminals

In Definition 2.12, we defined the notion of filled context for a recursive non-terminal. We now define the analogous notion for a leeching non-terminal; this is illustrated in Figure 2.5.

Definition 2.17 (Filled context). Let $A \in \mathcal{N}_{leech}$. A *filled context* for A is a parse tree for A with the following properties:

1. it contains at least one recursive non-terminal;

2. it is non-recursive; and

3. some recursive node is called the *distinguished node*. The path from the root to the distinguished node is called the *distinguished path*.

When the distinguished node is labeled with a recursive non-terminal R, we call the tree a *R-filled context* for A. The notions of (left and right) context yield, and of context length are defined in the same way as in Definition 2.12.

In analogy to Lemma 2.14, Lemma 2.15 and Lemma 2.16, we have the following.

Lemma 2.18. *For each $A \in \mathcal{N}_{leech}$ there exists a filled context.*

Proof. Consider a simple path π from A to a recursive non-terminal R that consists of only leeching non-terminals (π must exist, because A is leeching). The path has length at most $|\mathcal{N}| - 1$. We now show that we can expand π to an R-filled context for A.

Consider a node x_B on π labeled with a (leeching) node B which is followed by a node labeled with C. There must exist a production $B \to CD$ or $B \to DC$ for some non-terminal D. We know that D has a non-recursive parse tree τ_D of depth at most $|\mathcal{N}|$. In particular, we know that τ_D does not contain any non-terminals that appear before B on π, else these would be recursive, contradicting our assumption. Add τ_D as a child of x_B (left or right as indicated by the production). Apply this construction to all leeching nodes on π and replace R with a non-recursive parse tree τ_R for R. By similar reasoning, τ_R does not contain any non-terminals that appear before R on π. Denote the resulting tree with τ.

It is obvious that τ is a parse tree for A, which is non-recursive and contains at least one recursive non-terminal (R). Hence, τ is an R-filled context for A. □

Remark 2.19 (Context depth). Every filled context τ for a leeching non-terminal has a depth of at most $2|\mathcal{N}|$. ◇

Lemma 2.20. *Let $A \in \mathcal{N}_{leech}$ and let τ be a B-filled context for A having depth δ and distinguished path length ζ. Let τ_B be a filled context for B having depth δ_B and distinguished path length ζ_B. For all $i \geq |\mathcal{N}|$ there exists a parse tree τ_i for A, such that*

$$d(\tau_i) = \zeta + \delta_B + i \cdot \zeta_B.$$

Furthermore

$$|s_i| = \rho + \gamma_B + i \cdot \rho_B,$$

where s_i is the yield of τ_i, ρ is the context length of τ, γ_B is the length of the yield of τ_B and ρ_B is the context length of τ_B.

Proof. We know from Lemma 2.14 and Lemma 2.15 that B has parse trees $\tau_{B,j}$, for $j > 0$, such that $d(\tau_{B,j}) = \delta_B + j \cdot \zeta_B$ and $|s_{B,j}| = \gamma_B + j \cdot \rho_B$, where $s_{B,j}$ is the yield of $\tau_{B,j}$. Now, construct the parse tree τ_i, for $i \geq 0$, as follows: replace the distinguished node of τ by a parse tree $\tau_{B,i}$ for B of depth $\delta_B + i \cdot \zeta_B$. We write this as $\tau_i = \tau_s[\tau_{B,i}]$.

To see that actually $d(\tau_i) = \zeta + \delta_B + i \cdot \zeta_B$, consider a path π in τ_i from root to leaf. There are two possibilities:

1) π is fully contained in τ and hence has a maximal depth of $|\mathcal{N}|$.

2) π contains the distinguished node and ends in a leaf in $\tau_{B,i}$. In this case, π consists of the distinguished path, followed by a path of length at most $\delta_B + i \cdot \zeta_B$. The total length of π is at most $\zeta + \delta_B + i \cdot \zeta_B$.

Because $d(\tau_{B,i}) = \delta_B + i \cdot \zeta_B$, there exists at least one path in τ_i that has length $\zeta + \delta_B + i \cdot \zeta_B$. Hence $d(\tau_i) \geq \zeta + \delta_B + i \cdot \zeta_B$. Since clearly $\zeta + \delta_B + i \cdot \zeta_B \geq |\mathcal{N}|$ for $i \geq |\mathcal{N}|$, we obtain $d(\tau_i) \leq \zeta + \delta_B + i \cdot \zeta_B$. We may now conclude that $d(\tau_i) = \zeta + \delta_B + i \cdot \zeta_B$.

Finally, by construction, it is clear that $|s_i| = \rho + |s_{B,i}| = \rho + \gamma_B + i \cdot \rho_B$. □

Remark 2.21 (Small parse trees). From Lemma 2.20, we can use the same construction to get a different parse tree for each $i < |\mathcal{N}|$, all yielding different strings. But, when $i < |\mathcal{N}|$, the depth of the parse tree is equal to $\max(d(\tau), \zeta + \delta_B + i \cdot \zeta_B)$ as the tree that is plugged into the context might not increase the overall depth of the new tree. ◇

Lemma 2.22. *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{leech}, \forall i \geq 0 : |A^{c+|\mathcal{N}|\cdot i}| > i.$$

Proof. Consider a recursive non-terminal R, reachable by A, having a context of depth δ_R with a distinguished path length of ζ_R. By Lemma 2.20, for each $i \geq |\mathcal{N}|$, we have a different string with a parse tree of depth $\zeta + \delta_R + i \cdot \zeta_R$, where $\zeta \leq |\mathcal{N}| - 1$, $\delta_R \leq 2|\mathcal{N}|$ and $\zeta_R \leq |\mathcal{N}|$. Hence, each τ_i yields a new string lastly in iteration

$$|\mathcal{N}| - 1 + 2|\mathcal{N}| + i \cdot |\mathcal{N}| = 3|\mathcal{N}| + i \cdot |\mathcal{N}| - 1.$$

Therefore,

$$|A^{3|\mathcal{N}|+i\cdot|\mathcal{N}|-1}| - 1 \geq i - |\mathcal{N}|, \text{ or}$$
$$|A^{3|\mathcal{N}|+i\cdot|\mathcal{N}|-1}| > i - |\mathcal{N}|,$$

for $i \geq |\mathcal{N}|$. By substituting i' for $i - |\mathcal{N}|$ we obtain that, for $i' \geq 0$,

$$|A^{3|\mathcal{N}|+(i'+|\mathcal{N}|)\cdot|\mathcal{N}|-1}| > i', \text{ or}$$

$$|A^{3|\mathcal{N}|+i'\cdot|\mathcal{N}|+|\mathcal{N}|^2-1}| > i'.$$ □

Non-restricted non-terminals

The bounds obtained above for recursive and restricted non-terminals can be generalized as follows for all but the restricted non-terminals.

Corollary 2.23 (Non-terminal lower bound). *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}, \forall i \geq 0 : |\mathcal{N}| \cdot |A^{i+c}| > i.$$

Proof. Consider the constants c_1 and c_2 from Lemma 2.16 and Lemma 2.22, respectively. We can choose c_3 as the maximum of c_1 and c_2 and get:

$$\forall A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}, \forall k \geq 0 : |A^{c_3+|\mathcal{N}|\cdot k}| > k.$$

Now let $A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}$ and i be an arbitrary natural number. We distinguish the following two cases to show that the corollary holds:

1) i is a multiple of $|\mathcal{N}|$, i.e., $i = |\mathcal{N}| \cdot k$ for some natural number k. We can now argue as follows:

$$\begin{aligned} |A^{c_3+|\mathcal{N}|\cdot k}| > k &\Leftrightarrow |A^{c_3+|\mathcal{N}|\cdot\frac{i}{|\mathcal{N}|}}| > \frac{i}{|\mathcal{N}|} \\ &\Leftrightarrow |\mathcal{N}| \cdot |A^{c_3+i}| > i. \end{aligned}$$

2) i is not a multiple of $|\mathcal{N}|$. Then let i' be the next multiple of $|\mathcal{N}|$ larger than i. We have $i < i' < i + |\mathcal{N}|$ and by case 1 we know that $i' < |\mathcal{N}| \cdot |A^{c_3+i'}|$. Also $|A^j| \leq |A^{j'}|$ when $j \leq j'$, since $A^j \subseteq A^{j'}$. Combining these observations we get:

$$i < i' < |\mathcal{N}| \cdot |A^{c_3+i'}| < |\mathcal{N}| \cdot |A^{c_3+i+|\mathcal{N}|}|.$$

When we choose $c = c_3 + |\mathcal{N}|$, we have $|\mathcal{N}| \cdot |A^{i+c}| > i$, for $i \geq 0$. □

The corollary above readily implies the following.

Corollary 2.24 (Intermediate lower bound). *There exists a constant c such that*

$$\forall i \geq 0 : |\mathcal{N}| \cdot |\mathcal{T}^{i+c}| \geq i.$$

Proof. Fix some arbitrary $A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}$. Clearly $|\mathcal{T}^j| \geq |A^j|$ for all j. Then, by Corollary 2.23 there exists a constant c such that:

$$|\mathcal{N}| \cdot |\mathcal{T}^{i+c}| \geq |\mathcal{N}| \cdot |A^{i+c}| \geq i.$$ □

2.4.4 Length Bound

In this section, we establish a relation between the length of the longest string and the total number of generated strings up to an iteration. We begin by stating the following lemma, which expresses the existence of a shorter string of bounded length.

Lemma 2.25. *Let $s \in A^i$ with $|s| \geq 2^{|\mathcal{N}|}$. Then A^i also contains a shorter string s' with*

$$|s| - 2^{|\mathcal{N}|} < |s'| < |s|.$$

Proof. Consider a parse tree τ for s of depth δ. Since $|s| \geq 2^{|\mathcal{N}|}$, it follows from Lemma 2.6 that $\delta \geq |\mathcal{N}| + 1$.

Let π be a path of maximal length from root to leaf in τ. Let π' be the final segment of π of length $|\mathcal{N}|+1$ (π has length $\geq |\mathcal{N}|+1$ since $\delta \geq |\mathcal{N}|+1$). On π', some non-terminal B occurs more than once; let nodes x and y, in that order on π', be labeled with B. The yields of the subtrees rooted at x and y are denoted by s_x and s_y respectively.

In τ we can now replace the subtree τ_x rooted at x with the subtree τ_y rooted at y. Since the depth of τ' is at most that of τ, and $s \in A^i$, it follows from Lemma 2.5 that also $s' \in A^i$. The resulting parse tree τ' has a yield s' with length

$$|s'| = |s| - |s_x| + |s_y|. \tag{2.1}$$

Every node on the path from x to y (x included, y excluded) has precisely two non-terminal children: one ancestor of y and one non-ancestor of y. Each subtree rooted at a non-ancestor of y has a non-empty yield. It follows that the yield of τ_y is a strict substring of the yield of τ_x and hence $|s_y| - |s_x| < 0$. It now follows from (2.1) that

$$|s'| < |s|. \tag{2.2}$$

Furthermore, since the depth of τ_x is at most $|\mathcal{N}|+1$, the string yielded by τ_x has length at most $2^{|\mathcal{N}|}$: $|s_x| \leq 2^|\mathcal{N}|$. We also know that τ_y yields a string of at least length 1. It now follows from (2.1) that

$$|s'| \geq |s| - 2^{|\mathcal{N}|} + 1 \geq |s| - 2^{|\mathcal{N}|}. \tag{2.3}$$

Combining (2.2) and (2.3) gives us the desired length bounds for s'. □

The existence of shorter strings of bounded size makes it possible to bound the size of the maximum string.

Lemma 2.26. $\forall A \in \mathcal{N}, \forall i \geq |\mathcal{N}| - 1 : \omega_A^i < 2^{|\mathcal{N}|} \cdot |A^i|$.

Proof. Since $i \geq |\mathcal{N}| - 1$, we know by Corollary 2.9 that there exists some $s \in A^i$. If $\omega^i_A < 2^{|\mathcal{N}|}$ then the lemma is trivial. Else, let $j = \lfloor \omega^i_A / 2^{|\mathcal{N}|} \rfloor$. Lemma 2.25 can be repeatedly applied at least j times, starting from $s_0 = s$, yielding j additional distinct strings $s_1, s_2, \ldots, s_j \in A^i$. Hence, $|A^i| \geq j + 1$, and therefore $|A^i| > \omega^i_A / 2^{|\mathcal{N}|}$. □

Lemma 2.26 readily implies the bound on the length of the maximum string that was announced at the beginning of this subsection. We establish a relation between the length of the longest string and the total number of strings generated:

Corollary 2.27 (Length bound). $\forall i \geq |\mathcal{N}| - 1 : \omega^i_{\mathcal{T}} < 2^{|\mathcal{N}|} \cdot |\mathcal{T}^i|$.

Proof. For a given $i \geq |\mathcal{N}| - 1$, let $s \in \mathcal{T}^i$ be a string of maximal length: $s = \omega^i_{\mathcal{T}}$. By definition, $s \in A^i$ and $|s| = \omega^i_A$, for some $A \in \mathcal{N}$ (otherwise s could not have maximal length). Now by Lemma 2.26 we know obtain the desired inequality:

$$\omega^i_{\mathcal{T}} = |s| = \omega^i_A < 2^{|\mathcal{N}|} \cdot |A^i| \leq 2^{|\mathcal{N}|} |\mathcal{T}^i|.$$ □

2.4.5 Intermediate String Bound

In this section we bound the number of intermediate strings by the number of output strings.

Lemma 2.28 (Past bound). $\forall i \geq 0 : \forall 0 \leq k \leq i : |\mathcal{T}^i| \leq 2^{2^k - 1} \cdot |\mathcal{T}^{i-k}|^{2^k}$.

Proof. We prove the lemma by induction on i.

Basis For $i = 0$, the only possible value for k is 0 and the inequality $|\mathcal{T}^0| \leq 2^{2^0 - 1} \cdot |\mathcal{T}^0|^{2^0}$ becomes trivial.

Induction Step For $i > 0$, assume the lemma holds for $i - 1$:

$$|\mathcal{T}^{i-1}| \leq 2^{2^k - 1} \cdot |\mathcal{T}^{i-1-k}|^{2^k} \qquad (0 \leq k \leq i - 1).$$

By the remark made after introducing the concatenation scheme in Section 2.3, we have for all $j \geq 0 : \mathcal{T}^{j+1} \subseteq (\mathcal{T}^j \cdot \mathcal{T}^j) \cup \mathcal{T}^0$. We get:

$$\begin{aligned}
|\mathcal{T}^i| &\leq |(\mathcal{T}^{i-1} \cdot \mathcal{T}^{i-1}) \cup \mathcal{T}^0| \\
&\leq |\mathcal{T}^{i-1}| \cdot |\mathcal{T}^{i-1}| + |\mathcal{T}^0| \\
&= |\mathcal{T}^{i-1}|^2 + |\mathcal{T}^0| \\
&\leq 2 \cdot |\mathcal{T}^{i-1}|^2 && (\mathcal{T}^0 \subseteq \mathcal{T}^{i-1}) \\
&\leq 2 \cdot \left(2^{2^k-1} \cdot |\mathcal{T}^{i-1-k}|^{2^k}\right)^2 && (\text{by induction hypothesis, } 0 \leq k \leq i-1) \\
&= 2^{2^{k+1}-1} \cdot |\mathcal{T}^{i-(k+1)}|^{2^{k+1}} && (0 \leq k \leq i-1) \\
&= 2^{2^k-1} \cdot |\mathcal{T}^{i-k}|^{2^k} && (1 \leq k \leq i) \\
&= 2^{2^k-1} \cdot |\mathcal{T}^{i-k}|^{2^k} && (0 \leq k \leq i, k = 0 \text{ is immediate}).
\end{aligned}$$

Hence, for $0 \leq k \leq i$, we obtain

$$|\mathcal{T}^i| \leq 2^{2^k-1} \cdot |\mathcal{T}^{i-k}|^{2^k}.$$ □

Lemma 2.29 (String Growth). *For each edge $A \rightarrow B$ in the dependency graph the following holds:*

$$\forall i \geq |\mathcal{N}| - 1 : \forall s \in B^i : \exists s' \in A^{i+1} : s \textit{ is a strict substring of } s'.$$

Proof. In iteration $i \geq |\mathcal{N}| - 1$, every non-terminal contains at least one non-empty string (Corollary 2.9). The edge from A to B indicates the presence of either a rule $A \rightarrow BC$ or $A \rightarrow CB$ for some non-terminal C. The string $s \in B^i$ will be concatenated with a string $u \in C^i$ to form some string $s' \in A^{i+1}$, where $s' = s \cdot u$ or $s' = u \cdot s$, depending on the production. Clearly s is a strict substring of s'. □

We are now in a position to formulate the Future and Present bounds, which will be important in the proof of our main theorem. The future bound gives an upper bound for the number of intermediate strings in terms of the number of output strings in a later iteration. This is subsequently used together with the past bound to obtain the present bound which bounds the number of intermediate strings in terms of the number of output strings in *the same* iteration.

Lemma 2.30 (Future bound). $\forall i \geq |\mathcal{N}| - 1 : |\mathcal{T}^i| \leq (\omega_S^{i+|\mathcal{N}|})^2 \cdot |S^{i+|\mathcal{N}|}|$.

Proof. We first prove that every intermediate string will appear as a substring of an output string, several iterations later. Next, we bound the number of substrings of these output strings to obtain the desired bound.

Any string $s \in \mathcal{T}^i$ appears in A^i for some A. Consider a simple path π from S to A in the dependency graph; π has length at most $|\mathcal{N}|$. By repeatedly applying Lemma 2.29 we know that $S^{i+l(\pi)}$ contains a string s' that is a superstring of s. Because $l(\pi) \leq |\mathcal{N}|$ it holds that $s' \in S^{i+l(\pi)} \subseteq S^{i+|\mathcal{N}|}$. Hence, each string in $\mathcal{T}^i$ has a superstring in $S^{i+|\mathcal{N}|}$.

The number of substrings of a string s' is bounded by $|s'|^2$. Consequently, the number of substrings we can create using strings in $S^{i+|\mathcal{N}|}$ is bounded by $(\omega_S^{i+|\mathcal{N}|})^2 \cdot |S^{i+|\mathcal{N}|}|$. Together with the first observation, this yields

$$|\mathcal{T}^i| \leq (\omega_S^{i+|\mathcal{N}|})^2 \cdot |S^{i+|\mathcal{N}|}|. \qquad \square$$

Theorem 2.31 (Present bound). $\exists c \in \mathbb{N} : \forall i \geq |\mathcal{N}| - 1 : |\mathcal{T}^i| \leq c \cdot |S^i|^{2^{|\mathcal{N}|+2}}$.

Proof. We combine the Future and Past bounds to bound $|\mathcal{T}^i|$:

$$\begin{aligned}
|\mathcal{T}^i| &\leq 2^{2^{|\mathcal{N}|}-1} \cdot |\mathcal{T}^{i-|\mathcal{N}|}|^{2^{|\mathcal{N}|}} && \text{(Past bound)}\\
&\leq 2^{2^{|\mathcal{N}|}-1} \cdot \left((\omega_S^i)^2 \cdot |S^i|\right)^{2^{|\mathcal{N}|}} && \text{(Future bound)}\\
&= 2^{2^{|\mathcal{N}|}-1} \cdot (\omega_S^i)^{2^{|\mathcal{N}|+1}} \cdot |S^i|^{2^{|\mathcal{N}|}} &&\\
&\leq 2^{2^{|\mathcal{N}|}-1} \cdot (2^{|\mathcal{N}|} \cdot |S^i|)^{2^{|\mathcal{N}|+1}} \cdot |S^i|^{2^{|\mathcal{N}|}} && \text{(Lemma 2.26)}\\
&\leq c \cdot |S^i|^{2^{|\mathcal{N}|+2}}. && \square
\end{aligned}$$

Although we will not make use of the following, we find that we can also bound the size of the maximum intermediate string in the size of the maximum output string of the same iteration.

Lemma 2.32 (Present length bound). $\forall i \geq |\mathcal{N}| - 1 : \omega_{\mathcal{T}}^i \leq (\omega_S^i)^{2^{|\mathcal{N}|}}$.

Proof. Consider a string s in $\mathcal{T}^i$ having a length of $\omega_{\mathcal{T}}^i$, with $s \in A^i$. In the dependency graph, there exists a simple path π from S to A having a length at most $|\mathcal{N}|$. This means there exists a string $s' \in S^{i+|\mathcal{N}|}$ that is a *strict* superstring of s (Lemma 2.29). Therefore:

$$\omega_{\mathcal{T}}^i < \omega_S^{i+|\mathcal{N}|}.$$

On the other hand, it is obvious that in the worst case, the maximal string length doubles each iteration, therefore:

$$\omega_{\mathcal{T}}^i \leq (\omega_{\mathcal{T}}^{i-|\mathcal{N}|})^{2^{|\mathcal{N}|}}.$$

Combining the inequalities above gives us:

$$\omega_{\mathcal{T}}^i \leq (\omega_{\mathcal{T}}^{i-|\mathcal{N}|})^{2^{|\mathcal{N}|}} < (\omega_S^{i-|\mathcal{N}|+|\mathcal{N}|})^{2^{|\mathcal{N}|}} = (\omega_S^i)^{2^{|\mathcal{N}|}}. \qquad \square$$

2.5 IPT Proof for the Naive Algorithm

In order to prove that our naive generation algorithm yields an enumeration in *incremental polynomial time* in the sense of Johnson et al. [111], we only require the following proposition, which we prove using the results obtained above:

Proposition 2.33. *There exists a fixed polynomial p such that after each iteration i, the total time spent by the naive algorithm described in Algorithm 2 so far is bounded by $p(|S^{i-1}|)$.*

Proof. We will first look at the time necessary to generate one string, then at the time necessary to generate one iteration and finally at the time needed to generate strings up to an iteration i.

Consider an intermediate string $s \in A^i$. When $i = 0$, the only thing that needs to happen is to store s, given that there are no duplicate productions. When $i > 0$, the following steps need to be performed:

1. concatenate two strings to form s;
2. check if the string has already been generated for A (duplicate check);
3. save the string in order to check for duplicates later.

The concatenation of two strings, resulting in s, can be done in time $\mathcal{O}(|s|)$. A lookup and insertion, to keep track of the set of generated strings, can both be done in time $\mathcal{O}\big(|A^i| \cdot |s|\big)$.[9]

Next, we construct a bound for the total number of intermediate strings calculated in iteration $i > 0$. In the worst case, all strings in $\mathcal{T}^{i-1}$ will be pairwise combined, for each production. Hence, the total number of candidates in iteration i is bounded by $|\mathcal{P}| \cdot |\mathcal{T}^{i-1}|^2$, where $|\mathcal{P}|$ is equal to the number of productions in the grammar.

Combining the two observations above gives us an upper bound on the total work in iteration i: $\mathcal{O}\big(\omega^i_{\mathcal{T}} \cdot |\mathcal{T}^i| \cdot |\mathcal{T}^{i-1}|^2\big)$. From the Past bound we know that $|\mathcal{T}^i| = \mathcal{O}\big(|\mathcal{T}^{i-1}|^2\big)$. The total work done up to and including iteration i is therefore bounded by

$$\mathcal{O}\left(\sum_{j=1}^{i} \omega^j_{\mathcal{T}} \cdot |\mathcal{T}^{j-1}|^4\right).$$

Note that the work in iteration 0 is constant, since it requires storing just one string for each terminal production. The work in the first $|\mathcal{N}|$ iterations is

[9]Actually, a much better bound can be obtained. However, to avoid being overly technical we stick to the bound given here, as this does not compromise the desired polynomial bound.

also bounded by a constant:

$$\sum_{j=1}^{|\mathcal{N}|-1} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 \leq |\mathcal{N}| \cdot \omega_{\mathcal{T}}^{|\mathcal{N}|} \cdot |\mathcal{T}^{|\mathcal{N}|-1}|^4 = \mathcal{O}(1)\,.$$

Hence, the total time spent up to and including iteration $|\mathcal{N}|-1$ is considered constant.

In the remainder of the proof, we bound $\mathcal{O}\Big(\sum_{j=|\mathcal{N}|}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4\Big)$ by a polynomial in $|S^{i-1}|$. First, observe the following:

$$\begin{aligned}
\sum_{j=|\mathcal{N}|}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 &\leq i \cdot \omega_{\mathcal{T}}^{i} \cdot |\mathcal{T}^{i-1}|^4 \\
&< c_1 \cdot i \cdot |\mathcal{T}^{i}| \cdot |\mathcal{T}^{i-1}|^4 && \text{(Corollary 2.27)} \\
&\leq c_2 \cdot |\mathcal{T}^{i+c_3}| \cdot |\mathcal{T}^{i}| \cdot |\mathcal{T}^{i-1}|^4 && \text{(Corollary 2.24)} \\
&\leq c_4 \cdot |\mathcal{T}^{i-1}|^{c_5} && \text{(Past bound)} \\
&\leq c_6 \cdot |S^{i-1}|^{c_7} && \text{(Present bound)}
\end{aligned}$$

for constants $c_1, \ldots, c_7$. Note that the applied lemmas only hold from iteration $|\mathcal{N}|-1$ on. This is not a problem as they are only applied for $j \geq |\mathcal{N}|$. From the above we can conclude:

$$\sum_{j=1}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 = \mathcal{O}\big(|S^{i-1}|^c\big)\,,$$

for some constant c. Therefore, the time needed by the algorithm to calculate all intermediate strings up to and including interation i is bounded by $\mathcal{O}\big(|S^{i-1}|^c\big)$, which is clearly polynomial in the size of S^{i-1}, as desired. □

We are now ready to prove the result central to this chapter:

Theorem 2.34 (Naive algorithm is IPT). *There is a fixed polynomial p such that the entire language $\mathcal{L}(G)$ can be enumerated without duplicates in such a way that the time needed to output the $(m+1)$th output string is bounded by $p(m)$.*

Proof. Consider the $(m+1)$th output string s. We know that $s \in \Delta S^i$ for some i and we also know, by Proposition 2.33 that the time needed to calculate all strings up to and including iteration i is bounded by $\mathcal{O}\big(|S^{i-1}|^c\big)$, for some constant c. Since $|S^{i-1}| \leq m$, we obtain a polynomial in m as desired. □

Remark 2.35 (IPT correspondence). Theorem 2.34 states that the time between the start of the enumeration process and the $(m+1)$th string is bounded by a polynomial in m. This does not fully correspond to the original definition of IPT (see Section 2.1), where the elapsed time is measured *between output m and $m+1$*. It should be clear that both characterizations are equivalent, as the time since the previous string was output is even shorter. ◇

2.6 From Given-Length to Infinite Enumeration

The purpose of this section is to establish a link between two types of problems: given-length enumeration with polynomial delay (GLEPD) and infinite enumeration in incremental polynomial time (IEIPT). We show that we can always use an algorithm for GLEPD to obtain an algorithm for IEIPT.

For a fixed context-free grammar G, consider a GLEPD-algorithm that, given a natural number n, enumerates all strings $w \in \mathcal{L}(G)$ with $|w| = n$. We treat the algorithm as a black box and denote it by $\textsc{Enumerate}_G(n)$. The polynomial delay property holds for the algorithm: there exists a fixed polynomial p_D such that, on input n, the time before the first output, the time between two outputs and the time after the last output until the algorithm terminates, is bounded by $p_D(n)$.

From this algorithm, we can immediately derive Algorithm 3, which is denoted by $\textsc{Enumerate}_{G,\infty}$. We now prove that $\textsc{Enumerate}_{G,\infty}$ enumerates the entire language $\mathcal{L}(G)$ in IPT.

Algorithm 3 Infinite enumeration in incremental polynomial time (IEIPT) algorithm $\textsc{Enumerate}_{G,\infty}$, based on a given-length enumeration in polynomial delay (GLEPD) algorithm $\textsc{Enumerate}_G(i)$.

```
for $i := 1 \to \infty$ do
    $\textsc{Enumerate}_G(i)$
```

Lemma 2.36. *For each infinite context-free language $\mathcal{L}$, there exist two constants $c \in \mathbb{N} \setminus \{0\}$ and $d \in \mathbb{N}$ such that for each $l \in \mathbb{N}$ the language $\mathcal{L}$ contains at least one string of length $c \cdot l + d$.*

Proof. Consider a context-free grammar G such that $\mathcal{L}(G) = \mathcal{L}$. Let S be the start symbol of G. We know S must be recursive or leeching for $\mathcal{L}(G)$ to be infinite. From Lemma 2.15 and Lemma 2.20 we know that there exist two constants $c, d \in \mathbb{N}$ such that for each $l \in \mathbb{N}$ there is a string $s \in \mathcal{L}(G)$ with $|s| = c \cdot l + d$. □

Theorem 2.37 (Enumerate$_{G,\infty}$ is IPT). *Let G be a context-free grammar. There exists a fixed polynomial p such that in* Enumerate$_{G,\infty}$ *the time spent between the mth output and the $(m+1)$th output is bounded by $p(m)$, where $m > 0$.*

Proof. Consider the mth and the $(m+1)$th output strings that are generated consecutively by the algorithm and denote them by s_m and s_{m+1}, respectively. For algorithm Enumerate$_G(n)$ we have the polynomial $p_D(n)$, guaranteed by the polynomial delay property. We may assume p_D is monotonically increasing over the natural numbers.[10]

There are two cases to consider:

1) $|s_m| = |s_{m+1}|$.
   This means that the strings are generated in the same iteration i. Let c and d be the constants given by Lemma 2.36. We consider two additional subcases:

   (a) $i \leq d$.
   Let c_0 be the total time performed by algorithm Enumerate$_{G,\infty}$ in the iterations up to and including iteration d. Then clearly the time between the outputs s_m and s_{m+1} is bounded by c_0.

   (b) $i > d$.
   Let $l = \lfloor \frac{i-1-d}{c} \rfloor$. By Lemma 2.36, at least $l+1$ strings have already been generated before iteration i. Hence $l+1 < m$. As $l = \lfloor \frac{i-1-d}{c} \rfloor < m$, we obtain $i \leq m \cdot c + d$. As the time between s_m and s_{m+1} is bounded by $p_D(i)$, it is also bounded by $p_D(m \cdot c + d)$, because p_D is monotonically increasing. This is clearly a polynomial in m.

2) $|s_m| < |s_{m+1}|$.
   This means that the strings are generated in different iterations. Let i be the iteration in which s_m was generated and j be the iteration in which s_{m+1} was generated. Clearly, $1 \leq i < j$. The total time spent between outputs s_m and s_{m+1} consists of three parts:

   - the time spent in iteration i after the generation of s_m;
   - the time spent in iteration j before the generation of s_{m+1};
   - the time spent in iterations $i+1, \ldots, j-1$.

   The first two parts are bounded by $p_D(i)$ and $p_D(j)$ respectively. In every iteration k between i and j, the time needed to verify that there is no

[10] If this is not the case, p_D can be made monotonically increasing by converting all negative coefficients to positive.

string of length k in $\mathcal{L}(G)$ is bounded by $p_D(k)$. Hence, the total time spent between s_m and s_{m+1} is bounded by

$$p_D(i) + p_D(j) + \sum_{k=i+1}^{j-1} p_D(k) \leq p_D(j) + p_D(j) + (j-2) \cdot p_D(j) = j \cdot p_D(j).$$

We know from Lemma 2.36 that the maximal number of consecutive lengths for which no string exists is bounded by a constant. Hence, for some constant c_{wait} we have $j - i \leq c_{\text{wait}}$. The total time spent between s_m and s_{m+1} is therefore bounded by

$$p'(i) := (i + c_{\text{wait}}) \cdot p_D(i + c_{\text{wait}}), \qquad \square$$

which is clearly a polynomial in i. As in the previous case, $i \leq c \cdot m + d$, so we obtain $p'(c \cdot m + d)$ as a polynomial in m.

The proof is completed by taking for $p(m)$ the larger of the two polynomials from the two cases and adding the constant c_0.

2.7 Semi-Naive Optimization

We consider the semi-naive concatenation scheme that allows for faster enumeration in practice and show that the class of unambiguous grammars allows for an accurate prediction of the number of new strings in each iteration when used with this scheme.

2.7.1 The Semi-Naive Concatenation Scheme

The algorithm performs a lot of redundant concatenations. Consider, e.g., a production $A \to BC$. In iteration $i+1$, all strings in B^i and C^i are combined with each other and the *new* strings are added to ΔA^{i+1}. But, in iteration i, the strings of B^{i-1} and C^{i-1} have already been combined. Because $B^{i-1} \subseteq B^i$ and $C^{i-1} \subseteq C^i$ it is obvious that $|B^{i-1}| \cdot |C^{i-1}|$ concatenations are not new strings for A. This means in each iteration more and more redundant calculations are performed. A solution to this problem consists of using a so-called *semi-naive* scheme [54], which is often utilized in datalog evaluation where it serves the same purpose: avoiding repeated calculations [3, 6, 151].

The idea behind semi-naive evaluation is to only concatenate the new strings with each other and with the old strings and hereby avoiding old-old combinations. For a production $A \to BC$, we perform the following concatenations in iteration $i+1$:

Algorithm 4 Semi-naive concatenation scheme for context-free grammars.

```
    $A^0 = \{a \in \Sigma \mid A \to a \in \mathcal{P}\}$;
  $A^{i+1} = A^i \cup \{u \cdot v \mid \exists A \to BC \in \mathcal{P}$ :
           $(u \in B^i \wedge v \in \Delta C^i) \vee (u \in \Delta B^i \wedge v \in C^{i-1})\}$; and,
$\Delta A^{i+1} = A^{i+1} \setminus A^i$.
```

- ΔB^i with ΔC^i;
- ΔB^i with C^{i-1}; and,
- B^{i-1} with ΔC^i.

This yields the inductive scheme depicted in Algorithm 4, which we call the *semi-naive concatenation scheme.* The main advantage is summarized in the following theorem.

Theorem 2.38. *When applying the semi-naive concatenation schema, each pair of strings is concatenated at most once for each production, during the execution of the entire algorithm.*

Proof. Consider a production $A \to BC$, the strings $u \in \Delta B^i$ and $v \in C^i$ and let $s = uv \in A^{i+1}$. Now consider a concatenation in iteration $j \geq i+1$ for this production and two string u' and v'. This is only possible in the following situations

1. $u' \in B^j \wedge v' \in \Delta C^j$, or
2. $u' \in \Delta B^j \wedge v' \in C^{j-1}$.

Note that from the first case $v' \neq v$ and from the second case $u' \neq u$, hence, u and v cannot be concatenated again. The proof is analogous when $u \in B^i$ and $v \in \Delta C^i$, which covers the other case. □

Remark 2.39 (Duplicate string exception). Note that it *is* possible that the string $s = uv$ is generated multiple times by a certain production. This is the case when two other strings can be concatenated to yield s, i.e., $s = uv = u'v'$ with $u \neq u'$ and $v \neq v'$. Note that the parse tree for such strings is different.◇

This result allows us to conclude that up to iteration i at most $|\mathcal{T}^{i-1}|^2 \cdot |\mathcal{P}|$ concatenations have been performed using the semi-naive scheme, instead of $i \cdot |\mathcal{T}^{i-1}|^2 \cdot |\mathcal{P}|$ when using the original scheme. This has no influence on the IPT property that was shown in Proposition 2.33 as it only enables us to construct a lower-order polynomial to bound the work.

2.7.2 Unambiguous Grammars

A context-free grammar G is called *unambiguous* when every string in $\mathcal{L}(G)$ has exactly one parse tree rooted at S. If we use the semi-naive concatenation scheme with an unambiguous grammar, it is easy to see that every string that is generated for a non-terminal A is a previously unseen (i.e., new) string for A. Indeed, two productions of the same non-terminal cannot yield the same string anymore (only one parse tree per string) and one production can only yield the same string once (each parse tree is generated exactly once). These findings are summarized in the following lemma.

Lemma 2.40. *When using an unambiguous grammar in combination with the semi-naive scheme, every string calculated by the algorithm for a non-terminal A is* unique *for A.*

This makes it unnecessary check to see if the string has already been generated. Another advantage is that the depth of a string's parse tree now reflects the iteration it was generated. Previously, this was not the case, as multiple parse trees of different depths were allowed to represent the same string.

The lemma above allows us to calculate exactly how many strings will be concatenated for each production.

Theorem 2.41. *For an unambiguous grammar, enumerated using the semi-naive concatenation scheme, the number of new strings introduced in iteration $i+1$ for a production $A \to BC$ equals*

$$|B^i| \cdot |\Delta C^i| + |\Delta B^i| \cdot |C^{i-1}|.$$

Proof. Immediate from the scheme in Algorithm 4 and Lemma 2.40. □

2.8 Discussion

The fact that the simple algorithm, based on the naive bottom-up concatenation scheme and described in Section 2.3, already achieves the Incremental Polynomial Time criterion, is, we hope, an interesting theoretical (if not didactical) contribution, as we have not seen this mentioned elsewhere. An important caveat is that the context-free grammar G is considered fixed and not part of the input. An interesting question to investigate is what happens when G *is* part of the input.

An elementary approach as presented here has the best chances of being generalizable. Indeed, the insights developed here can possibly be extended to apply to the more general setting of context-free sets of arbitrary combinatorial objects as introduced by Courcelle and Engelfriet [62] and Flajolet et al. [81] and Flajolet and Sedgewick [82]. A major additional problem in this context is to keep the duplicate check (step 2 in the proof of Proposition 2.33) polynomial. Fortunately, in the HR approach to graph rewriting, every context-free graph language has bounded treewidth. In combination with imposing connectedness and a degree bound [133] this may produce a polynomial duplicate check.

We also note that for unambiguous grammars, the methods of Flajolet and Sedgewick [82] can be used to count exactly the number of strings (or derivation trees, which coincides for unambiguous grammars) of a given size.

3

Discovering XSD Keys from XML Data

We study the problem of XML key inference in the presence of a schema (XSD), which corresponds to an instance of the pattern mining problem. We discuss several other criteria for assessing the interestingness of a key and show that the key consistency problem is in PTIME. We present a key mining algorithm that follows a level-wise approach to discover all keys in XML documents and experimentally validate its performance and applicability using a real-world dataset.

3.1 Introduction

The automatic discovery of constraints from data is a fundamental problem in the scientific database literature, especially in the context of the relational model in the form of key, foreign key, and functional dependency discovery (see, e.g., [127]). Although the absence of DTDs and XML Schema Definitions (XSDs) for XML data occurring in the wild has driven a multitude of research on learning of XML schemas [39–43, 87], the automatic inference of constraints has been left largely unexplored (we refer to Section 3.2 for a discussion on related work). In this chapter, we address the problem of *XML key mining* whose core formulation asks to find all XML keys valid in a given XML document. We use a formalization of XSD keys (defined in Section 3.3) consistent with the definition of XML keys by W3C [161]. We develop a key mining algorithm within the framework of levelwise search that additionally leverages discovery algorithms for functional dependencies in the relational model. Our algorithm iteratively refines keys based on a number of quality requirements;

To illustrate the challenges of key mining in the presence of a schema, we first introduce the following example.

Example 3.1 (Basic XML key). Consider the key

$$\phi := \big(\underbrace{(\text{order}, q_{\texttt{order}})}_{\text{context } c}, \quad \underbrace{\texttt{.//book}}_{\text{target path } \tau}, \quad \underbrace{(\texttt{.//title}, \texttt{.//year})}_{\text{key paths } p_1, p_2, \dots}\big).$$

Here, the pair $(\text{order}, q_{\texttt{order}})$ is a *context* consisting of the label 'order' and the state or type[11] $q_{\texttt{order}}$, which identifies the context-nodes for which ϕ is to be evaluated. Furthermore, `.//book` is an XPath-expression, called *target path*, selecting within every context node a set of target nodes. The key constraint now states that every target node within a context must be uniquely identified by the record determined by the key paths `.//title` and `.//year`, which are XPath-expressions as well. In other words, no two target nodes should have both the same title and the same year. A schematic representation of the semantics of a key is given in Figure 3.2. So, over the XML document t displayed in Figure 3.1, the key ϕ gives rise to the table $R_{\phi,t}$:

$(\text{order}, q_{\texttt{order}})$	`.//book`	`.//title`	`.//year`
$(o_1,$	$b_1,$	*'Movie analysis'*,	*2012*)
$(o_1,$	$b_2,$	*'Programming intro'*,	*2012*)
$(o_2,$	$b_3,$	*'Programming intro'*,	*2012*)

In Figure 3.1, the names of the order and book nodes from left to right are o_1, o_2, and b_1, b_2, b_3, respectively, and every order node has type $q_{\texttt{order}}$. Then, ϕ holds in t if the functional dependency

$$(\text{order}, q_{\texttt{order}}), \texttt{.//title}, \texttt{.//year} \quad \to \quad \texttt{.//book}$$

holds in $R_{\phi,t}$. That is, within the same order node (a context node), 'title' and 'year' uniquely determine the 'book' element. △

As a necessary condition for a key to be valid on a tree t, the XML key specification [161, Section 3.11.4] requires every key path to always select precisely one node carrying a data-value.[12] The key is then said to *qualify* on t. As an example,

$$\phi' := \big((\text{bookshop}, q_{\texttt{bookshop}}), \quad \texttt{.//order}, \quad (\texttt{.//address})\big),$$

[11] Types are defined in the accompanying schema which is not given here but discussed in Section 3.3.2.

[12] Actually, the specification is a bit more general in allowing the use of attributes. For ease of presentation, we disregard attributes and let leaf nodes carry data values. We note that all the results in this chapter can be easily extended to include attributes.

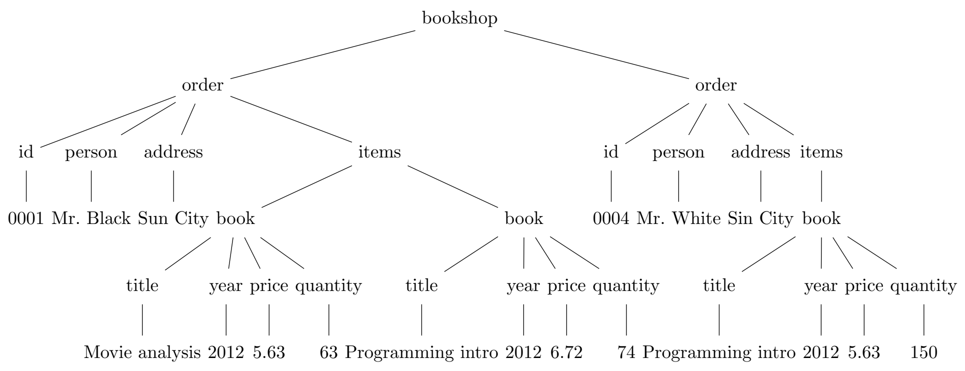


Figure 3.1: *XML tree example (order and book nodes are named o_1, o_2 and b_1, b_2, b_3 from left to right, respectively).*

qualifies for the particular tree given in Figure 3.1 (assuming every node labeled ‘bookshop’ has type $q_{\texttt{bookshop}}$) since every target node o_1 and o_2 has precisely one address node. But, the accompanying XSD might allow XML documents *without* an address or with *multiple* addresses, for which ϕ' would clearly not qualify. So, qualifying for the given document does not necessarily entail qualifying for every document in the schema. We say that a key is *consistent* w.r.t. an XSD if the key qualifies on every document satisfying the XSD. As a quality criterion for keys, we want our mining algorithm to only consider consistent keys. We therefore study the complexity of deciding consistency and obtain the pleasantly surprising result that consistency can be tested in polynomial time for keys disallowing disjunction on the topmost level.

In addition to consistency, we want to enforce a number of additional quality requirements on keys. In particular, we want to disregard keys that can only select an a priori bounded number of target nodes independent of the size of the input document. Since the main purpose of a key is to ensure uniqueness within a collections of nodes, it does not make sense to consider *bounded* keys for which the size of this collection is fixed in advance and can not grow with the size of the document. Similarly, we want to ignore so-called *universal* keys that hold in every document.

After laying the groundwork above, we turn to the theme of mining. Example 3.1 indicates how XML key mining can leverage algorithms for the discovery of functional dependencies (FDs) over a relational database. Indeed, once a context c and a target path τ are determined, any FD of the form $c, p_1, \ldots, p_n \to \tau$ that holds in the relational encoding $R_{(c,\tau,\overline{P}),t}$ entails the key $(c, \tau, (p_1, \ldots, p_n))$ in t where $\overline{P}$ is a sequence consisting of all possible consistent key paths. Of course, it remains to investigate how to efficiently explore the search space of candidate contexts c, target paths τ, and consistent key paths p. To this end, we embrace the framework of levelwise search (as, e.g., described by Mannila and Toivonen [128]) to enumerate target and key paths. The components of this framework consist of a search space U, a Boolean search predicate q, and a specialization relation $\preceq$ that is a partial order on U and monotone w.r.t. q. In particular, the partial order arranges objects from most general to most specific and when q holds for an object then q should also hold for all generalizations of that object. The solution then consists of all objects $u \in U$ for which $q(u)$ holds, enumerated according to the specialization relation while avoiding testing objects for which q cannot hold anymore given already obtained information.

We define a target path miner within the framework above as follows: the search predicate holds for a target path when the number of selected target nodes (the support) exceeds a predetermined threshold value; and, the partial order $\preceq$ is determined by containment among target paths. To streamline computation, we utilize a syntactic one-step specialization relation $\prec_1$ that we prove to be optimal w.r.t. the considered partial order. Furthermore, the search predicate can be solely evaluated on a much smaller prefix tree representation of the input document and, therefore, does not need access to the original document. In addition, we define a one-key path miner which searches for all consistent single key paths p (w.r.t. the already determined context and target path). Specifically, the search predicate holds for a key path p when p selects *at most one* key node (w.r.t. the given context and target path). Even though consistency requires the selection of exactly one key node, this mismatch can be solved by confining the search space to all key paths that appear as paths from target nodes in the prefix tree. Even though the search predicate can not always be computed on the much smaller prefix tree without access to the original document, we provide sufficient conditions for when this is the case. The partial order is defined as the set inclusion relation defined on key paths for which the one-step specialization relation is the inverse of $\prec_1$. Once all consistent one-key paths are determined, as explained above, a functional dependency miner (e.g., [44, 125, 126]) can be used to determine the corresponding XML key.

Contributions. To summarize, our contributions are as follows.

1. We propose several measures to determine the interestingness of XML keys: boundedness, satisfiability, universality, and implication of XML keys, as well as equivalence of target paths (Theorem 3.26).

2. We characterize the complexity of the consistency problem for XML keys w.r.t. an XSD for different classes of target and key paths (Theorem 3.15).

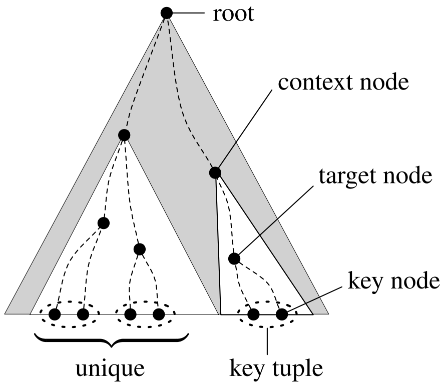


Figure 3.2: *Schematic representation of an XML key.*

3. We develop a novel key mining algorithm leveraging on algorithms for the discovery of relational functional dependencies and on the framework of levelwise search by employing an optimal one-step specialization relation for which the search relation can be computed, if not completely, then at least partly on a prefix tree representation of the document (Section 3.5).

4. We experimentally assess the effectiveness of the proposed algorithm on an extensive body of real world XML data and derive suitable values for several parameters for controlling the search space of the different components. (Section 3.6)

Outline. In Section 3.2, we discuss related work. In Section 3.3, we introduce the necessary definitions. In Section 3.4, we study key quality and investigate the complexity of key consistency w.r.t. an XSD. In Section 3.5, we discuss the XML key mining algorithm. In Section 3.6, we experimentally validate our algorithm. We conclude in Section 3.7.

3.2 Related work

XML Keys. One of the first definitions of keys for XML was introduced by Buneman et al. [52, 53]. These keys are of the form $(Q, (Q', P))$ where Q is the context-path, Q' is the target path and P is a set of key paths. Although the W3C definition of keys was largely inspired by this work, there are some important differences. First, Buneman et al.'s keys allow for more expressive target and key paths by allowing several occurrences of the descendant operator. Context paths, however are less expressive since W3C keys allow the context to be defined by an arbitrary Deterministic Finite State Automaton (DFA), while Buneman et al.'s keys limit themselves to path expressions. Furthermore, Buneman et al.'s key paths are allowed to select several nodes whereas W3C keys paths are restricted to select precisely one node. We stress that, in this work, we follow the W3C-specification for the definition of keys. As is the case for the relational model, much is known about the complexity of key inference for Buneman et al.'s keys [52, 53, 98]. Unfortunately, these results do not carry over to W3C keys as the latter are defined w.r.t. an XML Schema but the former are not.

Decision problems in the presence of a schema. A number of consistency problems of XML keys w.r.t. a DTD have been considered by Fan and Libkin [80]. They have shown, for instance, that key implication in the presence of a DTD is decidable in polynomial time. The keys that they consider, however, are much simpler than the W3C keys considered in this chapter. Basically, a key in their setting is determined by an element name and a number of attributes. Their model is subsumed by ours since each such key can be defined by an XML key and every DTD can be represented by an XSD. We point out that Fan and Libkin [80] provide many more results on the interplay between keys, foreign keys, inclusion dependencies and DTDs. Arenas et al. [27] discuss satisfiability[13] of XML keys w.r.t. a DTD. The result most relevant to this chapter is the NP-hardness of satisfiability w.r.t. a non-recursive DTD and for keys with only one key path. In the presence of XSDs, the problem becomes EXPTIME-hard (see Arenas et al. [25]).

XML constraint mining. The automatic discovery of Buneman et al.'s keys from XML data has previously been considered by a number of researchers. Grahne and Zhu [91] considered mining of approximatekeys and proposed an Apriori style algorithm which uses the inference rules of Buneman et al. [53] for optimization. Nečaský and Mlýnková [137] ignore the XML data but present an approach to infer keys and foreign keys from element/element joins in XQuery logs. Fajt et al. [76] consider the inference of keys and foreign keys building further on algorithms for the relational model. The algorithms above can not be used for W3C keys since they do not take the presence of XSDs into account and keys are not required to be consistent. Yu and Jagadish [169] consider discovery of functional dependencies (FDs) for XML. Similar to Buneman et al.'s keys, the considered FDs have paths that can select multiple data elements, and contexts are defined using a selector expression as opposed to using a DFA. For these reasons, W3C keys can not be encoded as a special case of FDs. Barbosa and Mendelzon [33] proposed algorithms to find ID and IDREFs attributes in XML documents. They show that the natural decision

[13]We note that satisfiability is called consistency in [27].

problem associated to this discovery problem is NP-complete, and present a heuristic algorithm. Abiteboul et al. [2] consider probabilistic generators for XML collections in the presence of integrity constraints but do not consider mining of such constraints.

3.3 Definitions

In this section, we introduce the required definitions concerning trees, XSDs, and XML keys, and formally define the XML key mining problem. The correspondence between our definition of XML keys and the W3C definition is discussed in Section 3.3.3.

3.3.1 Trees & XML

As is standard, we represent XML documents by means of labeled trees. Formally, for a set S, an *S-tree* is a pair (t, lab_t) where t is a finite tree and lab_t maps each node of t to an element in S. To reduce notation, we identify each tree simply by t and leave lab_t implied. We assume the reader to be familiar with standard common terminology on trees like *child*, *parent*, *root*, *leaf*, and so on. For a node v, we denote by $\text{anc-string}_t(v)$ the string formed by the labels on the unique path from t's root to (and including) v, called the *ancestor string* of v. By $\text{child-string}_t(v)$, we denote the string obtained by concatenating the labels of the children of v. If v is a leaf then $\text{child-string}_t(v)$ is the empty string, denoted by ε. Here, we assume that trees are sibling-ordered. We fix a finite set of element names Σ and an infinite set **Data** of data elements. An *XML-tree* is a $(\Sigma \cup \mathbf{Data})$-tree where non-leaf nodes are labeled with elements from Σ and leaf nodes are labeled with elements from $(\Sigma \cup \mathbf{Data})$. As the XSD specification does not allow 'mixed' content models[14] for fields in keys [161], we ignore mixed content models altogether to simplify presentation, and assume that when a node is labeled with a **Data**-element, it is the only child of its parent. We then denote by $\text{value}_t(v)$ the **Data**-label of v's unique child when it exists; otherwise we define $\text{value}_t(v) = \perp$, with $\perp$ a special symbol not in **Data**. When $\text{value}_t(v) \in \mathbf{Data}$, we also say that v is a **Data**-node.

Example 3.2. Figure 3.1 displays an XML-tree t. In this tree,

$$\text{anc-string}_t(b_1) = \texttt{bookshop order items book},$$

and also

$$\text{child-string}_t(b_1) = \texttt{title year price quantity}.$$

[14]A mixed content model allows nodes to have a mixture of **Data**-elements and normal nodes as its children.

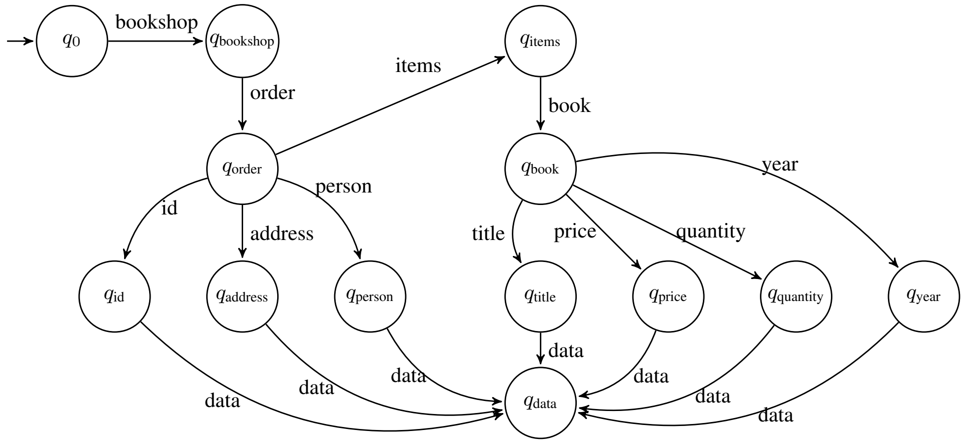


Figure 3.3: *Type automaton of XSD $X_{bookshop}$.*

Furthermore, every node labeled `id`, `person`, `address`, `title`, `year`, `price`, or `quantity` is a **Data**-node, while, for instance, b_1 is not. $\triangle$

3.3.2 XSDs

XML keys are defined within the scope of an XSD. We make use of the DFA-based characterization of XSDs introduced by Martens et al. [130]. An *XSD* is a pair $X = (A, \lambda)$ where $A = (\text{Types}, \Sigma \cup \{\texttt{data}\}, \delta, q_0)$ is a Deterministic Finite Automaton (or DFA for short) without final states (called the type-automaton) and λ is a mapping from Types to deterministic[15] regular expressions over the alphabet $\Sigma \cup \{\texttt{data}\}$. Here, Types is the set of states; `data` is a special symbol, not in Σ, which will serve as a placeholder for **Data**-elements; $\delta : \text{Types} \times \Sigma \cup \{\texttt{data}\} \to \text{Types}$ is the (partial) transition function; and $q_0 \in \text{Types}$ is the initial state. Additionally, the labels of transitions leaving q should be precisely the symbols in $\lambda(q)$. That is, for every $q \in \text{Types}$, $\mathit{Out}(q) = \mathit{Symb}(\lambda(q))$, where $\mathit{Out}(q) = \{\sigma \in \Sigma \cup \{\texttt{data}\} \mid \delta(q, \sigma) \text{ is defined}\}$ and $\mathit{Symb}(r)$ consists of all $(\Sigma \cup \{\texttt{data}\})$-symbols in regular expression r.

A *context* $c = (\sigma, q)$ is a pair in $\Sigma \times \text{Types}$. By $\text{CNodes}_t(c)$, we denote all nodes v of t for which $\text{lab}_t(v) = \sigma$ and A halts in state q when started in q_0 on the string $\text{anc-string}_t(v)$. Let $\mathcal{L}(r)$ denote the language defined by the regular expression r. We say that the tree t *adheres to* X, if for every context $c = (\sigma, q)$ and every v in $\text{CNodes}_t(c)$ one of the following holds:

[15]Also referred to as 1-unambiguous regular expressions [48].

- $\text{value}_t(v) \in \mathbf{Data}$ and $\texttt{data} \in \mathcal{L}(\lambda(q))$; or
- $\text{value}_t(v) = \bot$ and $\text{child-string}_t(v) \in \mathcal{L}(\lambda(q))$.

Intuitively, A determines the vertical context of a node v by the state q it reaches in processing $\text{anc-string}_t(v)$. When v is a **Data**-node, the content model specified by q, that is $\lambda(q)$, should contain the placeholder `data`. Otherwise, when v is not a **Data**-node, $\text{child-string}_t(v)$ should satisfy the content-model $\lambda(q)$. Recall that we do not allow mixed content models. We stress that this DFA-based characterization of XSDs corresponds precisely to the more traditional abstraction in terms of single-type grammars [131, 136]. We let $\mathcal{L}(X)$ denote the set of all trees adhering to XSD X. We assume that an XSD always defines trees with the same root label. In this way, the root is always assigned the same context, also referred to as the root context c_{root}.

Example 3.3 (XSD example). Let $X_{\text{bookshop}} = (A, \lambda)$ be the XSD where A is given in Figure 3.3 and λ is defined as follows:

$$\begin{aligned} q_0 &\mapsto \texttt{bookshop} \\ q_{\text{bookshop}} &\mapsto \texttt{order}^+ \\ q_{\text{order}} &\mapsto \texttt{id person address items}^+ \\ q_{\text{items}} &\mapsto \texttt{book}^+ \\ q_{\text{book}} &\mapsto \texttt{title year? price quantity} \end{aligned}$$

For all other types q, $\lambda(q) = \texttt{data}$. Here, r^+ and $r?$ are the usual abbreviations for rr^* and $r + \varepsilon$, where r is a regular expression. Then, tree t in Figure 3.1 adheres to X_{bookshop}. Moreover, $b_1 \in \text{CNodes}_t(\text{book}, q_{\text{book}})$ and $\text{child-string}_t(b_1) \in \mathcal{L}(\lambda(q_{\text{book}}))$. △

In some cases, it will be convenient to work with *trimmed* XSDs. Intuitively, we call an XSD *trimmed* if it does not contain useless states or transitions. The formal definition is as follows:

Definition 3.4 (Trimmed XSD). Let X be an XSD and let q_0 be its initial state. We call a context $c = (\sigma, q)$ *well-formed* (with respect to X) if there exists a type q' in X such that $\delta(q', \sigma) = q$, where δ is the transition function of the type-automaton of X. For type q in X, let X^q be the XSD obtained from X by replacing its start state by q. Then X is *trimmed* if (i) $\mathcal{L}(X^q) \neq \emptyset$ for all $q \neq q_0$; and, (ii) there is a tree $t \in \mathcal{L}(X)$ with $\text{CNodes}_t(\sigma, q) \neq \emptyset$, for every well-formed context (σ, q) of X.

Hence, in a trimmed XSD, there can be no state (except possibly the initial state q_0) that defines the empty tree language, and every well-formed context in A should be realized in at least one tree in $\mathcal{L}(X)$. The next lemma states that we can restrict attention to trimmed XSDs.

Lemma 3.5. *Every XSD can be converted in* PTIME *to a trimmed XSD defining the same language [25].*

3.3.3 XML Keys

A *selector expression* is a restricted XPath-expression [57] of the form

- $./l_1/l_2/\dots/l_k$ (starting with the child axis) or
- $.//l_1/l_2/\dots/l_k$ (starting with the descendant axis),

where $k \geq 1$, and $l_1, \dots, l_k$ are element names or the wildcard symbol *. A string $w = w_1 \cdots w_k$, where each w_i is an element name, is said to *match* $./l_1/l_2/\dots/l_k$ when $w_i = l_i$ or $l_i = *$ for each i. For selector expressions starting with the descendant axis, we say that w matches $.//l_1/l_2/\dots/l_k$ if a suffix of w matches $./l_1/l_2/\dots/l_k$. For a tree t, a node v of t, and a selector expression τ, the set $\tau(t, v)$ contains all nodes v' such that v' is a descendant of v and the path of labels from v (but excluding the label of v) to (and including) v' matches τ. A *disjunction of selector expressions* is of the form $\tau = \tau_1 \mid \cdots \mid \tau_m$ where each τ_i is a selector expression. In this case, $\tau(t, v)$ is defined as the union of all $\tau_i(t, v)$. When v is the root of the document, we simply write $\tau(t)$ for $\tau(t, v)$. We denote by $\mathcal{SE}$ and $\mathcal{DSE}$ the class of selector expressions and disjunctions of selector expressions, respectively. In proofs, we sometimes omit the leading dot in selector expressions and simply write $/l_1/l_2/\dots/l_k$ (or even $l_1/l_2/\dots/l_k$) or $//l_1/l_2/\dots/l_k$ to denote $./l_1/l_2/\dots/l_k$ and $.//l_1/l_2/\dots/l_k$, respectively.

Definition 3.6 (XML key). An *XML key*, defined w.r.t. an XSD X, is a tuple $\phi = (c, \tau, P)$, where (i) c is a context in X; (ii) $\tau \in \mathcal{DSE}$ is called the *target path*, and (iii), P is an ordered sequence of expressions in $\mathcal{DSE}$ called *key paths*. To emphasize that ϕ is defined w.r.t. X, we sometimes write a key simply as a pair (ϕ, X).

We stress that the definition of XML keys given above corresponds to the definition of keys in XML Schema [161]. In particular, the context is given implicitly by declaring a key inside an element which has a label and a certain type. Target paths are called *selector paths* and key paths are called *fields* [161, Paragraph 3.11.1]. They adhere to the same grammar as used here with the difference that we do not make use of attributes but require key paths to select data nodes.

The semantics of an XML key is as follows. The context c defines a set of context nodes which divides the document into separate (but not necessarily disjoint) parts. Specifically, each node in $\text{CNodes}_t(c) = \{v_1, \dots, v_n\}$ can be considered as the root of a separate tree. For each of those trees, i.e., for each

$i \in \{1, \ldots, n\}$, every node in $\tau(t, v_i)$ should uniquely define a record. Such a record is determined by the key paths in $P = (p_1, \ldots, p_k)$. That is, each v in $\tau(t, v_i)$ defines the record $[\text{value}_t(u_1), \ldots, \text{value}_t(u_k)]$, denoted by $\text{record}_P(t, v)$, where $p_j(t, v) = \{u_j\}$ for all $j \in \{1, \ldots, k\}$. We graphically illustrate the above in Figure 3.2.

Note that it is possible that $p_j(t, v)$ selects more than one node, selects a node u for which $\text{value}_t(u)$ is undefined, or selects nothing; these three cases are disallowed by the XML Schema specification as a key is required to qualify in a document.

Definition 3.7 (Key qualification). A key $\phi = (c, \tau, P)$ *qualifies* in a document t if for every $v \in \text{CNodes}_t(c)$, every $u \in \tau(t, v)$ and every $p \in P$, $p(t, u)$ is a singleton containing a **Data**-node.

Finally, following the W3C specification, we define satisfaction of an XML key w.r.t. a document as follows.

Definition 3.8 (Key satisfaction). An XML tree t *satisfies* a key $\phi = (c, \tau, P)$ or a key is *valid* w.r.t. t, denoted by $t \models \phi$, iff (i) ϕ qualifies in t; and, (ii) for every node v in $\text{CNodes}_t(c)$, $\text{record}_P(t, u) \neq \text{record}_P(t, u')$, for every two different nodes u and u' in $\tau(t, v)$.

Notice that there can be two causes for a key to be *invalid*: (i) the key does not qualify in the document and actually is ill-defined w.r.t. the document; or (ii) the data values in the document invalidate the key. The first cause can be seen as structural invalidation, while the second cause is semantical and more informative. We are mainly interested in inferring keys that *always* qualify in a document that satisfies the schema. We call such keys consistent. Deciding consistency is intractable when target and key paths are in $\mathcal{DSE}$ (proofs can be found in Arenas et al. [25]). However, when target and key paths are both in $\mathcal{SE}$, the problem becomes tractable, as we show in Section 3.4.

Definition 3.9 (Key consistency). A key is *consistent* w.r.t. a schema if the key qualifies in every document adhering to the schema.

Example 3.10 (Consistency). Consider the key from Example 3.1:

$$\phi := ((\text{order}, q_{\texttt{order}}), \texttt{.//book}, (\texttt{.//title}, \texttt{.//year}))$$

Then ϕ is valid w.r.t. the tree in Figure 3.1 but ϕ is not consistent w.r.t. X_{bookshop}. Indeed, X_{bookshop} defines the 'year'-element of a 'book'-element to be optional. Hence, it is possible to construct a document that adheres to X_{bookshop} for which the key does not qualify. $\triangle$

3.3.4 XML Key Mining Problem

Given an XML document t adhering to a given XSD, we want to derive all supported XML keys ϕ that are valid w.r.t. t.[16] We define the support of a key as the quantity measuring the number of nodes captured by the key. Define $\mathrm{TNodes}_t(\phi)$ as the set of target nodes selected by $\phi = (c, \tau, P)$ on t. That is,

$$\mathrm{TNodes}_t(\phi) = \bigcup_{v \in \mathrm{CNodes}_t(c)} \tau(t, v).$$

Then, following Grahne and Zhu [91], we define the *support* of ϕ on t to be the total number of selected target nodes: $\mathrm{supp}(\phi, t) = |\,\mathrm{TNodes}_t(\phi)|$. Since this support only depends on the context c and the target path τ of ϕ, we also write $\mathrm{supp}(c, \tau, t)$ for $\mathrm{supp}(\phi, t)$.

Example 3.11 (Support measure). Consider the document t depicted in Figure 3.1 and the key ϕ from Example 3.1. The context nodes are those that match the context $(\mathrm{order}, q_{\texttt{order}})$. There are two such nodes in t, both are labeled `order` and are direct children of the root node. The target nodes are those that can be selected from the two context nodes, using a path that matches `.//book`. There are three such target nodes, two for the first and one for the second context node. Therefore, the support of ϕ on t equals 3, that is, $\mathrm{supp}(\phi, t) = 3$. △

We are now ready to define the problem central to this chapter:

Definition 3.12 (XML key mining problem). For an XSD X, an XML document t adhering to X and a minimum support threshold N, the *XML key mining problem* consists of finding all keys ϕ consistent with X such that $t \models \phi$ and $\mathrm{supp}(\phi, t) > N$.

The above is only the core definition of the XML key mining problem. We will discuss some quality requirements that apply to the keys in Section 3.4.3.

3.4 Key Quality Complexity

A basic problem in data mining is the abundance of found patterns. In this section, we discuss several *key quality measures* that can be used to indicate low quality keys; these keys can then be removed from the output of the key mining algorithm or from the intermediate results. We focus on consistency testing and show that this problem becomes tractable when top-level

[16]W.l.o.g. and to simplify presentation, we restrict attention to a single document as multiple XML-documents can always be combined into one by introducing a common root.

disjunction is disallowed. In Section 3.4.1, we first define the main consistency problem and the XPath cardinality problem and show how we can use the latter to prove our main result. Section 3.4.2 contains the complexity analysis for the XPath cardinality problem in regard to the relevant XPath fragment. Finally, in Section 3.4.3, we give an overview of several alternative key quality measures that can be used to refine mining results and summarize the complexity classes of the associated decision problems.

3.4.1 Consistency & XPath Cardinality

As detailed in Section 3.3.3, the W3C specification requires keys to be consistent. We therefore define CONSISTENCY as the problem to decide whether ϕ is consistent w.r.t. X, given a key ϕ and an XSD X. In this section, we show that CONSISTENCY is in fact solvable in PTIME when patterns in keys are restricted to $\mathcal{SE}$. Actually, the PTIME result is also surprising since a minor variation of consistency is known to be EXPTIME-hard, as we explain below.

Consistency requires that on every document adhering to X, every key path should select precisely *one* data node for every target node. This is related to deciding whether an XPath selector expression selects *at least* and *at most* a given number of nodes, on every document satisfying a given XSD. Indeed, define $\forall^{\bullet k}_{\text{tree}}$ with $k \in \mathbb{N}$ and $\bullet \in \{<, =, >\}$ to be the problem of deciding, given an XSD X and a selector expression p, whether it holds that $|p(t)| \bullet k$, for every $t \in \mathcal{L}(X)$. We show in Lemma 3.14 that CONSISTENCY can be easily reduced to $\forall^{=1}_{\text{tree}}$. Although Björklund et al. [45] showed that $\forall^{>k}_{\text{tree}}$ is EXPTIME-complete, we obtain below that $\forall^{=1}_{\text{tree}}$ can in fact be solved in polynomial time through an intricate translation to the equivalence test for unambiguous tree automata [150].

XPath Cardinality Complexity

Because of its relevance to cardinality estimation of XPath result sets, we extend the problem $\forall^{\bullet k}_{\text{tree}}$ by restricting target and key paths to different fragments of XPath. To obtain a more complete picture, we also consider the class of all regular expressions, denoted by $\mathcal{RE}$.[17] For a regular expression r and a tree t, $r(t)$ then selects all nodes whose ancestor string[18] matches r. Furthermore, denote by $\mathcal{SE}$ the set of all selector expressions and by $\mathcal{SE}^{//}$ and $\mathcal{SE}^{*}$, the set of all selector expressions *without* descendant and wildcard, respectively (recall that $\mathcal{SE}$ denotes the set of all selector expressions without

[17]Hopcroft and Ullman [108] and Sipser [153] provide more information on regular languages and expressions.

[18]Defined in Section 3.3.2 as the string formed by the labels on the path from the root to the considered node.

$\mathcal{P}$	$\forall_{\text{tree}}^{>k,\mathcal{P}}$	$\forall_{\text{tree}}^{<k,\mathcal{P}}$	$\forall_{\text{tree}}^{=k,\mathcal{P}}$
$\mathcal{RE}$	EXPTIME-complete	in PTIME	in EXPTIME PSPACE-hard ($k \geq 1$)
$\mathcal{CSE}$	EXPTIME-complete	in PTIME	in EXPTIME PSPACE-hard ($k \geq 1$)
$\mathcal{DSE}$	EXPTIME-complete	in PTIME	in EXPTIME CONP-hard ($k \geq 1$)
$\mathcal{SE}$	EXPTIME-complete	in PTIME	in PTIME
$\mathcal{SE}^*$	in EXPTIME	in PTIME	in PTIME
$\mathcal{SE}^{//}$	in PTIME	in PTIME	in PTIME

Table 3.1: *Complexity of* $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$.

disjunction). In addition, we consider concatenations of selector expressions, as it is a natural extension to selector expressions. Denote by $\mathcal{CSE}$ the class of all expressions of the form $p_1p_2\cdots p_n$ where each p_i $(1 \leq i \leq n)$ is a selector expression (this allows for arbitrarily many descendant axes). The semantics for this fragment is inherited from the semantics for regular expressions. For a class of patterns $\mathcal{P} \in \{\mathcal{RE}, \mathcal{DSE}, \mathcal{CSE}, \mathcal{SE}, \mathcal{SE}^{//}, \mathcal{SE}^*\}$, we denote by $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ the problem $\forall_{\text{tree}}^{\bullet k}$ where expressions are restricted to the class $\mathcal{P}$.

Theorem 3.13. *The complexity of the problem* $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ *is as stated in Table 3.1.*

Note that these results also provide an upper bound for testing whether the number of nodes selected by a selector expression always lies within a fixed interval $[k, k']$.

Notice that the problem $\forall_{\text{tree}}^{=k,\mathcal{RE}}$ is PSPACE-hard for every value $k \geq 1$, while $\forall_{\text{tree}}^{=k,\mathcal{DSE}}$ is CONP-hard for every value $k \geq 1$. On the other hand, $\forall_{\text{tree}}^{=0,\mathcal{RE}} = \forall_{\text{tree}}^{<1,\mathcal{RE}}$ and $\forall_{\text{tree}}^{=0,\mathcal{DSE}} = \forall_{\text{tree}}^{<1,\mathcal{DSE}}$ and, thus, these two problems can be solved in polynomial time given the results in the second column of Table 3.1. In Section 3.4.2 we show that $\forall_{\text{tree}}^{=1,\mathcal{SE}}$ can be solved in polynomial time, which is a key problem in our study of consistency. The proofs for the other results can be found in Arenas et al. [25].

Consistency Complexity

For a class of patterns $\mathcal{P}$, we denote by CONSISTENCY($\mathcal{P}$) the problem CONSISTENCY restricted to keys using expressions[19] in $\mathcal{P}$. Thereto, we introduce the following definition. Let $k \in \mathbb{N}$, $\bullet \in \{<, =, >\}$, and $\mathcal{R}, \mathcal{S}$ be two pattern languages. We denote by $\forall_{\text{key}}^{\bullet k,\mathcal{R},\mathcal{S}}$ the problem to decide whether for a given

[19]This restriction applies both on the target path and the key paths.

XSD X and a key $\phi = (c, \tau, (p))$ with $\tau \in \mathcal{R}$ and $p \in \mathcal{S}$, it holds that $|p(t, u)| \bullet k$ for every $t \in \mathcal{L}(X)$, every node v in $\text{CNodes}_t(c)$, and for every node u in $\tau(t, v)$.

Let *root* stand for the class containing only the selector expression '.', that is, the expression which selects the root. The following lemma now allows us to transfer the upper and lower bounds from the previous section.

Lemma 3.14. *Let $\mathcal{P} \in \{\mathcal{RE}, \mathcal{DSE}, \mathcal{CSE}, \mathcal{SE}, \mathcal{SE}^{//}, \mathcal{SE}^*\}$, let $k \in \mathbb{N}$, and let $\bullet \in \{<, >, =\}$. Then*

1. *$\forall_{\text{key}}^{\bullet k, \mathcal{RE}, \mathcal{P}}$ is polynomial time reducible to $\forall_{\text{tree}}^{\bullet k, \mathcal{P}}$; and,*

2. *$\forall_{\text{tree}}^{\bullet k, \mathcal{P}}$ is polynomial time reducible to $\forall_{\text{key}}^{\bullet k, root, \mathcal{P}}$.*

Proof. We prove the two claims separately:

1. Let $\phi = (c, \tau, (p))$ be a key and let $X = (A = (Q_X, \Sigma, \delta_X, q_0^X), \lambda)$ be an XSD. Now, define $Q_{\tau,c}$ as the maximal subset of states $q \in Q_X$ for which there is a tree $t \in \mathcal{L}(X)$, a node v in $\text{CNodes}_t(c)$ and a node u in $\tau(t, v)$ such that A when executed on the ancestor-string of u ends in state q. This means that q is a state of the XSD which can be reached when evaluating τ from a node in $\text{CNodes}_t(c)$, for some $t \in \mathcal{L}(X)$. Intuitively, $Q_{\tau,c}$ contains all states in the XSD that can correspond to a target node. Denote by X^q the XSD X with start type q. Then $\bigcup_{q \in Q_{\tau,c}} \mathcal{L}(X^q)$ is the set of trees on which the number of matches of p needs to be tested. Therefore, (X, ϕ) is in $\forall_{\text{key}}^{\bullet k, \mathcal{RE}, \mathcal{P}}$ iff (X^q, p) is in $\forall_{\text{tree}}^{\bullet k, \mathcal{P}}$ for every $q \in Q_{\tau,c}$. It remains to explain how to compute $Q_{\tau,c}$. We can assume that X is trimmed (Lemma 3.5). We also assume that $c = (q_c, \sigma_c)$ is well-formed, which means that there is a state q' such that $\delta(q', \sigma_c) = q_c$. Let $\tau \in \mathcal{RE}$ and let $A_\tau = (Q_\tau, \Sigma, \delta_\tau, q_0^\tau, F_\tau)$ be the DFA accepting $\mathcal{L}(\tau)$. Define $\Gamma_0 = \{(q_c, q_0^\tau)\}$ and $\Gamma_i = \{(\delta_X(q_1, \sigma), \delta_\tau(q_2, \sigma)) \mid (q_1, q_2) \in \Gamma_{i-1}, \sigma \in \Sigma\} \cup \Gamma_{i-1}$. Then, $Q_{\tau,c} = \{q \mid \exists (q, q') \in \Gamma_{|Q_X| \times |Q_\tau|} \text{ and } q' \in F_\tau\}$. Essentially, an incremental product-construction [109] is performed to identify the states in the XSD DFA that are equivalent to the accepting states in the target path DFA.

2. Assume given (X, p). Let $\#_1$ be a new symbol and let $X_\#$ be the XSD defining the set $\{\#_1(t) \mid t \in \mathcal{L}(X)\}$. Then $(X, p) \in \forall_{\text{tree}}^{\bullet k, \mathcal{P}}$ iff $(X_\#, (c_{\text{root}}, ./*, (p)))$ is in $\forall_{\text{key}}^{\bullet k, \mathcal{SE}, \mathcal{P}}$ where c_{root} is the root context. $\square$

The main result of this section immediately follows from Theorem 3.13 and Lemma 3.14:

Theorem 3.15. *For the* Consistency *problem, the following results hold:*

1. Consistency($\mathcal{SE}$) *is in* ptime;

2. CONSISTENCY($\mathcal{DSE}$) *is* CONP*-hard and in* EXPTIME*;*

3. CONSISTENCY($\mathcal{RE}$) *is* PSPACE*-hard and in* EXPTIME*;*

3.4.2 $\forall^{=1,\mathcal{SE}}_{\text{tree}}$ is in PTIME

We will make use of unranked tree automata in what follows. A non-deterministic *unranked tree automaton* (UTA) is a tuple $A = (Q, \Delta, \delta, F)$ where Q is a finite set of states; Δ is a finite alphabet, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Delta \to 2^{(Q^*)}$ is a function mapping each $(q, \sigma) \in Q \times \Delta$ to a regular language over Q. A *run* of $A = (Q, \Delta, \delta, F)$ on a tree t is a function $\lambda : \text{nodes}(t) \to Q$ that assigns a state in Q to each node v of t such that for every $v \in \text{nodes}(t)$ with children $v_1, v_2, \ldots, v_n$ (noted in the order in which they appear in t) it is the case that $\lambda(v_1)\lambda(v_2)\cdots\lambda(v_n) \in \delta(\lambda(v), \text{lab}_t(v))$. A run is *accepting* if the root of t is labeled by a state in F. If an accepting run of A exists for tree t then we say that t is accepted by A. The set of all trees accepted by A is denoted by $\mathcal{L}(A)$. We say that A is *deterministic* iff $\delta(q, a) \cap \delta(q', a) = \emptyset$ for all $q \neq q' \in Q$ and $a \in \Delta$.

We will also make use of binary tree automata which operate in a top-down fashion over trees where every inner node has precisely two children. As this type of automaton has the same expressive power as the UTA's described above and complexity results can be easily transferred when NFA's are used to describe regular languages [138], we will use binary tree automata where convenient. Formally, a *binary (top-down) tree automaton* (BTA) is a tuple $A = (Q, \Delta, q_0, \delta, F)$ where Q is the set of states, Δ is the alphabet, $q_0 \in Q$ is the start state, $\delta : Q \times \Delta \to 2^{Q \times Q}$ is the transition function, and $F \subseteq Q$ is the set of final states. A *run* of A on a tree t is a function $\lambda : \text{nodes}(t) \to Q$ that assigns a state in Q to each node v of t such that the root of t is labeled by q_0 and for every inner node v with children v_1 and v_2 it is the case that $(\lambda(v_1), \lambda(v_2)) \in \delta(\lambda(v), \text{lab}_t(v))$. A run is *accepting* if for every leaf v, $\delta(\lambda(v), \text{lab}_t(v)) \in (F \times F)$. If an accepting run of A exists for tree t then we say that t is accepted by A. The set of all trees accepted by A is denoted by $\mathcal{L}(A)$. We say that A is *deterministic* iff $|\delta(q, a)| \leq 1$ for all $a \in \Delta$ and $q \in Q$. Finally, A is *k-ambiguous*, for a natural number k, if for every $t \in \mathcal{L}(A)$, there are at most k distinct accepting runs of A on t. We call A unambiguous if it is 1-ambiguous.

The following theorem lists well-known results on the complexity of tree automata that will be used in what follows.

Theorem 3.16. *Let A and A' be two tree automata which are either both unranked or both binary.*

1. *Deciding whether $\mathcal{L}(A) = \emptyset$ is in* PTIME*.*

2. *A tree automaton B with $\mathcal{L}(B) = \mathcal{L}(A) \cap \mathcal{L}(A')$ can be constructed in time polynomial in the sizes of A and A'. Furthermore, when A and A' are deterministic binary automata, then so is B.*

3. *Deciding whether $\mathcal{L}(A) = \mathcal{L}(A')$ is* EXPTIME*-complete [150].*

4. *Assume k to be fixed. When A and A' are binary tree automata and k-ambiguous, then deciding whether $\mathcal{L}(A) = \mathcal{L}(A')$ is in* PTIME *[150].*

Note that the tree automaton B computing $\mathcal{L}(A) \cap \mathcal{L}(A')$ is simply the usual product construction between A and A' which we also denote by $A \times A'$.

For an XML tree t, we denote by fcns(t) the usual *first-child next-sibling encoding* of t, defined as follows. To facilitate the definition, we define fcns on ordered forests which consists of concatenations of trees.[20] By ε we denote the empty forest. By $h := t_1 \cdots t_n$, we denote the concatenation of the trees $t_1, \ldots, t_n$. Then, define

- $\text{fcns}(\varepsilon) = \#$; and,
- $\text{fcns}(a(h)h') = a(\text{fcns}(h), \text{fcns}(h'))$, for forests h and h'.

Note that fcns(t) is always a binary tree. We denote $\Sigma \cup \{\#\}$ by $\Sigma_\#$. Intuitively, the fcns encoding of a tree t is a binary tree such that for every node v of t:

1) v is a node in fcns(t);

2) the left child of v in fcns(t) is the first child of v in t (if v is a leaf in t, then a node with label # is placed as the left child of v in fcns(t)); and

3) the right child of v in fcns(t) is the next sibling of v in t (if such a sibling does not exist in t, then a node with label # is placed as the right child of v in fcns(t)).

The following proof sketch gives an overview of the upcoming lemma's and their place in the overall idea of the proof.

Proof Sketch. In order to show that $\forall^{=1,\mathcal{SE}}_{\text{tree}} \in$ PTIME, we first need the following results:

1. given an XSD X, one can construct in polynomial time a deterministic BTA A_X such that for every tree t: $\text{fcns}(t) \in \mathcal{L}(A_X)$ if and only if $t \in \mathcal{L}(X)$ (Lemma 3.17);

2. there is a deterministic BTA $A_\#$ such that $t' \in \mathcal{L}(A_\#)$ if and only if $t' = \text{fcns}(t)$ for some XML tree t (Lemma 3.18);

[20]This simplifies the definition as children of a node form a concatenation of trees.

3. given a selector expression p, one can construct in polynomial time a non-deterministic BTA B_p such that for every XML tree t: $\text{fcns}(t) \in \mathcal{L}(B_p)$ if and only if $|p(t)| > 0$ (Lemma 3.23); and
4. given an XSD X and a selector expression p, testing whether $(X, p) \in \forall_{\text{tree}}^{<k,\mathcal{SE}}$ can be done in PTIME (Theorem 3.21).

With these ingredients, a polynomial time algorithm for $\forall_{\text{tree}}^{=1,\mathcal{SE}}$ works as follows. Let X be an XSD and p a selector expression. In order to test whether $(X, p) \in \forall_{\text{tree}}^{=1,\mathcal{SE}}$, we first verify whether $(X, p) \in \forall_{\text{tree}}^{<2,\mathcal{SE}}$, which can be done in polynomial time. If this is not the case, then we know that $(X, p) \notin \forall_{\text{tree}}^{=1,\mathcal{SE}}$, so the algorithm returns `False`. Otherwise, we still need to determine whether $(X, p) \notin \forall_{\text{tree}}^{=0,\mathcal{SE}}$ or $(X, p) \notin \forall_{\text{tree}}^{=1,\mathcal{SE}}$. The algorithm continues by computing the deterministic BTAs A_X, $A_\#$ and the non-deterministic BTA B_p. Then, to check whether $(X, p) \in \forall_{\text{tree}}^{=1,\mathcal{SE}}$, the algorithm needs to verify whether in all trees that adhere to the given XSD X at least one node is selected by the given selector expression p. This can be verified by checking whether $\mathcal{L}(A_X \times A_\#) \subseteq \mathcal{L}(B_p)$, where $A_X \times A_\#$ is the usual product of BTAs that accepts $\mathcal{L}(A_X) \cap \mathcal{L}(A_\#)$ and can be computed in polynomial time. The key observations to make here are:

1. containment for BTAs is an intractable problem, but it becomes tractable for unambiguous BTAs [150];
2. testing whether $\mathcal{L}(A_X \times A_\#) \subseteq \mathcal{L}(B_p)$ is equivalent to verifying whether $\mathcal{L}(A_X \times A_\#) \subseteq \mathcal{L}(A_X \times B_p)$;
3. $A_X \times A_\#$ is an unambiguous BTA as it is deterministic; and
4. although $A_X \times B_p$ is non-deterministic, by using the fact that $(X, p) \in \forall_{\text{tree}}^{<2,\mathcal{SE}}$, it is possible to prove that $A_X \times B_p$ is an unambiguous BTA.

Therefore, we can test in polynomial time whether $\mathcal{L}(A_X \times A_\#) \subseteq \mathcal{L}(A_X \times B_p)$ and, thus, we can test in polynomial time whether $(X, p) \in \forall_{\text{tree}}^{=1,\mathcal{SE}}$. □

In the remainder of this section, to simplify notation, we assume that `data` always belongs to Σ and therefore write Σ rather than $\Sigma \cup \{$`data`$\}$.

The next lemma states that we can always assume that an XSD is represented as a deterministic tree automaton.

Lemma 3.17. *Let X be an XSD. A deterministic binary tree automaton A_X can be constructed in time polynomial in the size of X such that for every XML tree t:*

$$t \in \mathcal{L}(X) \text{ if and only if } \mathit{fcns}(t) \in \mathcal{L}(A_X).$$

Proof. Let $X = (A, \lambda)$ be an XSD with $A = (S, \Sigma, s_0, \delta_A)$. As every deterministic regular expression can be transformed into an equivalent DFA in polynomial time (see, e.g., Brüggemann-Klein and Wood [48]), we assume that λ is a function that assigns to each $s \in S$ a DFA $A_s = (Q_s, \Sigma, q_s, \delta_s, F_s)$. Then define $A_X = (Q, \Sigma_\#, q_0, \delta, F)$ as follows. The set Q of states is defined as

$$\left(\bigcup_{s\in S}(\{s\} \times Q_s)\right) \cup \{q_0, q_\#\},$$

while the set F of final states is defined as $\{q_\#\}$. Furthermore, the transition function δ is defined as follows.

- For every $a \in \Sigma$: $\delta(q_0, a) = \big((\delta_A(s_0, a), q_{\delta_A(s_0,a)}),\ q_\#\big)$.
- For every $s \in S$, $q \in Q_s$ and $a \in \Sigma$:

$$\delta((s, q), a) \quad = \quad \Big((\delta_A(s, a), q_{\delta_A(s,a)}),\ (s, \delta_s(q, a))\Big).$$

- For every $s \in S$ and $q \in F_s$: $\delta((s, q), \#) = (q_\#, q_\#)$.
- Finally, $\delta(q_\#, \#) = (q_\#, q_\#)$.

With input fcns(t), A_X mimics the way the XSD X works with input t. Furthermore, A_X is deterministic as each A_s is a DFA. □

The next lemma shows that the set of fcns-encodings of all trees is in fact a regular tree language (is recognized by a binary tree automaton).

Lemma 3.18. *There is a deterministic binary tree automaton $A_\#$ such that $t' \in \mathcal{L}(A_\#)$ if and only if $t' = fcns(t)$ for some XML tree t.*

Proof. Let $A_X = (\{q_0, q_1, q_{stop}, q_f\}, \Sigma_\#, q_0, \{q_f, q_{stop}\})$. Then define δ as follows:

$$\begin{aligned} \delta(a, q_0) &= (q_1, q_{stop}) \text{ for every } a \in \Sigma, \\ \delta(a, q_1) &= (q_1, q_1) \text{ for every } a \in \Sigma, \\ \delta(\#, q_1) = \delta(\#, q_{stop}) &= (q_f, q_f). \end{aligned}$$

Basically, the automaton enforces the right child of the root to be a #-labeled node. It also enforces every Σ-labeled node distinct from the root to have two children, and every leaf to be #-labeled. □

It should be noticed that the following lemma is a well-known result about tree automata. Nevertheless, we show its proof as some of the results in the next section use the given construction.

Lemma 3.19. *Let $k \geq 2$ be a fixed constant. There is a polynomial-time algorithm that, given the UTAs $A_1, A_2, \ldots, A_k$, returns an UTA B such that $\mathcal{L}(B) = \bigcap_{i=1}^{k} \mathcal{L}(A_i)$.*

Proof. We begin by showing that there is a polynomial-time algorithm that, given two regular expressions r_1, r_2 over alphabets Σ and Δ, respectively, returns an automaton A_{r_1,r_2} over $\Sigma \times \Delta$ accepting the set of strings

$$(a_1, b_1)(a_2, b_2) \cdots (a_\ell, b_\ell),$$

such that $a_1 a_2 \cdots a_\ell \in \mathcal{L}(r_1)$ and $b_1 b_2 \cdots b_\ell \in \mathcal{L}(r_2)$.

Let $A_{r_1} = (Q, \Sigma, \delta, q_0, F_{r_1})$ and $A_{r_2} = (P, \Delta, \rho, p_0, F_{r_2})$ be two automata accepting the languages $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$ respectively. It is well known that these automata can be constructed in polynomial time. Define A_{r_1,r_2} as $(Q \times P, \Sigma \times \Delta, \mu, (p_0, q_0), F_{r_1} \times F_{r_2})$, where μ is defined as:

$$\mu((q,p),(a,b)) = \delta(q,a) \times \rho(p,b).$$

Then, it is easy to prove that this automaton accepts the set of strings $(a_1, b_1) \cdots (a_\ell, b_\ell)$ such that $a_1 a_2 \cdots a_\ell \in \mathcal{L}(r_1)$ and $b_1 b_2 \cdots b_\ell \in \mathcal{L}(r_2)$.

We use the previous construction in the proof of the lemma. More precisely, given two UTA A_1, A_2, we show how to construct an UTA B such that $\mathcal{L}(B) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. The extension of this construction for a fixed $k > 2$ can be easily obtained by associating automata in pairs. Assume that $A_1 = (Q, \Delta, \delta_1, F_1)$ and $A_2 = (P, \Delta, \delta_2, F_2)$. The new automaton accepting $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ is denoted by $B = (Q \times P, \Delta, \mu, F_1 \times F_2)$, where μ is defined by including the transition

$$\mu((q,p),\sigma) \quad = \quad \mathcal{L}\left(A_{\delta_1(q,\sigma),\delta_2(p,\sigma)}\right)$$

for every $(p,q) \in P \times Q$ and $\sigma \in \Delta$.

It is clear that all the above can be done in polynomial time. Now we prove that a tree t is accepted by B if and only if it is accepted by both A_1 and A_2. Suppose that t is accepted by B. Then there is an accepting run $\rho : \text{nodes}(t) \to Q \times P$ of B on t. By the definition of transition function μ, we know that for every node $v \in \text{nodes}(t)$, it is the case that child-string$_t(v)$ belongs to $\delta_1(\rho_1(v), \text{lab}_t(v))$, where ρ_1 denotes the projection of ρ over its first component. Thus, the projection of ρ to its first component is a run for the automaton accepting $\mathcal{L}(A_1)$. Moreover, it is an accepting run given that the first coordinate of $\rho(\mathit{root})$ belongs to F_1, from which we conclude that $t \in \mathcal{L}(A_1)$. For A_2 the proof is analogous by taking the projection over the second component. The converse is proved analogously. Let t be a tree and ρ_1, ρ_2 be accepting runs of A_1 and A_2 on t, respectively. Then it is clear that $\rho : \text{nodes}(t) \to Q \times P$ defined by $\rho(v) = (\rho_1(v), \rho_2(v))$ is an accepting run of B on t. $\square$

The next lemma shows that we can construct a UTA to check whether a regular expression selects at least k nodes in a given tree.

Lemma 3.20. *Let $k \geq 1$ be a fixed constant. There is a polynomial time algorithm that, given a regular expression r over an alphabet Δ, constructs an UTA $A_{r,k}$ such that for every Δ-tree t, it holds that $t \in \mathcal{L}(A_{r,k})$ if and only if $|r(t)| \geq k$.*

Proof. Let r be a regular expression. We first construct an automaton A_r accepting every tree t for which $|r(t)| > 0$. Then we take a small variation of the tree automata product between k copies of A_r in order to achieve the desired result.

Let $B_r = (Q_r, \Delta, q_0^r, \delta_r, \{q_r\})$ be an NFA accepting $\mathcal{L}(r)$. This automaton can be constructed in polynomial time and, moreover, we can assume that B_r has only one final state q_r for which no outgoing transitions are defined and that q_0^r has no incoming transitions. There is a slight mismatch between NFAs and UTAs: an NFA starts in the initial state and assigns a state to the first symbol of the string depending on its label; an UTA immediately 'assigns' a state to the root without knowing its label. In this sense, the UTA when simulating an NFA is always one step behind. We therefore modify B_r into B_r' to initially process two symbols at the same time: B_r will simply guess the next label and verify later whether it has guessed correctly. As a consequence, B_r' can not accept any string of length one. We deal with this later. Formally, $B_r' = (Q_r', \Delta, q_0^r, \delta_r', \{q_r\})$. Here, $Q_r' = (Q_r \setminus \{q_r\} \times \Delta) \cup \{q_0^r, q_r\}$. Furthermore, for all $a \in \Delta$, $\delta(q_0^r, a) = \{(q, b) \mid q \in \delta_r(q_0^r, ab), b \in \Delta\}$. We abuse notation here and write $\delta_r(q_0^r, ab)$ for the set of states that can be reached by B_r from the initial state when reading the string ab. For all $(q, a) \in Q_r \times \Delta$ and $b \in \Delta$:

$$\delta((q,a),b) = \begin{cases} \delta_r(q,a) \times \Delta & \text{if } a = b \text{ and } q_r \notin \delta_r(q,a); \\ ((\delta_r(q,a) \setminus \{q_r\}) \times \Delta) \cup \{q_r\} & \text{if } a = b \text{ and } q_r \in \delta_r(q,a); \\ \emptyset & \text{if } a \neq b. \end{cases}$$

Notice that B_r' guess the next symbol and only proceeds when that guess was correct. Furthermore, for all strings w of length greater than 1, $w \in \mathcal{L}(B_r)$ iff $w \in \mathcal{L}(B_r')$.

Next, we create an UTA A_r that will (non-deterministically) choose a path matching the regular expression r. Define the UTA A_r as $(Q_{A_r}, \Delta, \delta, \{q_0^r\})$, where $Q_{A_r} = Q_r' \cup \{q_\#\}$ and δ is defined as follows: (we make use of regular expressions to define the regular languages defining the transitions)

- For every pair $(\tau, a) \in (Q_{A_r} \setminus \{q_r, q_\#\}) \times \Delta$, if $\delta_r'(\tau, a) = \{\tau_1, \tau_2, \ldots, \tau_n\}$, then
  - if $q_r \notin \delta_r'(\tau, a)$, define the transition

$$\delta(\tau, a) = q_\#^* \tau_1 q_\#^* \mid q_\#^* \tau_2 q_\#^* \mid \cdots \mid q_\#^* \tau_n q_\#^*;$$

  – if $q_r \in \delta'_r(\tau, a)$, define the transition

$$\delta(\tau, a) = q^*_\# \tau_1 q^*_\# \mid q^*_\# \tau_2 q^*_\# \mid \cdots \mid q^*_\# \tau_n q^*_\# \mid q^*_\#;$$

- For all $a \in \Delta$ define the transitions

$$\delta(q_r, a) = q^*_\# \quad \text{and} \quad \delta(q_\#, a) = q^*_\#.$$

We need to prove some properties of A_r. Assuming that t is a Δ-tree, we show that (1) there is an injective mapping from the set $r(t)$ to the set of accepting runs of A_r on t, and (2) if there is an accepting run of A_r on t assigning state q_r to a node v, then $v \in r(t)$. Recall that we assume r does not select the first position of the string.

(1) If v is a node in t is selected by r, then there exists an accepting run ρ of B'_r on anc-string$_t(v)$ (by definition of A_r). We use this run to define the run ρ_v of A_r on t:

$$\rho_v(u) = \begin{cases} \rho(u) & \text{if } u \in \text{anc-string}_t(v) \\ q_\# & \text{otherwise} \end{cases}$$

By definition of A_r, it is clear that ρ_v is an accepting run of A_r on t. Moreover, given two distinct nodes $x, y \in r(t)$, we have that ρ_x is distinct from ρ_y. This can be proved by noticing that ρ_x assigns state q_r only to x and ρ_y assigns state q_r only to y. We conclude that the function that maps every $v \in r(t)$ to ρ_v is injective.

(2) Assume that $t \in \mathcal{L}(A_r)$, and let ρ be an accepting run of A_r on t assigning q_r to a node v. By definition of δ, if ρ assigns a state in $Q_{A_r} \setminus \{q_r, q_\#\}$ to a node u of t, then u has exactly one child u' such that $\rho(u') \in Q'_r \setminus \{q_r\}$. Moreover, if ρ assigns state $q_\#$ to a node u of t, then every child of u is assigned state $q_\#$ by ρ. This implies that every node in anc-string$_t(v)$ is assigned a state of $Q'_r \setminus \{q_r\}$. Moreover, by definition of δ and given that $\rho(v) = q_r$, the states assigned to the ancestors of v define an accepting run of B'_r over anc-string$_t(v)$. Thus, we conclude that the node v is selected by r.

By using k copies of UTA A_r, we construct a tree automaton accepting every tree t for which $|r(t)| \geq k$. More precisely, denote by A^i_r the i-th copy of A_r, and use superscript i when denoting its state q^i_r. Now define a tree automaton $A'_{r,k} = A^1_r \times A^2_r \times \cdots \times A^k_r$, where $\times$ denotes the product of tree automata defined in Lemma 3.19. Notice this product can be computed in polynomial time as k is fixed.

By the definition of the tree automata product in Lemma 3.19, we have that each state of $A'_{r,k}$ is a k-tuple $\bar{q}$, being the i-th component of $\bar{q}$ a state of A^i_r ($1 \leq i \leq k$). Notice that $\mathcal{L}(A'_{r,k}) = \mathcal{L}(A^1_r) \cap \cdots \cap \mathcal{L}(A^k_r) = \mathcal{L}(A_r)$, which is not the desired result. Then let $A_{r,k}$ be the automaton obtained from $A'_{r,k}$ by removing every state $\bar{q}$ having as components q^i_r and q^j_r, where $i \neq j$. Intuitively, automaton $A_{r,k}$ forces each copy of A_r to select a different node of a tree by using selector expression r. Next we prove that $A_{r,k}$ satisfies the statement of the lemma.

Let t be a tree. If $|r(t)| \geq k$, then by property (1) there exist pairwise distinct accepting runs $\rho_1, \ldots, \rho_k$ of A_r on t. Moreover, we have by definition of A_r that these accepting runs assign state q_r to pairwise distinct nodes of t. Thus, if ρ is a function defined as $\rho(s_1, \ldots, s_k) = (\rho_1(s_1), \ldots, \rho_k(s_k))$ for every tuple $(s_1, \ldots, s_k)$ of nodes from t, then we have that ρ is an accepting run of $A_{r,k}$ on t. Therefore, we conclude that $t \in \mathcal{L}(A_{r,k})$. Conversely, if ρ is an accepting run of $A_{r,k}$ on t, then there exists a sequence $\rho_1, \ldots, \rho_k$ of accepting runs of A_r on t such that these runs assign state q_r to pairwise distinct nodes of t (by definitions of A_r and $A_{r,k}$). Thus, by property (2), we have that $r(t)$ contains at least k distinct nodes. We conclude that $|r(t)| \geq k$.

As we do not consider the empty tree, it remains to deal with expressions r which can select the first position in a string. Let $S \subseteq \Delta$ be the set of symbols σ for which $\sigma \in \mathcal{L}(r)$. Then $A'_{r,k}$ is modified to accept any tree with a root in S when already $k-1$ copies of A_r accept (as opposed to requiring k copies to accept). □

We now study the complexity of the problem $\forall^{<k,\mathcal{P}}_{\text{tree}}$. The results in Table 3.1 concerning this decision problem all follow from the next theorem.

Theorem 3.21. *For every $k \geq 0$, $\forall^{<k,\mathcal{RE}}_{\text{tree}}$ is in* PTIME.

Proof. For $k = 0$ the theorem trivially holds. Thus, assume that $k \geq 1$. Let X be an XSD and r a regular expression. From Lemma 3.20, we can construct in polynomial time an UTA $A_{r,k}$ accepting every tree t for which $|r(t)| \geq k$. Then verifying whether $(X, r) \in \forall^{<k,\mathcal{RE}}_{\text{tree}}$ reduces to testing whether $\mathcal{L}(A_{r,k}) \cap \mathcal{L}(X) = \emptyset$ which by Lemma 3.19 and Theorem 3.16 can be tested in PTIME. □

We conclude this section by showing that $\forall^{=k,\mathcal{SE}}_{\text{tree}}$ is in PTIME which implies membership of $\forall^{=k,\mathcal{SE}^*}_{\text{tree}}$ and $\forall^{=k,\mathcal{SE}^{//}}_{\text{tree}}$ in PTIME as well. In this proof, we need the following two technical lemmas.

Lemma 3.22. *There exists a polynomial time algorithm that, given a selector expression p, computes a binary tree automaton A_p such that for every XML tree t, it holds that $|p(t)|$ is equal to the number of accepting runs of A_p on fcns(t).*

Proof. Let p be a selector expression of the form either $/\sigma_1/\sigma_2/\cdots/\sigma_\ell$ or $//\sigma_1/\sigma_2/\cdots/\sigma_\ell$. Then define the binary tree automaton

$$A_p = (Q, \Sigma \cup \{\#\}, q_0, \delta, F)$$

as follows. The set Q of states is defined as $\{q_0, q_1, \ldots, q_\ell, q_\#\}$, while the set F of final states is defined as $\{q_\#\}$. Furthermore, the transition function δ is defined as follows.

- If $p = /\sigma_1/\sigma_2/\cdots/\sigma_\ell$ and $\sigma_1 = *$, then for every $a \in \Sigma$:

$$\delta(q_0, a) \quad = \quad \{(q_1, q_\#)\}.$$

- If $p = /\sigma_1/\sigma_2/\cdots/\sigma_\ell$ and $\sigma_1 \neq *$, then:

$$\delta(q_0, \sigma_1) \quad = \quad \{(q_1, q_\#)\}.$$

- If $p = //\sigma_1/\sigma_2/\cdots/\sigma_\ell$ and $\sigma_1 = *$, then for every $a \in \Sigma$:

$$\delta(q_0, a) \quad = \quad \{(q_0, q_\#), (q_1, q_\#), (q_\#, q_0)\}.$$

- If $p = //\sigma_1/\sigma_2/\cdots/\sigma_\ell$ and $\sigma_1 \neq *$, then:

$$\begin{array}{rcll} \delta(q_0, \sigma_1) & = & \{(q_0, q_\#), (q_1, q_\#), (q_\#, q_0)\} & \\ \delta(q_0, a) & = & \{(q_0, q_\#), (q_\#, q_0)\} & \text{for every } a \in \Sigma \setminus \{\sigma_1\}. \end{array}$$

- Let $i \in \{2, \ldots, \ell\}$. If $\sigma_i = *$, then for every $a \in \Sigma$:

$$\delta(q_{i-1}, a) \quad = \quad \{(q_i, q_\#), (q_\#, q_{i-1})\}.$$

- Let $i \in \{2, \ldots, \ell\}$. If $\sigma_i \neq *$, then:

$$\begin{array}{rcll} \delta(q_{i-1}, \sigma_i) & = & \{(q_i, q_\#), (q_\#, q_{i-1})\} & \\ \delta(q_{i-1}, a) & = & \{(q_\#, q_{i-1})\} & \text{for every } a \in \Sigma \setminus \{\sigma_i\}. \end{array}$$

- For every $a \in (\Sigma \cup \{\#\})$: $\delta(q_\ell, a) = (q_\#, q_\#)$.

- For every $a \in \Sigma$: $\delta(q_\#, a) = \{(q_\#, q_\#)\}$.

- Finally, $\delta(q_\#, \#) = \{(q_\#, q_\#)\}$.

The resulting automaton has as many different runs as there are nodes that match to p in a given tree. $\square$

Lemma 3.23. *Let $k \geq 1$ be a fixed constant. There is a polynomial time algorithm that, given a selector expression p, computes a binary tree automaton B_p^k such that for every XML tree t:*

1) fcns$(t) \in \mathcal{L}(B_p^k)$ if and only if $|p(t)| \geq k$, and

2) if $t \in \mathcal{L}(B_p^k)$, then the number of accepting runs of B_p^k on t is

$$\frac{|p(t)|!}{(|p(t)| - k)!}.$$

Proof. We want to create an automaton accepting every XML tree for which p selects at least k distinct nodes starting from the root. Intuitively, given the automaton A_p described in the proof of Lemma 3.22, we will construct the product of k copies of this automaton, and then we will remove some states of this product to obtain the desired result. Let $\{A_p^i\}_{i=1}^k$ be k copies of A_p. The states of A_p^i will be denoted as in A_p but with a superscript i (that is, q_0^i, q_1^i, …, q_ℓ^i, $q_\#^i$). Moreover, let $B_p = A_p^1 \times A_p^2 \times \cdots \times A_p^k$, where $\times$ denotes the usual product of binary tree automata. It is important to notice that this product can be constructed in polynomial time as k is assumed to be fixed.

By the definition of B_p, we have that each state of B_p is a k-tuple $\bar{q}$, being the i-th component of $\bar{q}$ a state of A_p^i ($1 \leq i \leq k$). Notice that $\mathcal{L}(B_p) = \mathcal{L}(A_p^1) \cap \cdots \cap \mathcal{L}(A_p^k) = \mathcal{L}(A_p)$, which is not the desired result. Then let B_p^k be the automaton obtained from B_p by removing every state $\bar{q}$ having as components q_ℓ^i and q_ℓ^j, where $i \neq j$. Intuitively, automaton B_p^k forces each copy of A_p to select a different node of a tree by using selector expression p. Next we prove that B_p^k satisfies the statement of the lemma.

1) Let t be an XML tree. If $|p(t)| \geq k$, then by Lemma 3.22 we have that there exist pairwise distinct accepting runs ρ_1, …, ρ_k of A_p on fcns(t). Moreover, we have by definition of A_p that each of these accepting runs assigns state q_ℓ to a distinct node of fcns(t). Thus, if ρ is a function defined as $\rho(s_1, \ldots, s_k) = (\rho_1(s_1), \ldots, \rho_k(s_k))$ for every tuple $(s_1, \ldots, s_k)$ of nodes from fcns(t), then we have that ρ is an accepting run of B_p^k on fcns(t). Therefore, we conclude that fcns$(t) \in \mathcal{L}(B_p^k)$. Conversely, if ρ is an accepting run of B_p^k on fcns(t), then there exists a sequence ρ_1, …, ρ_k of accepting runs of A_p on fcns(t) such that: (a) each of these runs assigns state q_ℓ to a distinct node of fcns(t) (by definitions of A_p and B_p^k), and (2) $\rho(s_1, \ldots, s_k) = (\rho_1(s_1), \ldots, \rho_k(s_k))$ for every tuple $(s_1, \ldots, s_k)$ of nodes from fcns(t) (given that B_p^k is defined as the product $A_p^1 \times \cdots \times A_p^k$). Thus, given that ρ_1, …, ρ_k is a sequence of pairwise distinct accepting runs of A_p on fcns(t), we conclude by Lemma 3.22 that $|p(t)| \geq k$.

2) Assume that t is an XML tree accepted by B^k_p. From the discussion in the previous paragraph, we have that ρ is an accepting run of B^k_p on fcns(t) if and only if there exists a sequence ρ_1, ..., ρ_k of pairwise distinct accepting runs of A_p on fcns(t) such that $\rho(s_1, \ldots, s_k) = (\rho_1(s_1), \ldots, \rho_k(s_k))$ for every tuple $(s_1, \ldots, s_k)$ of nodes from fcns(t). Thus, in order to count the set of accepting runs of B^k_p on fcns(t), we first need to choose accepting run ρ_1, for which we have $|p(t)|$ possibilities by Lemma 3.22. Then we need to choose a different accepting run ρ_2, for which we have $(|p(t)| - 1)$ possibilities. More generally, when selecting ρ_i we have $(|p(t)| - i + 1)$ possibilities, as ρ_i has to be different from all the previously selected accepting runs. We conclude that the number of accepting runs of B^k_p on fcns(t) is equal to:

$$|p(t)| \times (|p(t)| - 1) \times \cdots \times (|p(t)| - k + 1) \quad = \quad \frac{|p(t)|!}{(|p(t)| - k)!}.$$

The binary tree automaton B^k_p constructed above therefore satisfies the desired properties. □

We are now ready to prove our main tractability result.

Theorem 3.24. *For every $k \geq 0$, $\forall^{=k,\mathcal{SE}}_{\text{tree}}$ is in* PTIME.

Proof. If $k = 0$, then the property is a corollary of Theorem 3.21, as $\forall^{<1,\mathcal{RE}}_{\text{tree}}$ and hence $\forall^{<1,\mathcal{SE}}_{\text{tree}}$ as well are in PTIME. Thus, assume that $k \geq 1$. Let X be an XSD and p a selector expression. By Lemma 3.23, we know that it is possible to construct in polynomial time a binary tree automaton B^k_p such that for every XML tree t, it holds that fcns(t) $\in \mathcal{L}(B^k_p)$ if and only if $|p(t)| \geq k$. Moreover, by Lemma 3.17, we know that it is possible to construct a deterministic binary tree automaton A_X such that for every XML tree t, it holds that fcns(t) $\in \mathcal{L}(A_X)$ if and only if $t \in \mathcal{L}(X)$. Let $A_\#$ be the deterministic binary tree automaton such that $t' \in \mathcal{L}(A_\#)$ if and only if $t' =$ fcns(t) for some XML tree t, as shown in Lemma 3.18. Now, to know whether (X, p) is in $\forall^{=k,\mathcal{SE}}_{\text{tree}}$, it suffices to check whether $(X, p) \in \forall^{<k+1,\mathcal{SE}}_{\text{tree}}$ and $\mathcal{L}(A_\# \times A_X) \subseteq \mathcal{L}(B^k_p)$. Notice that the latter condition is equivalent to checking whether $\mathcal{L}(A_\# \times A_X) \subseteq \mathcal{L}(A_X \times B^k_p)$. With this in mind, we can decide in polynomial time whether $(X, p) \in \forall^{=k,\mathcal{SE}}_{\text{tree}}$ by using the following algorithm:

1) Check whether $(X, p) \in \forall^{<k+1,\mathcal{SE}}_{\text{tree}}$. If this condition holds, then go to step (2). Otherwise return `False`. Notice that this step can be executed in polynomial time by Theorem 3.21, since every selector expression is a regular expression.

2) Compute $A_\# \times A_X$ and $A_X \times B^k_p$.

3) Check whether $\mathcal{L}(A_{\#} \times A_X) \subseteq \mathcal{L}(A_X \times B_p^k)$. Given that $(X,p) \in \forall_{\text{tree}}^{<k+1,\mathcal{SE}}$, we have that for every XML tree t in $\mathcal{L}(X)$, it holds that $|p(t)| \leq k$. Moreover, by Lemma 3.23 we have that for every XML tree t accepted by B_p^k, it holds that $|p(t)| \geq k$ and the number of accepting runs of B_p^k on t is:

$$\frac{|p(t)|!}{(|p(t)|-k)!}.$$

Therefore, for every XML tree t that belongs to $\mathcal{L}(A_X \times B_p^k)$, it holds that $|p(t)| = k$ and the number of accepting runs of $A_X \times B_p^k$ on t is bounded by (given that A_X is a deterministic binary tree automaton):

$$\frac{k!}{(k-k)!} \quad = \quad k!$$

We conclude that $A_X \times B_p^k$ is $k!$-ambiguous. Thus, given that $A_{\#} \times A_X$ is a deterministic binary tree automaton, we have that $A_{\#} \times A_X$ is also $k!$-ambiguous and, hence, we can verify whether $\mathcal{L}(A_{\#} \times A_X) \subseteq \mathcal{L}(A_X \times B_p^k)$ by using the polynomial time algorithm mentioned in Theorem 3.16. ☐

3.4.3 Other Quality Measures

We give an overview of a number of additional criteria that can be used to determine the quality of keys. The complete proofs of all complexity results that are presented in the remainder of this section (Theorems 3.25 and 3.26) can be found in Arenas et al. [25]. Since the number of keys mined from a given document can be quite large, we are interested in identifying irrelevant keys that can be disregarded from the output of any key mining algorithm. Examples are keys that hold in any document, that only address a bounded number of target nodes, and keys that are implied by keys that have already been found. Thereto, let X be an XSD, ϕ a key and Ψ be a set of keys such that every key in $\Psi \cup \{\phi\}$ is consistent w.r.t. X. Then,

- Universality is the problem to decide whether $t \models \phi$ for every tree in $t \in \mathcal{L}(X)$;
- Boundedness is the problem to decide whether there is an $N \in \mathbb{N}$, such that for every tree $t \in \mathcal{L}(X) : |\,\text{TNodes}_t(\phi)| \leq N$.
- Key Implication , denoted by $\Psi \sqsubseteq \phi$, is the problem to decide whether for all trees $t \in \mathcal{L}(X)$ such that $\bigwedge_{\psi \in \Psi} t \models \psi$ it holds that $t \models \phi$.
- Satisfiability is the problem to decide whether there is a tree $t \in \mathcal{L}(X)$ with $t \models \phi$;

Intuitively, a bounded key can only select a bounded number of target nodes independent of the size of the input document. Since the main purpose of a key is to ensure uniqueness of nodes within a collection of nodes, bounded keys are not very interesting.

It turns out that identifying universal and bounded keys is algorithmically feasible, while determining implication (and even satisfiability) of keys is intractable [25]. Therefore, determining a smallest set of keys (aka, a cover) is practically infeasible. Note that, while the EXPTIME-completeness of SATISFIABILITY is discouraging, it does not pose a problem for key mining algorithms in practice. Indeed, by Definition 3.12 a key mining algorithm will, on input (X, t) with $t \in \mathcal{L}(X)$ only return keys ϕ with $t \models \phi$ (which can efficiently be checked). As such, the keys ϕ it returns are necessarily satisfiable.

Similar to the previous section, we parametrize the problems above by a class $\mathcal{P}$ of expressions, to restrict attention to input keys that only use expressions in $\mathcal{P}$. We can summarize the complexity results for these problems as follows:

Theorem 3.25 (Key Quality Complexity). *The complexity of the decision problems related to key quality is as follows:*

1. UNIVERSALITY $(\mathcal{DSE})$ *is in* PTIME.
2. BOUNDEDNESS $(\mathcal{DSE})$ *is in* PTIME.
3. KEY IMPLICATION $(\mathcal{SE})$ *is* EXPTIME-*hard.*
4. SATISFIABILITY $(\mathcal{SE})$ *is* EXPTIME-*complete.*

Next, we consider *target path containment* and *equivalence*. Given an XSD X, a context c, and two selector expressions τ and τ', TARGET PATH CONTAINMENT is the problem to decide whether for every tree $t \in \mathcal{L}(X)$ and every node $v \in \text{CNodes}_t(c)$, $\tau(t, v) \subseteq \tau'(t, v)$. We denote the latter condition by $\tau \subseteq_{X,c} \tau'$. By TARGET PATH EQUIVALENCE we denote the corresponding equivalence problem.

Theorem 3.26. TARGET PATH CONTAINMENT *and* TARGET PATH EQUIVALENCE *are in* PTIME.

TARGET PATH EQUIVALENCE is a particularly relevant problem for key mining since it allows identifying, within the discovered set of keys, the semantically equivalent but distinct keys (c, τ, P) and (c, τ', P) with τ target path equivalent to τ'. In this sense, target path equivalence is a sufficient condition for key implication that can be solved efficiently.

Algorithm 5 XML key mining algorithm.

```
for all $c \in \text{ContextMiner}_{t,X}$ do
    for all $\tau \in \text{TargetPathMiner}_{t,X}(c)$ do
        $S = \text{OneKeyPathMiner}_{t,X}(c, \tau)$
        $\mathcal{P} = \text{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$
        for each $P \in \mathcal{P}$ return $(c, \tau, P)$
```

3.5 XML Key Mining Algorithm

In this section, we provide an algorithm for solving the XML key mining problem. Recall from Definition 3.12 that the input to this algorithm is an XSD X, an XML tree t and a minimum support threshold N, and that it should output keys that are consistent with X, are satisfied by t, and whose support exceeds N.[21] In what follows, let $X = (A_X, \lambda_X)$ with the type-automaton $A_X = (\text{Types}, \Sigma \cup \{\texttt{data}\}, \delta, q_0)$.

The overall structure of the XML key mining algorithm is outlined in Algorithm 5. Basically, the algorithm consists of four components:

- $\text{ContextMiner}_{t,X}$ returns a list of possible contexts based on t and X;
- $\text{TargetPathMiner}_{t,X}(c)$ returns a list of target paths that exceed the support threshold in t for a given context c;
- $\text{OneKeyPathMiner}_{t,X}(c, \tau)$ returns a maximal set S of key paths for which $(c, \tau, \{p\})$ is consistent for every $p \in S$; and,
- $\text{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$ returns a set $\mathcal{P}$ of minimal subsets P of S for which $t \models (c, \tau, P)$.

Both $\text{TargetPathMiner}_{t,X}(c)$ and $\text{OneKeyPathMiner}_{t,X}(c, \tau)$ are different instantiations of levelwise search [128]. $\text{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$ leverages on discovery algorithms for functional dependencies in the relational model. In the remainder of this section, we explain each function in detail. We will only consider target and key paths up to a given length k_{max} which can be at most the maximum depth of the document. Since the presence of top-level disjunction renders testing for consistency intractable (see Theorem 3.15), we focus on a key mining algorithm that disregards the union operator, i.e., we consider path expression from the class $\mathcal{SE}$.

To illustrate the different parts of the mining algorithm, we will use the XML document t depicted in Figure 3.1 as a running example.

[21]If no XSD is available, one can be derived, e.g., using algorithms from [42].

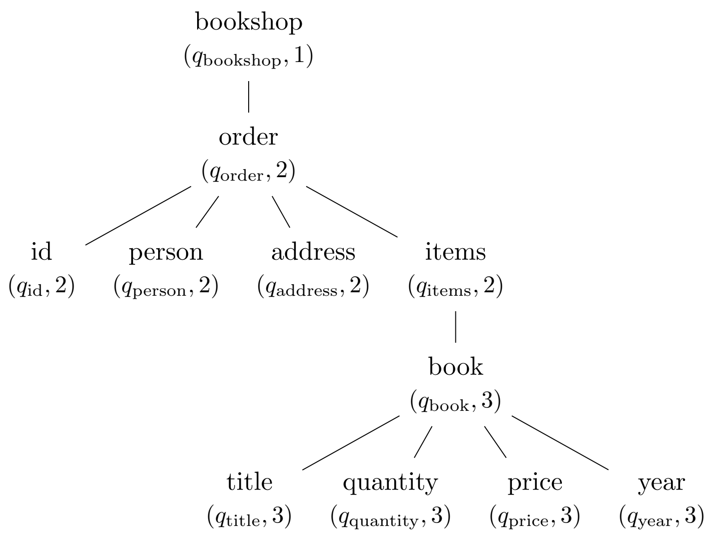


Figure 3.4: *Prefix tree for the XML tree in Figure 3.1. Each node has an associated state and number of matches.*

3.5.1 Prefix Tree and Context Miner

We first define a basic data structure that is used to speed-up various parts of the mining algorithm. Denote by PT(t) the prefix tree obtained from t by collapsing all nodes with the same ancestor string. Recall that the ancestor string of a node is the string obtained by concatenating all labels on the unique path from the root to (and including) that node. Let h be the function mapping each node in t to its corresponding node in PT(t). Then, we label every node m in PT(t) with a pair (q, i) where q is the state assigned to m by the type-automaton A_X and i is the number of nodes in t mapped to m, i.e., $|h^{-1}(m)|$. Note that this is made possible by the fact that the node type only depends on its ancestor string. As only nodes with identical ancestor strings are grouped together in the prefix tree, they all share the same type. Also note that PT(t) does not contain data nodes. The prefix tree can be computed in time linear in the size of t (see, e.g., Grahne and Zhu [91]).

We next discuss the context miner. Clearly, the set of all contexts $c = (\sigma, q)$ with $\sigma \in \Sigma$ and $q \in$ Types, can be directly inferred from the given XSD. But, since only contexts that are effectively materialized in t can give rise to a non-zero support, the context miner enumerates all unique contexts c occurring in PT(t) through a depth-first traversal.

Example 3.27 (Prefix tree). The prefix tree for the XML tree in Figure 3.1 is shown in Figure 3.4. The type automaton depicted in Figure 3.3 is used to assign a unique state to each node in the prefix tree. The set of materialised contexts can now be derived by combining for each node its label with the assigned context. In this case the context miner yields the following set:

$$\{(q_{\text{book}}, \text{book}), (q_{\text{bookshop}}, \text{bookshop}), (q_{\text{order}}, \text{order}), (q_{\text{quantity}}, \text{quantity}), (q_{\text{year}}, \text{year}), (q_{\text{title}}, \text{title}), (q_{\text{person}}, \text{person}), (q_{\text{price}}, \text{price}), (q_{\text{id}}, \text{id}), (q_{\text{address}}, \text{address}), (q_{\text{items}}, \text{items})\}.$$

Note that, in this example, all contexts that appear in the type automaton of the XSD effectively materialize in the XML document. △

3.5.2 Target Path Miner

Next, we describe the target path miner which finds all target paths exceeding the support threshold for a given context c. The algorithm follows the framework of *levelwise search* described by Mannila and Toivonen [128]. In brief, the algorithm is of a generate-and-test style that starts from the most general target path, .//* in our case, and generates increasingly more specific paths while avoiding paths that cannot be interesting given the information obtained in earlier iterations.

The components of any levelwise search algorithm consist of a set U called the *search space*; a predicate q on U called the *search predicate*; and a partial order $\preceq$ on U called the *specialization relation*. The goal is to find all elements of U that satisfy the search predicate. Obviously, in our case, U is the set of selector expressions up to a length k_{max}. A standard approach is to use a support threshold for the search predicate. Accordingly, we define the search predicate as $q(\tau) := \text{supp}(c, \tau, t) > N$, for the given input threshold N. That is, τ is deemed interesting when its support exceeds N.

For levelwise search to work correctly, q should be *monotone* (actually, monotonically decreasing) with respect to $\preceq$, meaning that if $\tau' \preceq \tau$ and $q(\tau)$ holds, then $q(\tau')$ holds as well. The intuition behind $\tau' \preceq \tau$ is that τ is more specific than τ', or in other words, that τ' is more general than τ. For our purposes, it would be ideal to use the semantic containment relation $\tau \subseteq_{X,c} \tau'$ in context c (as defined in Section 3.4). Although this containment relation is shown to be tractable (see Theorem 3.26), through a translation to the inclusion test of unambiguous string automata, it is not well-suited to be used within the framework of levelwise search which requires fast testing of specialization due to the large number of such tests. In strong contrast, as we show below, the containment of selector expressions that disregards the presence of a schema, has a syntactic counterpart which can be implemented efficiently. Therefore, we define $\tau' \preceq \tau$ if and only if for every XML tree t, the set $\tau(t)$ is a subset of $\tau'(t)$. With respect to this definition it is obvious that

Algorithm 6 Basic algorithm for levelwise search [128].

```
$C_0$ := set of most general elements of $U$;
$i := 0$;
**while** $C_i \neq \emptyset$ **do**
    $F_i := \{\tau \in C_i \mid q(\tau)\}$;
    $C_{i+1} := \{\tau \in U \mid \forall \tau' \in U : \tau' \prec \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\} \setminus \bigcup_{j \leq i} C_j$;
    $i := i + 1$;
**return** $\bigcup_{j<i} F_j$;
```

the search predicate q is monotone, as the support of τ will be at least the support of τ'. Notice also that $\tau \subseteq \tau'$ implies $\tau \subseteq_{X,c} \tau'$.

Now, levelwise search computes sets F_i iteratively as shown in Algorithm 6. Here, $\prec$ is the strict version of $\preceq$, so $\tau' \prec \tau$ if $\tau' \preceq \tau$ and $\tau' \neq \tau$. The step computing C_{i+1} is called *candidate generation*; those candidates that satisfy q then end up in the corresponding set F_{i+1} (the letter F is a shorthand for "frequent", referring to the support threshold). It can formally be shown that the union of all sets F_i indeed equals the set of all elements of U satisfying q [128]. Moreover, the algorithm is terminated as soon as C_i is empty, because then all later sets F_j and C_j with $j \geq i$ will be empty as well.

The abstract framework above, however, leaves a number of questions to be answered:

1. How can we efficiently evaluate the search predicate $q(\tau)$?
2. How can we efficiently generate candidate sets C_{i+1}?

We will next answer these questions in detail.

Search Predicate Evaluation

The search predicate supp(c, τ, t) can be entirely evaluated on the prefix tree PT(t) and does not need access to the original document t. A single XPath-expression can be used to aggregate the counts of all nodes matching τ below nodes in context c.[22] Indeed, for $c = (\sigma, q)$, the support can be obtained from PT(t) using the following XPath-expression:

`sum(//`σ`[@state=`id_q`]/`τ`/@matches)`,

where id_q is the internally used id of the state q. The attributes `@state` and `@matches` contain respectively the state id assigned to the node in the prefix tree and the number of nodes with the same ancestor path in t.

[22]Recall that in the prefix tree every node contains its corresponding context and the number of matching nodes from the original XML document.

Specialization Relation and Candidate Generation

Recall that our specialization relation corresponds to the following: $\tau' \preceq \tau$ iff $\tau(t) \subseteq \tau'(t)$ for every XML tree t. Since our chosen specialization relation is purely semantic, we need an equivalent algorithmic definition to show that containment can be effectively decided. Thereto, we define a "one-step specialization relation", whose repeated application corresponds to the semantic specialization relation, as follows: $\tau' \prec_1 \tau$ if τ is obtained from τ' by one of the following operations:

(a) if τ' starts with the descendant axis, replace it by the child axis;

(b) if τ' starts with the descendant axis, insert a wildcard step right after it; or,

(c) replacing a wildcard with an element name.

We establish that $\tau' \preceq \tau$ if and only if τ' can be transformed into τ by a sequence of $\prec_1$-steps, or, more formally:

Proposition 3.28. *The relation $\preceq$ equals the reflexive and transitive closure of the relation $\prec_1$.*

Proof. We denote the reflexive-transitive closure of $\prec_1$ by $\prec_1^*$. It is quite clear that $\prec_1$ is included in $\preceq$ (meaning that $\tau' \prec_1 \tau$ implies $\tau' \preceq \tau$), whence $\prec_1^*$ is included in $\preceq$ as well, since $\preceq$ is transitive. Hence we only have to show that, conversely, $\preceq$ is included in $\prec_1^*$.

So, consider arbitrary sequences of child steps $\pi = \ell_1/\ldots/\ell_p$ and $\pi' = \ell_1'/\ldots/\ell_{p'}$ and selector expressions τ and τ' where τ equals either $./\pi$ or $.//\pi$ and τ' equals either $./\pi'$ or $.//\pi'$. Now assume $\tau' \preceq \tau$; we must show $\tau' \prec_1^* \tau$. We begin by contemplating the possibility that $p' > p$. Consider any XML tree t in the form of a linear chain of p nodes, where the ith node is labeled ℓ_i if ℓ_i is an element name, and is labeled arbitrarily if ℓ_i is the wildcard. Clearly the last node of t belongs to $\tau(t)$; on the other hand, if $p' > p$, the set $\tau'(t)$ is empty, contradicting $\tau' \preceq \tau$. We may thus conclude that $p' \leq p$.

Now consider again a tree t that is a linear chain of p nodes, but now the node labels must be defined more carefully. The ith node of t must be labeled a_i, where a_i is defined as follows:

- If ℓ_i is an element name then a_i equals ℓ_i;
- if ℓ_i is the wildcard we further distinguish the following cases:
  - if i does *not* belong to the p' last positions of $\{1, \ldots, p\}$, i.e., if $i < p - p' + 1$, then a_i can be chosen arbitrarily;

- otherwise, let j be the position number of i in $\{1,\dots,p\}$ when counting backwards from p to 1, i.e., $j = p - i + 1$. Note that in $\{1,\dots,p'\}$, the corresponding element at position j from the back is $k = p' - j + 1 = i - (p - p')$. We distinguish two final cases:
  - if ℓ'_k is an element name, say a, then a_i equals some element name *different* from a;
  - otherwise, a_i can again be chosen arbitrarily.

To understand the definition above it helps to think of π and π' as being aligned at their ends. The following is an example for $p = 6$ and $p' = 4$:

$$\begin{array}{ccccccccccccc} \pi & = & a & / & * & / & b & / & c & / & * & / & * \\ \pi' & = & & & & & * & / & c & / & b & / & * \\ t & = & a & \to & c & \to & b & \to & c & \to & a & \to & c \end{array}$$

Now we note again that the last node of t belongs to $\tau(t)$, so it must also belong to $\tau'(t)$ by $\tau' \preceq \tau$. This implies that for each $i \geq p - p' + 1$, letting $k = i - (p - p')$ as above, if ℓ_i is a wildcard then also ℓ'_k must be a wildcard. Indeed, if not, we would have set a_i to a different element name and τ' would fail to select the last node of t. Furthermore, if ℓ'_k is an element name, then ℓ_i must be the same element name. Indeed, a_i equals ℓ_i, so ℓ'_k different from a_i would again fail τ' to select the last node of t. We conclude that, when aligning π and π' at their ends, and looking at the p' last steps of π, each step of π is either equal to the corresponding step of π', or the corresponding step of π' is a wildcard. As to the $p - p'$ first steps of π, they are unrelated to π'. We conclude that π can be obtained from π' by first, if necessary, filling up π' on the left so as to become of the same length as π, and then replacing some wildcards by element names.

This is almost what we need to conclude that $\tau' \prec_1^* \tau$ except that we still have to rule out the possibility that τ' might start with a child axis whereas τ starts with a descendant axis. This is indeed impossible: consider now a tree t that is a linear chain of $p + 1$ nodes. The first node is arbitrarily labeled, while for $i = 1,\dots,p$, the $i + 1$th node is labeled with ℓ_i if ℓ_i is an element name and is labeled arbitrarily otherwise. Then the last node of t belongs to $\tau(t)$ but cannot belong to $\tau'(t)$ since τ' is not long enough and starts with a child axis. This is again in contradiction with $\tau' \preceq \tau$. □

Note that the definition of $\prec_1$ makes it impossible that $\tau' \prec_1 \tau'' \prec_1 \tau$ while at the same time $\tau' \prec_1 \tau$. Hence, Proposition 3.28 implies that $\prec_1$ as defined above really is the "successor" relation of $\preceq$. More formally, $\tau' \prec_1 \tau$ holds precisely if and only $\tau' \prec \tau$ and there exists no intermediate τ'' such that $\tau' \prec \tau'' \prec \tau$. Moreover, $\prec_1$ is very efficient to compute. Thus armed, we can perform candidate generation in a effective manner as given in Algorithm 7.

Algorithm 7 TargetPathMiner$_{t,X}(c)$.

```
$C_0 :=$ set of minimal elements of $U$;
$i := 0$;
while $C_i \neq \emptyset$ do
    $F_i := \{\tau \in C_i \mid q(\tau)\}$;
    $G_{i+1} := \{\tau \in U \mid \exists \tau' \in F_i : \tau' \prec_1 \tau\}$;
    $C_{i+1} := \{\tau \in G_{i+1} \mid \forall \tau' : \tau' \prec_1 \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\}$;
    $i := i + 1$;
return $\bigcup_{j<i} F_j$;
```

Here, candidate generation is split up in two steps, which in practice can be interleaved. The set G_{i+1} takes all successors of the current set F_i; the set C_{i+1} then prunes away those elements that have a predecessor that does not satisfy q. We will formally show below that the sets F_i computed in this concrete manner are exactly the same as those prescribed by the levelwise algorithm. Note that each candidate is generated in only one level, which explains the omission of the extra candidate deletion step.

Theorem 3.29. *Algorithms 6 and 7 are equivalent.*

We begin by noting the following property which is folk knowledge but for which no rigorous proof seems to be available in the literature. Define the *depth* of an element $x \in U$ as the maximal number n such that there is a chain of n strict inequalities $x_0 \prec x_1 \prec \cdots \prec x_n = x$ where x_0 is minimal. The depth of a minimal element is defined to be zero. We have:

Proposition 3.30. *For any natural number i, consider the set Q_i of all elements of U that satisfy q and that have depth i. Then for each i, we have $F_i = Q_i$ where F_i is as specified by Algorithm 6.*

Proof. By induction on i. For $i = 0$, the claim holds by definition of F_0. For $i > 0$, let $i = i' + 1$. We must show that $F_i = \{x \in C_{i'+1} \mid q(x)\} = Q_i$.

For the inclusion from left to right, let $x \in C_{i'+1}$ such that $q(x)$. We must show that x has depth $i' + 1$. By definition of $C_{i'+1}$, every $y \prec x$ is in F_j for some $j \leq i'$ By induction, elements of F_j are in Q_j and thus have depth $j \leq i'$. Hence, the depth of x is at most $i' + 1$. The depth of x cannot be strictly less. Indeed, suppose the depth of x would be $j \leq i'$; then x would belong to $Q_j = F_j \subseteq C_j$. But $x \in C_{i'+1}$, so, by definition, $x \notin \bigcup_{j \leq i'} C_j$, which is a contradiction.

For the inclusion from right to left, let x be of depth $i' + 1$ such that $q(x)$. We must show that $x \in C_{i'+1}$, so we must show two things:

(a) Every $y \prec x$ is in F_j for some $j \leq i'$.

In proof, assume $y \prec x$. Since x has depth $i' + 1$, the depth j of y is at most i'. Since $q(x)$ and $y \prec x$, also $q(y)$ holds because q is monotone w.r.t. $\prec$. Hence, $y \in Q_j = F_j$.

(b) $x \notin \bigcup_{j \leq i'} C_j$.

In proof, suppose for the sake of contradiction that $x \in C_j$ for some $j \leq i'$. Then, since $q(x)$ holds, $x \in F_j = Q_j$. Hence, the depth of x would be $j \leq i'$, contradicting that the depth of x equals $i' + 1$. □

We can now show what is claimed by Theorem 3.29. Let us denote the sets C_i and F_i as computed by Algorithm 7 by C'_i and F'_i, to distinguish them from the computation of Algorithm 6. Then, the claim is the following:

Proposition 3.31. *For each i, the sets F'_i and F_i are equal.*

Proof. We show by induction that for every i, the sets F_i and F'_i are equal. For $i = 0$, the equality is trivial. For $i > 0$, let $i = i' + 1$. We must show $F_{i'+1} = F'_{i'+1}$.

For the inclusion from left to right, let $x \in F_{i'+1}$. Since $q(x)$, showing $x \in F'_{i'+1}$ amounts to showing $x \in C'_{i'+1}$, which in turn amounts to two things:

(a) $x \in G_{i'+1}$. To show this, we must show that there exists $y \in F'_{i'}$ such that $y \prec_1 x$. Since $x \in F_{i'+1}$, the depth of x equals $i' + 1$, so there exists a sequence $x_0, x_1, \ldots, x_{i'}, x_{i'+1}$ with x_0 minimal and $x_{i'+1} = x$ and $x_j \prec x_{j+1}$ for $j \in \{0, \ldots, i'\}$. Take $y = x_{i'}$. Since $y \prec x$ and $\prec_1^* = \preceq$, there exists a path in $\prec_1$ from y to x. This path cannot be of length more than one, as this would imply that the depth of x is strictly more than $i' + 1$. Hence, we have $y \prec_1 x$. The depth of y is clearly at least i', and also at most i' because the depth of x equals $i' + 1$. Finally, since $q(x)$, also $q(y)$. So, y has depth i' and satisfies q from which we conclude that $y \in F_{i'}$ as desired.

(b) Every $y \prec_1 x$ belongs to F'_j for some $j \leq i'$. This is shown using similar arguments as above. Let j be the depth of y. Since $y \prec_1 x$, also $y \prec x$, so $j \leq i'$ since the depth of x is $i' + 1$. Moreover, since $q(x)$, also $q(y)$. Hence $y \in F_j$ with $j \leq i'$ as desired.

For the inclusion from right to left, let $x \in F'_{i'+1}$. Since $q(x)$, showing $x \in F_{i'+1}$ amounts to showing $x \in C_{i'+1}$, which in turn amounts to two things:

(a) Every $y \prec x$ is in F_j for some $j \leq i'$. This is again shown using the same arguments as above.

(b) $x \notin \bigcup_{j \leq i'} C_j$. Indeed, if x would be in C_j for some $j \leq i'$, then $x \in F_j$, since $q(x)$ holds. But then the depth of x would be j, a contradiction.

Since both inclusions hold, we conclude that the sets F'_i and F_i are equal for each i. □

To conclude we note that the proof above is valid not just for $\prec_1$ but for any relation $\prec'$ whose reflexive and transitive closure equals $\prec$, which we have indeed shown to hold for $\prec_1$ in Proposition 3.28.

Example 3.32 (Specialization relation). We now illustrate the first iteration of Algorithm 7. Consider the context $c = (\text{order}, q_{\text{order}})$ from our running example, together with a support threshold of 3.

The target miner starts with the minimal elements as a set of candidates. In this case this corresponds to the most general path: $C_0 = \{\texttt{.//*}\}$. The predicate q is then checked for `.//*`. As this path selects all 23 non-data nodes below the `order` nodes, it is supported and the predicate evaluates to `True`. Hence, $F_0 = \{\texttt{.//*}\}$. Next, the specialization relation is used to generate the next level. Applying the 3 possible specialization operations on `.//*` yields the following set:

$$G_1 = \{\texttt{./*}, \texttt{.//*/*}, \texttt{.//items}, \texttt{.//book}, \ldots\}.$$

For each of these paths it is checked whether all the parent paths are supported (only `.//*` in this case). In the next iteration, the predicate will be evaluated for these new candidates, and a new level will be generated (if possible). Note that the sets C_i in Algorithm 6 would contain the same items. △

Duplicate Expression Elimination (Canonization)

Often, a nuisance in mining logical formulas such as selector expressions is duplicate elimination: different expressions may be logically equivalent. Fortunately, in our setting, it follows from Proposition 3.28 that only identical selector expressions can be equivalent. Indeed, two expressions can only be equivalent when they can be transformed in to one another through a series of allowed transformations. As a transformation yields a more specific expression that is not equivalent, equivalent expressions must be identical.

Regardless, it can happen that two derived, and therefore, non-equivalent, target paths τ and τ' select precisely the same set of target nodes on the given document t. As these paths are equivalent from the perspective of t, it holds that $t \models (c, \tau, P)$ iff $t \models (c, \tau', P)$ for all sets P. Therefore, w.r.t. the generation of key paths P, it does not make sense to consider all of these equivalent paths separately. Rather, we should choose among them one canonical path. One possibility, e.g., is to opt for the most specific path according to $\prec_1$ by preferring non-descendent paths, and by minimizing the length and number of wildcards. Notice that equivalence of target paths on t can be tested on the prefix tree $\mathrm{PT}(t)$ without access to the original document. This is accomplished by evaluating the expression on the prefix tree and comparing the sets of selected nodes. When these are equivalent, the target paths will select the same nodes in the underlying XML document as well.

Example 3.33 (Path equivalence). Consider the context q_{order} and the target paths `.//book` and `./items/book`. When evaluating these target paths on the small prefix tree in Figure 3.4, we notice that the same nodes are selected. We can therefore conclude that they are equivalent on the document. Indeed, both target paths will select all the book nodes in the larger XML document depicted in Figure 3.1. △

Boundedness Elimination

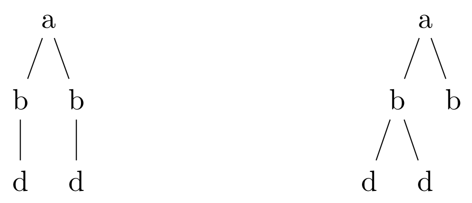
The quality of the mining result can be improved using the results of Section 3.4. Indeed, target paths that are bounded but that have still passed the support threshold N, which may happen with low values of N, may be eliminated at this stage.

Example 3.34 (Boundedness). Suppose the XSD in our running example would limit the number of `order` nodes in a document to a maximum of 10. We could then eliminate the target paths that select (descendants of) these nodes. △

3.5.3 One-Key Path Miner

The job of the one-Key path miner is to find all key paths p for which $(c, \tau, (p))$ is consistent on the given document: that is, for every $v \in \mathrm{CNodes}_t(c)$ and every $u \in \tau(t, v)$, it holds that $p(t, u)$ is a singleton containing a **Data**-node. In a second step, the key paths p for which $(c, \tau, (p))$ is consistent w.r.t. X, are selected for further processing. The reason for this two-step approach is to reduce the number of costly consistency tests. Although testing for consistency w.r.t. a schema can be done in polynomial time (see Theorem 3.15), it can be slow for large schemas and is ill-suited to be used directly as a search predicate. This will be made apparent in the experiments described in Section 3.6. Therefore, we test for document consistency (qualification) in a first step and make use of the fact that inconsistency on t implies inconsistency on X. That is, key paths which are not consistent on t (do not *qualify* on t) and which are therefore pruned in the first step, can never be consistent w.r.t. X.

The main idea here is that we look for paths that select at most one leaf per target node and that we start using very specific paths that are generalized



(a) *Document t_1.* **(b)** *Document t_2.*

Figure 3.5: *Two documents that yield the same prefix tree.*

in each step to select more nodes. When a path selects too many nodes, it is discarded. It turns out that again a levelwise search may be used, utilizing the converse of the specialization relation $\preceq$ for target-path mining. So, define $p' \preceq^{\text{key}} p$ iff $p \preceq p'$. That is, $p' \preceq^{\text{key}} p$ iff $p' \subseteq p$. The search predicate $q_\tau^{\text{key}}(p)$ is now defined to hold if p selects *at most* one node in t for each of the target nodes selected by τ in context c. This q_τ^{key} is indeed monotonically decreasing w.r.t. the converse of containment among selector expressions: $p' \preceq^{\text{key}} p \equiv p' \subseteq p$ and $q_\tau^{\text{key}}(p)$ together imply $q_\tau^{\text{key}}(p')$. We note that consistency requires the selection of *exactly one* node, rather than *at most one*. However, this mismatch can be solved by confining the search space U_{key} to all selector expressions up to length k_{max} that from a target node select a leaf node in the prefix tree: these expressions select at least one node by virtue of their being present in the prefix tree. The "most general" elements from which the levelwise search is started are in this case the *specific* paths in the prefix tree from target nodes to leafs. Obviously, U_{key} can be computed directly from $\text{PT}(t)$.

It remains to discuss how to compute q_τ^{key} efficiently. Unfortunately, q_τ^{key} can not always be computed solely on $\text{PT}(t)$. Indeed, consider the documents in Figure 3.5: $t_1 = a(b(d), b(d))$ and $t_2 = a(b(d, d), b)$, where each d-node is a **Data**-node. Then, $\text{PT}(t_1) = \text{PT}(t_2)$, yet ϕ is consistent on t_1 but inconsistent on t_2 for $\phi = (c_{\text{root}}, \texttt{./b}, (\texttt{./d}))$ with c_{root} the root context.

We next present a sufficient condition for inconsistency which can be tested on the prefix tree. Thereto, consider $\phi = (c, \tau, (p))$ and let $t' = \text{PT}(t)$. For a node m in t', we denote by $\#_{t'}(m)$ the number assigned to m in t', that is, $|h^{-1}(m)|$ for h as defined in Section 3.5.1. Define the following conditions:

C1 There exists a $v \in \text{CNodes}_{t'}(c)$ and a $u \in \tau(t', v)$ such that $\#_{t'}(u) < \sum_{w \in p(t',u)} \#_{t'}(w)$.

C2 There exists a $v \in \text{CNodes}_{t'}(c)$, a $u \in \tau(t', v)$, a $w \in p(t', u)$, and a node m on the path from u to w such that $\#_{t'}(m) < \#_{t'}(w)$.

Here, C1 says that the number of target nodes u is strictly smaller than the number of nodes selected by p, and C2 says that there is a leaf node selected by p and an ancestor with a smaller number of corresponding nodes in t. Both conditions imply that there are at least two nodes selected by p which belong to the same target node in t and which contradict consistency. The following proposition hence follows:

Proposition 3.35. *Given $\phi = (c, \tau, (p))$ and a document t. If condition C1 or C2 holds on* $\mathrm{PT}(t)$*, then ϕ is inconsistent on t.*

So, only when the tests for the two conditions above fail, we evaluate p on t to determine the value of $q_\tau^{\mathrm{key}}(p)$.

Finally, define $\prec_1^{\mathrm{key}}$ as the inverse of $\prec_1$, that is, $p' \prec_1^{\mathrm{key}} p$ iff $p \prec_1 p'$. Then, the first step of $\mathrm{OneKeyPathMiner}_{t,X}(c, \tau)$ is the same algorithm as depicted in Algorithm 7 with U, q, and $\prec_1$, replaced by U_{key}, q_τ^{key}, and $\prec_1^{\mathrm{key}}$, respectively. The second step in $\mathrm{OneKeyPathMiner}_{t,X}(c, \tau)$ retains from all the returned key paths p, those for which $(c, \tau, (p))$ is consistent w.r.t. X employing the algorithm of Theorem 3.15. A duplicate elimination step similar to the one of the previous section is performed as well, resulting in the removal of equivalent path expressions.

Example 3.36 (Key path miner). For the context $(\mathrm{order}, q_{\mathrm{order}})$ and the target path `.//book` the one-key path miner will generate candidate paths starting from the paths to leaf nodes in the prefix tree:

{`./quantity`, `./title`, `./year`, `./price`}.

None of these are found to be inconsistent by either C1 or C2, nor by the XML document itself (see Figure 3.1). This is because all of them appear exactly once. But, after the XSD consistency check, the path `./year` is removed. Indeed, when we inspect the XSD more closely (see Example 3.3), we see that year is optional. This means that there are XML documents that satisfy the XSD, but for which the key is inconsistent. In the next iterations, the algorithm will generate more general paths by applying the converse specialization relation, as described above. In this case, paths such as `./*` will violate the consistency requirement, while paths of the form `.//quantity` are equivalent to their non-descendant counterparts. Both types are removed. The final output of this phase is therefore:

{`./quantity`, `./title`, `./price`}. △

3.5.4 Minimal Key Path Set Miner

At this point, we have computed the maximal set S for which holds that for every $p \in S$, $(c, \tau, (p))$ is consistent w.r.t. X. Next, we are looking for minimal and meaningful sets $P \subseteq S$ such that $t \models (c, \tau, P)$, that is, such that (c, τ, P) is a key for t. Note that such a set P can be trivially converted to a sequence to satisfy the definition of an XML key, as defined in Section 3.3.3.

We capitalize on existing relational techniques for mining functional dependencies (see, e.g., [44, 125, 126]). To this end, we define a relation $R_{S,t}$ with the following schema

$$(\mathrm{CID}, \mathrm{TID}, p_1, p_2, \ldots, p_{|S|}),$$

where CID and TID are columns for the selected context nodes and target nodes, respectively, and every p_i corresponds to the unique **Data**-value selected by the corresponding key path p_i. Then, $(v, u, \bar{o}) \in R_{S,t}$ if and only if $v \in \mathrm{CNodes}_t(c)$, $u \in \tau(t, v)$ and $\mathrm{record}_S(t, u) = \bar{o}$. Now, it follows that $t \models (c, \tau, P)$ iff $\mathrm{CID}, p_1, p_2, \ldots, p_n \rightarrow \mathrm{TID}$ is a functional dependency in $R_{S,t}$ for $P = (p_1, \ldots, p_n)$. This means that when the values for the attributes $\mathrm{CID}, p_1, p_2, \ldots, p_n$ are identical for two tuples, the value for TID must be identical as well. We can now plug in any existing functional dependency discovery algorithm.

Example 3.37 (Key path set miner). Suppose we obtain the following consistent candidate key from the previous phases:

$$((\mathrm{order}, q_{\mathrm{order}}), \texttt{.//book}, \{\texttt{./title}, \texttt{./price}, \texttt{./quantity}\}),$$

yielding the relation in Table 3.2. We observe the following:

$$\begin{aligned} \mathrm{CID}, \mathrm{title} &\rightarrow \mathrm{TID}, \\ \mathrm{CID}, \mathrm{price} &\rightarrow \mathrm{TID}, \\ \mathrm{CID}, \mathrm{quantity} &\not\rightarrow \mathrm{TID}, \end{aligned}$$

and can hence derive the following final XML keys for the considered context and target path:

$$\begin{aligned} &((\mathrm{order}, q_{\mathrm{order}}), \texttt{.//book}, (\texttt{./title})), \\ &((\mathrm{order}, q_{\mathrm{order}}), \texttt{.//book}, (\texttt{./price})). \end{aligned}$$ △

3.6 Experiments

In this section, we analyse the performance of different parts of the mining algorithm. We also look at different optimizations to understand their impact on the execution time and number of derived keys.

For our experiments, we use a corpus of 90 high quality XML documents and associated XSDs obtained from Grijzenhout and Marx [93]. The input

CID	TID	title	price	quantity
o_1	b_1	Movie analysis	5.63	63
o_1	b_2	Programming intro	6.72	63
o_2	b_3	Programming intro	5.63	150

Table 3.2: *Relational table for the key used in Example 3.37 and the XML document in Figure 3.1.*

can therefore be seen as 90 pairs, each consisting of a unique XML-document and a unique XSD. The maximal and average number of elements occurring in documents is 91K and 5K, respectively, while the maximal and average number of elements occurring in XSDs is 532 and 52, respectively. All experiments are w.r.t. to this corpus; the time-related experiments were run on a 3GHz Mac Pro with 2GB of RAM, while others made use of the HPC-infrastructure of the VSC[23]. In all experiments, we set k_{max} to 4 for target paths and to 2 for key paths, unless explicitly mentioned otherwise.

Choosing Constants

We need to determine meaningful values for k_{max} and the support threshold. In this section we derive suitable bounds. To determine k_{max}, we examine the distribution of the target path lengths. To this end, we generated for all documents the set of target paths up to length 10, having a minimal support of 1, i.e., each target path must select at least one node in the XML document. Already 88.03% of the (canonical) target paths have a length up to 4, and 96.39% have a length up to 6. But, more importantly, when looking at the removal rates from canonization (removal of equivalent target paths) in Table 3.3, we see that larger paths have a much higher removal rate.[24] For example, for paths of length 6, 98.35% of the paths are covered by the canonical paths of length 6 or smaller. This means that, especially for larger path lengths, a significant portion of the candidate target paths are superfluous. The latter is exemplified in Figure 3.6, which shows the distribution of the target path length before and after canonization. This huge amount of unnecessarily generated paths motivates us to pick 2 and 4 as values for k_{max}. Finally, we think that, together with all possible contexts, paths of length up to 4 will provide enough freedom for selecting nodes in the XML document (recall that the use of wildcards and descendant axes is allowed).

For *key paths*, we make similar observations. The percentage of key paths that are pruned away using path equivalence is depicted in Table 3.4. These

[23]Flemish Supercomputer Center.

[24]Note that the total number of paths differs from the one presented in [25] as the article contained a small calculation error.

length	target paths	canonical	rem. rate
1	10974	3799	65.38%
2	17500	3344	80.89%
3	21929	2318	89.43%
4	27212	1584	94.18%
5	27379	784	97.14%
6	29771	491	98.35%
7	28942	280	99.03%
8	29178	170	99.42%
9	24349	73	99.70%
10	21492	28	99.87%
Total	238726	12871	94.61%

Table 3.3: *The fraction of target paths that is removed after a path equivalence test (without a schema).*

length	key paths	canonical	rem. rate
1	1505	681	54.75%
2	638	130	79.62%
3	421	38	90.97%
4	436	20	95.41%
Total	3000	869	71.03%

Table 3.4: *The fraction of key paths that is removed after a path equivalence test (without a schema).*

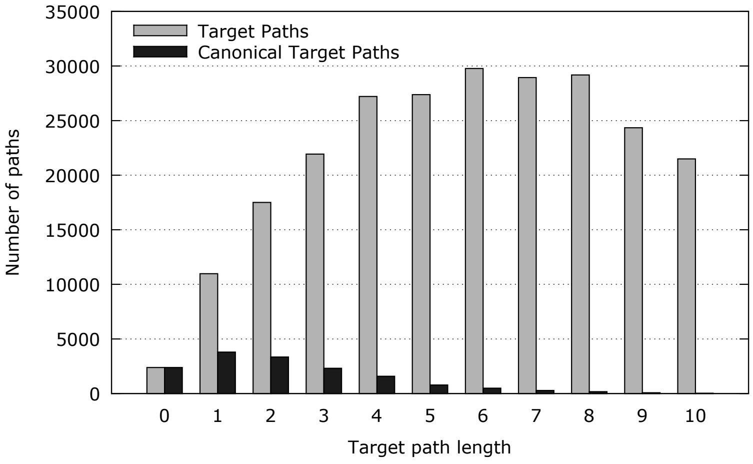


Figure 3.6: *Length distribution of the target paths and their canonical versions for a support threshold of 1 and a maximal length of 10.*

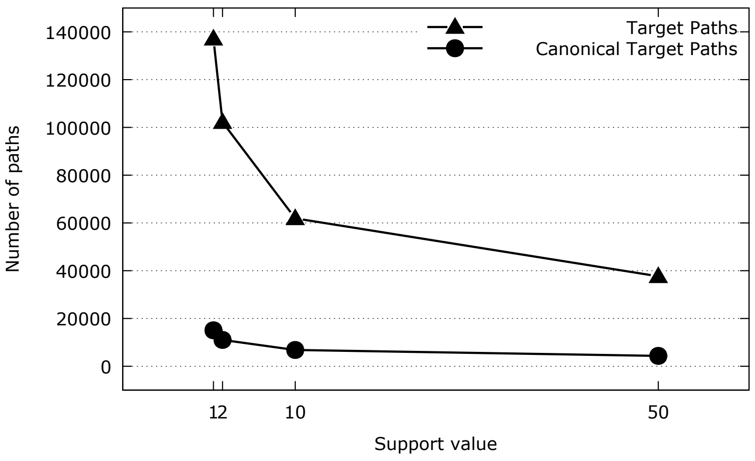


Figure 3.7: *Total amount of target paths found for support thresholds 1,2,10 and 50 and a maximal length of 6.*

observations motivate us to restrict to maximal key path lengths of 2 and 4.

Next, we derive a suitable value for the *support threshold*. Recall that the support of a key equals the number of target nodes that are selected by the target path. In Figure 3.7, we see the effects of an increasing support threshold on the number of target paths and on the number of canonical target paths. For larger values, the number of target paths passing the support threshold stabilizes quickly. This indicates that a large number of paths only selects a few target nodes and that even small support thresholds will prune away significant parts of the XML key search space. To decide on a good support value, we should strike a balance between removing paths with low support, while still keeping paths that select a significant portion of small documents. Indeed, small documents can yield low support values for a large portion of paths. For this reason, we mostly use support values of 2 and 10.

Finally, based on the observations made, we may already conclude that path equivalence *without* a schema, in conjunction with our canonization algorithm is an effective way of limiting an explosion of paths.

3.6.1 Prefix Tree

As different parts of the algorithm can avoid access to the input document t by operating directly on $\mathrm{PT}(t)$, it is instrumental to investigate the compression rate of $\mathrm{PT}(t)$ over t. Figure 3.8 plots the number of nodes in documents versus the number of nodes in the corresponding prefix trees. Note that the scale is logarithmic. In essence, every document is compressed to a prefix tree with at most 200 nodes, even the large documents which contain up to 91K nodes.

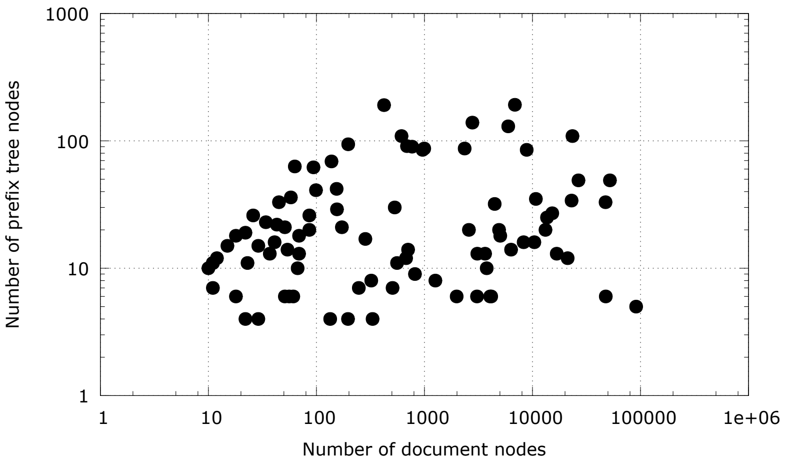


Figure 3.8: *Number of document nodes versus number of nodes in prefix trees. The prefix trees are considerably smaller than their full-sized counterparts.*

3.6.2 Contexts

A key $\phi = (c, \tau, P)$ consists of three interdependent components: target paths need only to be considered w.r.t. a context, and key paths need only to be considered w.r.t. a context and a target path. To avoid an explosion of the size of the search space it is paramount to reduce the number of considered contexts, target paths and key paths. We next assess the effectiveness of the algorithm in this respect.

We start with the number of contexts considered by the algorithm. An analysis comparing the number of contexts allowed by XSDs with the number of contexts actually used in the XML documents, shows that for 40% of the documents all allowable contexts materialize in the corresponding XML documents, i.e., there is no improvement as no allowable contexts can be omitted. More specifically, this means that considering only materialized contexts would not prune the search space for these documents. Nevertheless, it appears that this mostly happens for smaller XSDs. Indeed, the total sum of allowable contexts over all 90 documents is 4639 while the total sum of contexts found in actual documents is 2217 which indicates that over the complete data set 52% of all possible contexts do *not* have to be considered. Keeping in mind that every context that can be removed in this step eliminates a call to the target path *and* key path miner underlines the effectiveness of context search driven by the XML data at hand. In practice, the set of used contexts can be build during prefix tree construction (see Section 3.5.1), as type information is derived for each node anyway. This avoids additional inspection of the XSD.

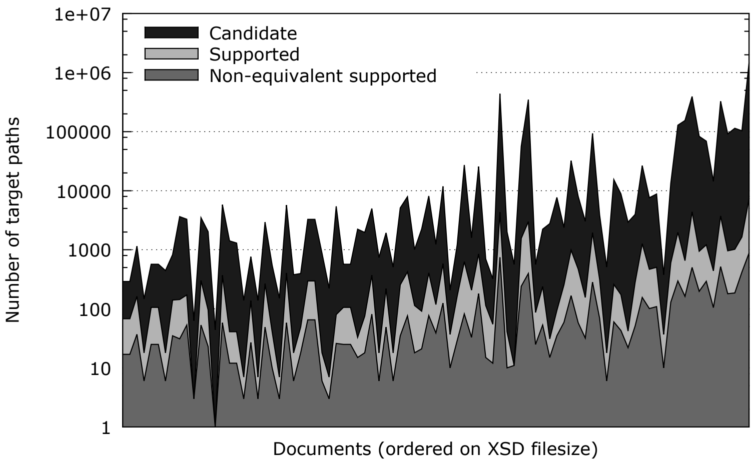


Figure 3.9: *Behavior of the target path miner. Number of candidate, supported and non-equivalent target paths per document for $k_{max} = 4$ and support threshold of 10.*

3.6.3 Target Paths

Next, we discuss the behavior of the target path miner when the support threshold N equals 10. The results are illustrated in Figure 3.9 (cases with $k_{max} = 5$ and/or lower support threshold were also tested but yield similar results and are therefore not shown). For presentation purposes, the x-axis enumerates all document-XSD pairs increasingly ordered by the size of the XSD. The figure then shows per pair, the number of candidate, supported, and non-equivalent derived target paths. Its purpose is to provide a visual inspection on the considered quantities on a per document basis. By candidate target paths we mean those that occurred in a candidate set C_i during the execution of Algorithm 7. Non-equivalent target paths are those that remain after duplicate elimination (as explained in Section 3.5.2). The number of possible target paths to consider (that is, the cardinality of the search space U times the number of allowable contexts) is not shown as the target path miner only considers a small fraction of those, to be precise, only 3% on average. Hence, when using a naive enumeration scheme to enumerate all the target paths, only 3% would actually be useful. Furthermore, on average, only 7% of all candidate target paths turn out to be supported and of all supported paths only 27% remains after duplicate elimination. To get a feeling for the magnitude of the reduction in target paths (TPs) provided by the algorithm, Table 3.5 shows the absolute numbers, which are summed up over the whole data set of document-XSD pairs.

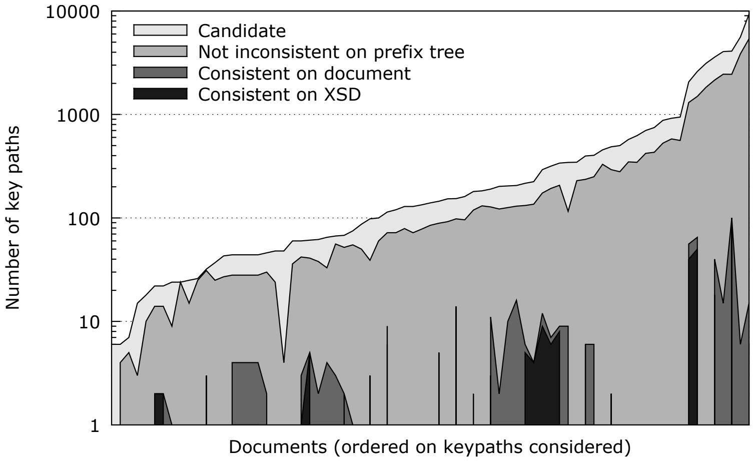


Figure 3.10: *Behavior of the one-key path miner for support threshold 10, max target path length 4 and max key path length 2.*

possible TPs	2.4×10^{11}
candidate TPs	6.7×10^{6}
supported TPs	8.4×10^{4}
unique TPs	1.3×10^{4}

Table 3.5: *Target path breakdown. This image shows the number of target paths in the search space and those that are effectively considered in different stages of the target path miner. The values shown are summed over the entire dataset.*

3.6.4 One-Key Paths

Figure 3.10 provides a visual interpretation of the reduction in number of key paths by the consecutive steps of the one-key path miner as described in Section 3.5.3. Again, for presentation purposes, the x-axis enumerates all document-XSD pairs increasingly ordered by the number of resulting candidate key paths. Specifically, the figure plots on a per document basis the following numbers: candidate key paths, paths for which the inconsistency test fails on the prefix tree, paths that are consistent on the document, and paths that are consistent w.r.t. the XSD. We first discuss the average improvement on a per document basis. Specifically, on average 39% of candidate paths are inconsistent over the prefix tree. This means that for 61% of the remaining key paths consistency needs to be tested on the document. On average (per document), only 6% of the key paths are consistent w.r.t. the document (qualify) and of these 68% turn out to be consistent w.r.t. the XSD. Table 3.6 shows

candidate KPs	48144
inconsistent KPs on prefix tree	29190
consistent KPs on document	484
consistent KPs on XSD	288

Table 3.6: *This image shows the number of key paths (KPs) that are effectively considered in different stages of the one-key path miner. The values shown are summed over the entire dataset.*

the absolute numbers summed up over the whole data set of document-XSD pairs, to give an indication of the effectiveness of the different stages of the one-key path miner.

It is interesting to observe that on the considered sample of real-world documents, consistency on the document does not always imply consistency w.r.t. the associated XSD. Specifically, Table 3.6 shows that overall only roughly 60% of key paths that are consistent w.r.t. the document are consistent w.r.t. the XSD as well.

3.6.5 Keys

Next, we discuss the actual keys returned by our algorithm. We use the hypergraph transversal algorithm to mine relational functional dependencies as, for instance, described by Mannila and Räihä [127], but any such algorithm can be readily plugged in. We consider keys with target path length at most 4 and key path length at most 2. In the following, we refer to testing consistency of a key w.r.t. its XSD (i.e., by applying the algorithm of Theorem 3.15) as the schema consistency test. Tables 3.7 and 3.8 then gather some statistics of discovered keys with and without the schema consistency test. First of all, it can be observed that not every document contains a key with the required support: only 30% and 16% of all documents have at least one key with support 10 and 100, respectively (Table 3.7). The latter might seem strange at first sight, but note that not all XML documents are in fact databases and that the requirement for a key to qualify (see Definition 3.7) is a severe one. Indeed, even lowering the support threshold to a value of two (experiment not shown here) only provides a key for 60% of the documents, but of course a key with support two is not likely to be very relevant. We note that the average support for discovered keys in this experiment equals 404 and 612 for support thresholds equal to 10 and 100, respectively, while the maximum support encountered is 2011, indicating that the discovered keys do cover a large number of elements.

The figures in the two tables nicely illustrate the effectiveness of schema consistency as a quality measure. Indeed, without schema consistency Ta-

	sup10	sup100
derived keys	107	54
docs with keys	27	15
avg keys per doc	4	3.6
max keys per doc	23	23
avg key paths in a key	1.3	1.3
max key paths in a key	2	2

Table 3.7: *Statistics of mined keys for support thresholds 10 and 100* without *the requirement to be consistent w.r.t. the associated XSD.*

ble 3.8 shows that 107 and 54 keys are derived for support threshold 10 and 100, respectively. Interestingly, in both cases, there is a document with a rather large number of keys: 23 to be specific. But, after the schema consistency test, each of these keys is removed as they all contain a key path that selects elements that are declared optional in the schema. Of course, one could debate about whether the schema is actually always correct or may be too liberal. One could always opt to *offer* keys that do not pass schema consistency to the user. However, after an inspection of the derived keys from our corpus, it becomes apparent that in many cases keys rejected by the schema are probably not keys at all. As an illustrative example, consider the three derived keys (all with support 340, and where *root* refers to the root context):

(root, `./Products`, (`./ID`))

(root, `./Products`, (`./Other_Information, ./Catalogue-Name`))

(root, `./Products`, (`./Type, ./Other_Information`))

After the schema consistency test only the first key remains. In this case, it should be clear that the second and third keys are not accurate, but are a glitch in the data as they both contain the field “other information”, which is a generic field that only happens to be unique when combined with another field. Therefore, one could say that the reduction from 107 to 43 and from 54 to 16 keys in Tables 3.7 and 3.8 actually improves the quality at the expense of lowering the quantity which in our opinion can be seen as a good thing as most data mining techniques suffer from an explosion in derived patterns.

3.6.6 Quality

It remains to discuss the quality of the keys. When the provided schema is accurate, the schema consistency test, as discussed above, provides a quality criterion in its own. A second quality criterion can be the high support of derived keys: as mentioned above, the support of the derived keys is on average

	sup10	sup100
derived keys	43	16
docs with keys	19	10
avg keys per doc	2.2	1.6
max keys per doc	9	4
avg key paths in a key	1.3	1.2
max key paths in a key	2	2

Table 3.8: *Statistics of mined keys for support thresholds 10 and 100* with *the requirement to be consistent w.r.t. the associated XSD.*

404 and 612 for support thresholds equal to 10 and 100, respectively, while the maximum support encountered is 2011.[25] Furthermore, when inspecting the discovered keys it appeared that in many cases keys select elements whose name contains 'ID'. Sometimes it is desirable to further refine the set of resulting keys, additional quality criteria such as the ones introduced in Section 3.4.3 may be utilized.

We finish with a discussion on implication of keys. Usually, in key discovery, the goal is to find a minimal set of keys, called a *cover*, from which all other keys can be derived. For instance, to this end, Grahne and Zhu [91] make use of the inference algorithms for XML keys that were investigated and shown to be polynomially computable by Buneman et al. [53]. Unfortunately, Theorem 3.25 shows that key implication in the presence of a schema is EXPTIME-hard. Still, there are opportunities for detecting duplicate keys. For instance, the next pair of discovered keys turns out to be equivalent (both with support 90):

((State: 188, Symbol: `ConstraintID`), `./*`, (`./*`))
((State: 167, Symbol: `PureOrMixtureData`),
`./Constraint/ConstraintID/*`, (`./*`))

Note that ConstraintID can only occur under a `Constraint`-element. We can therefore consider the keys to be equivalent as they select precisely the same set of target nodes.

3.6.7 Running Time

We next discuss the running time of the algorithm. Of course, the previous sections have already illustrated how the different mining steps succeed in

[25]Both thresholds impose a lower bound on the support value, hence the maximum values are not affected.

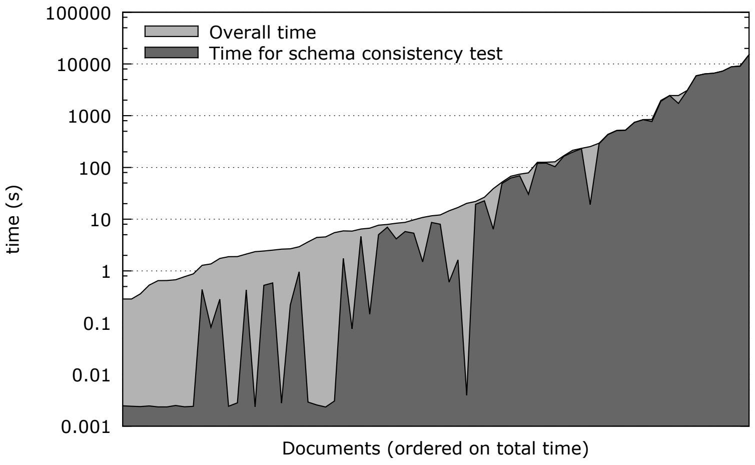


Figure 3.11: *Part of the overall running time that is spent on the schema consistency test; support threshold 2, max target path length 2 and max key path length 2.*

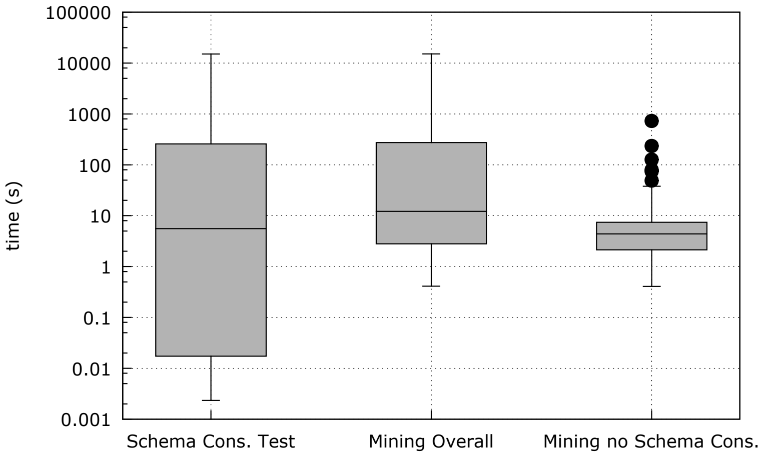


Figure 3.12: *Boxplots indicating average running times for schema consistency by itself, the entire mining algorithm and mining without schema consistency; support threshold 2, max target path length 2 and max key path length 2. The plots presented use* 1.5 *times the interquartile range for determining the whiskers. The black dots are outliers, each one depicting a document-XSD pair.*

reducing the number of considered contexts, target paths and key paths and every such reduction induces a gain in speed. Figure 3.12 gives insight in the overall running time. Here, we can see that a large fraction of the time is taken up by the schema consistency test. Furthermore, Figure 3.11 gives an indication of the proportion of time taken by the schema consistency test w.r.t. the overall running time. For presentation purposes, the x-axis enumerates all document-XSD pairs increasingly ordered by the time required for the schema consistency test. Note that the figure does not imply an exponential growth of the running time. In fact, as the x-axis does not correspond to a quantity, no inference can be made about the asymptotic growth of the running time.

To gain more insight in what part of the input controls the running time, we checked several metrics of the data and the schemas:

- XML metrics: number of nodes, depth, average children, number of labels, number of leafs, number of prefix tree nodes;
- XSD metrics: number of labels, number of states, number of contexts.

For the target miner, we found the number of nodes in the prefix tree to be correlated to the running time and the number of target paths. This is as expected, since the prefix tree is used for support calculation and equivalence tests; both are used continuously during in this part. For the key miner, we found that without XSD consistency tests, the same correlation is observed. Also, there is a correlation between the number of labels used in the document and the running time. Sadly, none of these metrics showed a clear connection to the running time when XSD consistency is used. We stress that key discovery is not a time critical task and that the algorithm only has to be run once for an XML-document and XSD. Nevertheless, the figures also show that the most room for improvement lies within a speedup of the schema consistency test and less in other components of the algorithm.

3.6.8 Optimizations

The execution of the algorithm can be tailored by switching several optimizations on or off. In this section, we take a look at some key optimizations and their effect on the running time.

We first focus on optimizations in the target miner. One of the most important optimizations is the *duplicate elimination*, described in Section 3.5.2. We consider three options:

1) no duplicate elimination;

2) duplicate elimination without a schema; and,

3) using duplicate elimination with a schema.

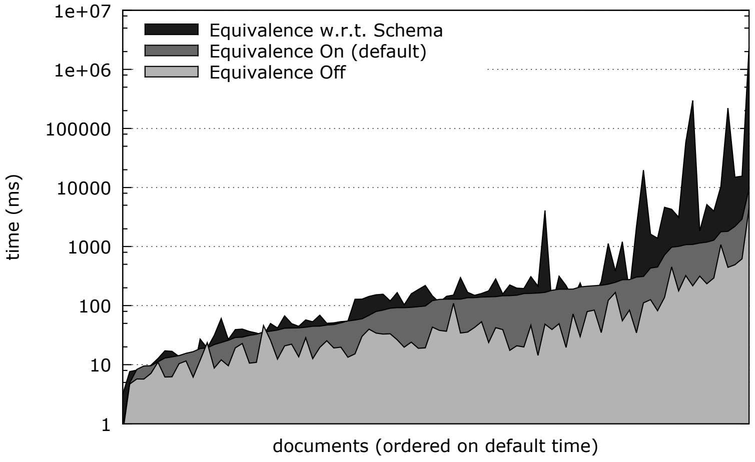


Figure 3.13: *Effect of equivalence optimizations on running time. Per-document running time of equivalence optimizations in the target miner; XSD consistency off, support threshold 10, max target path length 4.*

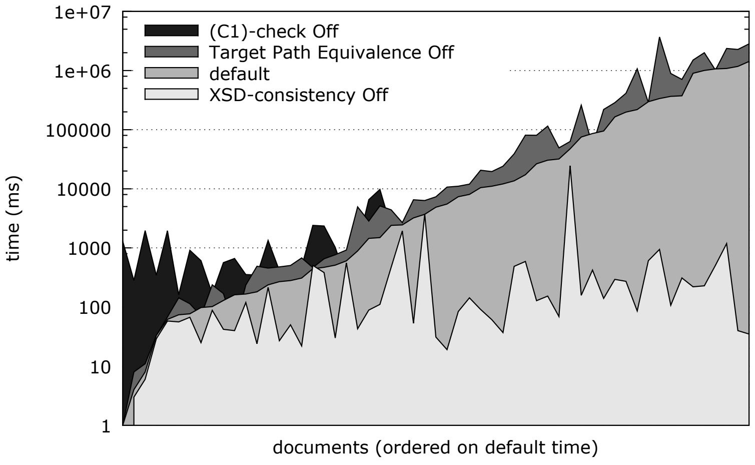


Figure 3.14: *Effect of different optimizations on running time. Per-document running time obtained by switching off different optimizations of the key path miner; support threshold 2, max target path length 2, max key path length 2.*

Figure 3.13 shows the running times for each of these, per document, ordered by (2). Option (1) is the fastest in almost every case, while (3) is the slowest in almost every case, sometimes even by several orders of magnitude. Notice that the time for options (2) and (3) does not exceed 10 seconds.

Although (1) may seem the best option for the target miner, it will produce a very large set of target paths (see above), each of them invoking a new run of the key path miner. Figure 3.14 shows the running time of the key miner, for the same documents, this time ordered by the default key miner time. When we compare the default time to the time where duplicate elimination of the target miner is switched off, we see the running time increase a lot, sometimes even by an order of magnitude. Note that the key path miner typically takes up a lot more time, hence it is advised to retain the duplicate elimination check (2), especially since duplicate elimination also improves the quality of the resulting keys.

As we have seen before, the XSD consistency test takes up a large portion of the running time. When switching this test off and resorting to XML consistency, we see that the running time improves by several orders of magnitude (Figure 3.14). Because of the applicability of the XSD consistency measure as a quality measure, however, we conclude that future work on the key miner software should make optimizing this part a priority.

3.7 Discussion

In this chapter, we initiated a fundamental study of properties of W3C XML keys in the presence of a schema and introduced an effective novel key mining algorithm leveraging on the formalism of levelwise search and on algorithms for the discovery of functional dependencies in the relational model.

An observed bottleneck of the proposed approach is to check consistency of a derived key w.r.t. the associated schema, even though the number of keys which have to be tested is greatly reduced by testing for inconsistency on the XML document, it should be investigated how schema consistency can be accelerated in practice. Another approach would be to try to find fast heuristic algorithms or to study the problem for subclasses of XSDs.

Another possible direction would be to investigate how the mining framework could be extended to top-level union in keys. It would be especially important to avoid an explosion of the size of the search space. The latter would also require finding heuristics for consistency testing in the presence of disjunction and a schema as this problem is CONP-hard.

It would also be interesting to see how the present framework can be extended to discover approximate XML keys. Grahne and Zhu [91] already provide a mechanism for doing this for other XML key formalisms. For this, we need a measure f over multisets which expresses how closely a multiset resembles a set. Then the *confidence* of the key $\phi = (c, \tau, P)$ can for instance be obtained by

$$\mathrm{agg}_{v \in \mathrm{CNodes}_t(c)} f(\{\mathrm{record}_P(u) \mid u \in \tau(t, v)\}),$$

where agg is an aggregate operator (as, e.g., sum). Our framework would then allow to plug in any algorithm for deriving relational approximate functional dependencies (see, for example King and Legendre [114] and Kivinen and Mannila [115]).

4 Parallel Evaluation of Multi-Semi-Joins

In this chapter, we study query evaluation and optimization in Big data systems. We introduce the multi-semi-join operator that allows for a reduction in total time for parallel MapReduce query plans for strictly guarded fragment queries. We provide an updated MapReduce cost model, show that the problem of finding an optimal query plan is NP-complete and give heuristic algorithms as an alternative solution.

4.1 Introduction

The problem of evaluating joins efficiently in massively parallel systems is an active area of research (e.g., [7, 10, 12, 34, 35, 56, 75, 116, 140, 148]). Here, efficiency can be measured in terms of different criteria, including net time, total time, amount of communication, resource requirements and the number of synchronization steps. As parallel systems aim to bring down the net time, i.e., the difference between query end and start time, it is often considered the most important criterium. Indeed, the amount of computing power itself seems to be no longer an issue because of the availability of services such as Amazon AWS. However, in pay-as-you-go plans, the cost *is* determined by the total time, that is, the aggregate sum of time spent by *all* computing nodes. On the other hand, for data analysts, a low net time, which consitutes the time between query submission and result collection, is of high importance. Hence, we focus on parallel evaluation of queries that minimizes total time while retaining a low net time. The low net time is achieved through the use of parallel query plans, while commonalities between different query parts are used to bring down the total time as redundant calculations can often be removed. In Chapter 5, we compare different parallel approaches with a sequential approach and study their properties in terms of total and net time using our own system called Gumbo.

Semi-joins have played a fundamental role in minimizing communication costs in traditional database systems through their role in semi-join reducers [36, 37], facilitating the reduction of communication in multi-way join computations. Using the relational algebra, the semi-join operator can be defined as $R \ltimes S = \pi_{R.*}(R \bowtie S)$. More specific, the result of a semi-join between two relations is the result of their natural join, projected on the schema of the first relation. This corresponds to the tuples of R for which a joining tuple exists in S. This is especially interesting in the context of a join: if we know which tuples will participate in the join, we can potentially avoid a lot of communication overhead. This is illustrated by the following query equivalence:

$$R \bowtie S = R \bowtie (S \ltimes R) = (R \ltimes S) \bowtie (S \ltimes R).$$

Avoiding communication overhead is important as this is a major bottleneck in distributed systems. Of course, this only holds if the semi-join can be calculated much more efficiently than the join itself. In more recent work, Afrati et al. [7] provide an algorithm for computing n-ary joins in MapReduce-style systems in which semi-join reducers play a central role. Furthermore, Nykiel et al. [139] and Wang and Chan [163] have demonstrated that combining multiple jobs in MapReduce can lead to significant cost reductions.

Motivated by the general importance of semi-joins and the cost reduction opportunities describe above, we study the system aspects of implementing semi-joins in a MapReduce context. In particular, we introduce a multi-semi-join operator $\ltimes\cdot$ that enables the evaluation of a set of semi-joins in one Mapreduce job while reducing resource usage like total time and requirements on cluster size, without sacrificing net time. We then use this operator to efficiently evaluate Strictly Guarded Fragment (SGF) queries [16, 84]. Not only can this query language specify all semi-join reducers, but also more expressive queries involving disjunction and negation.

We illustrate our approach by means of a simple example. Consider the following SGF query Q:

```
SELECT (x, y) FROM R(x, y)
WHERE (S(x, y) OR S(y, x)) AND T(x, z)
```

Intuitively, this query asks for all pairs (x, y) in R for which there exists some z such that

1. (x, y) or (y, x) occurs in S; and

2. (x, z) occurs in T.

To evaluate Q it suffices to (1) compute the following semi-joins

$$
\begin{array}{rcl}
X_1 & := & R(x, y) \ltimes S(x, y); \\
X_2 & := & R(x, y) \ltimes S(y, x); \\
X_3 & := & R(x, y) \ltimes T(x, z);
\end{array}
$$

(2) store the results in the binary relations X_1, X_2, or X_3; and (3) subsequently compute $\varphi := (X_1 \cup X_2) \cap X_3$. Our multi-semi-join operator $\ltimes\!\cdot(\mathcal{S})$ (defined in Section 4.5.2 and implemented as the algorithm MSJ) takes a number of semi-join-equations as input and exploits commonalities between them to optimize evaluation. In our framework, a possible query plan for query Q is of the form:

$$\begin{tikzcd}
& \boxed{\mathrm{EVAL}(R, \varphi)} \arrow[dl, dash] \arrow[dr, dash] & \\
\boxed{\mathrm{MSJ}(X_1, X_2)} & & \boxed{\mathrm{MSJ}(X_3)}
\end{tikzcd}$$

In this plan, the calculation of X_1 and X_2 is combined in a single MapReduce job, X_3 is calculated in a separate job; and $\mathrm{EVAL}(R, \varphi)$ is a third job responsible for computing the subset of R that adheres to φ. The combined evaluation of X_1 and X_2 makes it possible to exploit the overlap (R and S) between them. As numerous grouping options are possible in general, we provide a method that uses a cost model to determine the best[26] query plan for SGF queries. Note that, unlike the simple query Q illustrated here, SGF queries can be nested in general. In addition, we also show how to generalize the method to the simultaneous evaluation of multiple SGF queries.

The contributions of this chapter can be summarized as follows:

1. We introduce the multi-semi-join operator $\ltimes\!\cdot(\mathcal{S})$ to evaluate a set $\mathcal{S}$ of semi-joins and present a corresponding 1-round MapReduce implementation $\mathrm{MSJ}(\mathcal{S})$.

2. We present query plans for basic, that is, unnested, SGF queries and propose an improved version of the cost model presented in [139, 163] for estimating their cost (total time). As computing the optimal plan for a given basic SGF query is NP-hard, we utilize an existing greedy heuristic which we refer to as GREEDY-BSGF.

[26]One that is parallel and has a minimal total cost.

3. We show that the evaluation of (possibly nested) SGF queries can be reduced to the evaluation of a set of basic SGF queries in an order consistent with the dependencies induced by the former. In this way, computing an optimal plan for a given SGF query (which is NP-hard as well) can be tackled by a two-tier strategy, where first an optimal ordering of the basic SGF subqueries is determined, followed by an optimal evaluation of each of these basic subqueries. We present the greedy algorithm GREEDY-SGF.

While this chapter focuses on the above topics, the next chapter provides more practical details on the implementation of the presented techniques in a system called Gumbo, describes practical optimizations and gives insights in the performance of our methods through an extensive experimental evaluation.

Outline. This chapter is organized as follows. We discuss related work in Section 4.2. We introduce the strictly guarded fragment (SGF) queries in Section 4.3.1. Section 4.4 introduces the MapReduce computation model and the revised version of an existing cost model. In Section 4.5, we consider the evaluation of multi-semi-joins and SGF queries using MapReduce. Section 4.7 discusses the addition of new atom types and using bag semantics; we conclude in Section 4.8.

4.2 Related Work

Guarded Fragment Queries. Recall that first-order logic (FO) queries are equivalent in expressive power to the relational algebra (RA) [59] and form the core fragment of SQL queries (see, e.g., Abiteboul et al. [3]). Guarded fragment (GF) queries have been studied extensively by the logicians in the 1990s and 2000s, and they emerged from the intensive efforts to obtain a syntactical classification of FO queries with decidable satisfiability problems. For more details, we refer the reader to the highly influential paper by Andréka et al. [16], as well as survey papers by Grädel [90] and Vardi [159]. In traditional database terms, GF queries are equivalent in expressive power to semi-join algebra [119]. Closely related are freely acyclic GF queries, which are GF queries restricted to using only the $\wedge$ operator and guarded existential quantifiers [144]. Flum et al. [84] introduced the term strictly guarded fragment queries for queries of the form $\exists\bar{y}(\alpha \wedge \varphi)$. That is, guarded fragment queries without Boolean combinations at the outer level. We consider a slight generalization of these queries as explained in Remark 4.5.

Query Optimization. In general, obtaining the optimal plan in SQL-like query evaluation, even in centralized computation, is a hard problem Chaudhuri [55] and Ioannidis [110]. Classic works by Yannakakis and Bernstein advocate the use of semi-join operations to optimize the evaluation of conjunctive queries [36, 37, 168]. A lot of work has been invested to optimize query evaluation in Pig [1, 141], Hive [19, 105–107, 156] and SparkSQL [28, 152, 165] (previously called Shark) as well as in MapReduce settings in general [46]. Nykiel et al. [139] develop a method for optimally grouping MapReduce jobs that share map input and/or output based on a cost model. They show that the optimal grouping problem is NP-hard and provide a greedy heuristic. Wang and Chan [163] built upon this work by generalizing the technique and adding new optimizations. They also refine the proposed cost model. None of the work referenced above targets SGF queries directly.

Semi-joins in MapReduce. Tao et al. [155] studied *minimal* MapReduce algorithms, i.e., algorithms that scale linearly to the number of servers in all significant aspects of parallel computation such as reduce compute time, bits of information received and sent, as well as storage space required by each server. They show that, among many other problems, a single semi-join query between two relations can be evaluated by a one-round minimal algorithm. This is a simpler problem, as a single basic SGF query may involve multiple semi-join queries. Some other related work is the following: efficient MapReduce algorithms to evaluate recursive Datalog queries [6, 11] and handling skew in multiway joins [12]. Afrati et al. [7] introduced a generalization of Yannakakis' algorithm (using semi-joins) to a MapReduce setting. Note that Yannakakis' algorithm starts with a sequence of semi-join operations, which is a (nested) SGF query in a very restricted form. Hassan and Bamha [99] present a semi-join evaluation algorithm in the Map-Reduce-Merge framework (see Yang et al. [167]), and address the problem of skew by sending only histograms of the data over the network to bring down communication costs. This technique closely resembles our *tuple id optimization* presented in Section 5.3.1.

Query Evaluation in MapReduce. Afrati, Sarma, Salihoglu and Ullman obtained that the input to the reducer, i.e., the number of key-value pairs generated are crucial factors in determining the efficiency of a MapReduce program [8]. They defined the notion of *replication rate*, i.e., the average number of key-value pairs generated by the map function from each input tuple, and derive an interesting trade-off between replication rate and reduce input size for specific problems such as Hamming distance 1 and triangle finding. For a nice illustration, we refer the reader to Ullman's example of drug interaction [157]. This work was then extended to include similar tradeoffs for some other problems [5, 148]. Afrati and Ullman introduced a MapReduce algorithm for evaluating some specific n-ary queries in one round [9, 10]. Their algorithm later became the precursor to the Hypercube algorithm for evaluating join queries, studied extensively by Beame, Koutris and Suciu [34, 35, 116] in various settings such as a load-balanced setting, matching databases and skewed databases. Some interesting lower bounds are obtained, and a system called Myria has been developed based on their study [96]. MyriaL, the language of Myria, consists of SQL, Datalog, and PigLatin syntaxes, which are incompatible with the syntax of SGF queries that we consider here.

4.3 Preliminaries

4.3.1 Strictly Guarded Fragment Queries

In this section, we define the strictly guarded fragment queries (SGF) [84], but use a non-standard, SQL-like notation for ease of readability.

We assume given a fixed infinite set $\mathbf{D} = \{a, b, \dots\}$ of data values and a fixed collection of relation symbols $\mathbf{S} = \{R, S, \dots\}$, disjoint with $\mathbf{D}$. Every relation symbol $R \in \mathbf{S}$ is associated with a natural number called the *arity of* R. An expression of the form $R(\bar{a})$ with R a relation symbol of arity n and $\bar{a} \in \mathbf{D}^n$ is called a *fact*. A database DB is then a finite set of facts. Hence, we write $R(\bar{a}) \in \mathsf{DB}$ to denote that a tuple $\bar{a}$ belongs to the R relation in DB. Alternatively, we say a relation is a set of facts with a common relational symbol. The number of tuples in a relation R is denoted by $\|R\|$, while the total number of bytes belonging to a relation is denoted by $|R|$. We also assume given a fixed infinite set $\mathbf{V} = \{x, y, \dots\}$ of variables, disjoint from $\mathbf{D}$ and $\mathbf{S}$. A *term* is either a data value or a variable. An *atom* is an expression of the form $R(t_1, \dots, t_n)$ with R a relation symbol of arity n and each of the t_i a term, $i \in [1, n]$. Note that every fact is also an atom. A *basic strictly guarded fragment (BSGF) query* (or just a basic query for short) is an expression of the form

$$Z := \texttt{SELECT } \bar{x} \texttt{ FROM } R(\bar{t}) \text{ [} \texttt{WHERE } C \text{]}; \tag{4.1}$$

where $\bar{x}$ is a sequence of variables that all occur in the atom $R(\bar{t})$, and the `WHERE` C clause is optional. If it occurs, C must be a Boolean combination of atoms. Furthermore, to ensure that queries belong to the guarded fragment, we require that for each pair of distinct atoms $S(\bar{u})$ and $T(\bar{v})$ in C it must hold that all variables in $\bar{u} \cap \bar{v}$ also occur in $\bar{t}$. (See also Remark 4.5 below.) The atom $R(\bar{t})$ is called the *guard* of the query, while the atoms occurring in C are called the *conditional atoms*. We interpret Z as the output relation of the query.

On a database DB, BSGF query (4.1) defines a new relation Z containing all tuples $\bar{a}$ for which there is a substitution σ for the variables occurring in $\bar{t}$ such that $\sigma(\bar{x}) = \bar{a}$, $R(\sigma(\bar{t})) \in \mathsf{DB}$, and C evaluates to `True` in DB under substitution σ. Here, the evaluation of C in DB under σ is defined by recursion on the structure of C. If C is C_1 `OR` C_2, C_1 `AND` C_2, or `NOT` C_1, the semantics correspond to the usual Boolean interpretation. If C is an atom $T(\bar{v})$ then C

evaluates to `True` if $\sigma(\bar{t}) \in R(\bar{t}) \ltimes T(\bar{v})$, i.e., if there exists a T-atom in DB that equals $R(\sigma(\bar{t}))$ on those positions where $R(\bar{t})$ and $T(\bar{v})$ share variables.

Example 4.1 (Semi-join applications). The intersection $Z_1 := R \cap S$ and the difference $Z_2 := R - S$ between two relations R and S are expressed as follows:

$$\begin{aligned} Z_1 &:= \texttt{SELECT } \bar{x} \texttt{ FROM } R(\bar{x}) \texttt{ WHERE } S(\bar{x}); \\ Z_2 &:= \texttt{SELECT } \bar{x} \texttt{ FROM } R(\bar{x}) \texttt{ WHERE NOT } S(\bar{x}); \end{aligned}$$

The semi-join $Z_3 = R(\bar{x}, \bar{y}) \ltimes S(\bar{y}, \bar{z})$ and the antijoin $Z_4 = R(\bar{x}, \bar{y}) \rhd S(\bar{y}, \bar{z})$ are expressed as follows:

$$\begin{aligned} Z_3 &:= \texttt{SELECT } \bar{x}, \bar{y} \texttt{ FROM } R(\bar{x}, \bar{y}) \texttt{ WHERE } S(\bar{y}, \bar{z}); \\ Z_4 &:= \texttt{SELECT } \bar{x}, \bar{y} \texttt{ FROM } R(\bar{x}, \bar{y}) \texttt{ WHERE NOT } S(\bar{y}, \bar{z}); \end{aligned}$$

The following BSGF query selects all the pairs (x, y) for which $(x, y, 4)$ occurs in R and either $(1, x)$ or $(y, 10)$ is in S, but not both:

$$\begin{aligned} Z_5 := \; & \texttt{SELECT } (x, y) \texttt{ FROM } R(x, y, 4) \\ & \texttt{WHERE } (S(1, x) \texttt{ AND NOT } S(y, 10)) \\ & \texttt{OR } (\texttt{NOT } S(1, x) \texttt{ AND } S(y, 10)); \end{aligned}$$

Finally, the traditional star semi-join between $R(x_1, \dots, x_n)$ and relations $S_i(x_i, y_i)$, for $i \in [1, n]$, is expressed as follows:

$$\begin{aligned} Z_6 := \; & \texttt{SELECT } (x_1, \dots, x_n) \texttt{ FROM } R(x_1, \dots, x_n) \\ & \texttt{WHERE } S(x_1, y_1) \texttt{ AND} \dots \texttt{ AND } S(x_n, y_n); \end{aligned}$$

These queries illustrate the applicability of semi-joins in different types of queries. $\triangle$

Remark 4.2 (Semi-join algebra syntax). The BSGF query in Equation (4.1) can be rewritten in semi-join algebra syntax by replacing each atom $T(\bar{v})$ in C by $R(\bar{t}) \ltimes T(\bar{v})$, and projecting the end result of C onto the coordinates $\bar{x}$. $\diamond$

A *strictly guarded fragment (SGF) query* is a collection of BSGFs of the form

$$Z_1 := \xi_1; \dots ; Z_n := \xi_n;$$

where each ξ_i is a BSGF that can mention any of the predicates Z_j with $j < i$. On a database DB, the SGF query then defines a new relation Z_n where every occurrence of Z_i is defined by evaluating ξ_i.

Example 4.3 (Basic queries). Let **Amaz**, **B&N**, and **BD** be relations containing tuples (title,author,rating) corresponding to the books found at Amazon, Barnes and Noble, and Book Depository, respectively. Let **Upcoming** contain tuples (newtitle, author) of upcoming books. The following query selects all upcoming books (newtitle, author) of authors that have not yet received a "bad" rating for the same title at all three book retailers (Z_2 is the output relation):

$$\begin{aligned} Z_1 \quad &:= \quad \texttt{SELECT}\ \mathrm{aut}\ \texttt{FROM}\ \mathbf{Amaz}(\mathrm{ttl}, \mathrm{aut}, \text{`}\mathsf{bad}\text{'}) \\ & \qquad \texttt{WHERE}\ \mathbf{B\&N}(\mathrm{ttl}, \mathrm{aut}, \text{`}\mathsf{bad}\text{'})\ \texttt{AND}\ \mathbf{BD}(\mathrm{ttl}, \mathrm{aut}, \text{`}\mathsf{bad}\text{'}); \\ Z_2 \quad &:= \quad \texttt{SELECT}\ (\mathrm{new}, \mathrm{aut})\ \texttt{FROM}\ \mathbf{Upcoming}(\mathrm{new}, \mathrm{aut}) \\ & \qquad \texttt{WHERE NOT}\ Z_1(\mathrm{aut}); \end{aligned}$$

Note that this query cannot be written as a basic query, since the atoms in the query computing Z_1 must share the ttl variable, which is not present in the guard of the query computing Z_2. △

Remark 4.4 (Atom types). For ease of exposition, we only allow filters that test the equality between attributes, or between an attribute and a constant (e.g., 'bad' in $\mathbf{Amaz}(\mathrm{ttl}, \mathrm{aut}, \text{`}\mathsf{bad}\text{'})$). It is straightforward to add more advanced filters including inequalities and Boolean combinations of filters. In Section 4.7.1 we outline how the algorithms presented in this chapter can be extended to work with these additional atom types. ◇

Remark 4.5 (Syntax and nesting depth). The syntax we use here differs from the traditional syntax of the Guarded Fragment [84], and is actually closer in spirit to join trees for acyclic conjunctive queries [37], although we do allow disjunction and negation in the where clause. In the traditional syntax, a projection in the guarded fragment is only allowed in the form $\exists \bar{w} R(\bar{x}) \wedge \varphi(\bar{z})$ where *all* variables in $\bar{z}$ must occur in $\bar{x}$. One can obtain a query in the traditional syntax of the guarded fragment from our syntax by adding extra projections for the atoms in C. For example,

$$\texttt{SELECT}\ x\ \texttt{FROM}\ R(x,y)\ \texttt{WHERE}\ S(x,z_1)\ \texttt{AND NOT}\ S(y,z_2)$$

becomes

$$\exists y (R(x,y) \wedge (\exists z_1) S(x,z_1) \wedge \neg (\exists z_2) S(y,z_2)).$$

We note that this transformation increases the nesting depth of the query. ◇

4.4 A Cost Model for Hadoop MapReduce

As our aim is to reduce the total cost of already parallel query plans and our algorithms will search for the most cost-effective way to combine MapReduce

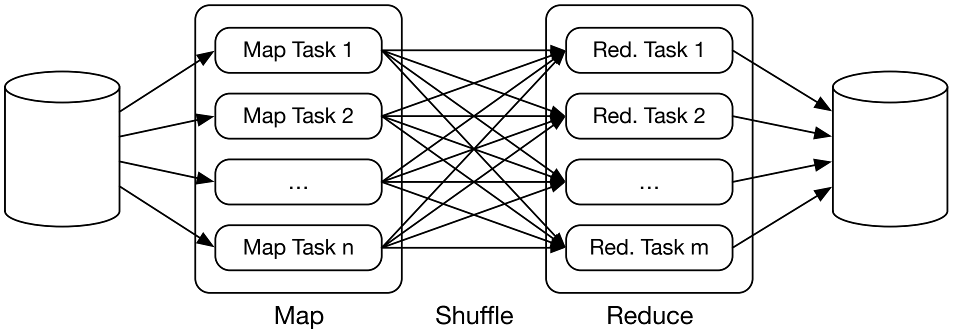


Figure 4.1: *Overview of the MapReduce computation model. We identify three different phases: map, shuffle and reduce.*

jobs, we need a cost model that is able to estimate this metric for a given job. Therefore, we first recall the MapReduce computation model (MR for short) that we can find in a variety of frameworks that are available today. Next, we discuss the details of the MR pipeline in the popular open-source Hadoop framework [69, 70, 164], and present an adaptation of an existing MR cost model [163], which is used in the algorithms discussed in Section 4.5.

4.4.1 The MapReduce Computation Model

An *MR job* is a pair (μ, ρ) of functions, where μ is called the map and ρ the reduce function. The execution of an MR job on an input dataset I proceeds in three stages, which are shown in Figure 4.1.

1. In the first stage, called the *map stage*, each fact $f \in I$ is processed by μ, generating a collection $\mu(f)$ of key-value pairs of the form $\langle k : v \rangle$.

2. The next stage, called the *shuffle stage*, is responsible for grouping the key-value pairs generated by the map phase, i.e., the collection $\bigcup_{f \in I} \mu(f)$, by their key. The result is a collection of groups

$$\langle k_1 : V_1 \rangle, \ldots, \langle k_n : V_n \rangle,$$

   where each V_i is a set of values. We also refer to these groups as *key-valueset pairs*.

3. In the final stage, called the *reduce stage*, each group $\langle k_i : V_i \rangle$ is processed by the reduce function ρ resulting again in a collection of values per group. The total collection $\bigcup_{i=1}^{n} \rho(\langle k_i : V_i \rangle)$ is the final output of the MR job.

MR jobs can be easily parallelized. Indeed, the maps can be applied in parallel to all facts $f \in I$ as the computations can be performed independently. The same holds for the reduce function, which can be applied in parallel to all key-valueset pairs $\langle k_1 : V_1 \rangle, \ldots, \langle k_n : V_n \rangle$. Hence, the amount of parallel operations in the map phase is bounded by the number of input facts $|I|$; in the reduce phase this amount is bounded by the number of distinct keys n. As the input for a reduce function may originate from different mappers, a reduce function cannot be applied before all map output is shuffled. Hence, the shuffle phase serves as a synchronization point between the map and reduce phases and could hinder the overall job progress when certain map tasks take too long.

An *MR program* is a directed acyclic graph of MR jobs, where an edge from job (μ, ρ) to (μ', ρ') indicates that (μ', ρ') operates on the output of (μ, ρ). We refer to the length of the longest path in an MR program as its *number of rounds*. Since each job must complete before the jobs that depend on it can start, each dependency serves as an additional synchronization point (next to the shuffle phase). Hence, when equivalent MR programs are available, those that have less rounds may result in lower net or total times as they exhibit less successive synchronization points.

4.4.2 MapReduce in Hadoop

While the idea behind the MR computation model is simple, executing an MR program on a real system requires an enviroment that manages cluster resources in a scalable way and fills in the necessary details: how are map/reduce functions applied? how is data transferred during the shuffle phase? where do the input and output reside? Also, the environment should provide a way of controlling the amount of parallelization in the map and reduce phases to find a good balance between parallel execution and job overhead. The open-source framework Apache Hadoop [94] is the system of our choice. We will now discuss the internals of this system that are necessary to understand the details of the cost model we present in Section 4.4.3.

Hadoop provides an environment for executing MR programs. Hadoop runs on a cluster of compute nodes and offers an environment where the user only needs to define the input data, a map function and a reduce function; by default, everything else is handled by the framework itself. As of Hadoop version 2, the system consists of two major components: a distributed file system called HDFS and a resource manager called Yarn. We now briefly discuss both components. For more details we refer to White [164].

Hadoop's default file system offers fault-tolerant distributed file storage and is called HDFS. Its fault-tolerance is accomplished by replicating each file across a given number of compute nodes (3 by default). When a node fails,

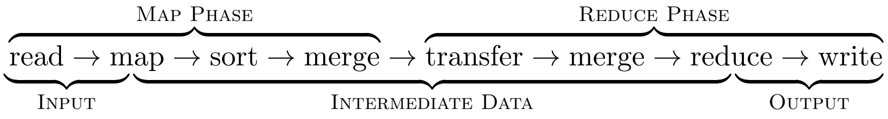

Map Phase
Reduce Phase
read → map → sort → merge → transfer → merge → reduce → write
Input
Intermediate Data
Output

Figure 4.2: *Overview of the MapReduce pipeline in Hadoop.*

spare copies are still be available at alternative locations and can be used to resume or restart calculations. As HDFS is designed to work with large files, each file is split into so-called *blocks* of a predefined size (128MB by default). These blocks can be managed (e.g., replicated) in an independent way.

Yarn is Hadoop's resource manager and controls the scheduling of jobs. While Yarn is also capable of executing non-MapReduce programs, we only focus on MapReduce in this work. Yarn supports the execution of MR jobs by first starting the required amount of map tasks. After completing the map tasks, Yarn creates a user-defined number of reduce tasks that each fetch their data from the mappers (shuffle phase) and apply the reduce function to it. Both map and reduce tasks run in a Java Virtual Machine (JVM) and have a fixed amount of memory at their disposal, which is used to read, process and buffer the data that is operated on. It is possible to fine-tune Hadoop's parameters to configure it specifically for a given application or node setup.

In the remainder of this section, we discuss the individual components of the entire Hadoop MR pipeline, which is depicted in Figure 4.2 and focus on the parts that are important to construct an accurate cost model in the next section. We choose to split up the pipeline into only two global phases that correspond to the type of tasks supported by Hadoop: a map and a reduce phase. The "missing" shuffle phase is spread across different parts of the pipeline (sort, map-side merge, transfer and reduce-side merge). The internals of the map phase are shown in Figure 4.3; we refer to White [164, Figure 7-4, Chapter 7] for more details on the different stages.

Input Read

For each MR job, a set of files is associated with the map function(s). Recall that these files are stored on HDFS in blocks of 128MB in size. For each *input split*, which maps to one HDFS block by default, a map task is created. The map task is responsible for applying the map function to the data that resides in the aforementioned split. The map task itself will apply the map function to the input values sequentially. This way of operation allows task overhead to be minimized, while at the same time a sufficiently large degree of parallelization can be obtained as multiple map tasks can run in parallel

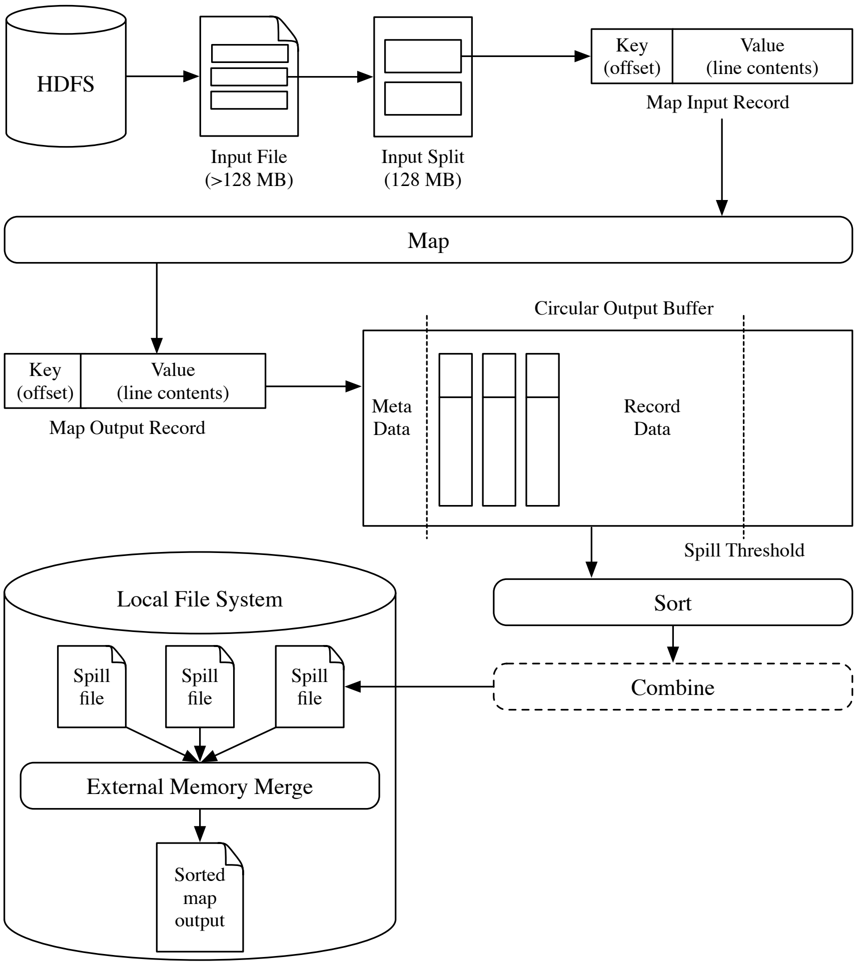


Figure 4.3: *Detailed overview of the Map phase.*

(typically about ten per compute node, depending on the available resources).

Hadoop prefers map tasks that run on the nodes that store their input data. These tasks are denoted *data-local map tasks* and aid to avoid data transfer between nodes over the network, which is typically a costly operation.

Hadoop allows adjusting the split size, i.e., the amount of data assigned to one map task. Care should be taken when making this change as this may have undesired side-effects. Indeed, when an input split is bigger than an HDFS block, the mapper may need to fetch input data from one or more *different* nodes, as we cannot guarantee data locality in this case. When the split size is too small, this may lead to a high number of mappers which cannot be scheduled to run in parallel due to resource constraints. This may in turn lead to increased job overhead and higher net times. We further explore this in the form of the map shaping optimization in Section 5.3.5.

Remark 4.6 (Small input files). Sometimes, it may not be possible to avoid small splits, e.g., when the input files all contain a tiny amount of data and are significantly smaller than the block size. Hadoop provides several solution to this problem such as the `CombineTextInputFormat` or `SequenceFiles`, which can serve as a wrapper or container for a set of small files. In this work we assume only files of significant size are considered. ◇

Map

A map task (or mapper) reads input records (facts) from its corresponding HDFS input block one by one and applies the user-defined map function to it. By default, an input record is a key-value pair where the key is the records's byte position in the file and the value is a string containing the contents of the current line.[27] The user-defined map function generates a set of records (key-value pairs) for each input. These records are collected into the map output buffer, which has a fixed size[28].

Sort, Combine & Spil

When the map output buffer fills up beyond a given threshold[29], its contents are *spilled* (written) to local disk (See Figure 4.3). An important caveat is that, besides being used for the raw map output records, the buffer also stores metadata, amounting to 16 bytes per output record [103]. This metadata is used to indicate the boundaries of the raw key and value bytes. When output records are relatively small, the metadata can take up a large part of the buffer, which may lead to unexpected spilling.

Before the buffer is spilled to disk, an in-memory sort is applied to the records to guarantee that those that have the same key and/or are destined for the same reducer, are grouped together. More details on how the groups are formed are given by White [164]. It is important to understand that, in general, multiple groups/keys will be processed by the same reducer to find a balance between parallelization and task overhead. A hash function is used to determine the reduce task for a given key. This hash function should be carefully chosen in order to avoid skew.

Remark 4.7 (Types of skew). Kwon et al. [117] give an overview of different types of skew that can appear in a MapReduce application. Map-side skew can

[27]This behaviour can be modified to obtain a better performance.

[28]**Hadoop setting:** `mapreduce.task.io.sort.mb`

[29]**Hadoop setting:** `mapreduce.map.sort.spill.percent`

arise due to input records that take too long to process, or multiple types of input data that each must be treated in a different way. Reduce-side skew can arise when the records are not evenly distributed across all reducers, or when certain key-valueset pairs require more processing time then others (expensive input). As map and reduce code can contain arbitrary user logic, it is possible to see the reduce-side skew methods in mappers too. ◇

After sorting, an optional combiner is applied to the result of the sort. The task of a combiner is to process the values that have the same key and reduce the number of records and/or their size *before* they are written to disk or transmitted over the network, as this reduces overhead. Often, the combiner strongly resembles the reduce function, but the availability and the effectiveness of a combiner both depend on the application.

Merge (Map-side)

During the map phase, the buffer contents may be spilled to disk several times, leading to a number of different spill files that are internally sorted. To obtain one final output file for the map task that contains a partition for each reducer, all the spill files need to be merged. This is accomplished through the use of an external-memory merge that operates with a merge factor[30] D in a number of rounds. When n spill files need to be merged, each *round* will perform a sequence of n/D merges. Each merge step processes D spill files and results in n/D spill files. This process repeats for $\lceil \log_D n \rceil$ rounds and results in one final sorted file, containing a partition for each reducer. As each stage involves reading and writing all map output from and to local disk, merging is an expensive operation that, as we will see later, has a definite impact on both net and total time. It is therefore of high importance that the number of spill files is kept low.

When more than three spill files are present, the combiner (if specified) is run on the intermediate outputs of a merge in order to further reduce the size of the map output data [164, p. 198].

Transfer

When all map tasks are completed, a user-defined number of reduce tasks (or reducers) is started. Each reducer contacts the nodes where the map output data resides and requests its data partition.[31] This phase is sometimes called the *shuffle phase*. Clearly, a too high number of mappers or reducers is undesired, as each reducer possibly needs to fetch data from every mapper,

[30]**Hadoop setting:** `mapreduce.task.io.sort.factor`
[31]The actual data is sent over the http-protocol.

creating a high overhead. On the other hand, a too low number will cause each task to last longer, causing an increased net time. This indicates that the number of reducers needs to be carefully chosen.

Hadoop offers an optimization that allows reducers to start when a given percentage of mappers has been completed[32]. This makes data transfer possible *before* all mappers are finished. Note that the actual application of the reduce function on the data values cannot start before all map tasks finish, as all map output data for each particular reducer needs to be available to ensure all values belonging to a particular key are present. This may lead to reducers that are idle for a long time.

Merge (Reduce-side)

When a reduce task has received the relevant data from all mappers, the data needs to be sorted again to ensure every key is associated with the correct values, possibly originating from different mappers. Therefore, all map outputs that are received are again merged together into one sorted file using an exernal-memory merge.

Hadoop offers an extra optimization that removes the need for the last merge round. Indeed, the final read-write step can be removed as the output of the final merge step can be delivered directly to the reduce function. This is possible only when the memory requirements for the reduce function itself are significantly low. For our reducers, this is generally the case.

Reduce

In this stage, the user-defined reduce function is applied to all key-valueset pairs. In Hadoop, the output is again a set of key-value pairs that are written to an output buffer. Hadoop allows output from one task to be directed to different files, which makes it possible for the reducer to perform different tasks on the same data and write output related to separate tasks to a separate location on HDFS. This functionality is used in our implementation in Gumbo to calculate the result of multiple queries using only one job. The algorithm for evaluating multiple semi-join queries in the same job (described in Section 4.5.2) requires this behaviour.

Output Write

The output buffer for a certain reducer is written to HDFS when a threshold is exceeded. The process of writing to HDFS consists of replicating each output block the required number of times across nodes of the cluster. This happens

[32]**Hadoop setting:** `mapreduce.job.reduce.slowstart.completedmaps`

l_r	local disk read cost (per MB)
l_w	local disk write cost (per MB)
h_r	hdfs read cost (per MB)
h_w	hdfs write cost (per MB)
t	transfer cost (per MB)
p	penalty for map-reduce task pairs
c_{b}	map buffer metadata per record (16 bytes)
c_{o}	map output metadata per record (2 bytes)
M_i^j	map output bytes for partition i, split j
$\widehat{M}_i^j$	map output records for partition i, split j
R_i	reduce i input bytes
$\widehat{R}_i$	reduce i input records
K_i	reduce i output size (in MB)
m_i	number of map tasks for $\mathcal{I}_i$
r	number of reduce tasks
D	external sort merge factor
buf_{map}	map task buffer limit (in MB)
buf_{red}	reduce task buffer limit (in MB)

Table 4.1: *Description of constants used in the cost model.*

through the use of a write pipeline that spans multiple nodes. For more details on this process, we refer to White [164]. It is important to realize that this procedure causes the cost of writing output to become significantly larger than reading from HDFS, or reading from or writing to local disk.

4.4.3 Cost Model

As the previous section implies, the inner workings of Hadoop's MapReduce are more intricate than the generic map-reduce procedure. Based on the discussion of the components of Hadoop MR above, we now introduce a cost model for analyzing the I/O complexity of an MR job based on the one introduced by Nykiel et al. [139] and Wang and Chan [163] but with a distinctive difference. The most important adaptation we introduce, and that is elaborated upon below, takes into account that the map function may have a different input/output ratio for different parts of the input data.

Map Phase

The cost of a single map task consists of (*i*) reading the data from HDFS and applying the map function to it, (*ii*) sorting, combining and merging the *local* key-value pairs produced by the map function, and (*iii*) writing the result to local disk. Let $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$ denote the partition of the input tuples. Let N_i be the size (in MB) of $\mathcal{I}_i$, M_i be the size (in MB) of the intermediate data output by the mapper on $\mathcal{I}_i$ and $\widehat{M_i}$ be the number of output records produced by the mapper on $\mathcal{I}_i$. For a partitioning of $\mathcal{I}_i$ into l input splits $\mathcal{I}_i^1, \dots, \mathcal{I}_i^l$, we denote by N_i^j the size (in MB) of $\mathcal{I}_i^j$, by M_i^j be the size (in MB) of the intermediate data output by the mapper on $\mathcal{I}_i^j$, and by $\widehat{M_i^j}$ the number of output records produced by the map function on $\mathcal{I}_i^j$. Finally, we assume $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$ are chosen in such a way that the map function behaves *uniformly* on all facts in each $\mathcal{I}_i$, which is defined as

$$\forall i, j : M_i^j = \frac{M_i}{l_i}, \tag{4.2}$$

where l_i is the number of input splits of $\mathcal{I}_i$.

The cost of map task m_i^j that processes input split $\mathcal{I}_i^j$ can be approximated by the following calculation:

$$\mathcal{C}_{map}(N_i^j, M_i^j, \widehat{M_i^j}) = h_r N_i^j + \text{merge}_{map}(M_i^j, \widehat{M_i^j}) + l_w(M_i^j + c_\text{o}\widehat{M_i^j}). \tag{4.3}$$

Here, $h_r N_i^j$ is the cost of reading the data and $l_w(M_i^j + \widehat{M_i^j})$ is the cost of storing the output (and meta) data to local disk. The cost of map-side sorting and merging, denoted by $\text{merge}_{map}(M_i)$, is expressed by

$$\text{merge}_{map}(M_i^j, \widehat{M_i^j}) = (l_r + l_w) M_i^j \log_D \left\lceil \frac{M_i^j + c_\text{b}\widehat{M_i^j}}{buf_{map}} \right\rceil. \tag{4.4}$$

Table 4.1 gives an overview of the meaning of the variables that are used here. Note that the costs for executing the map function, sorting the data and applying the combiner are omitted from the calculation as they are generally not the dominant part of the total cost. If desired, they can easily be integrated into the existing constants or added separately.

The merge cost calculation is explained as follows. The total output of a map task equals $M_i^j + c_\text{b}\widehat{M_i^j}$, which means that $S = {}^{M_i^j + c_\text{b}\widehat{M_i^j}}/_{buf_{map}}$ spill files are created. In order to merge these files together, $\log_D S$ rounds are necessary for a merge factor D. In each round, all data is read from and written back to local disk, explaining the $(l_r + l_w)M_i^j$ factor. The following example illustrates the calculation.

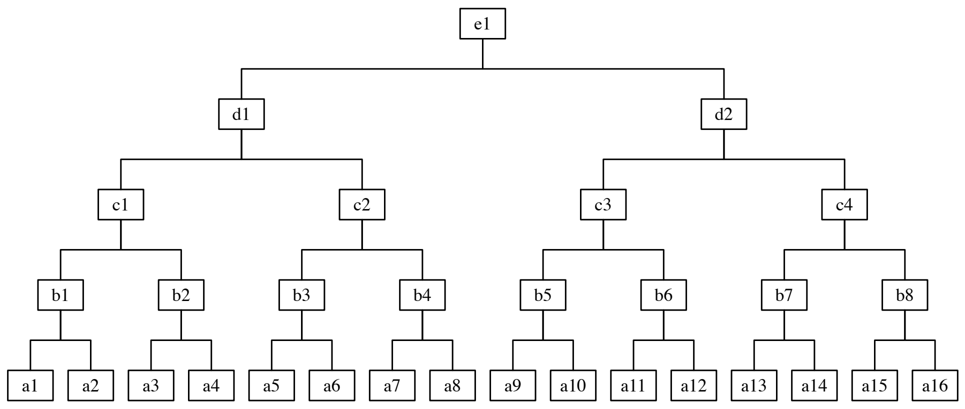


Figure 4.4: *Illustration of an external-memory merge for 16 spill files. Each box represent a sorted file and contains all data from the leafs below it. For example, c1 contains all data from a1-a4. In this case, all data is read from and written to disk four times.*

Example 4.8 (Merge-cost calculation). Consider a map task whose map output buffer threshold is exceeded (see Section 4.4.2) 16 times. This leads to 16 spill files being written to disk. We assume a merge factor of $D = 2$. Hence, after the map function finishes, the files are merged in groups of two. This means the number of files are reduces by a factor 2 after each merge-round. In total $\log_2 16 = 4$ rounds are necessary to obtain one final output file.

In one merge-round *all* data from the original 16 spill files will read from and written back to local disk. As four rounds are necessary, the merge-cost equals $4(l_r + l_w)M$, where M is the size of the output produced by the current map task, or equivalently, the total size of the original spill files. Note that the initial spill is not included in this calculation as this is covered by the $l_w M$ term in the overall cost (Equation (4.3)).

Finally, note that $\sum_{i=1}^{\log_D n} \frac{n}{D^i} = 8 + 4 + 2 + 1 = 15$ groups are merged, leading to 15 sequential (!) merge operations. △

Remark 4.9 (Rounding of $\log_D$*).* As opposed to the cost model used by Wang and Chan [163], the $\log_D$ factor is not rounded in the cost calculation above. The reason for this is to compensate for Hadoop's desire to achieve the merge factor in the last round [164]. This means that in the first round(s), some spill files may not be read, leading to an overestimate when rounding the log factor upwards. For a small number of spill files not rounding leads to more accurate results. When a large number of spill files is expected, rounding the result may be preferable. ◇

The total cost for an input partition $\mathcal{I}_i$ can now be calculated as follows:

$$\sum_j \mathcal{C}_{map}(N_i^j, M_i^j, \widehat{M_i^j}),$$

where j ranges over the input splits of $\mathcal{I}_i$. This can be further simplified:

$$\begin{aligned}
\mathcal{C}_{map}(N_i, M_i, \widehat{M_i}) &= \sum_j \mathcal{C}_{map}(N_i^j, M_i^j, \widehat{M_i^j}) \\
&= \sum_j \Big(h_r N_i^j + \text{merge}_{map}(M_i^j, \widehat{M_i^j}) + l_w(M_i^j + c_{\text{o}}\widehat{M_i^j})\Big) \\
&= \sum_j h_r N_i^j + \sum_j \text{merge}_{map}(M_i^j, \widehat{M_i^j}) + \sum_j l_w(M_i^j + c_{\text{o}}\widehat{M_i^j}) \\
&= h_r N_i + \sum_j \text{merge}_{map}(M_i^j, \widehat{M_i^j}) + l_w(M_i + c_{\text{o}}\widehat{M_i}).
\end{aligned}$$

As we assume that the map function behaves uniformly on $\mathcal{I}_i$ (and hence on all input splits of $\mathcal{I}_i$), we can simplify the merge cost function as follows:

$$\begin{aligned}
\text{merge}_{map}(M_i, \widehat{M_i}) &= \sum_j \text{merge}_{map}(M_i^j, \widehat{M_i^j}) \\
&= \sum_j \left((l_r + l_w) M_i^j \log_D \left\lceil \frac{M_i^j + c_{\text{b}}\widehat{M_i^j}}{buf_{\,map}} \right\rceil \right) \\
&= \sum_j \left((l_r + l_w) \frac{M_i}{m_i} \log_D \left\lceil \frac{(M_i + c_{\text{b}}\widehat{M_i})/m_i}{buf_{\,map}} \right\rceil \right) \qquad (Eq.\ 4.2) \\
&= m_i (l_r + l_w) \frac{M_i}{m_i} \log_D \left\lceil \frac{(M_i + c_{\text{b}}\widehat{M_i})/m_i}{buf_{\,map}} \right\rceil \\
&= (l_r + l_w) M_i \log_D \left\lceil \frac{(M_i + c_{\text{b}}\widehat{M_i})/m_i}{buf_{\,map}} \right\rceil,
\end{aligned}$$

where m_i equals the number of input splits for partition $\mathcal{I}_i$. This leads to a final map task cost calculation of

$$\begin{aligned}
\mathcal{C}_{map}(N_i, M_i, \widehat{M_i}) &= h_r N_i + \text{merge}_{map}(M_i, \widehat{M_i}) + l_w(M_i + c_{\text{o}}\widehat{M_i}), \text{ with} \\
\text{merge}_{map}(M_i, \widehat{M_i}) &= (l_r + l_w) M_i \log_D \left\lceil \frac{(M_i + c_{\text{b}}\widehat{M_i})/m_i}{buf_{\,map}} \right\rceil.
\end{aligned}$$

The total cost incurred in the map phase equals the sum

$$\sum_{i=1}^{k} \mathcal{C}_{map}(N_i, M_i, \widehat{M_i}). \qquad (4.5)$$

Note that the cost model in [139, 163] defines the total cost incurred in the map phase as

$$\mathcal{C}_{map}\left(\sum_{i=1}^{k} N_i, \sum_{i=1}^{k} M_i, \sum_{i=1}^{k} \widehat{M_i}\right). \tag{4.6}$$

The latter is not always accurate. Indeed, consider for instance an MR job whose input consists of two relations R and S where the map function outputs many key-value pairs for each tuple in R, while only one or zero key-value pairs for each tuple in S, e.g., because of filtering. This difference in map output may lead to a non-proportional contribution of both map functions to the total cost. Hence, as shown by Equation (4.5), we opt to consider different inputs separately. This cannot be captured by map cost calculation of Equation (4.6), as it considers the *global* average map output size in the calculation of the merge cost. In Section 5.5, we confirm the effectiveness of the proposed adjustment by means of an experiment. The following example illustrates the difference between the two approaches.

Example 4.10 (Cost model differences). For this example we use a split size of 100MB, a sort buffer of 100MB and a merge factor of 10. Suppose we have a dataset of 1100 records, each taking up 1MB, with the following map function: the first 100 of these records are duplicated 10 times and of the 1000 remaining records, 99% is removed. More concretely, let $\mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2)$ with $N_1 = 100, N_2 = 100$, $M_1 = 1000, M_2 = 10$ and $\widehat{M_1} = 1000, \widehat{M_2} = 10$. Then Equation (4.5) gives us a total map cost of approximately

$$\begin{aligned}&100h_r + 1000l_w + (l_r + l_w)1000 + 1000h_r + 10l_w\\ &= 1100h_r + 1010l_w + (l_r + l_w)1000,\end{aligned}$$

while Equation (4.6) only yields $1100h_r + 1010l_w$. Clearly, the averaging of the map output in the second case fails to account for the merge cost, which is rather significant in this case. Admittedly, this is an extreme example, but even in more common scenario's this may lead to significant errors in the cost estimations. △

Reduce Phase

Recall that the reduce stage involves (i) transferring the intermediate data (i.e., the output of the map function) to the correct reducer, (ii) merging the key-value pairs locally for each reducer, (iii) applying the reduce function, and (iv) writing the output to HDFS.

In order to analyze the cost in the reduce phase, let $\mathcal{J}$ denote the map output data, $R = \sum_{i=1}^{k} M_i$ denote the total reducer input size (in MB), K the total reducer output size (in MB). Furthermore, Let r be the number of reduce tasks that is created for the MR job and let $\mathcal{J}_1, \ldots, \mathcal{J}_r$ denote a partition of $\mathcal{J}$ such that the key-value pairs in $\mathcal{J}_i$ are processed by reduce task i. Also, let R_i denote the size of $\mathcal{J}_i$ (in MB) and K_i the number of output bytes (in MB).

The cost of a reduce task can now be calculated as follows:

$$\mathcal{C}_{red}(R_i, K_i) = tR_i + pmr + \text{merge}_{red}(R_i) + h_w K_i,$$

where the cost of merging equals

$$\text{merge}_{red}(R_i) \quad = (l_r + l_w) R_i \log_D \left\lceil \frac{R_i}{buf_{red}} \right\rceil.$$

The main cost consists of three parts. Transferring the input from the mappers to the reducer takes $tR_i + pmr$, merging the map outputs takes $\text{merge}_{red}(R_i)$, and writing the final output to HDFS takes $h_w K_i$. The cost for transferring has an extra term pmr, which is the *penalty* for a high number of mappers m and/or reducers r. We omit the cost for applying the reduce function, as its cost is generally not dominant. If desired, it can easily be incorporated into the existing constants or added as a separate term.

Note that there is a slight difference in the merge part when compared to the map phase. When n chunks need to merged using an external merge with factor D, the process is as follows. First, all data is written to disk. Next, $\log_D n$ levels are necessary to obtain a final sorted file on disk. In order to apply the reduce function on this data, the file is read one final time. Hence, all n chunks are written and read $\log_D n + 1$ times. Hadoop offers an optimization[33] that causes the final merge step to be skipped. The result of the merge is then fed directly to the reduce function, avoiding an additional read and write. Hence, when this optimization is active, all data is read and written $\log_D n$ times. In what follows, we assume the optimization is active. When this is not the case, an extra read cost can easily be added to the cost calculation by re-adding 1 to the log factor.

Remark 4.11 (Reduce-side merge skip). The optimization that skips the final merge step is not activated by default in Hadoop. The reason for this is that Hadoop assumes the reduce function needs all memory made available to the reduce task. However, most of the time this is not the case and enabling the optimization offers a significant increase in reducer performance. For the algorithms presented in Section 4.5, the reduce function has a low memory footprint and can therefore benefit from the optimization. $\diamond$

When the intermediate data is divided equally among the reduce tasks, i.e., $\forall i, j : R_i = R_j$, or $\forall i : M_i' = {}^{R}/_{m_i}$ the total cost for the reduce phase

[33]**Hadoop setting:** `mapreduce.reduce.merge.inmem.threshold`

becomes:

$$\begin{aligned}
\mathcal{C}_{red}(R,K) &= \sum_i \mathcal{C}_{red}(R_i,K_i)\\
&= \sum_i \left(tR_i + \text{merge}_{red}(R_i) + h_w K_i\right)\\
&= \sum_i tR_i + \sum_i \text{merge}_{red}(R_i) + \sum_i h_w K_i\\
&= tR + \text{merge}_{red}(R) + h_w K,
\end{aligned}$$

with

$$\begin{aligned}
\text{merge}_{red}(R) &= \sum_i \text{merge}_{red}(R_i)\\
&= \sum_i \left((l_r + l_w) R_i \log_D \left\lceil \frac{R_i}{buf_{red}} \right\rceil\right)\\
&= \sum_i \left((l_r + l_w) \frac{R}{r} \log_D \left\lceil \frac{R/r}{buf_{red}} \right\rceil\right)\\
&= r(l_r + l_w) \frac{R}{r} \log_D \left\lceil \frac{R/r}{buf_{red}} \right\rceil\\
&= (l_r + l_w) \frac{R}{r} \log_D \left\lceil \frac{R/r}{buf_{red}} \right\rceil.
\end{aligned}$$

Hence, the total cost incurred in the reduce phase equals

$$\begin{aligned}
\mathcal{C}_{red}(R,K) &= tR + \text{merge}_{red}(R) + h_w K, \text{ with}\\
\text{merge}_{red}(R) &= (l_r + l_w) R \log_D \left\lceil \frac{R/r}{buf_{red}} \right\rceil.
\end{aligned}$$

Total Cost

The total cost of an MR job can be approximated by

$$\mathcal{C}_{head} + \sum_{i=1}^{k} \mathcal{C}_{map}(N_i, M_i, \widehat{M_i}) + \mathcal{C}_{red}(\sum_{i=1}^{k} M_i, K),$$

where $\mathcal{C}_{head}$ is the overhead cost of starting an MR job.

We can briefly summarize the differences with the cost model in [163] as follows. First, we explicitly incorporate the meta-data $\widehat{M_i}$ into the cost merge$_{map}$. Second, we characterize the reduce function in a slightly different way. Third, we offer a way of penalizing a large number of mappers and/or reducers, which may cause significant overhead during the transfer stage. Fourth, we incorporate the cost of writing the output of reduce phase, which in [163] is omitted as the focus is one-round MR jobs. When estimating the cost of MR programs that consists of a sequence of jobs this cost can be significant as different MR jobs may yield a different amount of data passed from one round to the next. Furthermore, this data is written to HDFS, which incurs a higher cost than writing to the local file system, or reading from HDFS. Finally, we offer a partition of map input to account for non-uniform applications of the map function. This can greatly affect the total cost in certain situations as was demonstrated in Example 4.8.

4.5 Parallel MSJ and SGF Evaluation

In this section, we describe how SGF queries can be evaluated using MapReduce (MR). We start by introducing some necessary building blocks in Section 4.5.1 to 4.5.3, and describe the evaluation of BSGF queries and multiple BSGF queries in Sections 4.5.4 and 4.5.5, respectively. These are then generalized to the full fragment of SGF queries in Sections 4.5.6 and 4.5.7.

First, we introduce some additional notation. We say that a tuple$\bar{a} = (a_1, \ldots, a_n) \in \mathbf{D}^n$ of n data values *conforms* to a vector $\bar{t} = (t_1, \ldots, t_n)$ of terms, if

1. $\forall i, j \in [1, n]$, $t_i = t_j$ implies $a_i = a_j$; and,
2. $\forall i \in [1, n]$ if $t_i \in \mathbf{D}$, then $t_i = a_i$.

For instance, $(1, 2, 1, 3)$ conforms to $(x, 2, x, y)$. Likewise, a fact $T(\bar{a})$ conforms to an atom $U(\bar{t})$ if $T = U$ and $\bar{a}$ conforms to $\bar{t}$. We write $T(\bar{a}) \models U(\bar{t})$ to denote that $T(\bar{a})$ conforms to $U(\bar{t})$. If $f = R(\bar{a})$ is a fact conforming to an atom $\alpha = R(\bar{t})$ and $\bar{x}$ is a sequence of variables that occur in $\bar{t}$, then the projection $\pi_{\alpha;\bar{x}}(f)$ of f onto $\bar{x}$ is the tuple $\bar{b}$ obtained by projecting $\bar{a}$ on the coordinates in $\bar{x}$. For instance, let $f = R(1, 2, 1, 3)$ and $\alpha = R(x, y, x, z)$. Then, $R(1, 2, 1, 3) \models R(x, y, x, z)$ and hence $\pi_{\alpha;x,z}(f) = (1, 3)$.

4.5.1 Evaluating One Semi-join

As a warm-up, let us explain how single semi-joins can be evaluated in MR. A single semi-join is a query of the form

$$Z := \texttt{SELECT } \bar{w} \texttt{ FROM } \alpha \texttt{ WHERE } \kappa; \tag{4.7}$$

where both α and κ are atoms. For notational convenience, we will denote this query simply by $\pi_{\bar{w}}(\alpha \ltimes \kappa)$.

Algorithm 8 SJ $\big(\pi_{\bar{w}}\big(\alpha \ltimes \kappa\big)\big)$ – Single semi-join calculation in MapReduce.

```
1: function MAP(fact $f$)
2:     if $f \models \alpha$ then
3:         emit $\langle \pi_{\alpha;\bar{z}}(f) : [\textsc{Req}\,\kappa; \textsc{Out}\,\pi_{\alpha;\bar{w}}(f)]\rangle$
4:     if $f \models \kappa$ then
5:         emit $\langle \pi_{\kappa;\bar{z}}(f) : [\textsc{Assert}\,\kappa]\rangle$

6: function REDUCE($\langle k : V\rangle$)
7:     if $V$ contains [ASSERT $\kappa$] then
8:         for all [REQ $\kappa$; OUT $\bar{a}$] in $V$ do
9:             output $\bar{a}$
```

To evaluate Query (4.7), one can use the following one-round repartition join [46], which is depicted in Algorithm 8. The mapper distinguishes between guard facts (i.e., facts in DB conforming to α) and conditional facts (i.e., facts in DB conforming to κ). Specifically, let $\bar{z}$ be the join key, i.e., those variables occurring in both α and κ. For each guard fact f such that $f \models \alpha$, the mapper emits the key-value pair

$$\langle \pi_{\alpha;\bar{z}}(f) : [\textsc{Req}\,\kappa; \textsc{Out}\,\pi_{\alpha;\bar{w}}(f)]\rangle \,.$$

Intuitively, this pair is a request message sent by guard fact f to request whether a conditional fact $g \models \kappa$ with $\pi_{\kappa;\bar{z}}(g) = \pi_{\alpha;\bar{z}}(f)$ exists in the database, stating that if such a conditional fact exists, the tuple $\pi_{\alpha;\bar{w}}(f)$ should be output. Conversely, for each conditional fact $g \models \kappa$, the mapper emits a message of the form

$$\langle \pi_{\kappa;\bar{z}}(g) : [\textsc{Assert}\,\kappa]\rangle \,,$$

asserting the existence of a κ-conforming fact in the database with join key $\pi_{\kappa;\bar{z}}(g)$. On input $\big\langle \bar{b} : V\big\rangle$, the reducer outputs all tuples $\bar{a}$ to relation Z for which $[\textsc{Req}\,\kappa; \textsc{Out}\,\bar{a}] \in V$, provided that V contains at least one assert message. The final MR program is shown in Algorithm 8.

Example 4.12 (Single semi-join in MapReduce). Consider the query

$$Z \coloneqq \pi_x(R(x,z) \ltimes S(z,y))$$

and let I contain the facts $\{R(1,2), R(4,5), S(2,3)\}$. Then the mapper emits the key-value pairs

$$\begin{aligned}&\langle 2 : [\textsc{Req}\,S(z,y); \textsc{Out}\,1]\rangle\,;\\&\langle 5 : [\textsc{Req}\,S(z,y); \textsc{Out}\,4]\rangle\,;\ \text{and,}\\&\langle 2 : [\textsc{Assert}\,S(z,y)]\rangle\,;\end{aligned}$$

which after reshuffling result in the groups

$$\begin{aligned}&\langle 5 : \{[\textsc{Req}\, S(z,y); \textsc{Out}\, 4]\}\rangle \text{ ; and,}\\&\langle 2 : \{[\textsc{Req}\, S(z,y); \textsc{Out}\, 1], [\textsc{Assert}\, S(z,y)]\}\rangle \,.\end{aligned}$$

Only the reducer processing the second group produces an output, namely the fact $Z(1)$. △

The algorithm outlined above can be improved in several aspects. First, notice that both guard and conditional tuples are sent over the network exactly once. It is possible for multiple messages, originating from different facts, to share the same key and sometimes even the same value. When this is the case, a combiner can be used for packing messages together and reducing the map output. Secondly, the reducer for this algorithm needs to store all messages in its memory before processing can begin. It can be transformed into a constant memory (*streaming*) reducer by ensuring that assert messages always appear first. These optimizations are called packing and streaming reducers, respectively, and are discussed more in-depth in Section 5.3.4.

Cost analysis. To compare the cost of separate and combined evaluation of multiple semi-joins in the next section, we first illustrate how to analyze the cost of evaluating a single semi-join using the cost model described in Section 4.4. Hereto, let $|\alpha|$ and $|\kappa|$ denote the total size of all facts that conform to α and κ, respectively. Five values are required for estimating the total cost: N_1, N_2, M_1, M_2 and K. We can now choose $M_1 = |\alpha|$ and $M_2 = |\kappa|$. For simplicity, we assume that key-value pairs output by the mapper have the same size as their corresponding input tuples, i.e., $N_1 = M_1$ and $N_2 = M_2$.[34] Finally, the output size K can be approximated by its upper bound N_1.

4.5.2 Evaluating a Collection of Semi-joins

Since a BSGF query is essentially a Boolean combination of semi-joins, it can be computed by first evaluating all semi-joins followed by the evaluation of the Boolean combination. In the present section, we introduce a single-job MR program MSJ that evaluates a set of semi-joins in parallel using only one MR job. In the next section we introduce the single-job MR program EVAL to evaluate the Boolean combination.

We introduce a unary multi-semi-join operator $\ltimes\!\cdot(\mathcal{S})$ that takes as input a set of expressions $\mathcal{S} = \{X_1 := \pi_{\bar{x}_1}(\alpha_1 \ltimes \kappa_1), \ldots, X_n := \pi_{\bar{x}_n}(\alpha_n \ltimes \kappa_n)\}$. It is required that the X_i are all pairwise distinct and that they do not occur

[34] It is important to note that Gumbo, the system used in our experiments, does not make this simplifying assumption, but uses sampling to estimate M_1 and M_2 (see Section 5.4.2).

in any of the right-hand sides. The semantics are straightforward: the operator computes every semi-join $\pi_{\bar{x}_i}(\alpha_i \ltimes \kappa_i)$ in $\mathcal{S}$ and stores the result in the corresponding output relation X_i.

We now expand the MR job described in Section 4.5.1 into a job that computes $\ltimes\cdot(\mathcal{S})$ by evaluating all semi-joins in parallel. Let $\bar{z}_i$ be the join key of semi-join $\pi_{\bar{x}_i}(\alpha_i \ltimes \kappa_i)$. Algorithm 9 shows the single MR job MSJ($\mathcal{S}$) that evaluates all n semi-joins at once. More specifically, MSJ simulates the repartition join [46] of Algorithm 8, but outputs *all* request messages for the guard facts at once (i.e., those facts conforming to one of the α_i for $i \in [1, n]$). Similarly, *all* assert messages are generated simultaneously for the conditional facts (i.e., those facts conforming to one of the κ_i for $i \in [1, n]$). The reducer then reconciles the messages concerning the same κ_i. That is, on input $\langle \bar{b} : V \rangle$, the reducer outputs the tuple $\bar{a}$ to relation X_i for which $[\textsc{Req}\,(\kappa, i); \textsc{Out}\,\bar{a}] \in V$, provided that V contains at least one assert message of the form $[\textsc{Assert}\,\kappa]$. The output therefore consists of the relations $X_1, \ldots, X_n$, with each X_i containing the result of evaluating $\pi_{\bar{x}_i}(\alpha_i \ltimes \kappa_i)$. Recall that the MapReduce framework in Hadoop allows for writing output to different files from a single reduce task, which is necessary to obtain the desired behavior.

Combining the evaluation of a collection of semi-joins into a single MSJ job avoids the overhead of starting multiple jobs, reads every input relation only once and can reduce the amount of communication by packing similar messages together (see Section 5.3.2). At the same time, grouping all semi-joins together can potentially increase the average load of map and/or reduce tasks, which directly leads to an increased net time. These trade-offs are made more apparent in the cost analysis below and are taken into account in the algorithm Greedy-BSGF introduced in Section 4.5.4.

As was the case for the single semi-join MR program, this algorithm can also benefit from streaming optimizations (see Section 5.3), next to the packing optimization mentioned above.

Cost analysis. We consider the scenario where one relation is semi-joined with a set of relations, i.e., κ_i's are all different atoms, but $\alpha_1 = \cdots = \alpha_n = \alpha$. This scenario will be useful for our later analysis. Similar calculations can be performed for other scenarios. As before, to avoid clutter, we assume that the size of the key-value pair is the same as the size of the conforming input fact, and that every input tuple in the relation of α conforms to α, and every input tuple in the relation of κ_i conforms to κ_i. Then, for a set of input expressions

Algorithm 9 $\mathrm{MSJ}(X_1 := \pi_{\bar{x}_1}(\alpha_1 \ltimes \kappa_1), \ldots, X_n := \pi_{\bar{x}_n}(\alpha_n \ltimes \kappa_n))$ Multi-semijoin calculation in MapReduce.

```
1: function Map(fact $f$)
2:     buff = [ ]
3:     for every $i$ such that $f \models \alpha_i$ do
4:         add $\langle \pi_{\alpha_i;\bar{z}_i}(f) : [\textsc{Req}\,(\kappa_i, i); \textsc{Out}\,\pi_{\alpha_i;\bar{x}_i}(f)]\rangle$ to buffer
5:     for every $i$ such that $f \models \kappa_i$ do
6:         add $\langle \pi_{\kappa_i;\bar{z}_i}(f) : [\textsc{Assert}\,\kappa_i]\rangle$ to buffer
7:     emit buffer

8: function Reduce($\langle k : V\rangle$)
9:     for all [Req $\kappa_i$; Out $\bar{a}$] in $V$ do
10:        if $V$ contains [Assert $\kappa_i$] then
11:            output $\bar{a}$ to $X_i$
```

$\mathcal{S}$, the cost of $\mathrm{MSJ}(\mathcal{S})$, denoted by $\mathcal{C}(\mathrm{MSJ}(\mathcal{S}))$, equals

$$\begin{aligned}&\mathcal{C}_{head} + \mathcal{C}_{map}(|\alpha|, n|\alpha|, n\|\alpha\|) + \sum_{i=1}^{n} \mathcal{C}_{map}(|\kappa_i|, |\kappa_i|, \|\kappa_i\|)\\ &+ \mathcal{C}_{red}\Big(n|\alpha| + \sum_{i=1}^{n} |\kappa_i|, \sum_{i=1}^{n} |X_i|\Big),\end{aligned} \tag{4.8}$$

where $|X_i|$ equals the size of the output relation X_i in MB and $\|X_i\|$ equals the number of records in X_i. If we evaluate each X_i in a separate MR job, the total cost equals

$$\sum_{i=1}^{n} \begin{pmatrix} \mathcal{C}_{head} + \mathcal{C}_{map}(|\alpha|, |\alpha|, \|\alpha\|) + \mathcal{C}_{map}(|\kappa_i|, |\kappa_i|, \|\kappa_i\|) \\ + \mathcal{C}_{red}(|\alpha| + |\kappa_i|, |X_i|) \end{pmatrix}. \tag{4.9}$$

So evaluating all X_i's in one MR job is more efficient than evaluating each X_i in a separate MR job if and only if the resulting cost obtained using Equation (4.8) is less than that of Equation (4.9).

4.5.3 Evaluating Boolean Combinations

Let $X_0, X_1, \ldots X_n$ be relations with the same arity and let φ be a Boolean formula over $X_1, \ldots X_n$. Algorithm 10 depicts the straightforward MR program that evaluates $X_0 \wedge \varphi$ in a single job: on each fact $X_i(\bar{a})$, the mapper emits $\langle \bar{a} : i\rangle$. The reducer hence receives pairs $\langle \bar{a} : V\rangle$ with V containing all the indices i for which $\bar{a} \in X_i$, and outputs $\bar{a}$ only if the Boolean formula, obtained from $X_0 \wedge \varphi$ by replacing every X_i with `True` if $i \in V$ and `False` otherwise, evaluates to `True`. For instance, if $\varphi = X_1 \wedge X_2 \wedge \neg X_3$, it will emit $\bar{a}$ only if V contains 0, 1 and 2 but not 3.

We denote this MR job as EVAL(X_0, φ). We emphasize that multiple Boolean formulas $Y_1 \wedge \varphi_1, \ldots, Y_n \wedge \varphi_n$ with distinct sets of variables can be evaluated in one MR job which we denote as EVAL$(Y_1, \varphi_1, \ldots, Y_n, \varphi_n)$. The resulting algorithm can be obtained from Algorithm 10 by adding an iteration over the formula's $\varphi_1, \ldots, \varphi_n$ in the reducer. It should be clear that atoms that appear multiple times, possibly in different formulas, generate at most one message per input fact, as opposed to multiple when the formulas are evaluated by separate MR jobs.

Algorithm 10 EVAL(φ) – MapReduce program for evaluating one Boolean combination.

```
Require: $\varphi = BC(X_1, \ldots, X_n)$

 1: function MAP(FACT $f$)
 2:     for all $X_i$ do
 3:         if $f \models X_i$ then
 4:             emit $\langle f : i \rangle$

 5: function REDUCE($\langle k : V \rangle$)
 6:     for all $X_i$ do
 7:         if $i \in V$ then
 8:             $X_i \leftarrow$ True
 9:         else
10:             $X_i \leftarrow$ False
11:     if $eval(\varphi)$ then
12:         output $k$
```

Cost analysis. Let $|X_i|$ be the size of relation X_i in MB and $\|X_i\|$ be the number of records in X_i. The cost of an EVAL job with input relations $X_1, \ldots, X_n$ can then be approximated as follows:

$$\begin{aligned} \mathcal{C}(\text{EVAL}(\varphi)) &= \mathcal{C}_{head} + \sum_{i=0}^{n} \mathcal{C}_{map}(|X_i|, |X_i|, \|X_i\|) \\ &\quad + \mathcal{C}_{red}\Big(\sum_{i=0}^{n} |X_i|, |\varphi|\Big), \end{aligned} \tag{4.10}$$

where $|\varphi|$ is the size of the output.

4.5.4 Evaluating BSGF Queries

We now have the building blocks to discuss the evaluation of basic strictly guarded fragment (BSGF) queries. Consider the following BSGF query Q:

$$Z := \texttt{SELECT } \bar{w} \texttt{ FROM } R(\bar{t}) \texttt{ WHERE } C.$$

Here, C is a Boolean combination of conditional atoms κ_i, for $i \in [1, n]$, that can only share variables occurring in $\bar{t}$. Note that it is implicit that $\kappa_1, \dots, \kappa_n$ are all different atoms. Furthermore, let $\mathcal{S}$ be the set of equations $\{X_1 := \pi_{\bar{w}}\big(R(\bar{t}) \ltimes \kappa_1\big), \dots, X_n := \pi_{\bar{w}}\big(R(\bar{t}) \ltimes \kappa_n\big)\}$ and let φ_C be the Boolean formula obtained from C by replacing every conditional atom κ_i by X_i. Then, for every partition $\{\mathcal{S}_1, \dots, \mathcal{S}_p\}$ of $\mathcal{S}$, the following MR program computes Q:

$$\begin{tikzcd}
 & \boxed{\text{EVAL}(R, \varphi_C)} \arrow[dl, dash] \arrow[dr, dash] & \\
\boxed{\text{MSJ}(\mathcal{S}_1)} & \dots & \boxed{\text{MSJ}(\mathcal{S}_p)}
\end{tikzcd}$$

We refer to any such program as a *basic MR program for Q*. Notice that all MSJ jobs can be executed in parallel. So, the above program consists in fact of two rounds, but note that there are $p + 1$ MR jobs in total: one for each $\text{MSJ}(\mathcal{S}_i)$, and one for $\text{EVAL}(\varphi_C)$.

Example 4.13 (Alternative query plans). Figure 4.5 shows three alternative basic MR programs for the following query:

$$\begin{aligned} Z \quad := \quad & \texttt{SELECT } x, y \texttt{ FROM } R(x, y) \\ & \texttt{WHERE } S(x, z) \texttt{ AND } (T(y) \texttt{ OR NOT } U(x)) \end{aligned} \tag{4.11}$$

In alternative (a), all semi-joins X_1, X_2, X_3 are evaluated as separate MR jobs. In alternative (b), X_1 and X_3 are computed in one MR job, while X_2 is computed separately. In alternative (c), all semi-joins X_1, X_2, X_3 are computed in a single MR job. △

Cost analysis. When $\mathcal{S}$ is partitioned into $\mathcal{S}_1 \cup \dots \cup \mathcal{S}_p$, the cost of the MR program is:

$$\mathcal{C}(\text{EVAL}(R, \varphi_C)) \quad + \quad \sum_{i=1}^{p} \mathcal{C}(\text{MSJ}(\mathcal{S}_i)), \tag{4.12}$$

where $\mathcal{C}(\text{MSJ}(\mathcal{S}_i))$ is as in Equation (4.8).

Computing the optimal partition. By BSGF-Opt we denote the problem that takes a BSGF query Q as above and computes a partition $\mathcal{S}_1 \cup \dots \cup \mathcal{S}_p$ of $\mathcal{S}$

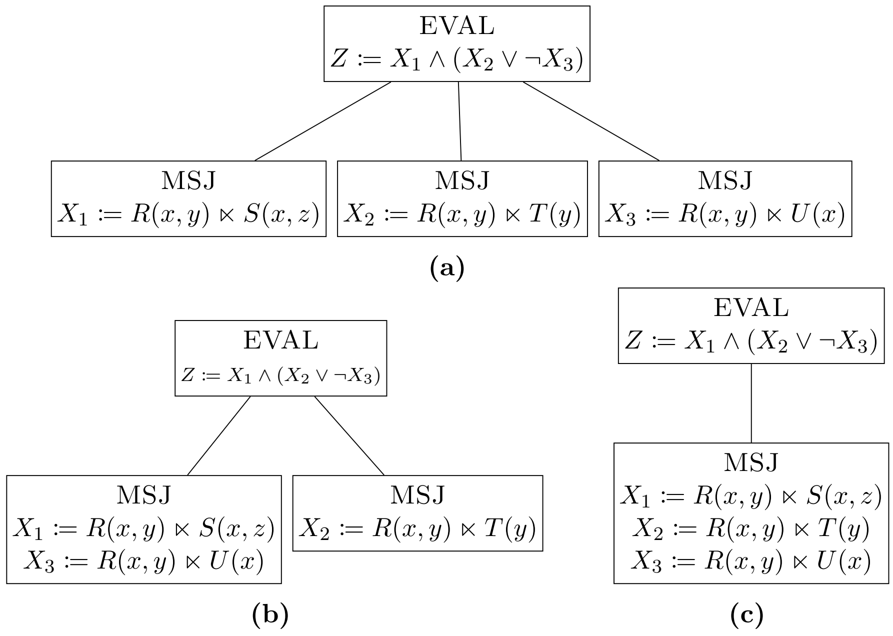

EVAL
$Z := X_1 \wedge (X_2 \vee \neg X_3)$
MSJ
$X_1 := R(x,y) \ltimes S(x,z)$
MSJ
$X_2 := R(x,y) \ltimes T(y)$
MSJ
$X_3 := R(x,y) \ltimes U(x)$
(a)
EVAL
$Z := X_1 \wedge (X_2 \vee \neg X_3)$
MSJ
$X_1 := R(x,y) \ltimes S(x,z)$
$X_3 := R(x,y) \ltimes U(x)$
MSJ
$X_2 := R(x,y) \ltimes T(y)$
(b)
EVAL
$Z := X_1 \wedge (X_2 \vee \neg X_3)$
MSJ
$X_1 := R(x,y) \ltimes S(x,z)$
$X_2 := R(x,y) \ltimes T(y)$
$X_3 := R(x,y) \ltimes U(x)$
(c)

Figure 4.5: *Some MapReduce query plan alternatives for the query given in Example 4.13.*

such that its total cost as computed in Equation (4.12) is *minimal*. The Scan-Shared Optimal Grouping, which is known to be NP-hard, is reducible to this problem (see Nykiel et al. [139]). Stating it formally, we have:

Theorem 4.14. *The decision variant of* BSGF-Opt *is* NP*-complete.*

Greedy heuristic. While for small queries the optimal solution can be found using a brute-force search, for larger queries we adopt the fast greedy heuristic introduced by Wang et al. [163]. For two disjoint subsets $\mathcal{S}_i, \mathcal{S}_j \subseteq \mathcal{S}$, define:

$$\text{gain}(\mathcal{S}_i, \mathcal{S}_j) = \mathcal{C}(\mathcal{S}_i) + \mathcal{C}(\mathcal{S}_j) - \mathcal{C}(\mathcal{S}_i \cup \mathcal{S}_j).$$

That is, $\text{gain}(\mathcal{S}_i, \mathcal{S}_j)$ denotes the reduction in cost obtained by evaluating $\mathcal{S}_i \cup \mathcal{S}_j$ in one MR job rather than evaluating each of them separately. For a partition $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_p$, our heuristic algorithm greedily finds a pair $i, j \in [1,p] \times [1,p]$ such that $i \neq j$ and $\text{gain}(\mathcal{S}_i, \mathcal{S}_j) > 0$ is the greatest. If there is such a pair i, j, we merge $\mathcal{S}_i$ and $\mathcal{S}_j$ into one set. We start with the trivial partition $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n$, where each $\mathcal{S}_i = \{X_i := \pi_{\bar{w}}\big(R(\bar{t}) \ltimes \kappa_i\big)\}$ and repeat this procedure until there is no pair i, j for which $\text{gain}(\mathcal{S}_i, \mathcal{S}_j) > 0$. We refer to this algorithm as Greedy-BSGF. For a BSGF query Q, we denote by $\text{OPT}(Q)$ the optimal (least cost) basic MR program for Q, and by $\text{GOPT}(Q)$ we denote the program computed by Greedy-BSGF.

4.5.5 Evaluating Multiple BSGF Queries

The approach presented in the previous section can be readily adapted to evaluate multiple BSGF queries. Indeed, consider a set of n BSGF queries, each of the form

$$Z_i := \texttt{SELECT } \bar{w}_i \texttt{ FROM } R_i(\bar{t}_i) \texttt{ WHERE } C_i$$

where none of the C_i can refer to any of the Z_j. A corresponding MR program is then of the following form:

$$\begin{tikzcd}[column sep=small] & \boxed{\mathrm{EVAL}(R_1, \varphi_{C_1}, \ldots, R_n, \varphi_{C_n})} \arrow[dl, dash] \arrow[dr, dash] & \\ \boxed{\mathrm{MSJ}(\mathcal{S}_1)} & \ldots & \boxed{\mathrm{MSJ}(\mathcal{S}_p)} \end{tikzcd}$$

The EVAL job is as described in Section 4.5.3. The child nodes constitute a partition of all the necessary semi-joins. Again, φ_{C_i} is the Boolean formula obtained from C_i. We assume that the set of variables used in the Boolean formulas are disjoint. For a set of BSGF queries F, we refer to any MR program of the above form as a *basic MR program for F*, whose cost can be computed in a similar manner as above. The optimal basic MR program for F and the program computed by the greedy algorithm of Section 4.5.4 are denoted by $\mathrm{OPT}(F)$ and $\mathrm{GOPT}(F)$, respectively, and their costs are denoted by $\mathcal{C}(\mathrm{OPT}(F))$ and $\mathcal{C}(\mathrm{GOPT}(F))$.

Remark 4.15 (Parallel EVAL *jobs).* While it is technically possible to execute parallel eval jobs, we choose not to focus on this optimization as (i) the round 1 reducers would need to output to multiple files, causing more overhead and smaller files, and (ii) the round 2 mappers would need to read in multiple smaller files, potentially causing the load of a map task to drop. Note that this reasoning assumes that Hadoop is used for MapReduce evaluation and other systems may not exhibit these disadvantages. Hence, parallelization of the EVAL job is worthwile to consider it as possible future work. $\diamond$

4.5.6 Evaluating SGF Queries

Next, we turn to the evaluation of SGF queries. Recall that an SGF query Q is a sequence of basic queries of the form:

$$Z_1 := \xi_1; \ldots; Z_n := \xi_n; \qquad (\ddagger)$$

where each ξ_i can refer to the relations Z_j with $j < i$. We denote the BSGF $Z_i := \xi_i$ by Q_i. The most naive way to compute Q is to evaluate the BSGF queries in Q sequentially, where each ξ_i is evaluated using the approach detailed in the previous section. This leads to a $2n$-round MR program. We would like to have a strategy that aims at decreasing the total time by allowing parallel evaluation and by combining the evaluation of different independent subqueries.

To this end, let $\mathcal{G}_Q$ be the dependency graph induced by Q. That is, $\mathcal{G}_Q$ consists of n nodes (one for each BSGF query) and there is an edge from Q_i to Q_j if relation Z_i is mentioned in ξ_j. A *multiway topological sort* (MTS) of the dependency graph $\mathcal{G}_Q$ is a sequence $(F_1, \dots, F_k)$ such that

1. $\{F_1, \dots, F_k\}$ is a partition of the nodes in $\mathcal{G}_Q$;
2. if there is an edge from node u to node v in $\mathcal{G}_Q$, then $u \in F_i$ and $v \in F_j$ such that $i < j$.

Notice that any multiway topological sort $(F_1, \dots, F_k)$ of $\mathcal{G}_Q$ provides a valid ordering to evaluate Q, i.e., all the queries in F_i are evaluated before F_j whenever $i < j$.

Example 4.16. Let us illustrate the latter by means of an example. Consider the following SGF query Q:

$$\begin{aligned}
Q_1: \quad & Z_1 & := \quad & \texttt{SELECT } x, y \texttt{ FROM } R_1(x, y) \texttt{ WHERE } S(x) \\
Q_2: \quad & Z_2 & := \quad & \texttt{SELECT } x, y \texttt{ FROM } Z_1(x, y) \texttt{ WHERE } T(x) \\
Q_3: \quad & Z_3 & := \quad & \texttt{SELECT } x, y \texttt{ FROM } Z_2(x, y) \texttt{ WHERE } U(x) \\
Q_4: \quad & Z_4 & := \quad & \texttt{SELECT } x, y \texttt{ FROM } R_2(x, y) \texttt{ WHERE } T(x) \\
Q_5: \quad & Z_5 & := \quad & \texttt{SELECT } x, y \texttt{ FROM } Z_3(x, y) \texttt{ WHERE } Z_4(x, x)
\end{aligned}$$

The dependency graph $\mathcal{G}_Q$ is as follows:

$$\begin{tikzcd}
& Q_3 \arrow[dl] & Q_2 \arrow[l] & Q_1 \arrow[l] \\
Q_5 & & & \\
& Q_4 \arrow[ul] & &
\end{tikzcd}$$

There are four possible multiway topological sorts of $\mathcal{G}_Q$:
1. $(\{Q_1, Q_4\},\ \{Q_2\},\ \{Q_3\},\ \{Q_5\})$,
2. $(\{Q_1\},\ \{Q_2, Q_4\},\ \{Q_3\},\ \{Q_5\})$,
3. $(\{Q_1\},\ \{Q_2\},\ \{Q_3, Q_4\},\ \{Q_5\})$, and
4. $(\{Q_1\},\ \{Q_2\},\ \{Q_3\},\ \{Q_4\}, \{Q_5\})$. $\triangle$

Let $\mathcal{F} = (F_1, \dots, F_k)$ be a topological sort of $\mathcal{G}_Q$. Since the optimal program $\text{OPT}(F_i)$, defined in Section 4.5.5, is intractable (due to Theorem 4.14), we will use the greedy approach to evaluate F_i, i.e., $\mathrm{GOPT}(F_i)$ as defined in Section 4.5.5. The cost of evaluating Q according to $\mathcal{F}$ is

$$\mathcal{C}(\mathcal{F}) = \sum_{i=1}^{k} \mathcal{C}(\mathrm{GOPT}(F_i)) \tag{4.13}$$

We define the optimization problem SGF-Opt that takes as input an SGF query Q and constructs a multiway topological sort $\mathcal{F}$ of $\mathcal{G}_Q$ with minimal $\mathcal{C}(\mathcal{F})$. Unfortunately, SGF-Opt is also hard (proof is given in Section 4.6.1):

Theorem 4.17. *The decision variant of* SGF-Opt *is* NP*-complete.*

Greedy heuristic. In the following, we present a novel heuristic for computing a multiway topological sort of an SGF query that tries to maximize the overlap between queries. To this end, we define the *overlap* between a BSGF query Q and a set of BSGF queries F, denoted by *overlap*(Q, F), to be the number of relations occurring in Q that also occur in F. For instance, in Example 4.16, the overlap between Q_2 and $\{Q_1, Q_3, Q_4, Q_5\}$ is 1 as they share only relation T.

Consider the following algorithm Greedy-SGF that computes a multiway topological sort $\mathcal{F}$ of an SGF query Q. Initially, all the vertices in the dependency graph $\mathcal{G}_Q$ are colored blue and $\mathcal{X} = ()$. The algorithm repeatedly performs the iteration shown in Algorithm 11 with the invariant that $\mathcal{X}$ is a multiway topological sort of the red vertices in $\mathcal{G}$. The iteration stops when every vertex in $\mathcal{G}_Q$ is red, and hence, $\mathcal{X}$ is a multiway topological sort of $\mathcal{G}_Q$. Clearly, the number of iterations is n, where n is the number of vertices in $\mathcal{G}_Q$. Each iteration takes $O(n^2)$, which means that our heuristic algorithm above runs in $O(n^3)$ time.

As mentioned above, the resulting partition of BSGF queries can then be evaluated using the MapReduce plan obtained from GOPT. Note that a dynamic evaluation strategy may consists of re-running Greedy-SGF after each BSGF evaluation, in order to obtain an updated MR plan. While this naive approach of re-calculating the MR plan certainly works, an incremental approach that avoids redundant calculation costs is preferred.

Remark 4.18 (Future overlap). Note that it is possible for multiple red vertices to have maximal overlap. In this case, an arbitrary node can be chosen. A possible improvement consists of using an extra measure called *future overlap*, which is defined as follows. The future overlap of a vertex $v \in D$ equals *overlap*(v, D'), where D' contains all vertices v' of $\mathcal{G}_Q$ such that $v' \notin D$ and there is no path from v' to v in $\mathcal{G}_Q$. This corresponds to the overlap that exists with independent queries that will be scheduled at a later point. Intuitively,

Algorithm 11 One iteration of GREEDY-SGF.

1. Suppose $\mathcal{X} = (F_1, \ldots, F_m)$, and $\mathcal{G}_Q$ still contains blue vertices.
2. Let D be the set of those blue vertices in $\mathcal{G}_Q$ for which none of the incoming edges are from other blue vertices.

   (Due to the acyclicity of $\mathcal{G}_Q$, the set D is non-empty if $\mathcal{G}_Q$ still has blue vertices.)
3. Find a pair (u, F_i) such that $u \in D$, $(F_1, \ldots, F_i \cup \{u\}, \ldots, F_m)$ is a topological sort of the vertices $\{u\} \cup \bigcup F_i$, and *overlap*(u, F_i) is non-zero.
4. If such a pair (u, F_i) exists, we pick one with maximal *overlap*(u, F_i), and set $\mathcal{X} = (F_1, \ldots, F_i \cup \{u\}, \ldots, F_m)$.

   Otherwise, we set $\mathcal{X} = (F_1, \ldots, F_m, \{u\})$.
5. Color the vertex u red.

$$\begin{tikzcd}
\boxed{Z_2(t_2, a) := \text{Upcoming}(t_2, a) \ltimes \neg Z_1(a)} & \boxed{Y_2(t_4) := \text{Amaz}(t_4, a, r) \ltimes Y_1(a)} \\
\boxed{Z_1(a) := \text{Amaz}(t_1, a, \text{`bad'}) \ltimes \text{B\&N}(t_1, a, \text{`bad'}) \wedge \text{BD}(t_1, a, \text{`bad'})} \arrow[u] & \boxed{Y_1(a) := \text{Upcoming}(t_3, a) \ltimes \text{Movie}(t_3)} \arrow[u]
\end{tikzcd}$$

Figure 4.6: *Example SGF queries.*

a larger future overlap indicates more overlap possibilities at a later stage. For this reason, among all queries that have maximal overlap, the ones with minimal future overlap are preferred. $\diamond$

4.5.7 Evaluating Multiple SGF Queries

Evaluating a collection of SGF queries can be done in the same way as evaluating one SGF query. Indeed, we can simply consider the union of all BSGF subqueries. Note that evaluating multiple subqueries in parallel allows for exploiting potential overlaps between the queries, potentially bringing down the overall total and/or net time.

Example 4.19 (Multiple SGF queries). Consider the two SGF queries in Figure 4.6. The first query is the same as the one in Example 4.3 and selects the titles of authors of upcoming books that don't have a bad rating on all websites. The second query selects the books of Amazon authors that have an upcoming book that is going to become a movie.

We can evaluate the queries all in parallel or one at a time (sequentially). But, as the queries have common input relations, we can treat them as one collection of BSGF queries and take advantage of the techniques above. One possible grouping of the BSGF queries is

$$(\{Z_1, Y_1\}, \{Z_2, Y_2\}), \tag{4.14}$$

which leads to groups that have no common input relations. Two other possibilities are

$$(\{Z_1\}, \{Z_2, Y_1\}, \{Y_2\}), \text{ or}$$

$$(\{Y_1\}, \{Z_1, Y_2\}, \{Z_2\}),$$

which both show overlap in the queries in their second group that can possibly benefit the total execution time. Note, however, that the net time will probably increase, as the dependencies between the queries require both of these plans to take $3 \times 2 = 6$ MapReduce rounds, as opposed to $2 \times 2 = 4$ rounds for the grouping in Equation (4.14). △

4.6 Decision Problems

In this section, we first show that the SGF-Opt problem is NP-complete. Next, for completeness, we focus on a related problem that involves minimizing the number of input reads, and show that this problem too is NP-complete.

4.6.1 SGF-Opt is NP-complete

To prove Theorem 4.17, we consider a more general problem. The same technique can then be used to prove the original theorem. We define the optimization problem Subset Cost-Opt that takes as input a set S and a cost function $w : 2^{|S|} \to \mathbb{N}$, and constructs a partition $\mathcal{S} = \{S_1, \ldots, S_n\}$ of S such that $\sum_{S_i \in \mathcal{S}} w(S_i)$ is minimized. The decision version of Subset Cost-Opt, denoted by Subset Cost, corresponds to deciding, for a given positive integer k, whether there exists a partition $\mathcal{S} = \{S_1, \ldots, S_n\}$ of S such that $\sum_i^n w(S_i) = k$. We now prove the NP-completeness of Subset Cost.

Theorem 4.20. *The* Subset Cost *problem is* NP-*complete.*

Proof. The problem is clearly in NP, as a given solution can be verified in polynomial time. We prove that this problem is NP-complete by a providing a polynomial time reduction from the Subset Sum problem, which consists of deciding, for given an integer k and a set of positive integers A, whether there exists a set $B \subseteq A$ such that $\sum_{b \in B} b = k$. This problem is NP-complete [86].

The reduction is as follows. Given an instance of the subset sum problem, i.e., a set of integers A and an integer k, we construct the following instance of SUBSET COST: The set of items S equals $A \cup \{\circ\}$, where $\circ$ is an element not present in A, and the cost function $w : 2^S \to \mathbb{S}$ is defined as

$$w(X) = \begin{cases} \gamma & \text{if } \circ \in X \\ \sum_{a \in X} a & \text{otherwise} \end{cases} \tag{4.15}$$

Here, γ is an arbitrary constant. Intuitively, w takes the sum of all items in a given set, except when the set contains the special item $\circ$. In that case, the value will be a fixed number γ.

In the remainder of this proof, we show that there exists a $B \subseteq A$ such that $\sum_{b \in B} b = k$ iff there exists a partition $\mathcal{S} = \{S_1, \dots, S_n\}$ of S such that $\sum_i^n w(S_i) = k + \gamma$.

Suppose there exists a partition $\mathcal{S} = \{S_1, \dots, S_n\}$ of S such that

$$\sum_i^n w(S_i) = k + \gamma.$$

Now, select the element of the partition that contains $\circ$, say $S_\circ$, and remove it from the partition to obtain a partition $\mathcal{S}'$ for $S \setminus S_\circ$. Now, let $B = \bigcup \mathcal{S}'$ and note that $B \subseteq A$. Clearly, the sum of the elements in B equals $k + \gamma - \gamma = k$, as the cost of the removed element equals γ by Equation (4.15). Hence, B is a k-cost subset of A.

Suppose there exists a $B \subseteq A$ such that $\sum_{b \in B} b = k$. Consider the partition $\mathcal{S} = \{B, C\}$ of S, where $C = (A \setminus B) \cup \{\circ\}$. As $w(B) = k$ and $w(C) = \gamma$, we have a total cost of $k + \gamma$. □

We conclude this section by providing a proof for Theorem 4.17. Consider the decision version of SGF-OPT, which we will denote by SGF: for a DAG of BSGF queries $\mathcal{G}_Q$ and a positive integer k, determine whether there exists a multiway topological sort $\mathcal{F}$ of $\mathcal{G}_Q$ with $\mathcal{C}(\mathcal{F}) = k$. We obtain the following:

Theorem 4.21. *The* SGF *problem is* NP*-complete.*

Proof. We can verify a solution for this problem in the following way: given a multiway topological sort $\mathcal{F}$ and a positive integer k, we can calculate its total cost using Equation (4.13), i.e.,

$$\mathcal{C}(\mathcal{F}) \;=\; \sum_{i=1}^{k} \mathcal{C}(\text{GOPT}(F_i)),$$

and check whether it equals k. As the greedy algorithm GREEDY-BSGF for GOPT runs in polynomial time, the total cost can be calculated in polynomial time as well and hence the problem is in NP.

We now provide a polynomial time reduction from Subset Sum to SGF. Given an instance of the Subset Sum problem consisting of a set of positive integers $A = \{a_1, \ldots, a_n\}$ and a positive integer k, we construct the following instance for SGF. Let $R_1, \ldots, R_n, R^\circ$ be a set of binary relations containing no tuples, a set of relations $S_1, \ldots, S_n$ be a set of binary relations where $|S_i| = a_i$, each tuple has a size of 1MB and every second field has value 0. Also, consider a set of BSGF queries $F = \{f_1, \ldots, f_n, f^\circ\}$, where f_i equals $R_i(x_i, y_i) \ltimes S_i(x_i, 1)$ and

$$f^\circ = R^\circ(x, 1) \ltimes R_1(x_1, y_1) \wedge \ldots \wedge R_n(x_n, y_n) \wedge S_1(x_1, 1) \wedge \ldots \wedge S_n(x_1, 1).$$

Notice that there are no dependencies between the queries. Finally, for the Hadoop system, we choose all I/O constants equal to 0, except $h_r = 1$.

Using Equation (4.12) and our cost model, we obtain that the cost of calculating each individual BSGF query f_i equals a_i:

$$\begin{aligned} \mathcal{C}(\text{GOPT}(\{f_i\})) &= \mathcal{C}(\text{EVAL}(R_i, \varphi_{C_i})) + \mathcal{C}(\{f_i\}) \\ &= 0 + \mathcal{C}_{map}(|f_i|, |f_i|) + \mathcal{C}_{red}(|f_i|, |f_i|) \\ &= 0 + h_r \cdot a_i + 0 \\ &= a_i. \end{aligned}$$

Note that the EVAL job has cost 0, as no output is provided by the MSJ job.

We also find that the cost of two jobs equals their individual sums, regardless whether or not they are grouped in the query plan provided by GOPT:

$$\mathcal{C}(\text{GOPT}(\{f_i, f_j\})) = a_i + a_j.$$

Finally, we note that GOPT will always group f_i with f° as all relations of f_i appear in f°. This leads to a cost of:

$$\mathcal{C}(\text{GOPT}(\{f_i, f^\circ\})) = \gamma,$$

where $\gamma = \sum_{i=1}^{n} a_i$.

Let $\mathcal{G}_F$ be the dependency graph that has nodes F and no edges (as there are no query dependencies). Now, there exists a multiway topological sort for $\mathcal{G}_F$ of cost $k + \gamma$ iff there exists a $B \subseteq A$ such that $\sum_{b \in B} b = k$.

Observe that the cost function behaves completely analogous to the one used in the proof of Theorem 4.20. Hence, the same reasoning can now be applied to complete the proof. □

4.6.2 Minimizing Reads

The main idea behind the Greedy-SGF algorithm is to avoid spreading queries with high overlap over different partitions in the multiway topological sort, as grouping them may have a beneficial effect on the total time. The problem SGF-Opt involves finding a partitioning of the query such that the total cost is minimized, where the total cost is estimated by the cost model we presented in Section 4.4. In this section, we consider the related problem of minimizing the number of input reads and show that this problem too is NP-hard.

Let Q be an SGF query consisting of BSGF queries $Q_1, \dots, Q_n$, the dependency graph induced by Q denoted by $\mathcal{G}_Q$ and $\mathcal{F} = (F_1, \dots, F_k)$ a multiway topological sort of $\mathcal{G}_Q$. As before, Q_i denotes a BSGF query $Z_i := \xi_i$. The *spread* of a BSGF query Q_i in a multiway topological sort $\mathcal{F}$ is defined as the sum of all spreads:

$$\mathit{spread}(Q_i, F) = |\{F_i \in \mathcal{F} \mid \exists Z_i \in F_i : Z_i \text{ occurs in } \xi_j\}|.$$

Intuitively, this is the number of partitions that read the output of Q_i. The spread of SGF query Q in a multiway topological sort $\mathcal{F}$ is then defined as follows:

$$\mathit{spread}(Q, F) = \sum_i \mathit{spread}(Q_i, F).$$

Recall that our BSGF evaluation strategy Greedy-BSGF aims to group queries with similar input relations into one MR job as this can improve overlap. Similarly, it makes sense to reduce the total spread as this reduces the number of distinct partitions a relation appears in. We define the optimization problem SPREAD-Opt that takes as input an SGF query Q and constructs a multiway topological sort $\mathcal{F}$ of $\mathcal{G}_Q$ that minimizes $\mathit{spread}(Q, \mathcal{F})$. Similar to SGF-Opt, SPREAD-Opt is also NP-hard, as the following theorem shows.

Theorem 4.22. *The decision variant of* SPREAD-Opt *is* NP*-hard.*

Proof. We provide a reduction from the 3-coloring problem [86]. Given an undirected graph $G = (V, E)$, we construct an SGF query Q_G such that G is 3-colorable if and only if there is an MTS $\mathcal{F}$ for Q_G whose spread is optimal, i.e., it equals $3|V| + 5|E| + 8$. For readability, instead of writing the query Q_G explicitly, we construct the dependency graph of Q_G. This graph can then be used to construct an actual query Q_G.

Let $v_1, \dots, v_n$ be the vertices in G. The vertices in the dependency graph of φ_G correspond to the following subqueries:

- one bottom-level subquery Q_0;
- one top-level subquery Q_T;
- two subqueries Q_U and Q_V;
- subqueries Q_R, Q_G, Q_B which correspond to the colors red (R), green (G) and blue (B), respectively;

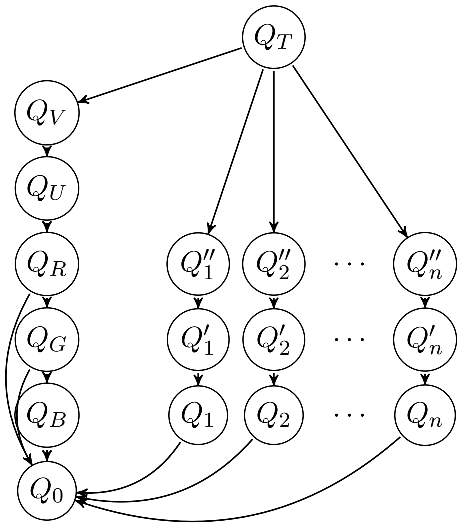


Figure 4.7: *Query dependencies introduced to ensure that every vertex is associated with exactly one color.*

- three subqueries Q_i, Q'_i, Q''_i for each vertex v_i.

The dependencies among these queries are depicted in Figure 4.7. This construction guarantees that each vertex is assigned exactly one color. In order to minimize the spread, each Q_i must appear in the same partition F_i as either Q_G, Q_B, or Q_R. Indeed, Q_i cannot appear together with Q_0 because of the dependency, and will not appear together with Q_U or Q_V because it would increase the spread. Indeed, Q_1 can be in the same group as Q_R, Q_G or Q_B without causing a spread increase, as these nodes all depend on Q_0; this is not the case for nodes Q_U and Q_V. Intuitively, the co-appearance of Q_i with either Q_R, Q_G, or Q_B represents the color assignment of vertex v_i. Hence, as desired, exactly one color is possible per vertex. The additional nodes Q'_i and Q''_i will help us to enforce a different color assignment to neighboring vertices, as we explain below.

Next, for each edge $(v_i, v_j) \in E$, we add the following nodes to the dependency graph: two subqueries $Q^{(1)}_{i,j}$ and $Q^{(2)}_{i,j}$. The dependencies associated to these nodes are depicted in Figure 4.8. This dependency construction enforces a different color assignment for vertices v_i and v_j by making sure these assignments have a lower spread.

It can be verified that the following properties hold for every MTS $\mathcal{F}$ for the dependency graph:

- *spread*$(Q_0) \geq 3$;

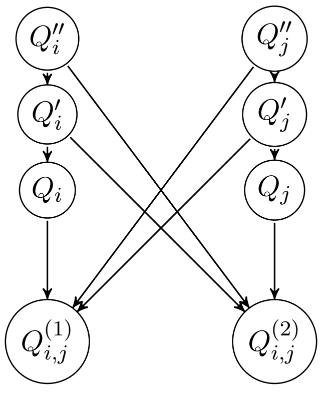


Figure 4.8: *Query dependencies introduced to ensure that every vertex to ensure that different colors are assigned to connected vertices.*

- the spread of Q_B, Q_G, Q_R equals 1;
- the spread of Q_U, Q_V equals 1;
- the spread of Q_i, Q'_i, Q'_i equals 1;
- for each edge $(v_i, v_j) \in E$,

$$\mathit{spread}(Q^{(1)}_{i,j}, \mathcal{F}) + \mathit{spread}(Q^{(2)}_{i,j}, \mathcal{F}) \geq 5.$$

Hence, we may conclude that for every MTS $\mathcal{F}$ for Q_G, we have $\mathit{spread}(Q, \mathcal{F}) \geq 3|V| + 5|E| + 8$.

We claim that G is 3-colorable if and only if there exists an optimal MTS $\mathcal{F}$ with $\mathit{spread}(Q, \mathcal{F}) = 3|V| + 5|E| + 8$. Suppose G is 3-colorable and let $\varphi : V \to \{R, G, B\}$ be the legitimate coloring. The optimal MTS $\mathcal{F} = (F_1, \ldots, F_7)$ is defined as follows. First,

$$\begin{array}{lll} Q_0 \in F_0 & & \\ Q_B \in F_1 & Q_G \in F_2 & Q_R \in F_3 \\ Q_U \in F_4 & Q_V \in F_5 & Q_T \in F_6, \end{array}$$

for each edge $(v_i, v_j) \in E$,

$$Q^{(1)}_{i,j}, Q^{(2)}_{i,j} \in F_0, \text{ and}$$

for each vertex $v_i \in \{v_1, \ldots, v_n\}$,

$$Q_i \in \begin{cases} F_1 & \text{if } \varphi(v_i) = B, \\ F_2 & \text{if } \varphi(v_i) = G, \\ F_3 & \text{if } \varphi(v_i) = R, \end{cases}$$

$$Q_i \in F_j \to Q'_i \in F_{j+1},$$
$$Q'_i \in F_j \to Q''_i \in F_{j+1}.$$

Note that $Q_i \in F_j \leftrightarrow Q_B \in F_j$, if vertex v_i is colored with B; $Q_i \in F_j \leftrightarrow Q_G \in F_j$, if it is colored with G, and $Q_i \in F_j \leftrightarrow Q_R \in F_j$, if it is colored with R. Furthermore, since for each $(v_i, v_j) \in E$, $\varphi(v_i) \neq \varphi(v_j)$, we have $Q_i \in F_j \leftrightarrow \neg(Q_j \in F_j)$. This means that either $Q_i, Q'_j, Q''_j \in F_k$ or $Q_j, Q'_i, Q''_i \in F_k$, for some k, which implies:

$$\mathit{spread}(Q^{(1)}_{i,j}, \mathcal{F}) + \mathit{spread}(Q^{(2)}_{i,j}, \mathcal{F}) = 5.$$

It is now easy to verify that $\mathit{spread}(Q, \mathcal{F}) = 3|V| + 5|E| + 8$.

For the converse, suppose that we have an MTS $\mathcal{F}$ for Q_G such that

$$\mathit{spread}(Q_G, \mathcal{F}) = 3|V| + 5|E| + 8.$$

This means that $\mathit{spread}(Q_0) = 3$ and for each edge $(v_i, v_j) \in E$,

$$\mathit{spread}(Q^{(1)}_{i,j}, \mathcal{F}) + \mathit{spread}(Q^{(2)}_{i,j}, \mathcal{F}) = 5.$$

The coloring $\varphi : V \to \{R, G, B\}$ is defined as follows.

$$\varphi(v_i) \quad = \quad \begin{cases} B & \text{if } Q_i \in F_1, \\ G & \text{if } Q_i \in F_2, \\ R & \text{if } Q_i \in F_3. \end{cases}$$

Since $\mathit{spread}(Q_0) = 3$, the MTS $\mathcal{F}$ assigns each Q_i at the same level as one of Q_G, Q_B and Q_R. Hence, the function φ is well-defined. Moreover, for each edge $(v_i, v_j) \in E$, we have $\mathit{spread}(Q^{(1)}_{i,j}, \mathcal{F}) + \mathit{spread}(Q^{(2)}_{i,j}, \mathcal{F}) = 5$. This means that for each $(v_i, v_j) \in E$, the MTS $\mathcal{F}$ does not assign Q_i and Q_j to the same level, as otherwise the spread would be larger than 5. Hence, as $\varphi(v_i) \neq \varphi(v_j)$ for each $(v_i, v_j) \in E$, we obtain a valid 3-coloring for G.

We have proven that each coloring corresponds to an MTS that has minimal spread and that each minimal spread corresponds to a coloring, which completes the proof. □

4.7 Extensions

4.7.1 Additional Atom Support

The MR programs described in this chapter only supports relational atoms, but they can be easily expanded to deal with equalities and inequalities. We now give a brief sketch on how to incorporate support for additional atom types.

First, we revise the definition of BSGF queries that was given in Section 4.3.1. A *basic strictly guarded fragment (BSGF) query* is an expression of the form

$$Z := \texttt{SELECT } \bar{x} \texttt{ FROM } R(\bar{t}) \text{ [} \texttt{WHERE } C \text{]}; \tag{4.16}$$

Now, C is a Boolean combination of atoms of one of the following forms:

1. $R(t_1, \ldots, t_n)$ with R a relation symbol of arity n and each of the t_i a term, $i \in [1, n]$,
2. x **op** y, where **op** $\in \{=, \neq\}$, or
3. x **op const**, where x is a variable, **op** $\in \{\leq, \geq, =, <, >\}$ and **const** $\in \mathbf{D}$.

Furthermore, when $S(\bar{x})$ is an atom in C, then all variables in $\bar{x}$ are a subset of those in $\bar{t}$.

Clearly, atoms of form 1 dictate the input relations. Atoms of forms 2 and 3 serve as a filtering mechanism, which can easily be checked by the EVAL reducer when evaluating the query.

Note that the query from Remark 4.5 is not allowed anymore, as variables z_1 and z_2 do not appear in the guard atom. Rewriting the query using the "traditional" notation yields a query which can be solved using one more round:

$$Z_1 := \texttt{SELECT } x \texttt{ FROM } S(x, y);$$
$$Z_2 := \texttt{SELECT } x \texttt{ FROM } R(x, y) \texttt{ WHERE } Z_1(x) \texttt{ AND NOT } Z_1(y).$$

As we were able to calculate the anwer to this query in two rounds before, this signifies there are opportunities to evaluate BSGF queries more efficiently. Consider, for example, the following query:

$$Z_1 := \texttt{SELECT } x \texttt{ FROM } S(x, y)\texttt{WHERE } y > 100;$$
$$Z_2 := \texttt{SELECT } x \texttt{ FROM } R(x, y) \texttt{ WHERE } Z_1(x) \texttt{ AND } T(y) \texttt{ AND } y < 10.$$

Here, the first query is only a projection. This can be done perfectly by the MSJ mapper. Furthermore, the $y < 10$ atom is handled by the EVAL reducer, but can also be used as a filter in the MSJ mapper to reduce the size of the intermediate data. Clearly, the newly added atoms open up opportunities to reduce the intermediate data. But, one has to be careful as the Boolean combinations may sometimes complicate these optimizations. For example, in the query

$$Z := \texttt{SELECT}\ x\ \texttt{FROM}\ S(x,y)\ \texttt{WHERE}\ y > 100\ \texttt{OR}\ (x = 10\ \texttt{AND}\ y = 3000)\ \texttt{OR}\ T(y),$$

the Boolean OR operations makes it impossible to perform a filtering step in the MSJ mapper. Instead, it might be interesting to define syntactical classes which can be evaluated more efficiently in practice.

4.7.2 Bag Semantics

In this entire chapter, we worked with set semantics, as this was the way we defined the semantics of strictly guarded fragment queries. It is possible, however, to work with bag semantics as well. The single semi-join and multi-semi-join (MSJ) algorithms need no adjustment, as the reducers output a result for every request message (if necessary, of course). The EVAL algorithm needs to be adjusted in such a way that the number of output tuples correspond to those in the first (guard) relation. Therefore, the reducer needs to record the number of X_0 facts that arrived and change the number of outputs accordingly. Our system Gumbo can deal with bag semantics as the optimizations allow every tuple to be uniquely identified (see Section 5.3.1).

4.8 Discussion

We provided a framework for evaluating SGF queries in a distributed setting using the MapReduce computation model. Our framework uses an updated cost model and the newly introduced multi-semi-join operator to detect and exploit overlap between queries. This allows us to transform 2-round parallel query plans for BSGF queries to be transformed into an equivalent version that potentially has a lower cost. As obtaining an optimal MapReduce program turned out to be NP-hard, we provided intuitive heuristics. We also showed that the simpler problem of minimizing the number of input reads is NP-hard as well. In the next chapter, we experimentally validate the proposed algorithms using our system called Gumbo and discuss the necessary implementation details.

5 Multi-Semi-Join Evaluation in Practice

In this chapter, we present Gumbo, an open-source system built on top of Hadoop in which we implemented the techniques proposed in the previous chapter. We use Gumbo to experimentally validate the algorithms by comparing them to standard techniques and to existing systems such as Hive and Pig. We find that our techniques are superior in several situations.

5.1 Contributions & Outline

With the necessary theoretical background in place, we are now ready to experimentally assess the effectiveness of Greedy-BSGF and Greedy-SGF and obtain that, backed by an updated cost model, these algorithms successfully manage to bring down total times of parallel evaluation, making it comparable to that of sequential query plans, while still retaining low new times. This is especially true in the presence of commonalities among the atoms of queries, i.e., atoms or relations that are re-used in different (parts of) queries.

Our open-source system called Gumbo contains a query compiler and execution engine that both support the methods described in the previous chapter. Alongside the basic implementations, we also find a multitude of optimizations that make the system usable in practice. As a lot of these optimizations can used in other MapReduce programs that serve a different purpose, we provide a thorough description of the ones that are most significant. Furthermore, we show through extensive experiments that Gumbo outperforms Pig and Hive in all aspects when it comes to parallel evaluation of SGF queries. Finally, as modern Big data systems need to scale well, we also provide insights in Gumbo's scaling abilities.

The contributions of this chapter are as follows:

1. We devise a 2-Round MapReduce algorithm based on the methods described in Chapter 4.
2. We describe a list of optimizations which allow our algorithm to be efficient in practice. Furthermore, these optimizations can be useful for other applications as well.
3. We give a high-level description of our open-source system called Gumbo, which can be used on top of an existing Hadoop 2.x cluster.
4. We experimentally assess the effectiveness of both greedy algorithms (Greedy-BSGF and Greedy-SGF) and find that these algorithms successfully manage to bring down total times for parallel query plans where common atoms appear and demonstrate that Gumbo outperforms Pig and Hive in all aspects when it comes to evaluation of SGF queries.

Outline. This chapter is organized as follows. First, we restate the BSGF evaluation algorithm in a more convenient form (MSJEVAL) in Section 5.2 and provide a list of optimizations that aim to improve its performance in Section 5.3. Next, in Section 5.4, we discuss the implementation of the algorithms in our own system called Gumbo, which runs on top of Hadoop. In Section 5.5, we perform an experimental evaluation using Gumbo. We conclude in Section 5.6.

5.2 Algorithm Revision

In order to thoroughly discuss the details of the 2-round BSGF evaluation algorithm outlined in the previous chapter, we provide a revision of the complete algorithm. Algorithm 12 shows the resulting algorithm, which is called MSJEVAL. An set of input BSGF queries needs to decomposed in to the required semi-joins, and the Boolean expressions. The former are used as input for the first round (MSJ), while the latter serve as input for the second round (EVAL).

The reworked algorithm follows a request-response model in which the main data consists of messages. Indeed, the mappers of the first round generate facts (Assert) and requests (Request), which are transformed into answers (Confirm or Deny). In the second round, these answers are "delivered" to the tuple that emitted the initial request and they are combined into a final query answer. We note the following:

- The number of BSGF queries (n) may not correspond precisely with the number of semi-joins (k) as multiple semi-joins may appear in one BSGF query and at the same time identical semi-joins may appear in different BSGF queries.
- Confirm and deny messages are used to signal the presence or absence of a conditional fact. These messages replace the EVAL map output, which explains why Map2 is an identity mapper.
- In Map1, the projection of guard atoms onto the output key is removed and now happens in Red2 for the simple reason that a "reply address" is needed to redirect the confirm messages.
- It is possible to evaluate multiple BSGF queries at once using this algorithm. This becomes apparent in Red2 where we find an iteration over all BSGF queries.
- The first mapper and reducer may be applied to a partition of $\mathcal{S}$ that is computed using Greedy-BSGF. As before, a partition of size m will lead to $m + 1$ MR jobs: m round 1 jobs and one round 2 job.

5.3 Optimizations

Most MR algorithms can greatly benefit from optimizations that allow a more efficient handling of the data (compression, sorting, avoiding redundant reads) or provide special solutions for certain input types. The optimizations we introduce in this section generally aim to reduce total and/or net times of MSJEVAL in Gumbo. Table 5.1 gives an overview of the optimizations, as well as an indication of their applicability to MR jobs in general. Table 5.2 shows which part of the MR pipeline is affected by the different optimizations.

5.3.1 Message Compression

Most MR jobs are I/O-bound, which means that the majority of time is spent in reading, writing or transmitting data. Therefore, it is of high importance to only store and transmit the data that is necessary and to choose an appropriate format.

In MSJEVAL, the information required by the final reducer consists of a truth value for each conditional atom and the (projected) tuple that needs to be output. In MSJEVAL, this information is supplied through the key, which contains the original fact. As this information is incorporated in all request, confirm and deny messages, this contributes to a significant data overhead in both rounds. When multiple conditional atoms are involved, the overhead of

Algorithm 12 MSJEVAL – 2-round MR algorithm for computing a set of BSGF queries.

```
Require: $\mathcal{S} = \{X_1 := \pi_{\bar{x}_1}(\alpha_1 \ltimes \kappa_1), \ldots, X_k := \pi_{\bar{x}_k}(\alpha_k \ltimes \kappa_k)\}$

 1: function MAP1(fact $f$)
 2:     buffer = [ ]
 3:     for every $i$ such that $f \models \alpha_i$ do
 4:         add $\langle \pi_{\alpha_i;\bar{z}_i}(f) : [\textsc{Req}\,(\kappa_i, i); \textsc{Reply}\, f] \rangle$ to buffer
 5:     for every $i$ such that $f \models \kappa_i$ do
 6:         add $\langle \pi_{\kappa_i;\bar{z}_i}(f) : [\textsc{Assert}\, \kappa_i] \rangle$ to buffer
 7:     emit buffer

 8: function REDUCE1($\langle k : V \rangle$)
 9:     for all [REQ $\kappa_i$; REPLY $f$] in $V$ do
10:         if $V$ contains [ASSERT $\kappa_i$] then
11:             emit $\langle f : [\textsc{Confirm}\, \kappa_i] \rangle$
12:         else
13:             emit $\langle f : [\textsc{Deny}\, \kappa_i] \rangle$

Require: $\mathcal{P} = \{Z_1 := (\alpha_1, \varphi_1), \ldots, Z_n := (\alpha_n, \varphi_n)\}$

14: function MAP2(fact $\langle k : v \rangle$)
15:     emit $\langle k : v \rangle$

16: function REDUCE2($\langle k : V \rangle$)
17:     $f \leftarrow k$
18:     for all $\kappa_i$ do
19:         if [CONFIRM $\kappa_i$] $\in V$ then
20:             $\kappa_i \leftarrow$ True
21:         else
22:             $\kappa_i \leftarrow$ False
23:     for all $Z_i$ do
24:         if $f \models \alpha_i \wedge eval(\varphi_i)$ then
25:             output $\pi_{\alpha_i;\bar{x}_i}(f)$ to $Z_i$
```

Optimization	Applicable to general MR
Mapper shaping	✓
Reducer shaping	✓
Message compression	?
Packing	?
Confirm Reduction	✗
Streaming Reducers	✗
Single Job Evaluation	✗
Map output estimation	✓
SequenceFiles	✓
Static Packing	✗

Table 5.1: *An overview of Gumbo's optimizations (top part) and specialized techniques (bottom part) with an indication of their applicability to general MR applications.*

the tuple data becomes even larger as multiple messages carry the same — and hence redundant — information.

A solution to this problem is to transmit tuple ids instead of actual tuples. In Gumbo, a *tuple id* is a pair consisting of the file id that contains the tuple and its offset within the file. Intuitively this serves the purpose of a tuple "address", allowing tuples to indicate where the response should be delivered. Tuples ids can have a variable-sized representation ranging from 2 to 18 bytes (two control bytes and maximum 64 bits per component). In most scenarios, this provides a smaller (and more predictable) representation of a tuple.

Recall that the final output tuples, generated by MAP1, are the result of projecting the original guard tuple. Hence, in order to compensate for the loss of the actual tuple *inside* the messages, we re-read the guard relation in the second round and emit a new type of message that delivers the actual tuple data to its unique tuple id or "address". This process is depicted in Figure 5.1.

Remark 5.1 (Tuple ids and bag semantics). When the tuple id optimization is enabled, a key-valueset pair that is processed by the second reducer is associated to precisely one guard input fact. This is not the case in the general MSJEVAL algorithm, where identical tuples are processed together. This makes it impossible in the general algorithm to assess the number of final outputs that should be generated for bag semantics and requires extra information to be transmitted in the messages. Using tuple ids resolves this issue. Note that because of this optimization, Gumbo offers bag semantics as opposed to the set semantics proposed in Chapter 4 (see also Section 5.3.1). Set semantics are available in a non-optimized version of Gumbo where tuple ids can be disabled; this requires using `gumbo.engine.hadoop.HadoopEngine`

	Message Compression	Streaming Reducers	Mapper Shaping	Reducer Shaping	Packing (Mapper)	Packing (Combiner)	Packing (Reducer)	Confirm Reduction
read								
map	$\odot$				$\odot$			
sort	●	●			●			
merge	●	●	●		●	●		
combine	●	●			$(\odot)$	$\odot$		
transfer	●	●	▼	▼	●	●		
merge	●	●		●	●	●		
reduce	●	●			$\odot$	$\odot$	$\odot$	$\odot$
write	●						●	●
read	$\odot$						●	●
map	●						●	●
sort	●						●	●
merge	●		○				●	●
combine	●						●	●
transfer	●		▼	▼			●	●
merge	●			○			●	●
reduce	$\odot$						$\odot$	$\odot$
write								

Table 5.2: *Overview of the effects of different optimizations on different parts of the 2-round algorithm (*MSJEVAL*). Necessary implementation changes are indicated with $\odot$, affected parts with ● or ○, where the latter is not implemented in Gumbo. An optional implementation change is shown as $(\odot)$, possible negative side-effects are shown as ▼.*

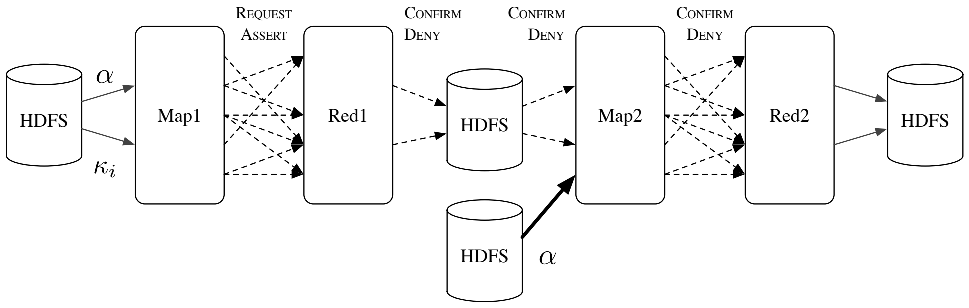


Figure 5.1: *Effect of tuple id compression on different stages of* MSJEVAL. *The dotted arrows indicate a data size reduction, while the bold arrow indicates a data increase.*

instead of the newer `gumbo.engine.hadoop2.HadoopEngine2`. ◇

Finally, a more obvious technique that reduces the size of messages is to use atom ids to refer to atoms inside the messages, instead of using their full representation. This can further reduce the size of the messages.

5.3.2 Message Packing

When messages that are send around in the MSJEVAL algorithm are relatively similar, this opens up opportunities for data compression. We use message *packing*, as also used in [163], which reduces network communication by combining messages associated with the same key into one message. In Gumbo, packing is used in three different forms:

- map output packing, or fact-based packing;
- combiner packing, or cross-fact packing;
- reduce output packing, or reply packing.

Map output packing or *fact-based packing* is applied to the map output of one input fact. Observe that MAP1 may emit multiple messages per fact f. A useful optimization that reduces network communication is to buffer all messages emitted per fact, and to *pack* the messages as follows. First, let us slightly generalize the format of a REQUEST message to be [REQUEST $\overline{(\bar{a}, \kappa)}$], where $\overline{(\bar{a}, \kappa)}$ denotes a sequence of $(\bar{a}, \kappa)$ pairs with $\bar{a}$ a tuple of data values and κ an atom. Likewise, let us generalize the format of ASSERT messages to be [ASSERT $\overline{\kappa}$] with $\overline{\kappa}$ a list of atoms. Then, for two messages with the same key, $m_1 = \langle \bar{c} : [\text{lab}; \text{list}_1] \rangle$, and $m_2 = \langle \bar{c} : [\text{lab}; \text{list}_2] \rangle$, that also have the same message label *lab* (either REQUEST or ASSERT) and where list$_1$ and list$_2$ are corresponding lists, we define

$$\text{pack}(m_1, m_2) = \langle \bar{c} : [\text{lab}; \text{list}_1 \cup \text{list}_2] \rangle \, .$$

Packing reduces both the number of output tuples and the total output size of the mapper, which can positively impact sort, merge and transfer times. Of course, if message packing is enabled, the reducer needs to unpack the messages before processing them. In particular, consider a fact f conforming to two conditional atoms κ_i and κ_j. Without packing, two ASSERT messages would be emitted. If, however, the join key $\bar{z}_i$ of κ_i with α_i is the same as the join key $\bar{z}_j$ of κ_j with α_j, then the key of both emitted messages is $\pi_{\bar{z}_i}(f)$ and we can pack both messages in a single message

$$\langle \pi_{\bar{z}_i}(f) : \text{ASSERT}\, \kappa_i, \kappa_j \rangle \, .$$

The same reasoning holds when f conforms to two guard atoms α_i and α_j and multiple REQUEST messages are normally emitted. If the join keys $\bar{z}_i = \bar{z}_j$ are equal, again a single REQUEST message of the form

$$\left\langle \pi_{\bar{z}_i}(f) : [\text{REQUEST}\, \pi_{\bar{x}_i}(f), \kappa_i,\ \pi_{\bar{x}_j}(f), \kappa_j] \right\rangle$$

suffices. Note that this is especially true in the specific case where $\alpha_i = \alpha_j$.

Remark 5.2 (Static vs. dynamic packing). The example above implies that an static inspection of the query (detecting common join keys) will reveal packing opportunities that are applicable for every fact. This allows for early optimizations that may improve map time as less data inspection is necessary. Gumbo incorporates this optimization by precalculating the packing opportunities (see Section 5.4.3). The more general version of packing, which is defined above, also allows packing of “accidental” projections that produce identical keys, even for different join keys. For example, $R(x, y) \wedge S(x) \wedge S(y)$ will produce two request messages with identical keys when $x = y$. $\diamond$

A second form of packing is called *combiner-based packing* or *cross-fact packing*. Here, the packing technique that was described above is applied to the entire output of one map task in the form of a combiner. Recall that a combiner is applied to a group of key-value pairs that have the same key. Hence, this creates the opportunity of packing messages that originate from different facts but still have an identical key. Note that the effectiveness of this technique strongly depends on both the join keys appearing in the query and the actual data characteristics.

This more aggressive form of packing is not always beneficial: we learned through our experiments that the additional overhead of using a combiner does not positively influence the total or net time. In contrast, packing the messages emitted per fact in an individual map task can be efficiently implemented and can significantly decrease the size of the communicated data as we will see in the experimental results (see Section 5.5).

Remark 5.3 (Fact-based vs. cross-fact packing). Note that fact-based packing can also be accomplished by using a combiner. The downside of this method is that the *unpacked* map output, which contains more messages, participates in the sort process and hereby causes an increase in sort time. Also, more spills are initiated, which can potentially increase map-side merge times. ◇

The final form of packing is called *reduce output packing*, or *reply packing*. Recall that round 1 reduce tasks output confirm and deny messages in order to send the truth values for atoms to the tuples that requested it. As we saw in fact-based packing, multiple request messages that have the same key may originate from the same fact and can hence be replied to using only one message. The same packing approach can be used here.

5.3.3 Confirm Reduction

Greedy-SGF uses confirm messages in the second round to indicate whether an atom should be set to `True` or `False`. These messages originate form Red1, are written to HDFS, and pass through almost every component of the EVAL job. At least half of these messages can be avoided by only sending one type of reply message. In Gumbo, only confirm messages are sent. Further improvement can consist of estimating the frequency of confirm and deny messages and only emitting the type that occurs the least.

5.3.4 Streaming Reducers

Another major challenge for MR programs is skew. Skew arises for example when key-value pairs are not distributed uniformly across all reducers (see Remark 4.7 on page 119). This may cause some reducers to take up more time then others, or even exceed the allowed resource usage. Especially the latter problem is undesired as this would cause the MR job to fail. For Red2, the number of confirm messages cannot exceed the number of request messages that were generated for a given input fact. Hence, the memory required only depends on the query structure (number of distinct atoms) and not on the input size. For Red1, on the other hand, the number of request and assert messages assigned to a reducer is not limited by the query structure. Indeed, one can always create a dataset that sends n messages to the same reducer by duplicating a fact n times (recall that we assume bag semantics in Gumbo, see Remark 5.1). This will cause n identical messages to be created by the mapper.[35] Recall that Red1 searches for the presence of assert messages, record the atoms linked to them (bounded by the size of the query) and needs to store all request messages. This means we can always create a data-set that causes the reducer to exceed its memory constraints for any query. Note that these constraints are not an issue for Hadoop itself as it uses external memory algorithms for the data operations.

A solution to this problem consists of using *streaming reducers*. The memory requirements of a streaming reducer only depend on the query size and not on data size, and it only performs one pass over the input. For MSJEVAL, we can transform Red1 into a streaming reducer: *if* we can assume all assert messages appear before the request messages, we only need to store the atom ids of the asserted conditional atoms, after which the request messages can be processed in a streaming fashion. Clearly, this defines a streaming reducer as the number of stored atom ids is data independent (linear in the size of the query). Fortunately, Hadoop allows user-overridden sort functions that make it possible to re-arrange the messages based on their type. As sorting is part of the MR pipeline anyway (see Section 4.4.2), this adjustment comes — in theory — at no additional cost. However, we found through our experiments that a highly optimized version of the sorting function is necessary in order to obtain a good performance of the sort step. For the details of this function, we refer to the Gumbo implementation [66].

5.3.5 Mapper Shaping

Another bottleneck in MR jobs is map spilling. When mappers produce too much output, this will be spilled to multiple files that have to be merged into one single sorted file. Possible causes for high spilling include a high replication factor,[36] large key-value pairs or incorrect configuration settings. The cost model of Section 4.4 clearly shows that a high number of spill files can significantly impact the total cost. Therefore, we propose an optimization that adjusts the input size of the mappers in such a way that spilling is avoided.

Recall from Section 4.4.2, in Hadoop, the number of map tasks created for a job equals its number of input splits. By default, one split corresponds to one HDFS block, but this can be adjusted by changing the maximum[37] and/or minimum[38] split size to a value that can be calculated as follows. Let N be the total input size, M the total map output size and B the size of the map output buffer (see Section 4.4.2). In order to avoid multiple spill

[35]Note that combiner-based packing (see Section 5.3.2) may help in some specific situations.

[36]Also called *replication rate* obtained by dividing map output size by map input size.

[37]**Hadoop setting:** `mapreduce.input.fileinputformat.split.maxsize`

[38]**Hadoop setting:** `mapreduce.input.fileinputformat.split.minsize`

files, each map task can only output a maximum of B bytes. Therefore, at least $m = \lceil M/B \rceil$ mappers are required and each mapper can process at most $S = \lfloor N/m \rfloor$ input bytes, where S equals the desired split size. More information on how to control the split size is given by White [164, p. 225].

Remark 5.4 (Split-block ratio). One has to be careful with split sizes that are larger than the block size, as extra blocks may need to be fetched from different nodes, possibly having an adverse effect on map performance [164, p. 224–226]. On the other hand, when the block size is not a multiple of the split size, similar undesired effects can occur.[39] Hence, it is recommended to correct the split size S such that the block size is a multiple of S.

5.3.6 Reducer Shaping

Similar to excessive spilling at the map-side, the reduce-side can also be troubled by the side-effects caused by a data overload. The task of the reduce-side merge step is to consolidate outputs from the different map tasks into one sorted file. As opposed to the number of mappers, which is based on the number of splits, the number of reducers has to be set directly by the user. Popular systems such as Hive and Pig use set the number of reducers in function of the input size. For example, Pig allocates 1 GB of input per reducer [164, p. 467]. As map input size has no direct relation to map output size, this strategy can be suboptimal. Indeed, when the replication factor (see Section 5.3.5) approaches 0, i.e., most input tuples are filtered out, individual reducers will operate on a tiny amount of data. When the replication factor far exceeds 1, i.e., a lot of outputs are generated for every input tuple, each reducer will operate on a large amount of data, possibly leading to excessive spilling. This illustrates the need for a better, and preferably automatic, way to determine the number of reducers. We therefore propose a more flexible mechanism, similar to mapper shaping, that automates this process and aims to eliminate spilling.

To obtain the desired number of reducers, Gumbo uses the total map output size[40] and divides this by the memory available per reduce task. This way, under the assumption that the partition function uniformly divides map output data over the reduce tasks, no reducer needs to perform excessive spills to merge the data.

[39]See `http://stackoverflow.com/a/14540272/787036` for a good example.

[40]The total map output sized can be estimated using basic sampling and simulation techniques (see Section 5.4.2).

5.3.7 Single-Job Evaluation

Instead of the 2-round MSJEVAL algorithm, an equivalent 1-round (and single-job) algorithm can be applied in certain situations. When the atoms that appear in the conditional part all have the same (or equivalent) join key, the reducers in the first round have all the required information at their disposal for providing an answer to the conditional part of the query. This is also the case when the conditional part of the query is a disjunction of (negated) atoms. In the latter case, an extra pass is necessary to guarantee unique results, if required.

The general 1-round MR program operates as follows. Consider an SGF query Q where all the join keys in the conditional atoms are the same, say $\bar{v}$. Then Q can be evaluated in a single MR round by combining MSJ and EVAL as follows. The mapper emits $\langle \pi_{\alpha;\bar{v}}(f) : [\text{Req}\,;\text{Out}\,\pi_{\alpha;\bar{w}}(f)]\rangle$ messages for all α-conforming facts f, and $\left\langle \pi_{X_i(\bar{v});\bar{v}}(g) : [\text{Assert}\,X_i]\right\rangle$ messages for all $X_i(\bar{v})$-conforming facts g, $i \in [1,n]$. On input $\left\langle \bar{b} : V\right\rangle$ the reducer uses the $[\text{Assert}\,X_i]$ messages in V to verify that φ is satisfied, and, if so, outputs $\bar{a}$ for every $[\text{Req}\,;\text{Out}\,\bar{a}]$ in V.

This algorithm can be used for multiple BSGF queries; only the addition of a query id is needed. Tuple-ids can be used as well. For this optimization to work, we can send the guard tuples as separate messages. This makes the one-round optimization typically a bit more expensive than the first round of the 2-round strategy, but this is more than compensated for by the omission of the second MR job. The resulting algorithm is denoted by MSJEVAL-1.

5.4 Gumbo

In order to validate the performance of the algorithms presented in Chapter 4, we implemented the MSJEVAL algorithm and the associated optimizations op top of Hadoop in a system called Gumbo. The resulting system is able to construct an optimized MR query plan, as well as execute this plan. We briefly discuss Gumbo's internal structure, its map output estimation method and possible extensions.

5.4.1 Internal Structure

Figure 5.2 shows the different components of Gumbo that are used to process an SGF query. SGF queries are provided together with the location of input and output relations. Figure 5.3 shows an example input file example which clarifies the structure. This input is processed by the compiler, which extracts the set of BSGF queries and creates a dependency graph as wel as a multiway topological sort. These are used by the execution engine to create actual MR

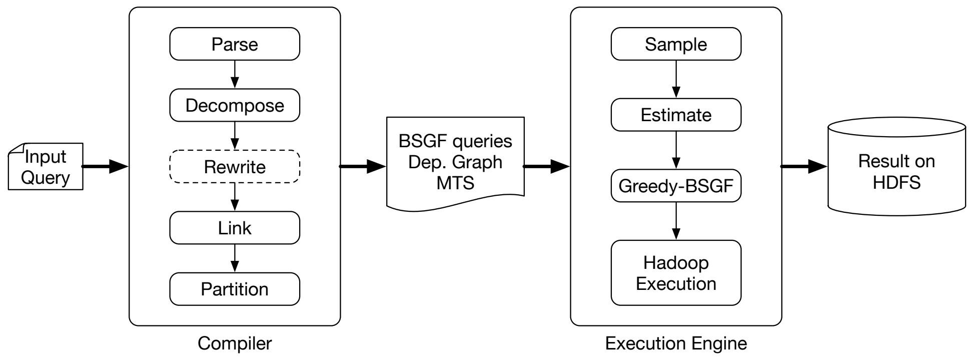


Figure 5.2: *The SGF workflow in Gumbo.*

jobs which are run using Hadoop. Aside from highlighting the most important aspects of the compiler and execution engine, we also outline how sampling is done in Gumbo to provide estimates for the cost model. Finally, we discuss some low-level optimizations and indicate how Gumbo could be ported to similar systems.

Compiler. The task of the compiler is to convert a set of SGF queries into a set of BSGF queries, determine their dependencies and provide a way of grouping the queries in the form of an multiway topological sort. For simplicity, we assume only one SGF query is given. This has no influence on the procedure of the compiler.

First, a `Parser` converts the raw input query into a `GumboQuery`, which contains the actual SGF queries, a relation-input file mapping, an output directory and temporary (or scratch) directory. The SGF query is then passed to the `Decomposer`, which breaks up the SGF query into a set of BSGF queries. Next, the optional `Rewriter` rewrites the BSGF queries into the desired form.[41] Then, the `Linker` determines the dependencies between the BSGF queries, based on their atoms, and constructs the corresponding dependency graph. Finally, the `Partitioner` creates a multiway topological sort (MTS, see Section 4.5.6) using the selected algorithm. Gumbo offers five strategies to create an MTS:

Unit no grouping is performed, i.e., all MTS subsets have size 1.

Depth groups BSGF queries based on the depth of their query subtree.

Height groups BSGF queries based on their distance to the top-level query.

[41]The rewriter can be used to, for example, unnest BSGF queries in such a way that no conjunctions appear in the conditional part.

```
Output -> output/EXP_033/
Scratch -> scratch/EXP_033/

R,4 <- input/experiments/EXP_033/1/R, CSV
S,1 <- input/experiments/EXP_033/1/S, CSV
T,1 <- input/experiments/EXP_033/1/T, CSV
U,1 <- input/experiments/EXP_033/1/U, CSV
V,1 <- input/experiments/EXP_033/1/V, CSV

#Out1(x,y,z,w)&R(x,y,z,w)&S(x)&T(y)&U(z)V(w);
```

Figure 5.3: *An example Gumbo input that contains the output directory, a temporary (scratch) directory, input relations (name, arity, location, format) and the actual query in prefix notation. Different input formats (infix, SQL-like) are also supported by Gumbo.*

Optimal brute-force implementation of SGF-OPT.

Greedy implementation of GREEDY-SGF.

Even though Optimal implements brute-force search, the running time is still acceptable for small inputs, but of course grows quickly with the number of BSGF queries, as expected. For larger queries, one may choose to use an SMT solver (e.g. Microsoft's Z3 system [68]) to obtain the optimal schedule more quickly.

Execution engine. Given a set of BSGF queries, their dependency graph and an MTS that gives a grouping for the BSGF queries, the execution engine creates suitable MR jobs that calculate the query anwer. For every group in the MTS, at least two MR jobs are created that correspond to the MSJEVAL algorithm. The greedy grouping algorithm (see Section 4.5.4) is used to obtain a good strategy for the first round, which consists of MSJ jobs. The resulting set of GREEDY jobs is then passed to Hadoop and executed. When these are all completed, the EVAL job is executed to calculate the final output. When the EVAL job is finished, the next MTS group is processed. During this entire process, Gumbo provides map output size estimates that are used for the cost model in the grouping algorithm (see below), manages the input and output paths associated to the jobs, creates the necessary (temporary) file paths, makes sure the final data ends up in the correct output directory, and enables certain optimizations, depending on the user settings.

5.4.2 Map Output Estimation

The cost model presented in Section 4.4 requires knowledge on map input, map output and reduce output sizes. We now outline a technique to obtain an accurate estimate for the map output size. Providing a precise value for this metric would require performing the map calculation on the entire set of input records. As this is infeasible in a practical setting, Gumbo uses basic sampling and estimation techniques to obtain an estimate. The reduce output size is omitted in Gumbo as it is more difficult to obtain in the presence of several optimizations and it does not play a major role in the overall cost when we disregard the packing optimization. Also, when multiple successive applications of MSJEVAL are required, the output is accounted for in the input cost of the next stage.

We assume that the map function behaves uniformly on all data of a given relation (see Section 4.4). If not, the relation can be split into different parts. Furthermore, we assume that there is a linear relation between the input size and the output size of the map function. This is a reasonable assumption for our algorithm, as it mainly consists of replicating messages and adding a constant amount of data. Gumbo aims to approximate this linear relation using the following basic extrapolation method. First, the input data is sampled. These samples are divided into two sets: a small and a large sample set. Next, the map function is applied to these samples to discover the corresponding map output sizes. Using linear extrapolation, we can obtain an estimate for the total map output size. We found that in our experiments the estimated values closely resembled the actual values.

Remark 5.5 (Possible sampling issues). The round 1 map function in our algorithm involves a projection operation. When all fields of a relation have a fixed size, this does not pose any problems, but, if fields have a variable size the linearity assumption may not hold. Also, when a lot of tuples are filtered out, it is important that a large enough set of samples is used. $\diamond$

5.4.3 Low-level Optimizations

Static packing. When a map or reduce task is initialized, the input queries are transformed into a series of transformations that contain a condition and an actual transformation that is directly applied on input tuples. When conditions are equal, and the transformations allow packing, they are merged together into one transformation that directly applies packing (see also Remark 5.2). This opens the door for more advanced pre-processing optimizations.

Job output compression. Gumbo compresses the output of the first job in the MSJEVAL algorithm (the MSJ job) by using a `SequenceFile` as the output type. This allows for a reduction in output size and faster input parsing in the second (EVAL) job.

5.4.4 Spark & Tez Support

Apache Spark is gaining more popularity as it provides quicker and more flexible query execution when compared to Hadoop. The power of Spark mainly resides in the use of its RDD datastructure, which enables data to reside in memory [170] as opposed to being read from and written to disk. As it only executes MR jobs, Gumbo's execution engine can be easily adjusted to operate with Spark. The input data would need to be stored in Spark's RDD data structure and the jobs should be translated to RDD transformations and actions such as `flatMap()` and `groupByKey()`. Assuming that the necessary relations are all stored in a single RDD A, a straightforward skeleton for our 2-round algorithm in Spark is as follows:

```
B = A.flatMapToPair(<Mapper 1>);
C = B.groupByKey();
D = C.flatMap(<Reducer 1>);
E = D.flatMapToPair(<Mapper 2>);
F = E.groupByKey();
G = F.flatMap(<Reducer 2>); // G is the output
```

To make this Spark algorithm more efficient, we can apply some optimizations. For example, Mapper 2 is actually just an identity mapper, so it can be omitted.

The observation above indicates that the second mapper in the Hadoop execution engine adds overhead to the job execution. Indeed, after round one, the "intermediate" data is written to HDFS. This is costly because the data is replicated and needs to be re-read by the next MR job. Tez [147] offers a way of avoiding this overhead by skipping all but the first map functions. More specifically, it allows reduce steps to immediate follow a previous reduce step without the need for intermediary maps. This way, the overhead of temporarily storing the inter-job data to HDFS is removed. This can greatly reduce the cost of multi-round MR programs such as MSJEVAL. Tez is implemented on top of Hadoop as well, which makes it easier to port Gumbo to this alternative framework.

5.5 Experimental Validation

In this section, we experimentally validate the effectiveness of our algorithms. First, we discuss our experimental setup in Section 5.5.1. In Section 5.5.2, we discuss the effectiveness of the MSJ program and compare it to Pig and Hive.

Hadoop Setting	Value
`io.file.buffer.size`	131072
`dfs.replication`	3
`mapred.child.java.opts`	-Xmx1024m
`mapreduce.map.memory.mb`	1280
`mapreduce.reduce.memory.mb`	1280
`mapreduce.task.io.sort.mb`	512
`mapreduce.reduce.merge.inmem.threshold`	0
`mapreduce.reduce.input.buffer.percent`	0.5
`mapreduce.job.reduce.slowstart.completedmaps`	1
`mapred.map.tasks.speculative.execution`	false
`mapred.reduce.tasks.speculative.execution`	false
`yarn.nodemanager.resource.memory-mb`	49152
`yarn.scheduler.minimum-allocation-mb`	4096
`yarn.scheduler.maximum-allocation-mb`	49152
`yarn.nodemanager.resource.cpu-vcores`	10

Table 5.3: *Hadoop settings used in the experiments.*

In Section 5.5.3, we discuss the evaluation of BSGF queries. In particular, we compare with Pig and Hive and address the effectiveness of the cost model. The experiments concerning nested SGF queries are presented in Section 5.5.4. Finally, we discuss various scaling characteristics of Gumbo in Section 5.5.5.

5.5.1 Experimental setup

All experiments are conducted on the HPC infrastructure of the Flemish Supercomputer Center (VSC). Unless otherwise noted, each experiment was run on a cluster consisting of 10 compute nodes. Each node features two 10-core "Ivy Bridge" Xeon E5-2680v2 CPUs (2.8 GHz, 25 MB level 3 cache) with 64 GB of RAM and a single 250GB hard disk. The nodes are linked to a IB-QDR Infiniband network. We used Hadoop 2.6.2, Pig 0.15.0 and Hive 1.2.1. Table 5.3 lists the settings for the Hadoop cluster that was utilized for our experiments. Table 5.4 shows the values that are used for the constants in the revised MapReduce cost model. The values were obtained using benchmark tests on the system used for the experiments. All experiments are run three times; average results are reported.

Queries typically contain a multitude of relations and the input sizes of our experiments go up to 100GB depending on the query and the evaluation strategy. The data that is used for the guard relations consists of 100M tuples that add up to 4GB per relation. For the conditional relations we use the same

Constant	Description	Value
l_r	local disk read cost (per MB)	0.03
l_w	local disk write cost (per MB)	0.085
h_r	hdfs read cost (per MB)	0.15
h_w	hdfs write cost (per MB)	0.25
t	transfer cost (per MB)	0.017
D	external sort merge factor	10
buf_{map}	map task buffer limit (in MB)	409MB
buf_{red}	reduce task buffer limit (in MB)	512MB

Table 5.4: *Values for constants used in the cost model.*

number of tuples that add up to 1GB per relation; 50% of the conditional tuples match those of the guard relation, unless otherwise noted.

We use the following performance metrics:

1. *total time*: the aggregate sum of time spent by all mappers and reducers;
2. *net time*: elapsed time between query submission to obtaining the final result;
3. *input cost*: the number of bytes read from hdfs over the entire MR plan;
4. *communication cost*: the number of bytes that are transferred from mappers to reducers.

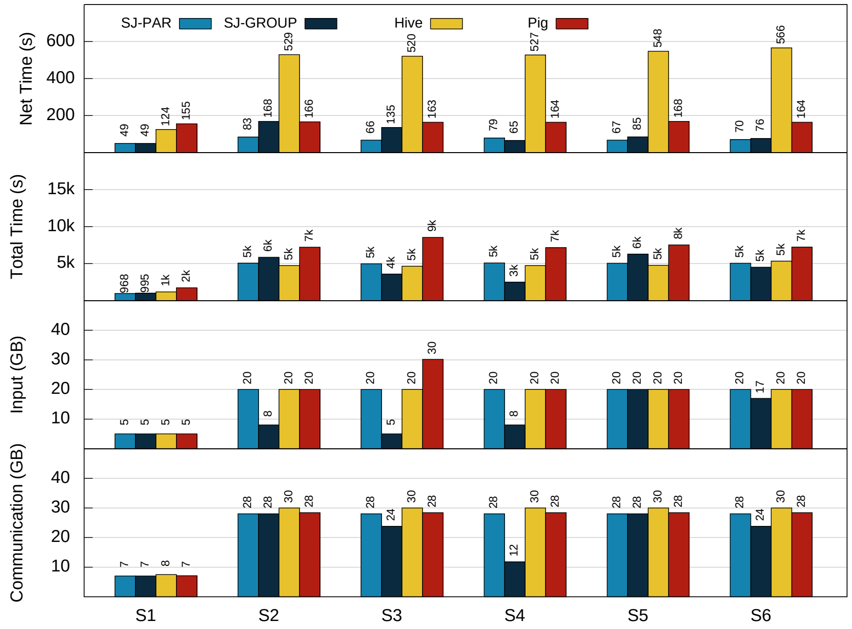
5.5.2 Multi-Semi-Join Queries

We start with the evaluation of sets of semi-joins. Table 5.5 lists the type of semi-join queries that are used in this section. We consider two evaluation strategies in Gumbo. The first strategy serves as the baseline and evaluates all semi-joins separately (SJ-PAR). That is, for a set of n semi-joins there are n MSJ jobs that are evaluated in parallel. The second strategy is the naive application of MSJ that groups all semi-join queries together into one MSJ job (SJ-GROUP). We also compare to Hive and Pig, for which we consider two approaches: using the semi-join operator in Hive and using the COGROUP operator in Pig. The results are depicted in Figure 5.4 and are discussed next.

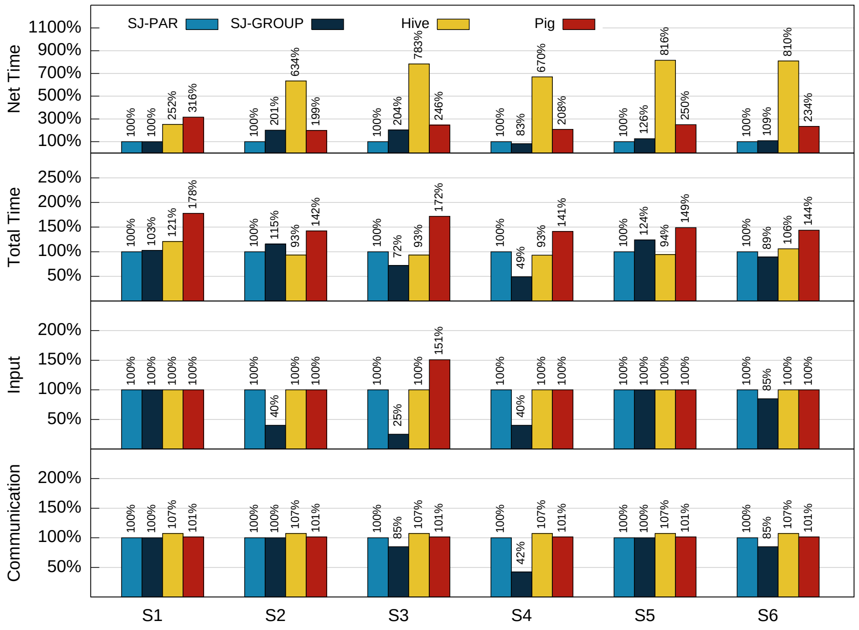
SJ-PAR versus SJ-GROUP. Figure 5.4 shows that SJ-GROUP can, for certain queries, provide benefits over SJ-PAR in terms of total time. Indeed, for queries S3, S4 and S6, we find a decrease in total time of 72%, 49% and 89% respectively. This is caused by the reduction of both the input data (up to 75% less) and intermediate data (up to 58% less). The latter is a direct

QID	Query	Type of query
S1	$R(x,y,z,w) \ltimes S(x)$	baseline
S2	$R(x,y,z,w) \ltimes S(x)$ $R(x,y,z,w) \ltimes T(y)$ $R(x,y,z,w) \ltimes U(z)$ $R(x,y,z,w) \ltimes V(w)$	guard name sharing
S3	$R(x,y,z,w) \ltimes S(x)$ $R(x,y,z,w) \ltimes S(y)$ $R(x,y,z,w) \ltimes S(z)$ $R(x,y,z,w) \ltimes S(w)$	guard & conditional atom name sharing
S4	$R(x,y,z,w) \ltimes S(x)$ $R(x,y,z,w) \ltimes T(x)$ $R(x,y,z,w) \ltimes U(x)$ $R(x,y,z,w) \ltimes V(x)$	guard & conditional atom key sharing
S5	$R(x,y,z,w) \ltimes S(x)$ $G(x,y,z,w) \ltimes T(y)$ $H(x,y,z,w) \ltimes U(z)$ $I(x,y,z,w) \ltimes V(w)$	no sharing
S6	$R(x,y,z,w) \ltimes S(x)$ $G(x,y,z,w) \ltimes S(y)$ $H(x,y,z,w) \ltimes S(z)$ $I(x,y,z,w) \ltimes S(w)$	conditional atom name sharing

Table 5.5: *Queries used in the MSJ experiment.*



(a) *Absolute values.*



(b) *Relative values.*

Figure 5.4: *Results for* SJ-PAR, SJ-GROUP, *Hive and Pig for the semi-join queries in Table 5.5*

consequence of the packing optimization, which can be applied when either keys or relation names are shared among the conditional atoms.

For queries S2 and S5, we find that SJ-GROUP causes an increase in total time. This is caused by an increased amount of spilling that cannot be compensated for by packing, as no commonalities are present among the conditional atoms.

In regard to net time we find that SJ-GROUP, in general, increases the net time. This effect is most visible for queries S2 and S3 where the guard has to emit four request messages per tuple, hereby increasing the average duration of the map tasks by 155% on average because of spilling. For query S4, the net time even decreases, mainly due to the high decrease in communication bytes. Finally, queries S5 and S6 show net times that are 26% and 9% higher, respectively. The main cause of this is the lack of separate synchronization points between map and reduce phases, as all tasks are now collected into one job.

Hive & Pig. We find that Hive exhibits total times similar to SJ-PAR, but net times that are more than 6 times higher in all cases. This is due to Hive blocking the different semi-join jobs to run in parallel, even though parallel execution is enabled. For Pig, we find that total time is on average 49,6% higher than for SJ-PAR and net times are on average 127,4% higher. The latter result is better than Hive's and is caused by the fact that Pig does allow for parallel execution of its COGROUP operations.

Conclusion. The naive implementation of semi-join queries Gumbo already outperforms Pig and Hive. Grouping all queries in one job affects total and net time in different ways, depending on the query. Together with the fact that these results can fluctuate even more depending on the actual input, this indicates the need for a cost-based approach.

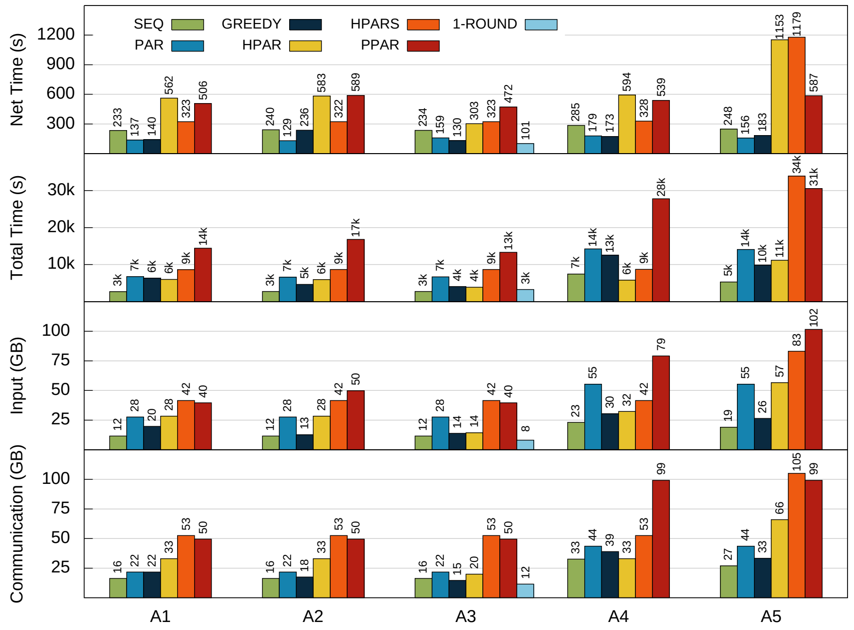
5.5.3 BSGF Queries

Table 5.6 lists the type of BSGF queries used in this section. The results obtained here generalize to non-conjunctive BSGF queries. Conjunctive BSGF queries were chosen to simplify the comparison with sequential query plans. Figures 5.5 and 5.6 show the results that are discussed next.

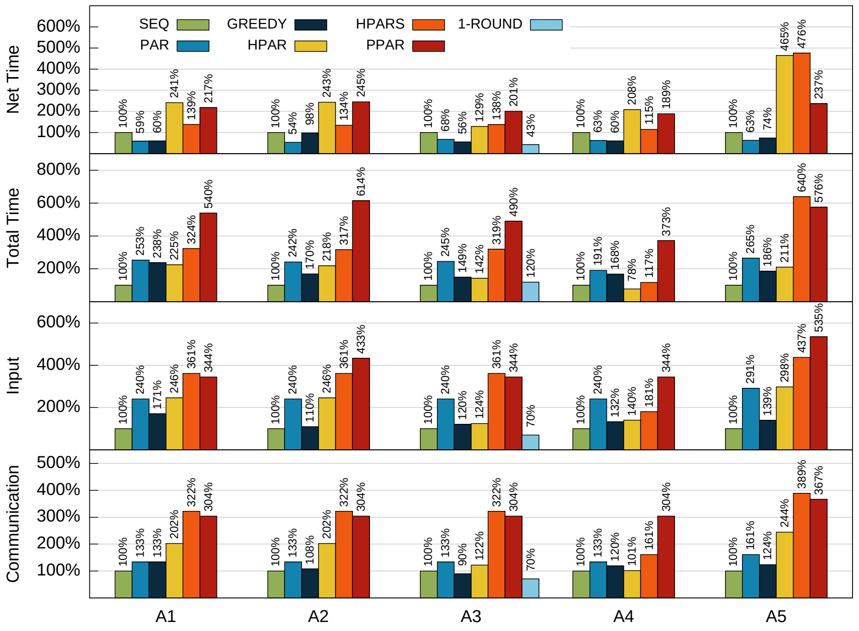
Sequential vs. Parallel. We first compare sequential and parallel evaluation of queries A1–A5 to highlight the major differences between sequential and parallel query plans and to illustrate the effect of grouping. In particular, we consider three evaluation strategies in Gumbo: (*i*) evaluating all semi-joins sequentially by applying a semi-join to the output of the previous stage (SEQ), where the number of rounds depends on the number of semi-joins; (*ii*) using the 2-round strategy with algorithm Greedy-BSGF (GREEDY); and, (*iii*)

QID	Query	Type of query
A1	$R(x,y,z,w) \ltimes$ $S(x) \wedge T(y) \wedge U(z) \wedge V(w)$	guard sharing
A2	$R(x,y,z,w) \ltimes$ $S(x) \wedge S(y) \wedge S(z) \wedge S(w)$	guard & conditional atom name sharing
A3	$R(x,y,z,w) \ltimes$ $S(x) \wedge T(x) \wedge U(x) \wedge V(x)$	guard & conditional atom key sharing
A4	$R(x,y,z,w) \ltimes$ $S(x) \wedge T(y) \wedge U(z) \wedge V(w)$ $G(x,y,z,w) \ltimes$ $W(x) \wedge X(y) \wedge Y(z) \wedge Z(w)$	no sharing
A5	$R(x,y,z,w) \ltimes$ $S(x) \wedge T(y) \wedge U(z) \wedge V(w)$ $G(x,y,z,w) \ltimes$ $S(x) \wedge T(y) \wedge U(z) \wedge V(w)$	conditional atom name sharing
B1	$R(x,y,z,w) \ltimes$ $S(x) \wedge T(x) \wedge U(x) \wedge V(x) \wedge$ $S(y) \wedge T(y) \wedge U(y) \wedge V(y) \wedge$ $S(z) \wedge T(z) \wedge U(z) \wedge V(z) \wedge$ $S(w) \wedge T(w) \wedge U(w) \wedge V(w)$	large conjunctive query
B2	$R(x,y,z,w) \ltimes$ $(S(x) \wedge \neg T(x) \wedge \neg U(x) \wedge \neg V(x)) \vee$ $(\neg S(x) \wedge T(x) \wedge \neg U(x) \wedge \neg V(x)) \vee$ $(S(x) \wedge \neg T(x) \wedge U(x) \wedge \neg V(x)) \vee$ $(\neg S(x) \wedge \neg T(x) \wedge \neg U(x) \wedge V(x))$	uniqueness query

Table 5.6: *Queries used in the BSGF experiment.*



(a) *Absolute values.*



(b) *Values relative to* SEQ.

Figure 5.5: *Results for evaluating the BSGF queries in Table 5.6 using different strategies.*

a more naive version of GREEDY where no grouping occurs, i.e., every semi-join is evaluated separately in parallel (PAR). As semi-join algorithms in MR have not received significant attention, we choose to compare to two extreme approaches: no parallelization (SEQ) and parallelization without grouping (PAR). Relative improvements of PAR and GREEDY w.r.t. SEQ are shown in Figure 5.5b.

Recall that parallel query plans aim to minimize net time. As expected, we find that both PAR and GREEDY result in lower net times. In particular, we see average improvements of 39% and 31% over SEQ, respectively. On the other hand, the total times for PAR are much higher than for SEQ: 132% higher on average. This is explained by the increase in both input and communication bytes, whereas the data size can be reduced after each step in the sequential evaluation. For GREEDY, total times vary depending on the structure of the query. Total times are significantly reduced for queries where conditional atoms share join keys and/or relation names. This effect is most obvious for queries A2, A3 and A5 where we observe reductions in total time of 30%, 29% and 30%, respectively, w.r.t. PAR.

For query A3, all conditional atoms have the same join key, making 1-round evalation possible. This is denoted by 1-ROUND[42] and further reduces the total and net time to only 49% and 63% of those of PAR, respectively.

Hive & Pig. We now examine parallel query evaluation in Pig and Hive and show that Gumbo outperforms both systems for BSGF queries. For this test, we implement the 2-round query plans of Section 4.5.4 directly in Pig and Hive. For Hive, we consider two evaluation strategies: one using Hive's left-outer-join operation (HPAR) and one using Hive's semi-join operations (HPARS). For Pig, we consider one strategy that is implemented using the COGROUP operation (PPAR). We also studied sequential evaluation of BSGF queries in both systems but choose to omit the results here as both performed drastically worse than their Gumbo equivalent (SEQ) in terms of net and total time.

First, similar to the semi-join experiment, we find that HPAR lacks parallelization. This is again caused by Hive's restriction that certain join operations are executed sequentially, even when parallel execution is enabled. This leads to net times that are 238% higher on average, compared to PAR. Note that query A3 shows a better net time than the other queries. This is caused by Hive allowing some form of grouping on certain join queries, effectively bringing the number of jobs (and rounds) down to 2.

Next, we find that HPARS performs better than HPAR in terms of net time but is still 126% higher on average than PAR. The lower net times w.r.t. HPAR are explained by Hive allowing parallel execution of semi-join operations, without allowing any form of grouping. This effectively makes

[42]See also MSJEVAL-1 in Section 5.3.7.

HPAR the Hive equivalent of PAR. The high net times are caused by Hive's higher average map and reduce input sizes, which lead to higher average map and reduce times, unavoidably increasing the overall net time.

Finally, Pig shows an average net time increase of 254% w.r.t. PAR. This is mainly caused by the lack of reduction in intermediate data and in input bytes, together with the strategy of determining the number of reducers based on the input size (1GB of map input data per reducer). For these queries, this leads to a low number of reducers, causing the average reduce time, and hence overall net time, to go up.

As the reported net times for Hive and Pig are much higher than for sequential evaluation in Gumbo (SEQ), we conclude that Pig and Hive, with default settings, are unfit for parallel evaluation of BSGF queries. For this reason, we restrict our attention to Gumbo in the following sections.

Large Queries. Next, we compare the evaluation of two larger BSGF queries B1 and B2 from Table 5.6. The results are shown in Figure 5.6. Query B1 is a conjunctive BSGF query featuring a high number of atoms. Its structure ensures a deep sequential plan that results in a high net time for SEQ. We find that PAR only takes 22% of the net time, which shows that parallel query plans can yield significant improvements in this aspect. Conversely, PAR takes up 261% more total time than SEQ, as the latter is more efficient in pruning the data at each step. Here, GREEDY is able to successfully parallelize query execution without sacrificing total time. Indeed, GREEDY exhibits a net time comparable to that of PAR and a total time comparable to that of SEQ.

Query B2 consists of a large Boolean combination and is called the *uniqueness query*. This query returns the tuples that can be connected to precisely one of the conditional relations through a given attribute. The number of distinct conditional atoms is limited, and the disjunction at the highest level makes it possible to evaluate the four conjunctive subexpressions in parallel using SEQ. Still, we find that the net time of PAR improves that of SEQ by 66%. As PAR only needs to calculate the result of four semi-join queries in its first round, we also find a reduction of 57% in total time. The GREEDY strategy further reduces both numbers.

Finally, for query B2, a 1-round evaluation denoted again by 1-ROUND can be considered, as only one key is used in the conditional atoms. This evaluation strategy brings down both the net and total time of SEQ by more than 80%.

Cost Model. As explained in Section 4.4, the major difference between our cost model and that of Wang et al. [163] (referred to as $\mathcal{C}_{gumbo}$ and $\mathcal{C}_{wang}$, respectively, from here onward) concerns identifying the individual map cost contributions of the input relations. For queries where the map input/output ratio differs greatly among the input relations, we notice a vast improvement for the GREEDY strategy. We illustrate this using the following query:

$$\begin{aligned} R(x,y,z,w) \ltimes\ & S_1(\bar{x}_1, c) \wedge \ldots \wedge S_1(\bar{x}_{12}, c) \wedge \\ & S_2(\bar{x}_1, c) \wedge \ldots \wedge S_2(\bar{x}_{12}, c) \wedge \\ & S_3(\bar{x}_1, c) \wedge \ldots \wedge S_3(\bar{x}_{12}, c) \wedge \\ & S_4(\bar{x}_1, c) \wedge \ldots \wedge S_4(\bar{x}_{12}, c), \end{aligned}$$

where $\bar{x}_1, \ldots, \bar{x}_{12}$ are all distinct keys and c is a constant that filters out all tuples from $S_1, \ldots, S_4$. The results for evaluating this query using GREEDY with $\mathcal{C}_{gumbo}$ and $\mathcal{C}_{wang}$ are unmistakable: $\mathcal{C}_{gumbo}$ provides a 43% reduction in total time and a 71% reduction in net time. The reason is that $\mathcal{C}_{wang}$ does not discriminate between different input relations as it averages out the intermediate data and therefore fails to account for the high number of map-side merges and the accompanying increase in total time. Note that these merges also have an adverse effect on the net time, which is reflected by our measurements.

For queries A1–A5 and B1–B2, where input relations have a contribution to map output that is proportional to their input size, we find that both cost models behave similarly. Indeed, when comparing two random jobs from our experiments, the cost models are capable of correctly identifying the highest cost job in 72.28% ($\mathcal{C}_{gumbo}$) and 69.37% of the cases ($\mathcal{C}_{wang}$). The slight advantage for $\mathcal{C}_{gumbo}$ can be attributed to the small set of jobs having unbalanced map contributions. Hence, we find that $\mathcal{C}_{gumbo}$ provides a more robust cost estimation as it can isolate input relations that have a non-proportional map output contribution, while it automatically resorts to $\mathcal{C}_{wang}$ in the case of an equal contribution.

Conclusion. We conclude that parallel evaluation effectively lowers net times, at the cost of higher total times. The GREEDY strategy, backed by an updated cost model, successfully manages to bring down the total times of parallel evaluation, especially in the presence of commonalities among the atoms of BSGF queries. For larger queries, total times similar to SEQ are obtained. Finally, Gumbo outperforms Pig and Hive in all aspects when it comes to parallel evaluation of BSGF queries.

5.5.4 SGF Queries

In this section, we show that the algorithm GREEDY-SGF succeeds in lowering the total time while avoiding significant loss in net time. Figure 5.8 gives an overview of the type of queries that are used. Results are depicted in Figure 5.7. Note that these queries all exhibit different properties. Queries C1 and C2 both contain a set of SGF queries where a number of atoms overlap;

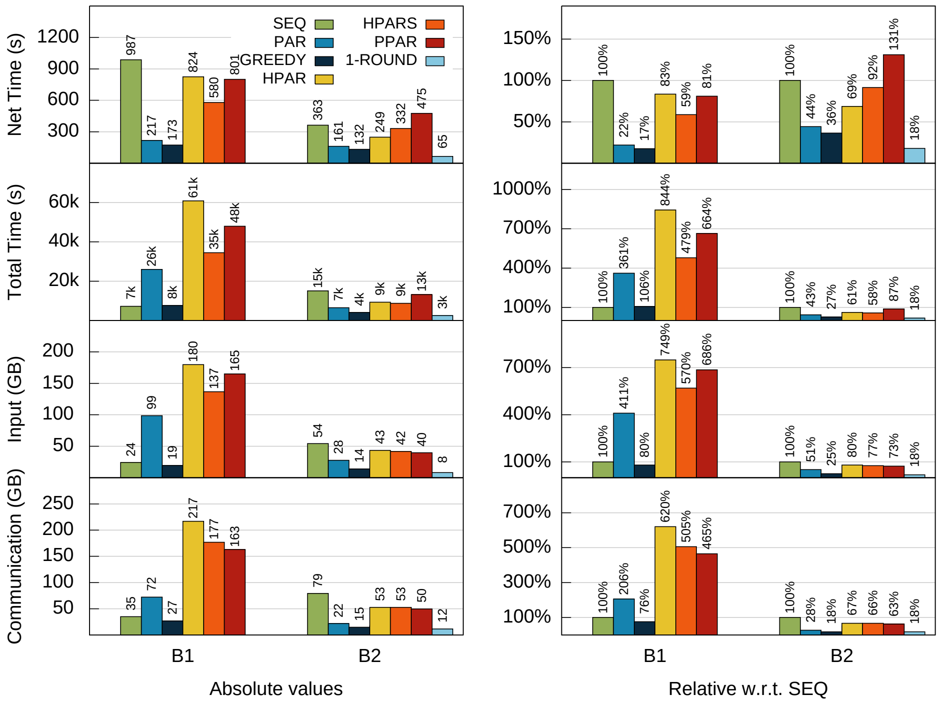


Figure 5.6: *Results for the large BSGF queries in Table 5.6.*

query C3 is a complex query that contains a multitude of different atoms; query C4 consists of two levels and many overlapping atoms.

We consider the following evaluation strategies in Gumbo: (i) sequentially, i.e., one at a time, evaluating all BSGF queries in a bottom-up fashion (SEQUNIT); (ii) evaluating all BSGF queries in a bottom-up fashion level by level where queries on the same level are executed in parallel (PARUNIT); and, (iii) using the greedily computed multiway topological sort combined with Greedy-BSGF (Greedy-SGF). Note that in both SEQUNIT and PARUNIT all semi-joins are evaluated using separate jobs. For all tests conducted here, we found that Greedy-SGF yields multiway topological sorts that are identical to the optimal topological sort (computed trough brute-force methods). Hence, we omit the results for the optimal plans.

Similar to our observations for BSGF queries, we find that full sequential evaluation (SEQUNIT) results in the largest net times. Indeed, PARUNIT exhibits 55% lower net times on average. We also observe that PARUNIT exhibits significantly larger total times than SEQUNIT for queries C1 and C2, while this is not the case for C3 and C4. The reason is that for C3 and C4, queries on the same level still share common characteristics, leading to a lower number of distinct semi-joins.

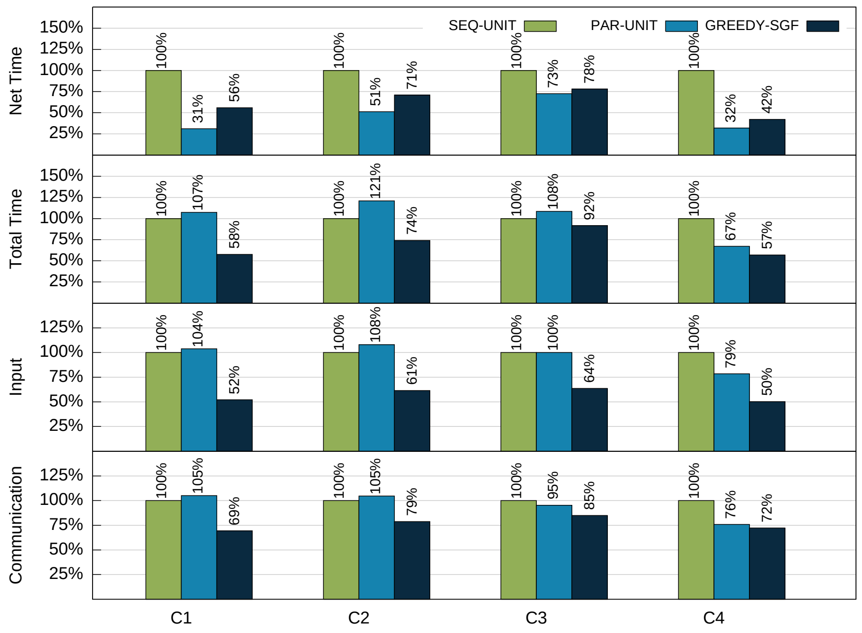


Figure 5.7: *Results for the SGF queries in Figure 5.8. Values are relative to* SEQUNIT*.*

For Greedy-SGF, we find that it exhibits net times that are, on average, 42% lower than SEQUNIT, while still being 29% higher than PARUNIT. The main reason for this is the fact that Greedy-SGF aims to minimize total time, and may introduce extra levels in the MR query plan to obtain this goal. Indeed, we find that total times are down 27% w.r.t. SEQUNIT, and 29% w.r.t. PARUNIT.

Finally, for these queries, the absolute savings in net time that can be obtained by using Greedy-SGF over SEQUNIT range from 115s to 737s, far outweighing the overhead cost of calculating the query plan itself, which typically takes around 10s (sampling included).[43] Hence, we conclude that Greedy-SGF provides an evaluation strategy for SGF queries that manages to bring down the total time (and hence, the resource cost) of parallel query plans, while still exhibiting low net times when compared to sequential approaches.

[43]Note that query plan calculation and sampling are not executed on the compute nodes themselves.

$$\begin{tikzcd}
\boxed{Z_3(x) \coloneqq G(\bar{x}) \ltimes Z_1(z) \lor Z_1(w)} & & \boxed{Z_4(x) \coloneqq H(\bar{x}) \ltimes Z_3(z) \lor Z_3(w)} \\
\boxed{Z_1(x) \coloneqq R(\bar{x}) \ltimes S(x) \land S(y)} \arrow[u] & \boxed{Z_2(x) \coloneqq G(\bar{x}) \ltimes T(x) \land T(y)} & \boxed{Z_3(x) \coloneqq H(\bar{x}) \ltimes U(x) \land U(y)} \arrow[u]
\end{tikzcd}$$

(a) *Query Set C1*

$$\begin{tikzcd}
\boxed{Z_4(\bar{x}) \coloneqq G(\bar{x}) \ltimes Z_1(x) \land Z_1(y)} & \boxed{Z_5(\bar{x}) \coloneqq H(\bar{x}) \ltimes Z_2(x) \land Z_2(y)} & \boxed{Z_6(\bar{x}) \coloneqq R(\bar{x}) \ltimes Z_3(x) \land Z_3(y)} \\
\boxed{Z_1(\bar{x}) \coloneqq R(\bar{x}) \ltimes S(x) \land S(y)} \arrow[u] & \boxed{Z_2(\bar{x}) \coloneqq G(\bar{x}) \ltimes T(x) \land T(y)} \arrow[u] & \boxed{Z_3(\bar{x}) \coloneqq H(\bar{x}) \ltimes U(x) \land U(y)} \arrow[u]
\end{tikzcd}$$

(b) *Query Set C2*

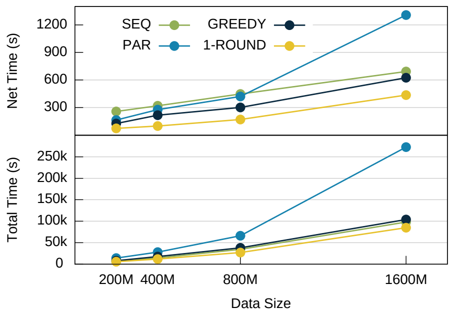
$$\begin{tikzcd}
 & \boxed{Z_{31}(z) \coloneqq I(\bar{x}) \ltimes Z_{22}(x) \land T(x) \land V(y)} & \\
\boxed{Z_{21}(z) \coloneqq G(\bar{x}) \ltimes Z_{11}(x) \land U(y)} \arrow[ur] & & \boxed{Z_{23}(z) \coloneqq R(\bar{x}) \ltimes U(x) \land T(y) \land V(z) \land Z_{13}(w)} \arrow[ul] \\
 & \boxed{Z_{22}(z) \coloneqq H(\bar{x}) \ltimes U(y) \lor V(y) \land Z_{12}(x)} \arrow[uu] & \\
\boxed{Z_{11}(z) \coloneqq R(\bar{x}) \ltimes S(x) \land T(y)} \arrow[uu] & \boxed{Z_{12}(z) \coloneqq R(\bar{x}) \ltimes T(y)} \arrow[u] & \boxed{Z_{13}(z) \coloneqq I(\bar{x}) \ltimes \neg S(w)} \arrow[uu]
\end{tikzcd}$$

(c) *Query C3*

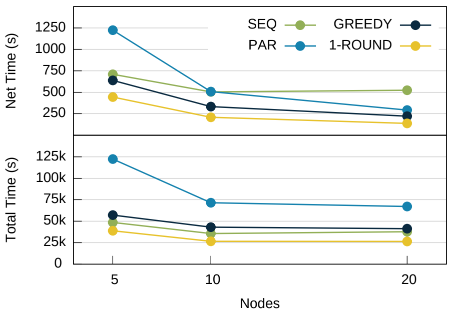
$$\begin{tikzcd}
 & & \boxed{Z_{21}(\bar{x}) \coloneqq H(\bar{x}) \ltimes Z_{11}(x) \lor Z_{12}(y) \lor Z_{23}(z) \lor Z_{24}(w)} & & \\
\boxed{Z_{12}(y) \coloneqq R(\bar{x}) \ltimes U(z) \lor S(x)} \arrow[urr] & & & & \boxed{Z_{14}(y) \coloneqq G(\bar{x}) \ltimes S(z) \lor U(x)} \arrow[ull] \\
 & \boxed{Z_{11}(y) \coloneqq R(\bar{x}) \ltimes S(x) \lor T(y)} \arrow[uur] & & \boxed{Z_{13}(y) \coloneqq G(\bar{x}) \ltimes U(x) \lor V(y)} \arrow[uul] &
\end{tikzcd}$$

(d) *Query C4*

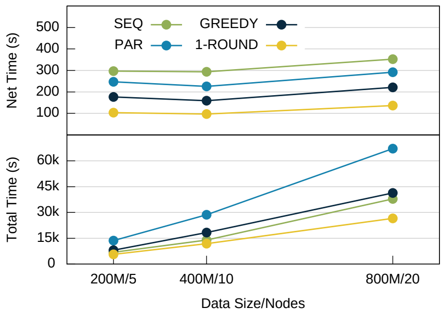
Figure 5.8: *Queries used in the SGF experiment. Each node represents one BSGF subquery ($\bar{x} = x, y, z, w$).*



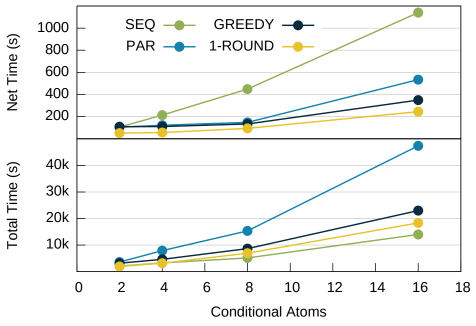
(a) *Varying data size (10 nodes).*



(b) *Varying cluster size (800M tuples).*



(c) *Varying data and cluster size.*



(d) *Varying the number of atoms.*

Figure 5.9: *Results for data and node variation tests in Gumbo.*

5.5.5 System Characteristics

In these final experiments, we study the behavior of Gumbo under the change of several parameters. We study the effect of growing data size, cluster size, query size, and selectivity. We consider queries similar to A3 as these allow us to (*i*) incorporate 1-ROUND in our results, and (*ii*) gain insights in the best possible scenario. Aside from GREEDY and 1-ROUND being faster than the other strategies, all conclusions hold for the other query types as well.

Data & Cluster Size. Figures 5.9a to 5.9c show the result of evaluating A3 using SEQ, PAR, GREEDY and 1-ROUND under the presence of variable data and cluster size. We summarize the most important observations:

1. in all scenarios, 1-ROUND performs best in terms of net and total time;
2. due to its lack of grouping, PAR needs a high number of mappers, which at some point exceeds the capacity of the cluster. This causes a large increase in net and total time, an effect that can be seen in Figure 5.9a.
3. with regard to net time, adding more nodes is very effective for the parallel strategies PAR, GREEDY and 1-ROUND (see Figure 5.9b). In contrast, adding more nodes does not improve SEQ substantially after some point. The reason for this is that the number of map tasks is lower in SEQ, as they are spread across multiple rounds. Therefore, less nodes are required to run all tasks in parallel and no improvement can be seen beyond 10 nodes. If desired, all strategies can be further parallelized by lowering the split size.
4. when scaling data and cluster size at the same time (see Figure 5.9c), all strategies are able to maintain their net times in the presence of an increasing total time. But, we find that w.r.t. total time PAR is the most sensitive for this type of scaling.

Query Size. We consider a set of queries similar to A3, where the number of conditional atoms ranges from 2 to 16. Results are depicted in Figure 5.9d. With regard to net time, we find that SEQ shows an increase in net time that is strongly related to the query size, while PAR, GREEDY and 1-ROUND are less affected. For total time, we observe the converse for PAR, as this strategy cannot benefit from the packing optimization in the same way as GREEDY and 1-ROUND can. This again shows that the GREEDY approach, which incorporates grouping, offers the best properties of both sequential and parallel evaluation strategies for this type of queries.

Selectivity. For a conditional relation, we define its *selectivity rate* as the percentage of guard tuples it matches. We tested queries A1–A3 for selectivity

	Net time			Total time		
	A1	A2	A3	A1	A2	A3
SEQ	10%	9%	8%	79%	95%	88%
PAR	33%	46%	69%	41%	47%	58%
GREEDY	23%	30%	13%	45%	57%	15%

Table 5.7: *Increase in net and total time when increasing the selectivity rate from 0.1 to 0.9 for queries A1–A3.*

rates 0.1 (high selectivity), 0.3, 0.5, 0.7 and 0.9 (low selectivity). The increase in net time and total time between selectivity rates 0.1 and 0.9 is summarized in Table 5.7. In general, we find that the selectivity has the most influence on the net times of PAR and GREEDY, and on the total times of SEQ. This can be explained by the structure of the query plans: at high selectivity rates, sequential query plans loose their filtering advantage. The jobs typically use less resources and can therefore increase their resource usage (reflected in total time) without sacrificing net time. For parallel query plans, the selectivity affects both metrics mainly because a low selectivity rate reduces the input to the EVAL job when only confirm messages are used. Finally, we observe that the filtering characteristics of SEQ disappear in the presence of low selectivity data, causing total times to become comparable to GREEDY for queries where packing is possible, such as A3. This can be explained by GREEDY being less sensitive to selectivity for queries where conditional atoms share a common join key, making an effective compression of intermediate data possible through packing.

5.6 Discussion

We have shown that, when compared to sequential evaluation, naive parallel evaluation of semi-join and (B)SGF queries can greatly reduce the net time of query execution. But, as expected, this generally comes at a cost of an increased total time. We presented several methods that aim to reduce the total cost (total time) of parallel MR query plans, while at the same time avoiding a too high increase in net time. The greedy approach was shown to be effective for evaluating (B)SGF queries in practice through several experiments that were performed using our own implementation called Gumbo. For certain classes of queries, our approach makes it even possible to evaluate (B)SGF queries in parallel with a total time similar to that of sequential evaluation. We have also shown that the profuse number of optimizations that are offered in Gumbo allow it to outperform Pig and Hive in several aspects. Detailed study of these optimization techniques could be the topic of future work.

A more general observation that can be derived from the experiments is that the input is not always the most important factor for determining the total or net time. Other factors such as communication cost, replication rate, etc. have to be taken into account. This backs up the choice of our cost model in Section 4.4.3 and the study of the decision problem in Section 4.6.1. Although, for completeness, we also provided a study of minimizing the number of input reads in Section 4.6.2.

We note that the techniques introduced in this chapter generalize to any map/reduce framework (as, e.g., [165]) given an appropriate adaptation of the cost model. Also, a large number of the proposed optimizations can be directly applied to any MR job.

Even though the algorithms in this chapter do not directly take skew into account, the presented framework can readily be adapted to do so when information on so-called heavy hitters is available or can be computed at the expense of an additional round (see, e.g., [1, 142, 146, 156]).

6 Discussion & Future Work

At the start of this work, we identified data mining as the core component of the knowledge discovery process and focused on two subtopics: pattern mining and big data query optimization. In regard to pattern mining, we studied the problem of enumerating all strings in a context-free language as well as the XML key mining problem which requires finding all keys that hold in an XML document in the presence of a schema (XSD). For Big data query optimization, we focused on optimizing the evaluation of parallel MapReduce query plans for multi-semi-join (or SGF) queries. In this final chapter, we attempt to draw some general conclusions and indicate possible directions for future work which complement the more topic-specific directions that were already presented at the end of each chapter.

Candidate Generation

For candidate generation, we studied a way of characterizing the generation speed of enumeration schemes: the incremental polynomial time property, which provides an upper bound on the time between two pattern outputs. We found that simple methods such as the naive concatenation scheme for context-free languages already abide by this property. In practice, a semi-naive approach is preferred in conjunction with a non-ambiguous grammar, as this allows for more accurate bounds to be obtained. As indicated in Chapter 2, the results could be generalizable to more complex types of pattern languages, such as graph languages. With the advent of social networks, this is a topic that we think deserves some special attention. We provide a mechanism for describing graph languages in Costa Florêncio et al. [61]. Here, the enumeration scheme could be used in conjunction with a matching algorithm such as subgraph isomorphism [158], or a more flexible notion of simulation [77, 79, 102, 121, 122] to find patterns from a predefined graph pattern language in large data graphs. In the context of distributed systems, it is also worth looking at the parallelization possibilities for this and other generation schemes and the type of guarantees we can expect in this setting. A practical evaluation and comparison of different schemes would then be advised.

A data-driven candidate generation procedure was developed for the XML Key Mining problem. This is an example where the naive search space is too large, requiring smart ways of exploring and pruning. Indeed, the naive enumeration of all possible XPath expressions would be too time-consuming, even if we could provide guarantees on the time between two candidate expressions. The proposed algorithm leverages both existing and new techniques to accomplish this: prefix trees, an efficient ordering of path expressions, levelwise search and relational FD mining techniques. We also indicated that extending our algorithm with a notion of approximate keys might be an interesting path to follow, as this could easily be plugged into our mining framework. Another interesting direction would be to transfer the notion of keys to the slightly different semi-structured data format called JavaScript Object Notation, or JSON [47]. This data format has taken an important place next to XML in the last few years, and a schema language has also been developed for it [113]. Together with the fact that many No-SQL solutions offer the storage of JSON documents, a key mining algorithm may provide additional insight in the data, which could lead to more advanced optimization techniques (e.g., indexing). Another interesting direction in this research is situated within the topic of graph keys, which can be used in, e.g., entity resolution [78]. Indeed, several methods are already in place, but it should be studied whether the XML key definition can be ported to graphs in a way that allows the mining algorithm to be reused.

Quality

In regard to data quality, we have devised several measures for XML Keys that allow us to decide whether a key should be considered interesting or not. These quality measures have the potential to positively affect the performance as they might allow for early pruning of the search space (e.g., boundedness). Hence, it is important to understand the meaning of these measures and to understand the impact of incorporating them in a key mining algorithm. Although incorporating them is often easy, this can also cause a major slowdown of the data mining process either due to the hard complexity class of the problem, or due to inefficient implementations. Consistency is an example of the latter, as we found that a large part of the running time of our mining algorithm can be attributed to the consistency check, despite the fact that this was proven to be in PTIME.

The cardinality estimation problem complexities, which were derived for different fragments of XPath and lie at the basis of the complexity of the consistency quality measure, might find use in areas outside XML or XML key mining. These results have already been transferred to similar string problems in Arenas et al. [25].

Querying Big Data

We believe that our extensive study of SGF query optimization and evaluation is a major contribution in cost-based optimization techniques for modern Big data systems. As an open-source implementation of our methods is available [66], we hope that our results will be incorporated into newly developed optimizers for established MapReduce-based systems such as Hive and Pig, as we have shown that there are possibilities for achieving a great speedup and reduction in resource requirements and/or costs.

The Big Picture

The growing interest in Big data has led to the development of new technologies and has boosted several research areas. The number of "Data Scientist" positions that are available today is still growing, indicating a high interest from the industry as well. This has caused the community to look for what is beyond MapReduce: new technologies and new data modeling methods are needed to bring order to the chaos of rapidly emerging solutions. Indeed, a lot of advanced frameworks under development and try to overcome the shortcomings of the MapReduce computation model (e.g., the multitude synchronization steps, its inability to operate on streaming data, lack of iteration support). A few popular examples include Spark [23, 170], HaLoop [49], Tez [24, 147], and Google Dataflow [14]. The latter is now available as the open-source Apache Beam [18] and aims to provide methods to model both batch and streaming data. Babu and Herodotou [30] give a thorough overview of the existing systems and the upcoming challenges.

As the results presented in this work are both fundamental and practical in nature, we hope that they will contribute to the development of this next generation of Big data systems. While it is difficult to forsee all possible uses, our results could be used as, for example, part of data mining toolkits, indexing mechanisms or optimization techniques.

For each of the topics we addressed in this work, fundamental theoretical problems were studied and used to devise practical algorithms. Both the theoretical background and the transfer of this knowledge to a practical setting, as well as the reuse of existing results are what the author considers to be one of the core principles of computer science, which are paramount in the quickly evolving Big data community.

Bibliography

[1] A. Gates, J. Dai, and T. Nair. Apache Pig's Optimizer. *IEEE Data Engineering Bulletin*, 36(1):34–45, 2013.

[2] S. Abiteboul, Y. Amsterdamer, D. Deutch, T. Milo, and P. Senellart. Finding Optimal Probabilistic Generators for XML Collections. In *Proceedings of the 15th International Conference on Database Theory (ICDT 2012)*. ACM, 2012, pages 127–139. DOI: 10.1145/2274576.2274591.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] M. Ackerman and E. Mäkinen. Three New Algorithms for Regular Language Enumeration. In *Proceedings of the 15th Annual International Conference on Computing and Combinatorics (COCOON 2009)*. Springer-Verlag, 2009, pages 178–191. DOI: 10.1007/978-3-642-02882-3_19.

[5] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating Subgraph Instances Using MapReduce. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE 2013)*. IEEE, 2013. DOI: 10.1109/icde.2013.6544814.

[6] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. Map-Reduce Extensions and Recursive Queries. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT 2011)*. ACM, 2011. DOI: 10.1145/1951365.1951367.

[7] F. N. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A Multiround Join Algorithm in MapReduce. 2014. arXiv: 1410.4156.

[8] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Vision Paper: Towards an Understanding of the Limits of Map-Reduce Computation. 2012. arXiv: 1204.1754.

[9] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT 2010)*. ACM, 2010, pages 99–110. DOI: `10.1145/1739041.1739056`.

[10] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, 2011. DOI: `10.1109/tkde.2011.47`.

[11] F. N. Afrati and J. D. Ullman. Transitive Closure and Recursive Datalog Implemented on Clusters. In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT 2012)*. ACM, 2012. DOI: `10.1145/2247596.2247613`.

[12] F. N. Afrati, J. D. Ullman, and A. Vasilakopoulos. Handling Skew in Multiway Joins in Parallel Processing. 2015. arXiv: `1504.03247`.

[13] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994)*. Morgan Kaufmann, 1994, pages 487–499.

[14] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB 2015)*, 8(12):1792–1803, 2015. DOI: `10.14778/2824032.2824076`.

[15] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS 1967)*. ACM, 1967, pages 483–485. DOI: `10.1145/1465482.1465560`.

[16] H. Andréka, I. Németi, and J. van Benthem. Modal Languages and Bounded Fragments of Predicate Logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998. DOI: `10.1023/a:1004275029985`.

[17] C. Antunes and A. Oliveira. Using Context-free Grammars to Constrain Apriori-based Algorithms for Mining Temporal Association Rules. In *Proceedings of the Workshop on Temporal Data Mining in the ACM International Conference on Knowledge Discovery and Data Mining (TDM@KDD 2002)*, 2002, pages 11–24.

[18] Apache Beam (incubating). URL: `http://beam.apache.org` (visited on 08/21/2016).

[19] Apache Calcite - Dynamic Data Management Framework. URL: http://calcite.apache.org (visited on 08/21/2016).

[20] Apache CouchDB. URL: https://couchdb.apache.org (visited on 08/21/2016).

[21] Apache Hive. URL: http://hive.apache.org (visited on 08/21/2016).

[22] Apache Pig. URL: https://pig.apache.org (visited on 08/21/2016).

[23] Apache Spark - Lightning-Fast Cluster Computing. URL: http://spark.apache.org (visited on 08/21/2016).

[24] Apache Tez - Welcome to Apache Tez. URL: https://tez.apache.org (visited on 08/21/2016).

[25] M. Arenas, J. Daenen, F. Neven, M. Ugarte, J. Van den Bussche, and S. Vansummeren. Discovering XSD Keys from XML Data. *ACM Transactions on Database Systems (TODS)*, 39(4):28:1–28:49, 2014. DOI: 10.1145/2638547.

[26] M. Arenas, J. Daenen, F. Neven, M. Ugarte, J. Van den Bussche, and S. Vansummeren. Discovering XSD keys from XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, 2013, pages 61–72. DOI: 10.1145/2463676.2463705.

[27] M. Arenas, W. Fan, and L. Libkin. What's Hard about XML Schema Constraints? In *Database and Expert Systems Applications: 13th International Conference (DEXA 2002)*. Springer, 2002, pages 269–278. DOI: 10.1007/3-540-46146-9_27.

[28] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*. ACM, 2015, pages 1383–1394. DOI: 10.1145/2723372.2742797.

[29] D. B. Arnold and M. R. Sleep. Uniform Random Generation of Balanced Parenthesis Strings. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):122–128, 1980. DOI: 10.1145/357084.357091.

[30] S. Babu and H. Herodotou. Massively Parallel Databases and MapReduce Systems. *Foundations and Trends in Databases*, 5(1), 2013. DOI: 10.1561/1900000036.

[31] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of Bisimulation Equivalence for Process Generating Context-free Languages. *Journal of the ACM*, 40(3):653–682, 1993. DOI: 10.1145/174130.174141.

[32] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. *ACM SIGMOD Record*, 15(2):16–52, 1986. DOI: 10.1145/16856.16859.

[33] D. Barbosa and A. Mendelzon. Finding ID Attributes in XML Documents. In *Database and XML Technologies*. Springer, 2003, pages 180–194. DOI: 10.1007/978-3-540-39429-7_12.

[34] P. Beame, P. Koutris, and D. Suciu. Communication Steps for Parallel Query Processing. In *Proceedings of the 32nd Symposium on Principles of Database Systems (PODS 2013)*. ACM, 2013. DOI: 10.1145/2463664.2465224.

[35] P. Beame, P. Koutris, and D. Suciu. Skew in Parallel Query Processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2014)*. ACM, 2014. DOI: 10.1145/2594538.2594558.

[36] P. A. Bernstein and D.-M. W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, 28(1):25–40, 1981. DOI: 10.1145/322234.322238.

[37] P. A. Bernstein and N. Goodman. The Power of Inequality Semijoins. *Information Systems*, 6(4):255–265, 1981. DOI: 10.1016/0306-4379(81)90002-8.

[38] G. J. Bex. Discovering Structure in Semi-Structured Data. PhD thesis. Hasselt University, 2008.

[39] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML Schema. In *Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD 2009)*. ACM, 2009. DOI: 10.1145/1559845.1559922.

[40] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data. *ACM Transactions on the Web*, 4(4):1–32, 2010. DOI: 10.1145/1841909.1841911.

[41] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of Concise Regular Expressions and DTDs. *ACM Transactions on Database Systems*, 35(2):1–47, 2010. DOI: 10.1145/1735886.1735890.

[42] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema Definitions from XML Data. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)*. VLDB Endowment, 2007, pages 998–1009.

[43] G. J. Bex, F. Neven, and S. Vansummeren. SchemaScope. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*. ACM, 2008, pages 1259–1262. DOI: `10.1145/1376616.1376750`.

[44] D. Bitton, J. Millman, and S. Torgersen. A Feasibility and Performance Study of Dependency Inference. In *Proceedings of the International Conference on Data Engineering (ICDE 1989)*, 1989, pages 635–641. DOI: `10.1109/icde.1989.47271`.

[45] H. Björklund, W. Martens, and T. Schwentick. Validity of Tree Pattern Queries with Respect to Schema Information. In *Mathematical Foundations of Computer Science (MFCS 2013)*. Volume 8087. Springer, 2013, pages 171–182. DOI: `10.1007/978-3-642-40313-2_17`.

[46] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proceedings of the 2010 International Conference on Management of data (SIGMOD 2010)*. ACM, 2010. DOI: `10.1145/1807167.1807273`.

[47] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. 2014. URL: `https://tools.ietf.org/html/rfc7159` (visited on 08/21/2016).

[48] A. Brüggemann-Klein and D. Wood. One-Unambiguous Regular Languages. *Information and Computation*, 140(2):229–253, 1998. DOI: `10.1006/inco.1997.2688`.

[49] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, 3(1):285–296, 2010. DOI: `10.14778/1920841.1920881`.

[50] P. Buneman. Semistructured Data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1997)*. ACM, 1997, pages 117–121. DOI: `10.1145/263661.263675`.

[51] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In *Proceedings of the 6th International Conference on Database Theory (ICDT 1997)*. Springer-Verlag, 1997, pages 336–350.

[52] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML. *Computer Networks*, 39(5):473–487, 2002. DOI: 10.1016/s1389-1286(02)00223-2.

[53] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Reasoning about Keys for XML. *Information Systems*, 28(8):1037–1063, 2003. DOI: 10.1016/s0306-4379(03)00028-0.

[54] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990. DOI: 10.1007/978-3-642-83952-8.

[55] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1998)*. ACM, 1998. DOI: 10.1145/275487.275492.

[56] S. Chu, M. Balazinska, and D. Suciu. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*. ACM, 2015. DOI: 10.1145/2723372.2750545.

[57] J. Clark and S. DeRose. XML Path Language (XPath). URL: http://www.w3.org/TR/xpath (visited on 08/23/2016).

[58] J. Clark and M. Murata. Relax NG Specification. URL: http://www.relaxng.org/spec-20011203.html (visited on 08/21/2016).

[59] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970. DOI: 10.1145/362384.362685.

[60] C. Costa Florêncio, J. Daenen, J. Ramon, J. Van den Bussche, and D. Van Dyck. Naive Infinite Enumeration of Context-free Languages in Incremental Polynomial Time. *Journal of Universal Computer Science*, 21(7):891–911, 2015. DOI: 10.3217/jucs-021-07-0891.

[61] C. Costa Florêncio, J. Ramon, J. Daenen, J. Van den Bussche, and D. Van Dyck. Context-free Graph Grammars as a Language-Bias Mechanism for Graph Pattern Mining. In *Proceedings of the 7th International Workshop on Mining and Learning with Graphs (MLG 2009)*. Extended Abstract, 2009.

[62] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press, 1st edition, 2012.

[63] A. Cron, H. L. Nguyen, and A. Parameswaran. Big Data. *XRDS*, 19(1):7–8, 2012. DOI: 10.1145/2331042.2331045.

[64] J. Daenen, F. Neven, and T. Tan. Gumbo: Guarded Fragment Queries over Big Data. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT 2015)*, 2015, pages 521–524. DOI: `10.5441/002/edbt.2015.48`.

[65] J. Daenen, F. Neven, T. Tan, and S. Vansummeren. Parallel Evaluation of Multi-Semi-Joins. *Proceedings of the VLDB Endowment*, 9(10):732–743, 2016. DOI: `10.14778/2977797.2977800`.

[66] J. Daenen and T. Tan. Gumbo v0.4. 2016. URL: `http://dx.doi.org/10.5281/zenodo.51517`.

[67] A. De Mauro, M. Greco, and M. Grimaldi. A Formal Definition of Big Data Based on Its Essential Features. *Library Review*, 65(3):122–135, 2016. DOI: `10.1108/LR-06-2015-0061`.

[68] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference (TACAS 2008)*. Springer-Verlag, 2008, pages 337–340. DOI: `10.1007/978-3-540-78800-3_24`.

[69] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. USENIX Association, 2004, pages 10–10.

[70] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. DOI: `10.1145/1327452.1327492`.

[71] Document Structure Description (DSD). URL: `http://www.brics.dk/DSD` (visited on 08/21/2016).

[72] P. Dömösi. Unusual Algorithms for Lexicographical Enumeration. *Acta Cybernetica*, 14(3):461–468, 2000.

[73] Y. Dong. Linear Algorithm for Lexicographic Enumeration of CFG Parse Trees. *Science in China Series F: Information Sciences*, 52(7):1177–1202, 2009. DOI: `10.1007/s11432-009-0132-7`.

[74] A. G. Duncan and J. Hutchinson. Using Attributed Grammars to Test Designs and Implementations. In *Proceedings of the 5th International Conference on Software Engineering (ICSE 1981)*. IEEE Press, 1981, pages 170–178.

[75] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and Adaptive Online Joins. *Proceedings of the VLDB Endowment*, 7(6):441–452, 2014. DOI: `10.14778/2732279.2732281`.

[76] S. Fajt, I. Mlynkova, and M. Necasky. On Mining XML Integrity Constraints. In *Sixth IEEE International Conference on Digital Information Management (ICDIM 2011)*. IEEE, 2011, pages 23–29. DOI: 10.1109/icdim.2011.6093314.

[77] W. Fan. Graph Pattern Matching Revised for Social Network Analysis. In *Proceedings of the 15th International Conference on Database Theory (ICDT 2012)*. ACM, 2012, pages 8–21. DOI: 10.1145/2274576.2274578.

[78] W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for Graphs. *Proceedings of the VLDB Endowment*, 8(12):1590–1601, 2015. DOI: 10.14778/2824032.2824056.

[79] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph Pattern Matching. *Proceedings of the VLDB Endowment*, 3(1-2):264–275, 2010. DOI: 10.14778/1920841.1920878.

[80] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. *Journal of the ACM*, 49(3):368–406, 2002. DOI: 10.1145/567112.567117.

[81] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic Average-Case Analysis of Algorithm. *Theoretical Computer Science*, 79(1):37–109, 1991. DOI: 10.1016/0304-3975(91)90145-R.

[82] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511801655.

[83] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A Calculus for the Random Generation of Labelled Combinatorial Structures. *Theoretical Computer Science*, 132(1):1–35, 1994. DOI: 10.1016/0304-3975(94)90226-7.

[84] J. Flum, M. Frick, and M. Grohe. Query Evaluation via Tree-decompositions. *Journal of the ACM*, 49(6):716–752, 2002. DOI: 10.1145/602220.602222.

[85] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2nd edition, 2008.

[86] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[87] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning Document Type Descriptors from XML Document Collections. *Data Mining and Knowledge Discovery*, 7(1):23–56, 2003. DOI: 10.1023/a:1021560618289.

[88] D. Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13, 2005. DOI: 10.1109/MC.2005.160.

[89] V. Gore, M. Jerrum, S. Kannan, Z. Sweedyk, and S. Mahaney. A Quasi-polynomial-time Algorithm for Sampling Words from a Context-Free Language. *Information and Computation*, 134(1):59–74, 1997. DOI: 10.1006/inco.1997.2621.

[90] E. Grädel. Description Logics and Guarded Fragments of First Order Logic. In *Proceedings of the 1998 International Workshop on Description Logics (DL 1998)*, 1998.

[91] G. Grahne and J. Zhu. Discovering Approximate Keys in XML data. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management (CIKM 2002).* ACM Press, 2002, pages 453–460. DOI: 10.1145/584792.584867.

[92] S. Grijzenhout and M. Marx. The Quality of the XML Web. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM 2011).* ACM, 2011, pages 1719–1724. DOI: 10.1145/2063576.2063824.

[93] S. Grijzenhout and M. Marx. University of Amsterdam XML Web Collection. 2010. URL: http://data.politicalmashup.nl/sgrijzen/xmlweb/ (visited on 08/21/2016).

[94] Hadoop. URL: http://hadoop.apache.org (visited on 08/21/2016).

[95] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, 1st edition, 2010.

[96] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria Big Data Management Service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD 2014).* ACM, 2014, pages 881–884. DOI: 10.1145/2588555.2594530.

[97] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns Without Candidate Generation. *ACM SIGMOD Record*, 29(2):1–12, 2000. DOI: 10.1145/335191.335372.

[98] S. Hartmann and S. Link. Efficient Reasoning about a Robust XML Key Fragment. *ACM Transactions on Database Systems (TODS)*, 34(2):1–33, 2009. DOI: 10.1145/1538909.1538912.

[99] M. A. H. Hassan and M. Bamha. Semi-Join Computation on Distributed File Systems Using Map-Reduce-Merge Model. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010).* ACM, 2010, pages 406–413. DOI: 10.1145/1774088.1774174.

[100] HBase - Apache Software Foundation project home page. URL: https://hbase.apache.org/ (visited on 08/21/2016).

[101] S. Henderson, J. Kolb, B. Lehman, and J. Montague. Trend Detection in Social Data. Whitepaper. 2014. URL: https://github.com/jeffakolb/Gnip-Trend-Detection/raw/master/paper/trends.pdf (visited on 08/21/2016).

[102] M. Henzinger, T. Henzinger, and P. Kopke. Computing Simulations on Finite and Infinite Graphs. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 1995. DOI: 10.1109/sfcs.1995.492576.

[103] H. Herodotou. Hadoop Performance Models. 2011. arXiv: 1106.0940.

[104] M. Hilbert. Big Data for Development: A Review of Promises and Challenges. *Development Policy Review*, 34(1):135–174, 2015. DOI: 10.1111/dpr.12142.

[105] Hive 0.14 Cost Based Optimizer (CBO) Technical Overview. URL: http://hortonworks.com/blog/hive-0-14-cost-based-optimizer-cbo-technical-overview (visited on 08/21/2016).

[106] Hive - Language Manual Join Optimization. URL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+JoinOptimization.

[107] Hive - Language Manual Join Optimization. URL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins (visited on 08/21/2016).

[108] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1st edition, 1979.

[109] S. C. Hsieh. Product Construction of Finite-State Machines. In *Proceedings of the World Congress on Engineering and Computer Science 2010 Vol I (WCECS 2010)*. Newswood Limited, 2010, pages 141–143.

[110] Y. E. Ioannidis. Query Optimization. *ACM Computing Surveys*, 28(1):121–123, 1996. DOI: 10.1145/234313.234367.

[111] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On Generating All Maximal Independent Sets. *Information Processing Letters*, 27(3):119–123, 1988. DOI: 10.1016/0020-0190(88)90065-8.

[112] S. Joshi, R. Jadon, and R. Jain. Sequential Pattern Mining Using Formal language Tools. *IJCSI International Journal of Computer Science Issues*, 9(5):316–325, 2012.

[113] JSON Schema and Hyper-Schema. URL: http://json-schema.org (visited on 08/21/2016).

[114] R. S. King and J. J. Legendre. Discovery of Functional and Approximate Functional Dependencies in Relational Databases. *Journal of Applied Mathematics and Decision Sciences*, 7(1):49–59, 2003. DOI: `10.1155/s117391260300004x`.

[115] J. Kivinen and H. Mannila. Approximate Inference of Functional Dependencies from Relations. *Theoretical Computer Science*, 149(1):129–149, 1995. DOI: `10.1016/0304-3975(95)00028-u`.

[116] P. Koutris and D. Suciu. Parallel Evaluation of Conjunctive Queries. In *Proceedings of the 30th Symposium on Principles of Database Systems of Data (PODS 2011)*. ACM, 2011. DOI: `10.1145/1989284.1989310`.

[117] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A Study of Skew in MapReduce Applications. In *Proceedings of the 5th Open Cirrus Summit*, 2011.

[118] R. Lämmel. Grammar Testing. In *Fundamental Approaches to Software Engineering: 4th International Conference, (FASE 2001)*. Volume 2029. Springer, 2001, pages 201–216. DOI: `10.1007/3-540-45314-8_15`.

[119] D. Leinders, M. Marx, J. Tyszkiewicz, and J. Van den Bussche. The Semijoin Algebra and the Guarded Fragment. *Journal of Logic, Language and Information*, 14(3):331–343, 2005. DOI: `10.1007/s10849-005-5789-8`.

[120] G. Linden, B. Smith, and J. York. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, 2003. DOI: `10.1109/MIC.2003.1167344`.

[121] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing Topology in Graph Pattern Matching. *Proceedings of the VLDB Endowment*, 5(4):310–321, 2011. DOI: `10.14778/2095686.2095690`.

[122] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong Simulation. *ACM Transactions on Database Systems (TODS)*, 39(1):1–46, 2014. DOI: `10.1145/2528937`.

[123] E. Mäkinen. On Lexicographic Enumeration of Regular and Context-Free Languages. *Acta Cybernetica*, 13(1):55–62, 1997.

[124] J. I. Maletic and A. Marcus. *Data Cleansing Data Mining and Knowledge Discovery Handbook*. Springer, 2005.

[125] H. Mannila and K.-J. Räihä. Practical Algorithms for Finding Prime Attributes and Testing Normal Forms. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS 1989)*. ACM, 1989, pages 128–133. DOI: `10.1145/73721.73734`.

[126] H. Mannila and K.-J. Räihä. Algorithms for Inferring Functional Dependencies from Relations. *Data & Knowledge Engineering*, 12(1):83–99, 1994. DOI: `10.1016/0169-023x(94)90023-x`.

[127] H. Mannila and K.-J. Räihä. *The Design of Relational Databases.* Addison-Wesley Longman, 1992.

[128] H. Mannila and H. Toivonen. Levelwise Search and Borders of Theories in Knowledge Discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997. DOI: `10.1023/a:1009796218281`.

[129] I. L. Markov. Limits on Fundamental Limits to Computation. *Nature*, 512(7513):147–154, 2014. DOI: `10.1038/nature13570`.

[130] W. Martens, F. Neven, and T. Schwentick. Simple off the Shelf Abstractions for XML Schema. *ACM SIGMOD Record*, 36(3):15, 2007. DOI: `10.1145/1324185.1324188`.

[131] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and Complexity of XML Schema. *ACM Transactions on Database Systems (TODS)*, 31(3):770–813, 2006. DOI: `10.1145/1166074.1166076`.

[132] M. Mathioudakis and N. Koudas. TwitterMonitor: Trend Detection over the Twitter Stream. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD 2010)*:1155–1158, 2010. DOI: `10.1145/1807167.1807306`.

[133] J. Matoušek and R. Thomas. On the Complexity of Finding Iso- and Other Morphisms for Partial k-Trees. *Discrete Mathematics*, 108(1):343–364, 1992. DOI: `10.1016/0012-365X(92)90687-B`.

[134] P. M. Maurer. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Software*, 7(4):50–55, 1990.

[135] MongoDB. URL: `http://www.mongodb.org` (visited on 08/21/2016).

[136] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory (TOIT). *ACM Transactions on Internet Technology*, 5(4):660–704, 2005. DOI: `10.1145/1111627.1111631`.

[137] M. Nečaský and I. Mlýnková. Discovering XML Keys and Foreign Keys in Queries. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC 2009).* ACM, 2009, pages 632–638. DOI: `10.1145/1529282.1529414`.

[138] F. Neven. Automata Theory for XML Researchers. *ACM SIGMOD Record*, 31(3):39, 2002. DOI: `10.1145/601858.601869`.

[139] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MR-Share: Sharing Across Multiple Queries in MapReduce. *Proceedings of the VLDB Endowment*, 3(1-2):494–505, 2010. DOI: `10.14778/1920841.1920906`.

[140] A. Okcan and M. Riedewald. Processing Theta-Joins Using MapReduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD 2011)*. ACM, 2011, pages 949–960. DOI: `10.1145/1989323.1989423`.

[141] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *USENIX 2008 Annual Technical Conference*. USENIX Association, 2008, pages 267–273.

[142] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*. ACM, 2008, pages 1099–1110. DOI: `10.1145/1376616.1376726`.

[143] K. Pal. How to Combat Financial Fraud by Using Big Data? URL: `http://www.kdnuggets.com/2016/03/combat-financial-fraud-using-big-data.html` (visited on 08/21/2016).

[144] F. Picalausa, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. Principles of Guarded Structural Indexing. In *Proceedings of the 17th International Conference on Database Theory (ICDT 2014)*, 2014, pages 245–256. DOI: `10.5441/002/icdt.2014.26`.

[145] P. Purdom. A Sentence Generator for Testing Parsers. *BIT*, 12(3):366–375, 1972. DOI: `10.1007/bf01932308`.

[146] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing Reducer Skew in MapReduce Workloads Using Progressive Sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC 2012)*. ACM, 2012, 16:1–16:14. DOI: `10.1145/2391229.2391245`.

[147] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*. ACM, 2015, pages 1357–1369. DOI: `10.1145/2723372.2742790`.

[148] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and Lower Bounds on the Cost of a Map-Reduce Computation. *Proceedings of the VLDB Endowment*, 6(4):277–288, 2013. DOI: `10.14778/2535570.2488334`.

[149] Schematron - A Language for Making Assertions about Patterns Found in XML Documents. URL: http://www.schematron.com/ (visited on 08/21/2016).

[150] H. Seidl. Deciding Equivalence of Finite Tree Automata. *SIAM Journal on Computing*, 19(3):424–437, 1990. DOI: 10.1137/0219027.

[151] M. Shaw, P. Koutris, B. Howe, and D. Suciu. Optimizing Large-scale Semi-Naïve Datalog Evaluation in Hadoop. In *Proceedings of the Second International Conference on Datalog in Academia and Industry.* Springer, 2012, pages 165–176. DOI: 10.1007/978-3-642-32925-8_17.

[152] S. Sidhanta, W. M. Golab, and S. Mukhopadhyay. OptEx: A Deadline-Aware Cost Optimization Model for Spark. 2016. arXiv: 1603.07936. URL: http://arxiv.org/abs/1603.07936.

[153] M. Sipser. *Introduction to the Theory of Computation.* Thomson Course Technology, 2nd edition, 2006.

[154] I. Sommerville. *Software Engineering.* Addison-Wesley, 1996.

[155] Y. Tao, W. Lin, and X. Xiao. Minimal MapReduce Algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD 2013).* ACM, 2013, pages 529–540. DOI: 10.1145/2463676.2463719.

[156] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010).* IEEE, 2010. DOI: 10.1109/icde.2010.5447738.

[157] J. D. Ullman. Designing Good MapReduce Algorithms. *XRDS: Crossroads, The ACM Magazine for Students*, 19(1):30–34, 2012. DOI: 10.1145/2331042.2331053.

[158] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976. DOI: 10.1145/321921.321925.

[159] M. Y. Vardi. Why is Modal Logic So Robustly Decidable? In *Descriptive Complexity and Finite Models: Proceedings of a DIMACS Workshop.* Volume 31. American Mathematical Society, 1996, pages 149–184.

[160] W3C. XML Document Type Declaration. 2004. URL: http://www.w3.org/TR/2004/REC-xml11-20040204/%5C#NT-doctypedecl (visited on 08/21/2016).

[161] W3C. XML Schema Part 1: Structures, 2nd edition. 2004. URL: http://www.w3.org/TR/xmlschema-1/%5C#cIdentity-constraint_Definitions (visited on 08/21/2016).

[162] M. M. Waldrop. The Chips are Down for Moore's Law. *Nature*, 530(7589):144–147, 2016. DOI: 10.1038/530144a.

[163] G. Wang and C.-Y. Chan. Multi-query Optimization in MapReduce Framework. *Proceedings of the VLDB Endowment*, 7(3):145–156, 2013. DOI: 10.14778/2732232.2732234.

[164] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 4th edition, 2015.

[165] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*. ACM, 2013, pages 13–24. DOI: 10.1145/2463676.2465288.

[166] Z. Xu, L. Zheng, and H. Chen. A Toolkit for Generating Sentences from Context-Free Grammars. *International Journal of Software and Informatics*, 5(4):659–676, 2011.

[167] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD 2007)*. ACM, 2007, pages 1029–1040. DOI: 10.1145/1247480.1247602.

[168] M. Yannakakis. Algorithms for Acyclic Database Schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases (VLDB 1981)*. VLDB Endowment, 1981, pages 82–94.

[169] C. Yu and H. V. Jagadish. XML Schema Refinement Through Redundancy Detection and Normalization. *The VLDB Journal*, 17(2):203–223, 2007. DOI: 10.1007/s00778-007-0063-0.

[170] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud 2010)*. USENIX Association, 2010.

Index