UCLA
UCLA Electronic Theses and Dissertations

Title
Approximation and Search Optimization on Massive Data Bases and Data Streams

Permalink
https://escholarship.org/uc/item/4pv2n0vs

Author
ZENG, KAI

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
Los Angeles

Approximation and Search Optimization on Massive Data Bases and Data Streams

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Kai Zeng

2014

© Copyright by

Kai Zeng

2014

The dissertation of Kai Zeng is approved.

Junghoo Cho

Tyson Condie

Yingnian Wu

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2014

To my parents and my wife . . .

for their unconditional love

TABLE OF CONTENTS

1 Introduction **1**

1.1 Search Challenges 1

1.2 Analytics Challenges 5

1.3 Overview and Contributions 7

1.3.1 K*SQL and XSeq: Expressive and Efficient CEP Languages . 7

1.3.2 Trinity.RDF: Web Scale Graph Engine 7

1.3.3 EARL and ABM: Bootstrap-Based Approximation Techniques for Interactive Data Analytics 8

I SEARCH OPTIMIZATION **12**

2 Background: Nested Words and Visibly Pushdown Automata **13**

2.1 Nested Words 13

2.2 Visibly Pushdown Automata 15

2.3 Difference between Nested Words and VPAs 17

3 KSQL: Unifying Languages and Query Execution for Relational and XML Sequences **18**

3.1 K*SQL By Examples 20

3.1.1 Nested Kstars 23

3.1.2 Linear-Hierarchical Data 26

3.2 Expressive Power 32

3.2.1 K*SQL vs. XPath 32
3.2.2 K*SQL vs. Other Sequence Languages 33
3.2.3 Monadic Second Order Logic 35
3.3 Optimization 36
3.3.1 Compile-time Optimization 36
3.3.2 Optimization for Nested Constructs 38
3.4 Experiments 40
3.4.1 XML queries in K*SQL 41
3.4.2 Query Execution Time 42
3.4.3 Number of Backtracks 43
3.5 Related Work 43
3.6 Summary of K*SQL 44
3.7 K*SQL Syntax and Expressive Power 45
3.7.1 K*SQL Syntax 45
3.7.2 K*SQL for Other Domains 46
3.7.3 Proof of Theorem 1 (Algorithm) 47
3.7.4 XPath for Sequence Queries 52
3.7.5 From VPE to K*SQL 53
3.7.6 Aggregates and Complexity 57

4 XSeq: High-Performance Complex Event Processing over Hierarchical Data 58

4.1 XSeq Query Language 63

4.2 Advanced Queries from Complex Event Processing 72
4.2.1 Stock Analysis 72
4.2.2 Social Networks 74
4.2.3 Inventory Management 75
4.2.4 Directory Search 76
4.2.5 Genetics 77
4.2.6 Protein, RNA and DNA Databases 78
4.2.7 Temporal Queries 79
4.2.8 Software Trace Analysis 81
4.3 XSeq Optimization 82
4.3.1 Efficient Query Plans via VPA 83
4.3.2 Static VPA Optimization 88
4.3.3 Run-time VPA Optimization 90
4.4 Formal Semantics of XSeq 91
4.5 Expressiveness and Complexity 96
4.5.1 CXSeq 97
4.5.2 Regularity of CXSeq and Complexity 101
4.5.3 Query Evaluation Complexity 101
4.6 Experiments 102
4.6.1 Effectiveness of Different Optimizations 103
4.6.2 Sequence Queries vs. XPath Engines 105
4.6.3 Conventional Queries vs. XPath Engines 107
4.6.4 Throughput for Different Types of Queries 109

4.7 Previous Work 111

4.8 Summary of XSeq 113

4.9 Core XSeq Proof of Regularity 113

4.9.1 Core XSeq with Variable Concatenation 114

4.9.2 Core XSeq Basic 119

5 Trinity.RDF: A Distributed Graph Engine for Web Scale Graphs 124

5.1 Join vs. Graph Exploration 128

5.1.1 RDF and SPARQL 128

5.1.2 Using Join Operations 129

5.1.3 Using Graph Explorations 130

5.2 System Architecture 132

5.3 Data Modeling 134

5.3.1 Modeling Graphs 134

5.3.2 Graph Partitioning 135

5.3.3 Indexing Predicates 137

5.3.4 Basic Graph Operators 138

5.4 Query Processing 139

5.4.1 Overview 139

5.4.2 Single Triple Pattern Matching 140

5.4.3 Multiple Pattern Matching by Exploration 143

5.4.4 Final Join after Exploration 145

5.4.5 Exploration Plan Optimization 146

5.4.6 Cost Estimation 151
5.5 Experiments 153
5.6 Related Work 163
5.7 Summary of Trinity.RDF 165

II APPROXIMATION OPTIMIZATION 166

6 EARL: Early Accurate Results for Advanced Analytics on MapReduce 167
6.1 Architecture 170
6.1.1 Extending MapReduce 171
6.2 Estimating Accuracy 175
6.2.1 Accuracy Estimation Stage 176
6.2.2 Sample Size and Number of Bootstraps 177
6.2.3 Sampling 179
6.2.4 Fault Tolerance 183
6.3 Optimizations 184
6.3.1 Inter-Iteration Optimization 184
6.3.2 Intra-Iteration Optimization 187
6.4 Current Implementation 188
6.5 Experiments 191
6.5.1 A Strong Case for EARL 191
6.5.2 Approximate Median Computation 192
6.5.3 EARL and Advanced Mining Algorithms 193

6.5.4 Sample Size and Number of Bootstraps 193
6.5.5 Pre-map and Post-map Sampling 194
6.5.6 Update Overhead 195
6.6 Related Work 196
6.7 Summary of EARL 198

7 The Analytical Bootstrap: a New Method for Fast Error Estimation in Approximate Query Processing 200

7.1 Problem Statement 203
7.1.1 An Example of Bootstrap 205
7.1.2 Scope of Our Approach 207
7.2 Background 209
7.2.1 Semirings and Relational Operators 209
7.2.2 Semiring Random Variables 212
7.3 Semantics & Query Evaluation 213
7.3.1 Formal Semantics 214
7.3.2 Intensional Query Evaluation 215
7.4 Extensional Query Evaluation 219
7.4.1 The Multinomial Representation 219
7.4.2 Queries without Aggregates 220
7.4.3 Queries with Aggregates 226
7.4.4 Correctness and Complexity 228
7.5 Efficient Approximation 228

7.6 Extensions of ABM 231
7.7 Experiments 232
7.7.1 Experiment Setup 232
7.7.2 Error Quantification Accuracy 233
7.7.3 Error Quantification Performance 238
7.7.4 Using Stratified Samples 240
7.8 The ABS System 241
7.9 Related Work 242
7.10 Summary of ABM 244
7.11 Correctness of Intensional & Extensional Evaluation 245
7.11.1 Background 245
7.11.2 Semantics & Query Evaluation 246
7.11.3 Extensional Query Evaluation 247
7.12 Constructing Distributions for General Queries without Aggregates . . 250

8 Conclusion and Future Work 251

References 253

LIST OF FIGURES

2.1 Tiny examples of nested words in different domains: XML and genomics. 15

3.1 A double-bottom or W-shape stock pattern. 29

3.2 Sample XML document for ancestry information. 29

3.3 KMP versus VPSearch for pattern matching against visibly pushdown words. 40

3.4 (a) XML queries in K*SQL vs. native XML engines. (b) W-shape pattern in K*SQL: optimized vs. straightforward implementation. (c) Contribution of different parts of the K*SQL optimization on the overall performance. 42

3.5 Formal syntax for K*SQL. The starting rule for K*SQL is ⟨sequence query spec⟩, which extends the ⟨simple table⟩ construct of SQL:2003. 46

3.6 Syntax of Core XPath 1.0 combined with 2.0. 48

4.1 A query in XPath 2.0/XQuery for a sequence of 'falling price' in Nasdaq's XML. 60

4.2 XSeq Syntax (QName, Variable, BoolExpr, Constant, and ArithmeticExpr are defined in the text). 64

4.3 (a) The β-meander motif and (b) the falling wedge pattern 78

4.4 VPAs for (a) `/son@Bdate` and (b) `/daughter son`. 85

4.5 Visual correspondence of VPA states and XSeq axes. 86

4.6 `//book/year/text()` 90

4.7 CXSeq Syntax 97

4.8 Contribution of different optimization techniques. 104

4.9 XSeq vs. XPath/XQuery engines: (a) 'V'-pattern query over Nasdaq stocks, (b) Sequence queries over Nasdaq stocks, (c) Regular XPath queries over XMark data, and (d) conventional XPath queries from XMark. 106

4.10 Effect of different types of XSeq queries on total execution time (a) and memory usage (b). 110

4.11 The effect of different types of queries on (a) Total query execution time, (b) Throughput in terms of tuple processing, and (c) Throughput in terms of datasize. 110

4.12 CXSeqA 114

4.13 CXSeqB 119

5.1 An example RDF graph 131

5.2 Distributed query processing framework 134

5.3 An example of model (5.1) 135

5.4 An example of model (5.2) 137

5.5 The query graph of Example 31 140

5.6 Distribution of the RDF graph in Figure 5.1 142

5.7 Expansion and combination examples 147

5.8 Data scalability 161

5.9 Machine scalability 161

6.1 A simplified EARL architecture 171

6.2 (a) Effect of B on c_v, (b) Effect of n on c_v 177

6.3 Work saved using our intra iteration optimization vs. sample size . . . 188

6.4 An example of how a user_job would work with the EARL framework 189

6.5 (a) Computation of average using EARL and stock Hadoop, (b) Computation of median using EARL and stock Hadoop, (c) Computation of K-Means using EARL, (d) Empirical sample size and number of bootstraps estimates vs. a theoretical prediction 192

6.6 (a) Processing times of pre-map and post-map sampling, and (b) Processing time with the update procedure 195

7.1 (a) An example of a database sample D with one relation R (named *stock*), and (b) a resample instance of R^r 205

7.2 The possible multiset worlds of D^r 206

7.3 (a) The result of $q(D_1)$ and $q(D_2)$, (b) all possible answers of q, and (c) their marginal probabilities 207

7.4 (a) Query plan of Example 32, (b) initial annotation of R^r, (c-d) intensional evaluation steps, and (f) truth table of $\pi_q(t_a) = 1$ 218

7.5 Comparing the distributions given by ABM and bootstrap on (a) Quantiles & existence probabilities, (b) KS distance and (c) User-defined quality measures; (d) Comparing user-defined quality measures given by ABMand bootstrap to ground truth 234

7.6 (a) ABM vs. bootstrap on user-defined quality measures for Skewed TPC-H; effect of varying (b) number of bootstrap trails, and (c) sampling rate; comparing time performance of ABM & various techniques (d) under 10% sampling rate, (e) under different sampling rates 239

7.7 ABM vs. bootstrap under stratified sampling 240
7.8 ABS Architecture 241

List of Tables

5.1 Base tables and bound variables. 130

5.2 Individual matching result of q_1 143

5.3 Individual matching result of q_2 144

5.4 Matching result of q_2 after matching q_1 144

5.5 Results after incorporating q_2 and q_3 145

5.6 Statistics of datasets used in experiments 154

5.7 Statistics of queries used in experiments 154

5.8 Query run-time in milliseconds on the LUBM-160 dataset (21 million triples) 154

5.9 Query run-time in milliseconds on the DBPSB dataset (15 million triples) 155

5.10 Query run-times in milliseconds for the LUBM-10240 dataset (1.36 billion triples) 157

5.11 Query run-times in seconds for the LUBM-100000 dataset (9.96 billion triples) 158

5.12 Query run-times in milliseconds for BTC-10 dataset (3.17 billion triples) 159

5.13 The space overhead of MapReduce-RDF-3X compared with the original datasets 160

5.14 The result sizes of LUBM queries 161

6.1 Symbols used 172

7.1 Classes of SQL queries supported by different techniques, and their coverage of TPC-H and Conviva queries 210

7.2 Summary of Notations 214

ACKNOWLEDGMENTS

Let me start by acknowledging the collaborators with whom I shared in this research: their names and the subjects of their contributions are listed in the Publication page that follows this. But moving past those wonderful papers, I would like to revisit the memories of my life as a graduate student at UCLA. These few lines are an attempt to reflect the support and love I received from so many over the years.

I thank my parents for their unconditional love. Starting from my undergraduate study, I have been living away from them for so many years. I am deeply grateful to my parents, whose selfless love is always with me no matter the distance.

I would like to express my deepest gratitude to my advisor, Professor Carlo Zaniolo, for the guidance, mentorship and trust he provided to me, all the way through the first day I met him when I was an undergraduate, to the completion of this degree. His insightful mind, optimism, patience and caring have made these five years memorable and rewarding.

I would also like to give my earnest appreciations to my committee, Professor John Cho, Professor Tyson Condie, and Professor Yingnian Wu, for their precious time and suggestions not only on my research, but also on my career and life.

My special thanks go to my beloved wife, Han, who has been a real source of love and support since the day we first met at Beijing. Her love has made every day in my life — no matter how tough it is — full of happiness and excitement.

Special acknowledgement must go to Haixun Wang and Barzan Mozafari, who after my adviser, have the greatest role in mentoring me about research. I really appreciate their generous help on developing my background and skills.

I am also grateful to my friends and fellow students. This list includes but not limited to Shi Gao, Nikolay Laptev, Alexander Shkapsky, Mohan Yang, Hamid Mousavi,

Jiacheng Yang, Jiaqi Gu, Hsuan Qiu, Massimo Mazzeo, Bin Shao, Weiguang Si, Xufeng Kou, Yuting Wang, Yang Yang, Yanbin Fan, Jinchao Li, Yuan Tian, Hao Wu, Jinwen Wang, Murong Lang, Tingyi Liu, Lingyan Ruan, Wenjia Huang, Tiansheng Yao, and Dan Xie.

VITA

2008-2009	RSDE Intern, Microsoft, Beijing, China.
2009	Bachelor of Engineering in Software Engineering, Zhejiang University, Hangzhou, China.
2010–2011	Teaching Assistant, Computer Science Department, UCLA. Taught sections of 'CS 111: Operating Systems Principles' and 'CS 33: Introduction to Computer Organization'.
2011	Research Intern, Microsoft Research Asia, Beijing, China.
2013	Research Intern, IBM Almaden Research, California, USA.
2009–2014	Research Assistant, Computer Science Department, UCLA.

PUBLICATIONS

Kai Zeng, Shi Gao, Barzan Mozafari and Carlo Zaniolo. The Analytical Bootstrap: a New Method for Fast Error Estimation in Approximation Query Processing. In Proceedings of the ACM **SIGMOD** Conference, Snowbird, Utah, June 22-27, 2014.

Kai Zeng, Shi Gao, Jiaqi Gu, Barzan Mozafari and Carlo Zaniolo. ABS: the Analytical Bootstrap System for Fast Error Estimation in Approximate Query Processing. Demo. In Proceedings of the ACM **SIGMOD** Conference, Snowbird, Utah, June 22-27, 2014.

Alexander Shkapsky, **Kai Zeng**, and Carlo Zaniolo. Graph Queries in a Next Generation Datalog System. Demo. In Proceedings of the 39th International Conference on Very Large Data Bases (**VLDB**), Rival del Garda, Trendo, August 26-30, 2013.

Barzan Mozafari, **Kai Zeng**, Loris D'Antoni, and Carlo Zaniolo. High-Performance Complex Event Processing over Hierarchical Data. In ACM Transactions on Database Systems (**TODS**), 38(4):21, 2013.

Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed Graph Engine for Web Scale RDF Data. In Proceedings of the 39th International Conference on Very Large Data Bases (**VLDB**), Rival del Garda, Trendo, August 26-30, 2013.

Kai Zeng, Mohan Yang, Barzan Mozafari, and Carlo Zaniolo. Complex Pattern Matching in Complex Structures: the XSeq Approach. Demo. In Proceedings of the 29th International Conference on Data Engineering (**ICDE**), Brisbane, Australia, April 8-11, 2013.

Nikolay Laptev, **Kai Zeng**, and Carlo Zaniolo. Very Fast Estimation for Result and Accuracy of Big Data Analytics: the EARL System. Demo. In Proceedings of the 29th International Conference on Data Engineering (**ICDE**), Brisbane, Australia, April 8-11, 2013.

Nikolay Laptev, **Kai Zeng**, and Carlo Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. In Proceedings of the 38th International Conference on Very Large Data Bases (**VLDB**), Istanbul, Turkey, August 27-31, 2012.

Barzan Mozafari, **Kai Zeng**, and Carlo Zaniolo. High-Performance Complex Event Processing over XML Streams. In Proceedings of the ACM **SIGMOD** Conference, Scottsdale, Arizona, USA, May 20-24, 2012.

Barzan Mozafari, **Kai Zeng** and Carlo Zaniolo “From Regular Expressions to Nested Words: Unifying Languages and Query Execution for Relational and XML Sequences”, In Proceedings of the 36th International Conference on Very Large Data Bases (**VLDB**), Singapore, September 13-17, 2010.

Barzan Mozafari, **Kai Zeng**, and Carlo Zaniolo. K*SQL: A Unifying Engine for Sequence Patterns and XML. Demo. In Proceedings of the ACM **SIGMOD** Conference, Indianapolis, Indiana USA, June 6-11, 2010.

ABSTRACT OF THE DISSERTATION

Approximation and Search Optimization on Massive Data Bases and Data Streams

by

Kai Zeng

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2014

Professor Carlo Zaniolo, Chair

A fast response is critical in many data-intensive applications, including knowledge discovery analytics on big data, and queries searching for complex patterns in sequences, data streams and graphs. Moreover, the volume of data and the complexity of the analytical tasks they must support are now growing at such a torrid rate that the vigorous progress in performance and scalability of computer systems cannot keep up with it. This situation calls for (i) effective optimization techniques to reduce the cost of complex pattern queries, and (ii) approximation techniques to produce results of predictable accuracy using a small subset of the data. In this dissertation we (i) introduce new query languages and optimization techniques for pattern matching in sequences, data streams and graphs, and (ii) formulate a general approximation model for analytics queries. Thus, in this dissertation we have made the following contributions:

(i) We have designed and demonstrated optimized implementation techniques for K*SQL and XSeq, which provide a unified framework for complex pattern searching on relational and XML DBs, respectively. In particular, we introduced efficient execution exploiting recent advances in automata theory known as Nested Words.

(ii) We have designed and demonstrated efficient scalable graph search engine based on novel distributed memory-based system architecture, and exploit graph exploration operations for implementing an efficient graph search algorithm.

(iii) We have introduced support for bootstrap methods in MapReduce. Bootstrap is a very useful estimation technique for sampling-based approximation. Thus we designed the EARL of Hadoop system, that facilitates and optimizes the use bootstrap methods on parallel MapReduce systems.

(iv) We have then invented and demonstrated an analytical model for bootstrap, whereby the Monte-Carlo evaluation of the standard method is replaced by a probabilistic query. Thus, we provided a semiring-based extension of relational algebra and related query optimization techniques to support fast execution of the resulting probabilistic query. We finally developed an Analytical Bootstrap System (ABS) for parallel and distributed computing platforms. ABS is applicable to most relational database queries and delivers very accurate estimates at speeds that outperforms the traditional bootstrap method by orders of magnitude.

CHAPTER 1

Introduction

The last decade has seen a soaring number of data-intensive applications. This increase can be attributed to many contributing factors, including (1) the wide adoption of computer systems across science, business, education and health-care, (2) the widespread usage of smart devices and sensor networks, and (3) the unprecedented influence of World Wide Web in almost every aspect of daily life. A fast response is critical in these data-intensive applications. However, the volume of data they must process, and the complexity of analytical tasks the applications must support have undergone an exponential growth, which even the vigorous progress in performance and scalability of computer systems cannot keep up with. This renders many traditional data management techniques irrelevant, and poses new challenges to data science research. These new challenges and the rich body of important applications behind them motivate many of the problems discussed in this dissertation. In this dissertation, we classify the general data analytical tasks into two classes: search and analytics. Next we categorize some of these challenges in each of these classes, and propose our solutions to tackle these challenges.

1.1 Search Challenges

We see a fast growth in the number and importance of data-intensive applications that need to extract useful knowledge from their massive data. Among the many different

kinds of knowledge, “pattern” is one of the most succinct, popular and human-readable form. Formally, in this dissertation a “pattern” is a *set or sequence* of data values which satisfy certain conditions that are of special interest to the users.

Searching for patterns from massive volume of streaming and stored data plays a vital role in many application areas. High-frequency trading[1], stock market and auction monitoring [SZZ04], publish-subscribe systems [DGH06], logistics and inventory management [Ba07a], click stream analysis [Sa01], commercial search engines and social network analysis are only a few examples of such application areas. In financial services, the insurance companies and the banks monitor the transaction flows to signify patterns of potential fraud behaviors. Recommendation systems might be interested in the sequences of users’ web browsing history, which can bring insights into more precise recommendation and targeted advertising. High-frequency trading systems requires real-time detection of a sequence of stock tickers that represent a new market trend. Logistics management systems track and monitor sequences of RFID readings to spot anomalies in the supply chains. Modern commercial search engines, e.g., Google and Bing, rely on pattern queries on knowledge graphs consisting of billions of edges (facts) to retrieve more structured high quality answers in order to augment search results. Social medias, suck as facebook[2], search patterns in huge social graphs to find potential friends for users, or to deliver more targeted ads.

The massive volume and increasing complexity of streaming and stored data pose difficult technical challenges to these pattern searching applications, which become even more severe in the presence of real-time constraints. We briefly summarize the three main challenges: *complexity*, *efficiency* and *scalability*.

Complexity. Complex Event Processing (CEP) applications represent an important class of applications that need to search patterns in streaming data or stored sequence

[1] http://en.wikipedia.org/wiki/High-frequency_trading
[2] https://www.facebook.com/

data. Many real-world sequence queries used in CEP applications involve nested hierarchical structures. Furthermore, much data exchanged over the Internet is embedded in XML or JSON, where the data itself have a hierarchical structure (when XML elements or JSON objects are enclosed in one another) besides their sequential structures (e.g. there is a global order of the tags in a JSON or XML file based on the order that they appear in the document). Such hierarchical structures need to be described by a new model *Nested Words*, and expressed using Kleene-* constructs.

Despite of the importance and pervasiveness of hierarchical structures, current languages lack the expressive power for such structures. The power and flexibility of Kleene-* constructs for expressing relatively complex patterns of events have long been recognized in CEP applications. Research work of extending query languages for relational data with Kleene-* constructs can be dated back to [SZZ01, Sa01], and lead to the recent SQL-MR proposal for their including into the SQL standards [ZWC07]. However, these hierarchical structures in complex real-world queries are not expressible in current languages. Such limitation is worsened for complex real-world data, e.g., when the data is embedded in XML or JSON. XPath and XQuery, which are the most widely used languages for XML data, lack explicit constructs for expressing Kleene-* and sequence patterns. This limitation highly hinders their usability in CEP applications.

Efficiency. CEP applications usually process high-speed streaming data. For instance, the New York Stock Exchange captures 1 TB of trade information during each single trading session; modern cars have close to 100 sensors that constantly monitor items such as fuel level and tire pressure. Such high speed requires that CEP applications have the ability to find the asked patterns in almost real time. However, existing systems lack of efficient implementation for supporting complex patterns. Nested Kleene-* structures need to be modeled by Nested Words and Visibly Pushdown Automata,

which go well beyond Finite State Automata, which are the common underlying automata model of many CEP applications. Furthermore, several Kleene-* extensions of XPath have been previously proposed in the literature [Cat06, CM07b, CM07a]. But the efficient implementation of these extensions remain an open problem.

What's more, queries with Kleene-* constructs involve a high degree of non-determinism. Running non-deterministic automata has always been an efficiency bottleneck. The difficulty lies in the exponential number of runs in the automata that need to be traced and followed. Providing effective techniques for handling non-determinism represents another main challenge.

Scalability. Even when processing stored data, when the data volume grows to web scale, answering pattern matching queries at interactive speed is still really challenging. Web scale graphs such as modern knowledge bases and social networks contains hundreds of millions of entities and billions of edges, which makes traditional data management technique irrelevant. Actually, despite of tremendous efforts devoted to building high performance graph pattern matching systems, scalability remains the biggest hurdle. The data volume has gone far beyond the capacity of a single machine, and thus has to be stored in a distributed environment. However, a graph pattern query usually comprises a large number of joins, where these joins need to access data in a random manner that has little locality. This incurs undesirably high network cost during matching. Furthermore, graph data is usually highly asymmetric. For example, in DBpedia [ABK07], over 90% nodes have less than 5 neighbors, while some top nodes have more than 100,000 neighbors. This asymmetry makes it really hard to balance work load, and hinders the query response time as it significantly slows down the stragglers.

1.2 Analytics Challenges

Data-driven activities in business, science and engineering are rapidly growing in terms of both significance and the size of the data they use. The need for timely and cost-effective analytics over such 'Big Data' which is common in all these application domains, poses great challenges to modern data management systems. The state-of-the-art massive data management systems include parallel databases and MapReduce systems. Parallel databases, such as Teradata [ter] and GAMMA projects [DGG86], use an architecture based on a cluster of shared nothing computers connected by a high-speed interconnect, where each database table is distributed across a cluster of nodes using various partitioning methods. MapReduce systems [had] distributes data uniformly across the nodes in the cluster, and utilize a special two-phase computation model to evenly distribute the work loads across all the nodes. However, even with modern hardware, analytics queries (e.g., OLAP-like queries) over massive datasets can still take hours, even days, rendering interactive data exploration impractical [AMP13], and can be entirely unsatisfactory to a user. In general, overloaded systems and high delays are incompatible with a good user experience; moreover approximate answers that are accurate enough and generated quickly are often of much greater value to users than tardy exact results.

This situation has brought even more attention to the already-active area of Approximate Query Processing (AQP), and in particular to sampling approaches as a critical and general technique for coping with the ever-growing size of big data. The first line of research work was that of Hellerstein et al. [HHW97a], where early results for simple aggregates were returned. Since then, sampling techniques have been widely used in databases [AGP99b, BCD03, CDN07, HHW97a, JAP07, OBE09], stream processors [BDM04, MZ10], and even Map-Reduce systems [AMP13, LZZ12]. For most applications on large datasets, performing careful sampling and computing early re-

sults from such samples provide a fast and effective way to obtain approximate results within the prescribed level of accuracy.

Of course, the approximate query answers so obtained are of very limited utility unless they are accompanied by some *accuracy guarantees*. For instance, in estimating income from a small sample of the population, a statistician seeks assurance that the answer so derived falls within a given margin of the correct answer computed on the whole population, with high confidence, say within $\pm 1\%$ of the correct answer with probability $\geq 95\%$. This enables the user to decide whether the current approximation is "good enough" for his/her purpose. Thus, assessing the quality (i.e., error estimation) of approximate answers is a fundamental aspect of AQP.

Previous error estimation technique [AMP13, CCM00, CDN07, HHW97a, HSS09, JAP07, JJ09, PBJ11a, WOT10] uses asymptotic theory to analytically derive closed-form error estimates for common aggregate functions in a database, such as `SUM`, `AVG`, etc. Although computationally appealing, analytic error quantification is restricted to a very limited set of queries. Thus, for every new type of query, a new closed-form formula must be derived. This derivation is a manual process that is ad-hoc and often impractical for complex queries [PJ05a].

This lack of an automatic and general way to access the quality of approximate answers greatly hinders the utilities of AQP. After many years of database research, AQP is still limited to simple aggregate queries [AMP13, CCM00, CDN07, HHW97a, HSS09, JAP07, JJ09, PBJ11a, WOT10]. In addition, although the need for approximation techniques obviously grow with the size of the data sets, general methods and techniques for handling complex tasks are still lacking in both MapReduce systems and parallel databases even though these claim 'big data' as their forte.

1.3 Overview and Contributions

In this dissertation, we set out to tackle the challenges mentioned above. Next, we list and elaborate on the main contributions made in this dissertation.

1.3.1 K*SQL and XSeq: Expressive and Efficient CEP Languages

We carefully design a natural and minimal Kleene-* extension of SQL and XPath called K*SQL and XSeq, respectively, which can easily model both sequential structures and well-nested hierarchical structures. We exploit Nested Words [Alu07] and Visibly Pushdown Automata (VPA) [AM04, AM06] as the underlying automata model for K*SQL and XSeq, which enables K*SQL and XSeq highly amenable to optimization. (We provide a brief introduction of nested words and VPA in Chapter 2.) Using nested words and VPA has several key benefits: (1) It allows for expressing complex queries that are common in CEP applications. (2) It allows for efficient stream processing algorithms. This gives us languages with expressiveness, versatility, ease-of-use and efficiency.

Furthermore, we introduce extensive optimization techniques, including both compile-time optimizations and run-time optimization, for K*SQL and XSeq. In particular, we develop generalizations of Knuth-Morris-Pratt algorithm [KJP77] for nested words and visibly pushdown words, which extend the pattern search optimization algorithms to non-deterministic mode.

1.3.2 Trinity.RDF: Web Scale Graph Engine

Trinity.RDF is a general graph engine that can manage and search web-scale graphs, including RDF data for which we demonstrate several example applications. Specifically, we propose to store the graph data in its native graph format in a distributed

in-memory key-value store. This approach satisfies the scalability requirement, and also supports fast random accesses on the data, which is the characteristic data access pattern of graph pattern matching. Therefore, modeling the data as an in-memory graph enables in-memory graph exploration — a new graph pattern matching paradigm other than the traditional join approach. Graph exploration exploits the dependency information in the graph pattern to eagerly prune the size of the intermediary results. This pruning is of particular importance in distributed environment, due to its ability of reducing the amount of intermediate data needed to shuffle across network, and in turn boosting the query performance and system scalability. We demonstrate the effectiveness of our technique through extensive experimental study, and show that even without sophisticated graph partitioning scheme, Trinity.RDF can achieve several orders of magnitude speed-up on web scale graph data compared to the state-of-the-art systems.

1.3.3 EARL and ABM: Bootstrap-Based Approximation Techniques for Interactive Data Analytics

To tackle the limitations of previous error estimation technique, and develop AQP systems that support efficient approximation with error estimation for general data analytics tasks, we propose *bootstrap* [ET93] — a more general method for error estimation. Bootstrap is essentially a Monte-Carlo procedure, which for a given initial sample, (i) repeatedly forms simulated datasets by resampling tuples i.i.d. (identically and independently) from the given sample, (ii) recomputes the query on each of the simulated datasets, and (iii) assesses the quality of answer on the basis of the empirical distribution of the query answers so produced. The wide applicability and automaticity of bootstrap is confirmed both in theory [BF81, VW00] and practice [KTA13, LZZ12, PJ05a]. Unfortunately, bootstrap tends to suffer from its high

computational overhead, since hundreds, or even thousands of bootstrap trials are typically needed to obtain reliable estimates [KTA13, PJ05a]. Therefore, we study two approaches to remove the computational hindrance of bootstrap: (1) optimizing the Monte-Carlo simulation process of bootstrap using delta computation, which is developed in our *EARL* system; (2) bypassing the Monte-Carlo simulation process of bootstrap, which is developed in our *Analytical Bootstrap Method*.

EARL. We introduce a novel incremental computation model into traditional batch-processing big data platforms such as Hadoop, and develop Early Accurate Result Library (EARL). EARL optimizes the work-flow computation on massive datasets by (1) starting computing the users' analytics tasks on a small sample of the original dataset, (2) constantly evaluating the quality of the approximate answers using bootstrap method, (3) gradually enlarging the samples if the accuracy of the approximate answers is not enough, and (4) stopping and returning the approximate results as early as possible once the desired approximation quality is met. This incremental computation model achieves the desired accuracy while minimizing the time and the resources required for the users's analytics tasks.

We also proposes both intra- and inter- iteration optimization techniques to minimize the repeated computation introduced by bootstrap and incremental computation. We further propose two sampling methods: pre- and post- sampling methods that allows the users to express their application-specific logics.

Analytical Bootstrap. Many real-life AQP applications can be expressed in SQL. Since SQL queries can be expressed using well-structured relational algebra, we introduce a new technique, called *Analytical Bootstrap Method* (ABM), which is both computationally efficient and automatically applicable to a large class of SQL queries, and thus combines the benefits of analytical approach and bootstrap approach for error estimation.

ABM models the bootstrap process through a probabilistic relational model. In specific, ABM annotates the sampled database with random variables to mimic the resampling process of bootstrap but without doing the actual resampling, and extends relational operators to manipulate these random variables. The query results produced by ABM will encode the correct distribution of all possible answers that would be produced *if* we actually performed bootstrap on the database. In particular, using a single-round query evaluation, ABM accurately estimates the empirical distribution of the query answers that would be produced by hundreds or thousands of bootstrap trials on the sampled database.

We evaluate ABM through extensive experiments on both synthetic and real-life data and queries. Our results show that ABM is an accurate prediction of the simulation-based bootstrap, while it is 3–4 orders of magnitude faster than the state-of-the-art parallel implementations of bootstrap [KTS12].

In summary, in this dissertation we have made the following contributions:

(i) We have designed and demonstrated optimized implementation techniques for K*SQL and XSeq, which provide a unified framework for complex pattern searching on relational and XML DBs, respectively. In particular, we introduced efficient execution exploiting recent advances in automata theory known as Nested Words.

(ii) We have designed and demonstrated efficient scalable graph search engine based on novel distributed memory-based system architecture, and exploit graph exploration operations for implementing an efficient graph search algorithm.

(iii) We have introduced support for bootstrap methods in MapReduce. Bootstrap is a very useful estimation technique for sampling-based approximation. Thus we designed the EARL of Hadoop system, that facilitates and optimizes the use bootstrap methods on parallel MapReduce systems.

(iv) We have then invented and demonstrated an analytical model for bootstrap, whereby

the Monte-Carlo evaluation of the standard method is replaced by a probabilistic query. Thus, we provided a semiring-based extension of relational algebra and related query optimization techniques to support fast execution of the resulting probabilistic query. We finally developed an Analytical Bootstrap System (ABS) for parallel and distributed computing platforms. ABS is applicable to most relational database queries and delivers very accurate estimates at speeds that outperforms the traditional bootstrap method by orders of magnitude.

Part I

SEARCH OPTIMIZATION

CHAPTER 2

Background: Nested Words and Visibly Pushdown Automata

2.1 Nested Words

The nested words model [Alu07] is a recently proposed notion from the field of automata [AM04, AM06] that can model data having both sequential and hierarchical structures. Common examples include XML, procedural programming traces or even genomic data [AM04].

Informally[1], in a nested word there is a sequential ordering among all the elements, while there is a secondary, hierarchical structure which is formed by nested edges between some of the elements, i.e., the edges do not cross. In this sense, nested words generalize both words and ordered trees, and allow both word and tree operations. In a nested word, the elements (a.k.a. positions) are divided into three disjoint sets: (i) call elements, where there is an outgoing hierarchical edge, (ii) return elements, where there is an incoming hierarchical edge, and (iii) internal elements that lack any hierarchical edges. A nested word is allowed to have pending edges, that are incoming edges without any call positions or outgoing edges without return positions. A nested word without pending edges is called *well-matched*. This terminology is owed to the software verification literature [AM06], where a program consists of several nested

[1] The formal definitions can be found in [Alu07].

function calls and returns, while other instructions (internal positions) form the sequential execution. However, nested words can be also used in several other domains. In Figure 2.1, n_1 is a nested word that represents a portion of an XML document that is not well-matched (it is still a valid nested word, as none of the edges cross). In Figure 2.1, white circles are internal positions, while blue and black circles represent calls and returns, respectively.

Another appealing application area for nested words is genomics. RNA sequences are not simply long strands of nucleotides. Rather, intra-strand base pairing leads to structures such as the one depicted in Figure 2.1. The covalent chemical bonds between subsequent nucleotides in each strand can be seen as the primary structure, while the hydrogen bonds between the bases (G&C, A&U) form a secondary structure [Aa90]. Since these bonds do not cross, each RNA sequence can be modeled as a nested word, e.g. n_2 in Figure 2.1.

Decision properties. Traditionally, dual structures such as XML have been modeled as ordered trees, and thus, have been queried using tree automata. Various classes of automata over nested words have been defined that have higher expressiveness and succinctness compared to word and tree automata [AM06]; however, their decision complexity and closure properties are analogous to the corresponding word and tree special cases. For example, regular languages of nested words are closed under union, intersection, complementation, concatenation, and Kleene-* [AM06]; deterministic nested word automata are as expressive as their non-deterministic counterparts; and membership, emptiness, language inclusion and equivalence are all decidable [Alu07].

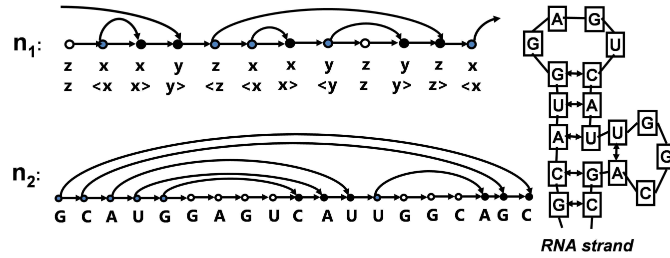


Figure 2.1: Tiny examples of nested words in different domains: XML and genomics.

2.2 Visibly Pushdown Automata

Similar to their closely related nested words model, visibly pushdown words also model a sequence of letters (i.e., a “normal” word) together with hierarchical edges connecting certain positions along the word. The edges are properly nested (i.e., edges do not cross), but some edges can be pending. Visibly pushdown words have found applications in many areas, ranging from program analysis to XML, and even representations of genomic data [AM06].

Visibly Pushdown Automata (VPA) are a natural generalization of finite state automata to visibly pushdown words. Visibly pushdown languages (VPLs) consist of languages accepted by VPAs. While VPLs enjoy higher expressiveness and succinctness compared to word and tree automata, their decision complexity and closure properties are analogous to the corresponding word and tree special cases. For example, VPLs are closed under union, intersection, complementation, concatenation, and Kleene-* [AM04]; deterministic VPAs are as expressive as their non-deterministic counterparts; and membership, emptiness, language inclusion and equivalence are all decidable [AM04, AM06]. Next, we briefly recall the formal definition of a VPA. Readers are referred to the seminal paper [AM04] for more details.

Let Σ be the finite input alphabet, and let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ be a partition of Σ. The intuition behind the partition is: Σ_c is the finite set of *call* (push) symbols, Σ_r is the

finite set of *return* (pop) symbols, and Σ_i is the finite set of *internal* symbols. Visibly pushdown automata are formally defined as follows:

Definition 1. *A visibly pushdown automaton (VPA) M over S is a tuple $(Q, Q_0, \Gamma, \delta, F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, Γ is a finite stack alphabet with a special symbol $\perp$ (representing the bottom-of-stack), and $\delta = \delta_c \cup \delta_r \cup \delta_i$ is the transition relation, where $\delta_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \backslash \{\perp\})$, $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$, and $\delta_i \subseteq Q \times \Sigma_i \times Q$.*

If $(q, c, q', \gamma) \in \delta_c$, where $c \in \Sigma_c$ and $\gamma \neq \perp$, there is a *push-transition* from q on input c where on reading c, γ is pushed onto the stack and the control changes from state q to q'; we denote such a transition by $q \xrightarrow{c/+\gamma} q'$. Similarly, if $(q, r, \gamma, q') \in \delta_r$, there is a *pop-transition* from q on input r where γ is read from the top of the stack and popped (if the top of the stack is $\perp$, then it is read but not popped), and the control state changes from q to q'; we denote such a transition $q \xrightarrow{r/-\gamma} q'$. If $(q, i, q') \in \delta_i$, there is an *internal-transition* from q on input i where on reading i, the state changes from q to q'; we denote such a transition by $q \xrightarrow{i} q'$. Note that there are no stack operations on internal transitions. We write St for the set of *stacks* $\{w\perp | w \in (\Gamma \backslash \{\perp\})*\}$. A *configuration* is a pair (q, σ) of $q \in Q$ and $\sigma \in St$. The transition function of a VPA can be used to define how the configuration of the machine changes in a single step: we say $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if one of the following conditions holds:

1. If $a \in \Sigma_c$ then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/+\gamma} q'$ and $\sigma' = \gamma \cdot \sigma$
2. If $a \in \Sigma_r$, then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/-\gamma} q'$ and either $\sigma = \gamma \cdot \sigma'$, or $\gamma = \perp$ and $\sigma = \sigma' = \perp$
3. If $a \in \Sigma_i$, then $\gamma \in \gamma'$ and $\sigma = \sigma'$.

A (q_0, w_0)-*run* on a word $u = a_1 \cdots a_n$ is a sequence of configurations $(q_0, w_0) \xrightarrow{a_1} (q_1, w_1) \cdots \xrightarrow{a_n} (q_n, w_n)$, and is denoted by $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$. A word u is accepted

by M if there is a run $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$ with $q_0 \in Q_0$, $w_0 = \bot$, and $q_n \in Q_F$. The language $L(M)$ is the set of words accepted by M. The language $L \subseteq \Sigma^*$ is a visibly pushdown language (VPL) if there exists a VPA M with $L = L(M)$.

2.3 Difference between Nested Words and VPAs

The input to a Nested Word Automaton (NWA) must come as a word with a *parsed nested structure*, i.e., upon seeing a call position we know its corresponding return position and vice versa. However, in many situations the input is given as word and the nested structure yet needs to be parsed/inferred. For instance, given a streaming XML, we do not know the return positions of the calls, at least during the first scan of the data. Thus, to handle such situations, Alur and Madhusudan [AM04] have proposed Visibly Pushdown Languages (VPL) where a stack is used to store the pending call and return symbols. VPLs are a subclass of context-free languages that are accepted by Visibly Pushdown Automata (VPA). Here again, the alphabet is split into three disjoint sets of Σ_c, Σ_r and Σ_i and upon reading a call symbol ($a \in \Sigma_c$), the VPA *has to push* on the stack, and upon reading $a \in \Sigma_r$ it *has to pop* the stack. For $a \in \Sigma_i$, the VPA *cannot use the stack.*

CHAPTER 3

KSQL: Unifying Languages and Query Execution for Relational and XML Sequences

There is much interest in extending relational query languages with Kleene-* (Kstar) constructs for matching complex patterns of events in data streams and stored sequences. The power and flexibility of Kstar constructs for SQL, which were introduced in [SZZ01, Sa01], have recently attracted the attention of DBMS vendors and DSMS start-up companies, leading to the recent SQL-MR proposal for their inclusion into the SQL standards [ZWC07]. This is hardly surprising, given Kstar's proven effectiveness in application areas as diverse as stock market and auction monitoring [SZZ04], publish-subscribe systems [DGH06], RFID-based inventory management [Ba07a], click stream analysis [Sa01], and electronic health systems [HH05, La10]. In financial services, for instance, a brokerage customer may be interested in a sequence of stock trading events that represent a new market trend. In RFID-based tracking and monitoring, applications may want to track valid paths of shipments and detect anomalies in the supply chains.

In this chapter we show that, in spite of the many success stories mentioned above, we have only begun to explore the variety of new applications made possible by the Kstar constructs. We introduce a new language and system, called K*SQL, that reaches well beyond existing proposals to provide:

1. A unifying framework. Many query languages have been proposed, each de-

signed for a different domain. K*SQL is powerful enough to express and support efficiently both set and sequence queries on both relational and XML data, residing in the database or flowing in as a data stream. The many domain-specific languages previously proposed for various combinations of the above retain their validity and desirability in their own application realm, but because of its superior query optimization technology and expressiveness, K*SQL can be used to support and extend them—e.g., XPath 2.0 can be efficiently implemented by a simple translation into equivalent K*SQL queries. Thus, even when programmers prefer to continue to write their XML queries in XPath, they will still benefit from the performance improvement brought by K*SQL as a query execution backend.

Furthermore, K*SQL provides a natural query language for *nested words*—a recently proposed model from the field of formal verification[AM06], which generalizes both words and tree structures. To the best of our knowledge, this is the first *database query language* proposed for this very powerful and useful data model.

2. More complex patterns. In addition to supporting new data models, the power of K*SQL allows it to match more powerful patterns on standard relational sequences and streams. These are critical in advanced applications, such as stock analysis, RFID processing and trajectory mining. For instance, many real-world sequence queries that involve nested Kstar patterns are not expressible in current languages, such as the proposed SQL-MR standards [ZWC07]. Also, when data is embedded in XML, which is a common practice with data exchange over the internet, K*SQL can express sequence queries that are not expressible in XPath 1.0 or 2.0.

We achieve these goals through the following contributions:

1. We study the formal properties of sequence extensions for SQL by incrementally extending our query language to support pattern matching over (i) bounded regular expressions, (ii) regular expressions, and finally (iii) regular expressions over nested

words. This methodology allows us to characterize the expressiveness of K*SQL, and compare it to other existing languages (Section 3.2).

2. Based on our study of expressiveness, we carefully design a natural extension of SQL that provides versatility and ease-of-use, while minimizing syntactic additions (Section 3.1).

3. We develop extensive optimization techniques for K*SQL, including generalizations of the KMP [KJP77] algorithm to the case of nested words and visibly pushdown words (Section 3.3).

4. We implement and validate our optimization techniques on well-known benchmarks and real-world data (Section 3.4).

5. We provide compilation algorithms and tools for automatic translation of several (e.g., XPath, SASE+ [GAD08]) languages into K*SQL, thus allowing for both (i) code-base migrations and (ii) the use of the proposed optimizations also as a *back-end query execution engine* when users prefer those languages as an interface (Section 3.2.1, Section 3.7.3).

This chapter is organized as follows. We briefly introduce the basic syntax of K*SQL through examples in Section 3.1, followed by a summary of our complexity results in Section 3.2. We highlight our main algorithms for implementation and optimization of K*SQL in Section 3.3 which are empirically validated in Section 3.4. We review the related work and conclude in Sections 3.5 and 3.6, respectively.

3.1 K*SQL By Examples

K*SQL extends the syntax of a previous SQL-based sequence language (SQL-TS [SZZ01]) with a few but powerful constructs.[1]. Thus, we first use a simple example that could

[1]For the formal syntax and semantics of K*SQL see Section 3.7.1.

also be expressed in most of existing languages, before considering examples involving our extensions. Similar to [ZWC07], our pattern extensions are meant to be effective on both DB tables and data streams. So, as our first example, let us consider a DB table containing recent Nasdaq stock transactions (we discuss data streams later):

Example 1. *A table with Nasdaq transactions.*

```
CREATE TABLE NasdaqTable (seller Varchar(20), buyer Varchar(20),
  stockName Varchar(8), shares Integer, price Integer,
  datetime Timestamp)
```

Here, `price` is the price per share. As an example, consider the following well-known query from stock market analysis:

Example 2 (Double-bottom or 'W' pattern). *Find those stocks whose price has formed a W-shape. That is, the price has been going down to a local minimum, then rising up to a local maximum and then again, decreasing to another local minimum, and finally, followed by another rise. The starting price should be at least 50. (See Fig. 3.1).*

```
SELECT S.stockName, D.avg(price) AS runningAvg,
  avg(D.price) AS finalAvg, last(D.price) AS finalPrice
FROM NasdaqTable
  PARTITION BY stockName
  ORDER BY datetime
  AS PATTERN (S A* B+ C+ D+)
WHERE S.price >= 50 AND S.price > first(A).price
    AND A.price < prev(A).price
    AND B.price > prev(B).price
    AND C.price < prev(C).price
    AND D.price > prev(D).price AND maximal(D)
```

The above is a typical K*SQL query. The semantics are based on 'immediately

follows' relationship between ordered tuples. Thus, the syntax is very similar to SQL, except that we have sequential semantics:

- The `PARTITION BY` clause splits the tuples according to their `stockName` value, as if they were separate streams.
- The `ORDER BY` clause defines how the tuples in each partition should be ordered, e.g., in the above example we order the transactions in their chronological order. Similar to SQL, the `DESC` keyword can be added for descending order.
- The `AS PATTERN` clause defines the sequential pattern that we are searching for. In Example 2, `S`, `A`, `B`, `C` and `D` refer to *consecutive* tuples. Variable `S` is *singleton* and matches with exactly one tuple, while the other variables are *group variables* (or Kstar variables): $*$ allows for arbitrary repetition, while $+$ requires a repetition of at least 1. These variable names can be used in the `WHERE` predicates to express the relationship between these tuples.

K*SQL supports both *running aggregates* (e.g. `D.avg(price)`) as well as *final (a.k.a. blocking) aggregates* (e.g., `avg(D.price)`). K*SQL also supports the four typical sequence modifiers, namely `first`, `last`, `prev` and `next` which can be applied to group variables. In K*SQL, `maximal(D)` denotes that we will remain in the `D+` state until this fails–i.e., until the price is no longer increasing. In the absence of the `maximal` predicate, the default behavior is to return all the matches, namely any number of successive occurrences satisfying the predicates.

As mentioned, Example 2 could be expressed in most of the previously proposed languages as well, modulo minor variations in keywords and syntax. K*SQL uses the same constructs for both stored tables and data streams; however, the `ORDER BY` clause will be omitted in continuous queries on data streams where the order follows from the

very declaration of the stream, such as that in Example 3. For instance, in our system, an input stream of Nasdaq transactions can be defined as follows:

Example 3. *A stream of Nasdaq transactions.*

```
CREATE STREAM Nasdaq (seller Varchar(20), buyer Varchar(20),
      stockName Varchar(8), shares Integer,
      price Integer, datetime Timestamp)
  ORDER BY datetime SOURCE 'port4446';
```

In this example `SOURCE 'port4446'` declares the port at which the input data is arriving; `ORDER BY datetime` declares that tuples in our stream are ordered according to their timestamp `datetime`. In the absence of such declaration, the data stream is assumed ordered by its arrival order. But in either case, continuous queries assume and maintain this order, and thus our K*SQL queries over data streams do not contain any explicit `ORDER BY` clause–which will therefore be omitted in the rest of this chapter.

Our next order of business is to allow nested Kstars in the definition of patterns. Although it requires only a minor syntactic extension, nested Kstars significantly improve the usability and expressiveness of our language.

3.1.1 Nested Kstars

The 'W-shape' pattern of Example 2 consists of two 'V-shape' patterns. However there are many more complex queries that involve nested Kstars. For instance, consider the following example from stock analysis, known as *uptrend falling wedge* pattern[2].

Example 4 (Wedge pattern). *Find those stocks whose price fluctuates as a series of 'V-shape' patterns, where in each 'V' the range of the fluctuation becomes smaller, and eventually, the price rises up to higher than its starting point.*

[2] http://www.chartpatterns.com/wedges.htm

Since each 'V' sub-pattern is itself two Kstars, say X^+Y^+, we need to somehow express the *arbitrary number of repetition* of this sub-pattern with a nested Kstar, say $(X^+Y^+)^*$. Next questions are then how to clearly express the complex conditions on such patterns, and how to run them efficiently? The following K*SQL query is the answer to this query.

```
SELECT first(first(Z).X).stockName,
          first(first(Z).X).price AS startPrice,
          E.price AS finalPrice
FROM Nasdaq
     PARTITION BY stockName
     ORDER BY datetime %Optional for streams
     AS PATTERN ((Z: X+ Y+)+ E)
WHERE Z.X.price  < prev(Z.X).price
    AND Z.Y.price  > prev(Z.Y).price
    AND max(Z.price)-min(Z.price)  <
      max(prev(Z).price)-min(prev(Z).price)
    AND first(first(Z).X).price  < E.price
```

Here, K*SQL goes beyond SASE+ [GAD08], SQL-TS and SQL-MR by supporting nested Kstars. Next, we briefly explain the new features introduced in the query above.

Aliases. As shown in this Example, K*SQL allows the use of aliases for subpatterns: $(Z : X^+Y^+)$ defines Z as an alias for the sequence X^+Y^+, one for each 'V'-phase in the example considered. Now, $(Z : X^+Y^+)^+$ denotes one or more occurrences of Z. We have thus moved from patterns consisting of linear sequences to patterns consisting of sequences of sequences. K*SQL allows for any depth of nested Kstars, e.g. here the depth is 2. Perhaps two main reasons why previous languages did not support nested Kstars were (i) they would lead to ambiguity in the aggregates and, (ii) they require

much more complex optimizations. Use of aliases in K*SQL overcomes the former obstacle, as described next.

Aggregates on nested Kstars. K*SQL assumes that each instance of Z has virtual attributes whose values are derived from the instances of X^+ and Y^+ occurring in this instance of Z, e.g. min(Z.price) is the minimum price among the X's and Y's of the current repetition of Z. We could also calculate the running and final averages of the falling prices in the current Z by Z.avg(X.price) and avg(Z.X.price), respectively. Also, max(prev(Z).price) refers to the maximum price in the previous repetition of Z.

Similarly, the running aggregate first is available on Z, with unchanged semantics, i.e. Z.first(Y.price) denotes the sequence of the rising prices of the first Z, while first(first(Z).X).price returns the price of the first tuple of X in the first Z.

Therefore, the K*SQL syntax for nested Kstars is powerful and unambiguous, and only requires the user to assign a new alias variable to each compound Kstar [3], i.e. a Kstar expression consisting of more than one variable. In fact, even though partial optimizations for nested Kstars were proposed in [KMS08], they did not allow aggregates on such constructs due to the ambiguity that such combinations would cause. Thus, K*SQL syntax achieves the unambiguity while allowing aggregates on nested Kstars, mainly through the aliases and the simple semantics introduced above. Efficiency concerns are addressed in Sections 3.3 and 3.4.

So far we have only considered relational data, but in practice, many data streams are embedded in XML tags, as XML allows for generality, and usability of data exchange over the Internet. For instance, stock/financial transactions are often encoded and published as XML streams. Thus, a next natural question is 'whether and how a sequence language can query such data'? And if possible at all, 'what types of XML

[3]No alias is required for simple Kstars, since B+ is viewed as equivalent to (B: _ +), where _ is the anonymous variable, as in Datalog.

data and queries can be expressed in our language'? Next, we answer these questions for K*SQL.

3.1.2 Linear-Hierarchical Data

Consider the following DTD for an XML schema:

```
< !DOCTYPE company [
< !ELEMENT company (name, (transaction)*) >
< !ELEMENT transaction (price, buyer, date) >
< !ELEMENT name (#PCDATA) >
< !ELEMENT price (#PCDATA) >
< !ELEMENT buyer (#IDREF) >
< !ELEMENT date (#PCDATA) > ] >
```

Throughout this chapter, we use SAX-3 [Za06] representation of XML[4], a slightly modified version of the famous SAX API. Thus, every XML is processed as a stream of SAX events represented by triplets (`type`, `token`, `value`). The order in which these triplets appear in the sequence reflect their pre-order traversal position in the document. By having a unique tag name for the root element ('company' in this example), we can easily extend the same format even to represent a stream of several XML documents with the same schema[5]. The following is the beginning portion of an XML document, within a stream that consists of the XML documents for several Nasdaq companies:

```
(type, token, value)
...
106: ('open', 'company', -),
107: ('open', 'name', -),
108: ('text', 'IBM', -),
109: ('close', 'name', -),
110: ('open', 'transaction', -),
```

[4] Any relational format for pre-order traversal of the stored/streaming XML file(s) is acceptable.

[5] If the document tokens are out-of-order, a fourth column can be used for `documentId`.

. . .

Here, the numbers represent the relative position of each tag within the stream of company XMLs. Assume that the stock transactions under each company appear according to their date attribute. Thus, a transaction occurred earlier has an open tag with a smaller position number. We begin by searching the same 'W'-shape pattern as in Example 2, but this time from XML data.

Example 5. *The K*SQL query below, returns all the W-shape stocks in a given Nasdaq XML document stream (See Figure 3.1).*

```
SELECT C.token as CompanyName,
   first(Z.first(X.G.token)) as price1,
   first(Z.last(X.G.token)) as price2,
   first(Z.last(Y.K.token)) as price3,
   last (Z.last(X.G.token)) as price4,
   last (Z.last(Y.K.token)) as price5,
FROM NasdaqStream
AS PATTERN ( A B C D (Z: (X: E F G H I^6 J)*
                         (Y: E F K H I^6 J)*
                      )^2 L
           )
WHERE  A = open('company')
    AND B = open('name')
    AND D = close('name')
    AND E = open('transaction')
    AND F = open('price')
    AND H = close('price')
    AND J = close('transaction')
    AND X.G.price <= prev(X.G).price
    AND Y.K.price >= prev(Y.K).price
    AND L = close('company')
```

Syntactic shorthands. Note that for XML documents, the tuples are processed according to their appearance order, and hence the ORDER BY clause for XML queries is omitted [6]. Here, open() and close() are merely convenient shorthands to recognize open or close tags, e.g. B = open('name') could be replaced by a condition that B.type =' open' AND B.token =' name'. Here, we also used the notation I^6 as a shorthand for the repetition, i.e. IIIIII; likewise (Z : ...)^2 stands for (Z : ...)(Z : ...). Also observe that, due to their similar definitions, variables E, F, G, H, I and J, are used under both X and Y, and thus we can use the path notation X.E or Y.E to refer to one or the other. When their path notation is missing, the parser disambiguates them by duplicating their predicates for each subpattern that they appear in.

Query explanation. In the query above, the first part of the pattern, namely ABCD, parses the ⟨company⟩⟨name⟩somename⟨/name⟩ header. Next, we use Z to alias the definition for a 'V'-shaped pattern and use Z^2 to capture a 'W'. In each 'V', the falling and rising phases are defined by X* and Y*, respectively. To recognize each occurrence of X, we use four variables, EFGH, recognizing ⟨transaction⟩ ⟨price⟩ somePrice⟨/price⟩, which are followed by I^6, where I's act as wildcards to skip the next six tags, namely ⟨buyer⟩Name⟨/buyer⟩ ⟨date⟩somedate⟨/date⟩, and so on. Here, J and L refer to the corresponding close tags for transaction and company. The rest is obvious (consider the W-shape pattern in Figure 3.1).

Limitations of other languages. This example illustrates the power of nested Kstars (with aliasing) in K*SQL. Kaghazian et al. [KMS08] also allow nested Kstars but they do not support aggregates on such expressions. However, the hierarchical aliasing in K*SQL allows us to select subsequent occurrences of X and then compare the G prices within and so on. On the other hand, expressing queries such as Example 2

[6]The order of appearance of the tags in the XML, is referred to as the total 'document order' in XPath 2.0 and XQuery.

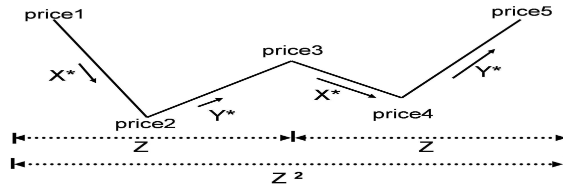


Figure 3.1: A double-bottom or W-shape stock pattern.

in XPath is often difficult[7] if not impossible[8] (e.g., for an extension of XPath with Kstar see [Cat06]).

The next question is whether these constructs (i.e., nested Kstars plus aliasing and aggregates) are also capable of querying XML data with *recursive schemas*?, as a recursive nature can represent a serious challenge for a relational sequence language.

```
<familyroot id="31602">
    <son name="John">
        <son name="Brian">
        </son>
        <son name="Bob">
            <son name="Paul">
            </son>
        </son>
        <daughter name="Alice">
        </daughter>
    </son>
</familyroot>
```

Figure 3.2: Sample XML document for ancestry information.

[7] The 'W'-shape query is expressible in XPath but is hard to write, read and optimize (see Section 3.7.4).

[8] For examples, see Section 3.7.4.

XML with recursive schema. Consider the tiny ancestry XML in Figure 3.2, in which, for example, a *son* can contain other sons to an arbitrary depth. Now consider the following example.

Example 6. *For an ancestry XML (e.g., the one in Figure 3.2), return the names of all those sons whose father is named 'John'.*

Such queries are very easy to write in XPath, here:

```
//son[name = "John"]/son/@name
```

Current Kstar languages cannot express such queries simply because they cannot determine (i) how many intermediate ⟨son⟩'s they should skip before reaching all the sons of John, and (ii) they cannot detect that, e.g., Paul is Bob's son and not Brian's, say, e.g., by considering their depth in the XML. To overcome these limitations for recursive structures, K*SQL supports a simple but powerful aggregate, called `isElement`. The following K*SQL query is equivalent to the XPath expression above:

```
SELECT Y.value as sonNames
FROM AncestryRelation
AS PATTERN (A X N* B Y N* C N* D)
WHERE A = open('son')
    AND X.type = 'attribute' AND X.token = 'name'
    AND X.value = 'John' AND isElement(N)
    AND B = open('son')
    AND Y.type = 'attribute' AND Y.token = 'name'
    AND C = close('son') AND D = close('son')
```

In K*SQL, `isElement()` is a built-in function that is internally implemented using a stack which evaluates to true on every tuple, until a violation of well-nestedness occurs, at which point, it evaluates to false. For the example above, when a new tuple

is assigned to state `N*`, it is added to the stack if its token is an open tag. But if the new tuple's token is a close tag, we check if the top of the stack is its corresponding open tag. If yes, we pop it, and otherwise there is a stack violation, and the tuple will be passed to the next state, e.g. `B`. For tokens that are neither open nor close tags, we do not touch the stack but remain in `N*` depending on the query mode, e.g., in maximal mode, we stay in `N*` until a stack violation occurs, but in all-match mode, we consider all the options non-deterministically. In Section 3.7.2, we explain how, in K*SQL, `isElement()` is implemented in a generic form (i.e., not limited to XML or its specific SAX representation).

Query explanation. Here, each time a ⟨`son`⟩ tag is found (element `A`), the `X` element checks its name attribute, the `N*` elements skip the well-nested elements to ignore the intermediate children of the current node. Since the default setting is non-deterministic, at some point, the automaton will follow the `B` element instead of `N`, and if it is another ⟨`son`⟩ tag, the automaton will proceed with the rest of the pattern. Once all possible traces of this automaton are explored (either success or failure), the first element (i.e., `A`) will be moved forward until the next ⟨`son`⟩ is found, and the same process is repeated.

Nested structures other than XML. The capabilities of K*SQL, in querying data with both sequential and hierarchical structures, is not limited and specific to XML. In fact, K*SQL provides for pattern matching over nested words as well as over words. Nested words were originally proposed for static program analysis [Alu07, AM06], but can model other dual linear-hierarchical structures as well. XML represents only one example of such data. Procedural programming traces and genomic data are other examples. A brief background on these notions can be found in Chapter 2. Interestingly, in Section 3.2.3, we prove that K*SQL can query any data that can be modeled using nested words (or visibly pushdown words [AM04], a closely related notion). The

examples in this chapter were chosen from the XML domain due to the importance of XML and its familiarity to the database community, and also its long history of rich languages in the field. However, we emphasize that our constructs are not specific to XML or its particular SAX representation. A brief explanation on the application of K*SQL for other domains such as program traces and RNA sequences are provided in Section 3.7.2.

Many interesting questions arise at this point: Can K*SQL express all XPath queries? What is the true expressive power that our built-in `isElement` construct brings to K*SQL? How does K*SQL compare with other existing sequence languages? What is the query evaluation complexity in K*SQL? What if we allow aggregates? How can we optimize K*SQL queries and ensure efficiency? The next two sections address these questions.

3.2 Expressive Power

In this section, we briefly present our main results on the expressiveness of K*SQL, and compare it to other existing models and languages. The proofs are in Appendices 3.7.3 and 3.7.5.

3.2.1 K*SQL vs. XPath

Core XPath 2.0 [CM07a] represents a fragment of XPath that is complete for First Order (FO) logic over trees [CM07b]. In Section 3.7.3, we prove the following theorem:

Theorem 1. *For every Core XPath 2.0 query, there is an equivalent K*SQL query.*

Moreover, we show later (Theorem 5) that K*SQL is as expressive as VPLs which are equivalent to monadic second order (MSO) logic over nested words [Pit05]. Thus,

K*SQL is strictly more expressive than Core XPath 2.0, which is in turn strictly more expressive than Core XPath 1.0 [CM07b]. Section 3.7.4 further elaborates on the limited expressivity of XPath for sequence queries. In Section 3.7.3, we provide a simple constructive proof[9], that shows we can algorithmically construct an equivalent K*SQL query for any given Core XPath expression.

3.2.2 K*SQL vs. Other Sequence Languages

In this section, we study the expressiveness of K*SQL and compare it with other sequence languages. We first disallow aggregates in these query languages, to differentiate between the real power of the language core itself from that brought about by aggregates. In Section 3.7.6, we briefly address the effect of allowing aggregates on the complexity and expressiveness.

While a full query language can return additional information about the matches, in order to simplify the presentation, here we only consider the decision version of these query languages, i.e. the select clause returns a 'TRUE' answer when a match is found. Thus, the language membership for a given query is the *decision* of whether the input sequence satisfies the pattern described by the PATTERN construct and the WHERE conditions. For a query language L, and for a given alphabet Σ, we use $\mathcal{D}(L)$ to denote the class of all the decision problems that can be encoded/expressed as a query written in L running on a sequence of input symbols from Σ.

A hierarchy of constructs. Here, we start from a restricted version of K*SQL, then incrementally add back its main constructs, leading to the following hierarchy of languages: **K*SQL1**: when we don't allow any of `isElement`, nested Kstars, or query composition; **K*SQL2**: when we allow nested Kstars but no `isElement` or

[9] While we could also derive Theorem 1 from Theorem 5, we inductively prove the former in Section 3.7.3, since it gives us a linear-time algorithm for intuitive translation of XPath queries.

query composition; and finally **K*SQL3**: where both nested Kstars and `isElement` are allowed but no query composition [10]. This hierarchy has enabled us to (i) analyze the effect of these critical constructs on the usability of the language, (ii) decide on what extensions are needed for expressiveness and which ones are only syntactic sugar or help the optimizer, and finally, (iii) compare with other existing languages while providing insights on a unified approach to querying both words and nested words.

Lemma 2 (K*SQL1)**.** *Let $A \subseteq \Sigma^*$. The following statements are equivalent:*

*1. $A \in \mathcal{D}$(K*SQL1).*

2. $A \in \mathcal{D}$(SQL-MR [ZWC07] without query composition).

3. $A \in \mathcal{D}$(SASE+ [GAD08] restricted to its 'strict contiguity' and 'partition contiguity' modes, and without query composition).

4. $A \in \mathcal{D}$(Cayuga [Da07]).

K*SQL1 can be formulated using a 'bounded' NFA (i.e., contains no loops) where the transitions between the states are labeled with regular formulas. Moreover, SQL-TS [SZZ01] becomes a strict subset of $\mathcal{D}$(K*SQL1), due to the lack of non-determinism in SQL-TS. The SASE+ language runs in different modes for the matching condition: *strict contiguity*, *partition contiguity*, *skip till next match*, and *skip till any match*. The latter two modes increase the expressiveness of SASE+. Using these modes, SASE+ (under query composition) is equivalent to NFA^b [GAD08] which is in turn equivalent to class of regular languages (when the predicates are regular[GAD08]). This, and the following lemma lead us to Theorem 4.

Lemma 3 (K*SQL2)**.** *Let $A \subseteq \Sigma^*$. The following statements are equivalent:*

1. A is recognizable using a regular expression (RE).

[10] Query composition does not add to the expressiveness of K*SQL.

*2. $A \in \mathcal{D}$(K*SQL2).*

Theorem 4. *$\mathcal{D}$(K*SQL2) is equal to $\mathcal{D}$(SASE+ with query composition).*

From SASE+ to K*SQL2. The translation of SASE+ queries into K*SQL2 is simple. The `PATTERN` and `WHERE` clauses of SASE+ are analogous to K*SQL2. However, SASE+ supports *skip till next match* and *skip till any match* in its query modes, whereby irrelevant tuples in the middle of a match are skipped. To emulate these modes in K*SQL2, we use wildcards and nested Kstars as follows: every SASE+ pattern `X`* is replaced with (`A : B X C`)* in K*SQL, where `B` and `C` are wildcards, thus allowing arbitrary tuples between consecutive `X`'s.

In summary, in the absence of aggregates, and once we allow query compositions, from the previous languages, SQL-MR [ZWC07] (using all match mode), SASE+ [GAD08] (using its 'skip till any match' mode) become equivalent to K*SQL2. Also, Cayuga under query composition is contained in K*SQL2. This containment is strict, if the class of DSPACE[log n] problems are strictly contained in NSPACE[log n]. This is due to Theorem 4 and complexity results from [Da08], showing that Cayuga is a subset of DSPACE[log n] and can express some complete problems in this class.

3.2.3 Monadic Second Order Logic

While the unified support and optimization of sequence and XML queries represent a significant result, that is ready for commercial deployment, even higher level of expressive power and more exciting applications can be envisioned with the approach proposed in this chapter. In fact, the expressive power of K*SQL can be formally characterized in terms of a recently proposed model, called Visibly Pushdown Languages (VPL), and thus, K*SQL can query other hierarchical structures besides XML,

such as procedural traces and genomic data (e.g., see Section 3.7.2).

Similar to regular languages, VPLs can be recognized by two equivalent representations: Visibly Pushdown Automata (VPA) and Visibly Pushdown Effects (VPEs)[Pit05]. Also, VPLs are equivalent to languages definable in Monadic Second Order logic with a matching relation μ, a.k.a. MSO_μ [AM04]. For background on VPL and VPE, and the proof of the following theorem see Section 3.7.5.

Theorem 5 (K*SQL3)**.** *K*SQL3 can express all Visibly Pushdown Expressions, and therefore can recognize all Visibly Pushdown languages and nested words.*

3.3 Optimization

Here we briefly cover some of the core ideas that we have developed for the optimization of K*SQL.

3.3.1 Compile-time Optimization

At the compile time, we perform two important steps: query rewriting, and pre-calculating several offline matrices which are used by the optimization engine at run-time.

3.3.1.1 Query Re-writing

The compiler translates the K*SQL query into a special VPA (Visibly Pushdown Automata) where the transitions are made based on the predicates of the `WHERE` clause, and the states correspond to the pattern variables. The K*SQL parser categorizes the predicates into three types: **Context Free (CF)**, **Running Context Sensitive (RCS)**, and **Final Context Sentitive (FCS)**. In summary, running predicates (i.e., CF and RCS)

are preconditions which are assigned to the states, and are evaluated upon examining each tuple for that state, while final predicates (i.e., FCS) are postconditions which are assigned to the outgoing edges, and are examined only upon leaving a state. CF predicates are those predicates whose latest results can be cached in our in-memory history structure (part of the run-time system). For instance, the results of predicates that involve aggregates, are considered context sensitive (they depend on the assignment of more than one tuple), and thus, are not cached.

In a naive implementation, an impossible match with a Kstar may not be detected until the end of the input window, i.e. when the post-conditions are finally checked. To avoid this, we re-write the FCS predicates into an equivalent form, by splitting them into a running part (weaker version) and a final one (the stronger condition). This way, the running part serves as a precondition and prunes many unpromising attempts earlier on, even before the end of the input is reached. For example, `max(B.price) = 18` is equivalent to `B.max(price) ≤ 18 AND max(B.price) = 18` while the first conjunct in the latter form, is RCS and hence, can be checked as a precondition. Analogous rewritings are possible for `min` and `count` and even for more complex postconditions involving a combination of these aggregates. Another important case of such query re-writing applies to our nested constructs. For instance, `isElement(B)` is split into two separate checks: (i) the stack for `B` must be empty in the end, and (ii) the stack for `B` must stay valid at all times. Thus, while `isElement` is by its nature a context sensitive postcondition, it is translated into FCS and RCS parts. The compile-time optimizer adds all these weaker preconditions to the `WHERE` clause, in order to optimize the execution.

3.3.1.2 Off-line Optimization Matrices

K*SQL infers an implication graph [SZZ04] from the `WHERE` clause to capture the implications between different parts of the pattern. In order to optimize the pattern search, several offline tables are pre-calculated which will later guide the pattern search at run-time. We briefly mention the more important ones (P_j refers to the j'th element of a given pattern P):

• $Jump[j]$: How far should the pattern be shifted to the right, if a mismatch occurs on P_j.

• $Next[j]$: The earliest position in the pattern that we need to check for a match, once we shift the pattern by $Jump[j]$.

• $NETB$ (Not Even Try Before): A table to infer and remember the earliest position before which we should not attempt any matches. This is mainly used for `isElement` where the distance between an open and its close tag is used to skip many unpromising tuples, as soon as a mismatch occurs.

$NETB$ can be simply calculated by recording the matching close tag for each open tag, and the calculation for the first two are similar to [SZZ04, KMS08] (with some corrections).

3.3.2 Optimization for Nested Constructs

The main construct of K*SQL for querying hierarchical structures, is the `isElement`. The K*SQL system applies several compile-time optimizations for this construct. For run-time optimizations of the nested constructs, we have developed another algorithm, called *VPSearch* which generalizes the Knuth-Morris-Pratt algorithm [KJP77] to the case of pattern matching over visibly push-down words.

Assume that we are searching for pattern $P = \langle \mathtt{a} \rangle \mathtt{b} \langle \mathtt{a} \rangle \langle \mathtt{c} \rangle \mathtt{b} \langle /\mathtt{c} \rangle \langle /\mathtt{a} \rangle \langle /\mathtt{a} \rangle$. Failing to recog-

nize the hierarchical structure, any word search algorithm will consider $\hat{\Sigma} = \{a, \langle a \rangle, \langle /a \rangle, b, \langle b \rangle, \langle /b \rangle, c, \langle c \rangle, \langle /c$
as the alphabet. For instance, KMP [KJP77] or OPS [SZZ04] will start scanning the input from left to right, until a mismatch occurs, as shown in the example of Figure 3.3, where the first failure is when P_4 mismatches with T_4 (step I). Using their prefix functions, KMP/OPS shift the pattern by 2 positions, and since $Next[4] = 2$, their next comparison will be between T_4 and P_2 (step II). After the second failure, since $Next[2] = 1$, those algorithms compare T_4 with P_1 (step III), and only after the third failure, they finally move the input pointer to T_5.

However, by exploiting the hierarchical structure, we could avoid most of these unnecessary checks. In fact, by analyzing the pattern P, we knew a priori the distance of each open tag from its close tag. For instance, for P_1, this distance is 7 (since it matches with P_8) while for the second $\langle a \rangle$ this distance is 4. Thus, after the first mismatch in step I, we could immediately infer that T_3 is an open tag that closes after 1 tuple, and thus can never match with either of P_1 or P_3. This would allow us to skip the next two checks (steps II, III) and immediately resume the search from T_7. Note that KMP/OPS were not able to skip those checks, since they only look at the equality of the symbols but not at the hierarchical edges.

The observation made above is the main idea behind the VPSearch algorithm where we use a 2-dimensional prefix array instead of the KMP's 1-D prefix. In summary, when the implication graph for a given query is complete, the VPSearch achieves the same linear-time optimality for nested words as KMP does for words. The memory complexity is $O(d)$ where d is the maximum depth of the given XML.

i	1	2	3	4	5	6	7	8	9	10	11
T	$\langle a \rangle$	b	$\langle a \rangle$	$\langle /a \rangle$	$\cdots$						
P	$\langle a \rangle$	b	$\langle a \rangle$	$\langle c \rangle$	b	$\langle /c \rangle$	$\langle /a \rangle$	$\langle /a \rangle$			
(I)				$\uparrow$							
			$\langle a \rangle$	b	$\langle a \rangle$	$\langle c \rangle$	b	$\langle /c \rangle$	$\langle /a \rangle$	$\langle /a \rangle$	
(II)				$\uparrow$							
				$\langle a \rangle$	b	$\langle a \rangle$	$\langle c \rangle$	b	$\langle /c \rangle$	$\langle /a \rangle$	$\langle /a \rangle$
(III)				$\uparrow$							

$$T\colon \langle a \rangle\ b\ \overbrace{\langle a \rangle\ \langle /a \rangle}^{1}\ \cdots$$

$$P\colon \underbrace{\langle a \rangle\ b\ \overbrace{\langle a \rangle\ \langle c \rangle\ b\ \langle /c \rangle\ \langle /a \rangle}^{4}\ \langle /a \rangle}_{7}$$

Figure 3.3: KMP versus VPSearch for pattern matching against visibly pushdown words.

3.4 Experiments

The goal of our experiments is to study the amenability of K*SQL queries to efficient execution. Thus, we compare the efficiency of XML queries written in K*SQL to those run on the state-of-the-art XPath/XQuery engines. We also study the effectiveness of our optimization on the execution time, as well as the contribution of each of our optimization techniques to the overall performance.

We have implemented the parser, optimizer and the run-time query execution engine for K*SQL, all in Java. For data I/O and storage, we use the Stream Mill [Ba07a] API which is an extensible DSMS, providing access methods for both stored and streaming data.

Experiments were conducted on a 1.6GHz Intel Quad-Core Xeon E5310 Processor running Ubuntu 6.06, with 4GB of RAM. For complex sequence queries we used real-world datasets including world crude oil prices[11], a year of historical data for the S&P 500 stocks[12] ($125K$ records), and more than $7.6M$ NASDAQ records[13] since 1970.

[11] Official energy statistics of the US government, www.eia.doe.gov
[12] http://biz.swcp.com/stocks/
[13] http://infochimps.org/dataset/stocks_yahoo_NASDAQ

For XML, we used well-known benchmarks: Protein Sequence Database[14] (600MB, avg depth 5), Shakespeare plays[15] (8MB, avg depth 6) and XMark [Sa02]. Due to lack of space and the similarity of the results, for each experiment we only report the results on one dataset.

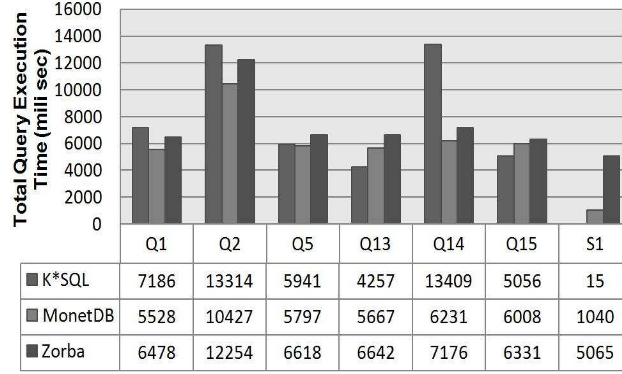
3.4.1 XML queries in K*SQL

We used the XMark benchmark to compare the execution time of their queries on native XML processors, versus the same queries that were run in K*SQL (using our XPath translation algorithm, Section 3.7.3). We compared against two of the fastest academic and industrial engines, MonetDB/XQuery [Ba06] and Zorba [Ba09], respectively. Since these two engines are written in C/C++, we transformed our java bytecodes into binary executables using Excelsior JET 7.0. (Natively coded C/C++ algorithms are typically much faster than JET generated binaries).

Out of the 20 XMark XQuery queries, Q1, Q2, Q5, Q13, Q14, Q15 were easily expressible in XPath. In Figure 3.4(a), we report the total execution time for these queries, on an XMark dataset of size 57MB. We have also run several sequence queries on Nasdaq transactions (embedded in XML tags). For instance, in Figure 3.4(a), S1 is the 'V'-shape query (similar to Example 5) that we ran for 20KB of data (the XPath engines could not easily handle larger data, since the XPath query for finding 'V' patterns involves several nested joins). In summary, despite the maturity of the research on XPath optimization, K*SQL achieves a very competitive performance on conventional queries, while for sequence queries involving Kstars (such as S1), K*SQL queries are consistently faster than their XPath counterparts, by several orders of magnitude.

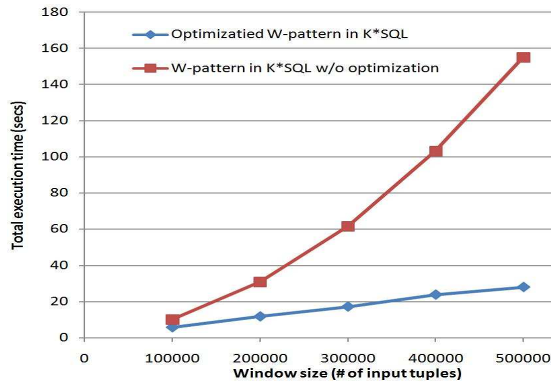
---

[14] http://www.cs.washington.edu/research/xmldatasets
[15] http://www.cafeconleche.org/examples/shakespeare

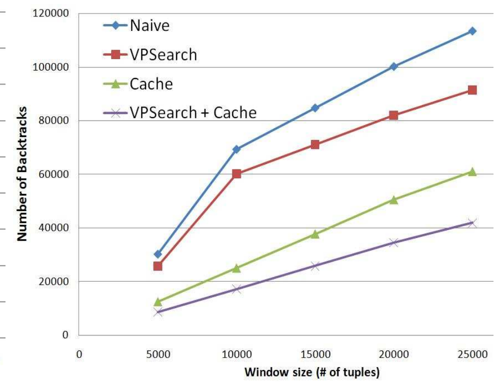


	Q1	Q2	Q5	Q13	Q14	Q15	S1
K*SQL	7186	13314	5941	4257	13409	5056	15
MonetDB	5528	10427	5797	5667	6231	6008	1040
Zorba	6478	12254	6618	6642	7176	6331	5065

(a)



(b)



(c)

Figure 3.4: (a) XML queries in K*SQL vs. native XML engines. (b) W-shape pattern in K*SQL: optimized vs. straightforward implementation. (c) Contribution of different parts of the K*SQL optimization on the overall performance.

3.4.2 Query Execution Time

Sequence queries written in K*SQL enjoy a high level of efficiency through the proposed optimization techniques. Depending on the query and input, our optimization can improve the execution time of a K*SQL query by several folds. Due to lack of space, here we only report the results for double-bottom (W-shape) query over the NASDAQ dataset, shown in Figure 3.4(b). The optimized query runs from 1.5x to 6x times faster, and the gap becomes larger as the number of input tuples increases.

3.4.3 Number of Backtracks

We further evaluated each part of our optimization techniques, in isolation, to gain better insight on their effect on the execution of K*SQL queries. In Figure 3.4(c), we report the number of backtracks during the execution of the 'V'-shape query (i.e., $\mathtt{A}^+\mathtt{B}^+$), over Nasdaq transactions, embedded in XML format. Here, we only focus on two main parts of K*SQL optimization for XML queries, namely VPSearch and caching—whereby a compact bitmap retains the result of predicate evaluations on the recent tuples. For this query, on average, caching (which itself uses the implication graph) reduced the number of unnecessary backtracks by 55% (compared to the naive implementation). The contribution of VPSearch to the overall performance of this query is limited (i.e., 16%) due to the low depth (i.e., 3) of the XML structure for Nasdaq transactions which only allows for a few tags to be skipped after each mismatch. However, VPSearch combined with the cache structure reduce the backtracks by 70%.

3.5 Related Work

The original SQL-TS language [Sa01, SZZ01, SZZ04], led to the recently proposed extension of SQL standards called SQL Match-Recognize (SQL-MR) [ZWC07] which features K*SQL1 kind of constructs. Simple optimizations of nested Kstars were addressed in [KMS08]. The use of these languages in temporal queries was discussed in [Zan09a, JS08], and further applications were demonstrated in the recent implementation of SQL-MR in [Da09].

Another major area that benefits from our proposal is Complex Event Processing (CEP), where pattern matching is a means for discovering complex events. The SASE language [WDR06], was designed for CEP over data streams, and was recently extended in SASE+ [GAD08] which provides a special syntax for allowing (i.e., skip-

ping) irrelevant tuples in between those that match a given pattern (see Section 3.2.2). Another CEP system is Cayuga [Da07] that comes with a SQL-like language (called CEL) for expressing queries over event streams. CEL has a FOLD operator, that skips an a-priori unknown number of tuples. However, expressing a pattern with more than one Kstar element requires writing nested queries that are inherently hard to optimize. The patterns expressible in CEL are a subset of those expressible in SQL-MR. The CEDR language [Ba07b] also has sequencing operators, but does not support Kstars. A recent system is the Microsoft CEP server[Aa09] which is based on the LINQ language (an extension to `.NET`, as a built-in query language).

Query automata have been recently proposed [MV09] for the evaluation of MSO formulas on nested words.

3.6 Summary of K*SQL

In this chapter, we proposed powerful generalizations for the Kleene-closure constructs that have recently been the focus of much research and commercial interest. Our extensions support more complex pattern queries both on linear sequences and on XML data—in fact the queries supported by XPath are a subset of those supported by our K*SQL language. The chapter also introduced powerful query optimization techniques whereby K*SQL can be implemented very efficiently on both relational sequences and hierarchical data such as XML. Having a unified execution engine that efficiently supports different data models and their query languages represents an exciting development for both data bases and data stream management systems. There is also potential for further benefits, given that K*SQL can express Visibly Pushdown Expressions—a powerful generalization of regular expressions that has been successfully applied to software analysis and genomic data. The competitive performance, compared to mature XML technology, achieved by by K*SQL is remarkable consider-

ing that the latter is still in its infancy and provides greater expressive power than Core XPath 2.0.

3.7 K*SQL Syntax and Expressive Power

3.7.1 K*SQL Syntax

The K*SQL syntax extends the $\langle$simple table$\rangle$ construct of the SQL:2003 standard. The BNF grammar is provided in Figure 3.5. The definition of several non-terminal symbols, such as $\langle$identifier$\rangle$ and $\langle$derived column$\rangle$, have been omitted from Figure 3.5, since they are identical to those in the ANSI/ISO standard[16] for SQL:2003.

The syntax for the additional method invocations that K*SQL supports as built-in functions are as follows. Both `open()` and `close()` methods accept an expression of type string as argument and return a string value. The `isElement()` function accepts a $\langle$column reference$\rangle$ as argument and return a boolean value.

Note that throughout the chapter, for clarity purpose, we have used:
$\langle$pattern element base$\rangle$ = `open/close`($\langle$expression$\rangle$)
as a shorthand for:
$\langle$pattern element base$\rangle$ · `XmlColName` = `open/close`($\langle$expression$\rangle$).
Similarly, we have used
`isElement`($\langle$pattern element base$\rangle$)
as a shorthand for:
`isElement`($\langle$pattern element base$\rangle$ · `XmlColName`).

[16] ISO/ANSI Foundation (SQL/Foundation), http://www.iso.org.

⟨simple table⟩	←	⟨sequence query spec⟩|⟨query specification⟩
		|⟨table value constructor⟩|⟨explicit table⟩
⟨sequence query spec⟩	←	SELECT ⟨seq select list⟩
		⟨from clause⟩
		PARTITION BY ⟨column reference⟩
		⟨order by clause⟩
		⟨pattern clause⟩
		⟨where clause⟩
⟨seq select list⟩	←	⟨derived column⟩[, ⟨derived column⟩...]
⟨pattern clause⟩	←	AS PATTERN ′(′ ⟨pattern⟩ ′)′
⟨pattern⟩	←	(⟨atomic pattern⟩|⟨compound pattern⟩)[⟨pattern⟩]
⟨atomic pattern⟩	←	⟨pattern element⟩[⟨pattern repetition⟩]
⟨compound pattern⟩	←	(⟨pattern element⟩ : ⟨pattern list⟩)[⟨pattern repetition⟩]
⟨pattern element⟩	←	⟨identifier⟩
⟨pattern repetition⟩	←	+| ∗ |⟨unsigned integer⟩
		|{⟨unsigned integer⟩ : ⟨unsigned integer⟩}
		|{ : ⟨unsigned integer⟩}|{⟨unsigned integer⟩ : }
⟨pattern list⟩	←	⟨pattern⟩[⟨pattern list⟩]
⟨column reference⟩	←	⟨pattern base⟩ ′.′ ⟨column name⟩
⟨pattern base⟩	←	⟨pattern element⟩|⟨pattern base⟩ ′.′ ⟨pattern element⟩
		| (PREV |NEXT |FIRST |LAST) ′(′ ⟨pattern base⟩ ′)′

Figure 3.5: Formal syntax for K*SQL. The starting rule for K*SQL is ⟨sequence query spec⟩, which extends the ⟨simple table⟩ construct of SQL:2003.

3.7.2 K*SQL for Other Domains

As briefly mentioned in Section 3.1.2, the power of K*SQL in querying linear-hierarchical data is not limited to XML, and its `isElement` construct is not dependant on a particular SAX representation. To see the latter, note that `isElement(B)` is used as a shorthand for `isElement(B.myXmlTag)` where we could replace `myXmlTag` with any other

column name under which the original xml tags are stored (same applies to `open()` and `close()`). Also, these constructs are not XML-specific: in general, for any domain that can be represented by nested words, the user only needs to redefine the `open()` and `close()` functions, which are, internally invoked by `isElement()`, and thus, we do not need to re-implement `isElement` for every new domain. For instance, in running static analysis over programming traces, the `open()` function detects a *function call*, while the `close()` detects it's corresponding *return statement(s)*. Similarly, in RNA sequences (genomics), intra-strand base pairing occurs between guanine (G) and cytosine (C) pair which can be modeled as corresponding open and close symbols, and so can adenine (A) and uracil (U) pair (see [Aa90, AM04] for more on the representation of RNAs as nested words.).

3.7.3 Proof of Theorem 1 (Algorithm)

Here, we provide a simple constructive proof for Theorem 1, that shows we can algorithmically construct an equivalent K*SQL query for any given XPath expression. Our algorithm starts by rewriting the leftmost axis-step into a K*SQL query. Then, at each step, iteratively, the pattern clause of the existing K*SQL query is updated, depending on the type of the current axis specifier. The predicates on the current level of XML nodes are moved to the WHERE clause of K*SQL, while nested expression patterns are independently translated into K*SQL, which then will be intersected with the answer set of the current K*SQL query.

To focus on the navigational fragment of XPath, in the following, we use the syntax of core XPath 1.0 [CM07b] combined with XPath 2.0 [CM07a]. This syntax is presented in Figure 3.6. To further simplify the discussion, we also omit the 'reference' and 'for loop' of XPath 2.0, as they can be trivially emulated in K*SQL using variables and conjunctions, respectively.

Axis	:=	self | child | parent
		| descendant | ancestor
		| following | preceding
		| following_sibling | preceding_sibling
NameTest	:=	QName | *
Step	:=	Axis::NameTest | Axis::NameTest[NodeExpr]
PathExpr	:=	Step
		| PathExpr/Step
		| PathExpr union PathExpr
		| PathExpr intersect PathExpr
		| PathExpr except PathExpr
NodeExpr	:=	PathExpr | not NodeExpr
		| NodeExpr and NodeExpr
		| NodeExpr or NodeExpr

Figure 3.6: Syntax of Core XPath 1.0 combined with 2.0.

In core XPath, the start production is `PathExpr`. We inductively translate a `PathExpr` into an equivalent K*SQL query. Whenever the production rule is 'union', 'intersect', or 'except' we rewrite the expression into separate paths, and then inductively, translate each path expression independently; in the end we use respectively use union operator |, K*SQL intersection, and negation of the predicates to combine the sub-queries. Therefore, we only need to concentrate on the `PathExpr/Step` production. As our induction hypothesis we assume that we know how to translate the first $k-1$ steps of the given expression from the left, into an equivalent K*SQL query, as follows:

SELECT $X_{i_1}, \cdots, X_{i_k}$
FROM XmlStream
ORDER BY tokenId
AS PATTERN $((X_1 : A_1 \cdots E_j^* \cdots (X_2 : A_2 \cdots B_2)\hat{\ }^{t_2} \cdots B_1)\hat{\ }^{t_1}$
$\cdots E_{j'}^* \cdots (X_3 : A_3 \cdots B_3)\hat{\ }^{t_3})$

WHERE where_clause

After translating each step, the pattern clause consists of a list of well-nested elements, i.e. $(X_i : A_i \cdots B_i)$ or E_j^* where A_i and B_i are corresponding open/close tags and E_j is a well-nested element. The t_i's denote the occurrence of their element, i.e. whether they are a star element ($t_i = *, +$) or a simple singleton ($t_i = 1$). The select clause outputs a subset of X_i's, such that the selected tuples are precisely the XML tags that correspond to the XPath expression upto the current Step. For the base case of $k = 1$, the select and where clauses are empty and the pattern clause consists of a simple (E_0^*). Now, assuming that we have an K*SQL query in the format above that is equivalent to the first (leftmost) $k - 1$ steps of the given `PathExpr`, we show how to construct a new K*SQL query that is equivalent to the first $k > 0$ steps. Depending on the `Axis` of the k'th step, we have the following cases for `Axis::NameTest` (node filter `[NodeExpr]` is addressed separately):

self: If the `NameTest` is a `QName`, for every X_i in the current select clause, we add the following predicate to the where clause: $A_i = \mathit{open}(QName)$. When the `NameTest` is '*' we do not need to change the where clause.

child: For every X_i in the current select clause, we replace it with all of its 'immediate children' in the current pattern definition, as follows: an immediate child of X_i is defined as either an ${E_j}^{t_j}$ or an $(X_j : A_j \cdots B_j)^{t_j}$ that appears between A_i and B_i without being enclosed in any sub-patterns of X_i. For the immediate children of X_i that are of form X_j, we just add X_j to the select clause, while for the immediate children of form ${E_j}^{t_j}$, we first replace them with the new[17] pattern ${E_{j_1}}^*(X_j : A_j {E_{j_2}}^* B_j)^{t_j} {E_{j_3}}^*$, and then we add the new X_j to the select clause. Trivially, in the where clause we declare all the new E, A and B variables as `isElement`, open and close, respectively.

[17]In this proof, whenever we add new variables to the pattern clause we assure that the new variable names are different from the existing names.

In the end, we remove the original X_i and also duplicate X_j's from the select clause. The where clause is also updated to reflect the `NameTest` requirement, similar to the 'self' axis above.

parent: For every X_i in the current select clause, we replace it with its 'immediate parent' in the current pattern definition, as follows: an immediate parent of X_i is defined as the first upper level $(X_j : A_j \cdots B_j)$ that encloses X_i. If such X_j does not exist for a given X_i (i.e., when X_i is the root element) we eliminate X_i from the select clause without adding any new variables. Otherwise, we replace X_i with X_j and update the where clause appropriately to reflect the `NameTest` requirement for A_j. Duplicate X variables are removed from the select clause to avoid identical outputs.

descendant (ancestor): For every X_i in the current select clause, we replace it with its 'descendants' ('ancestors'), as follows: a descendant (ancestor) of X_i is defined as either an ${E_j}^{t_j}$ or an $(X_j : A_j \cdots B_j)^{t_j}$ (for ancestor, it can be only of form $(X_j : A_j \cdots B_j)^{t_j}$) that is enclosed between A_i and B_i (for ancestor, X_j should be enclosing X_i definition). For the descendants of X_i that are of form X_j, we just add X_j to the select clause, while for the descendants of form ${E_j}^{t_j}$, we first replace them with the new pattern ${E_{j_1}}^*(X_j : A_j {E_{j_2}}^* B_j)^{t_j} {E_{j_3}}^*$, and then we add the new X_j to the select clause. (For ancestors, we simply replace X_i with all its ancestor X_j's.) Trivially, in the where clause we declare all the new E, A and B variables as `isElement`, open and close, respectively. In the end, we remove the original X_i and also duplicate the X_j's from the select clause. The where clause is also updated to reflect the `NameTest` requirement for A_j.

following_sibling (preceding_sibling): For every X_i in the current select clause, we replace it with its 'next' ('previous'), as follows: next (previous) of X_i is defined as the variable that immediately follows (precedes) the definition of X_i in the pattern clause, and has the same immediate parent as X_i. If such a variable does not exist,

we simply remove X_i from the select clause. If the next (previous) is of form X_j, we just add X_j to the select clause, while for variables of form ${E_j}^{t_j}$, we first replace them with the new pattern $(X_j : A_j {E_{j_1}}^* B_j){E_{j_2}}^{t_j}$ (for previous, we replace ${E_j}^{t_j}$ with ${E_{j_2}}^{t_j}(X_j : A_j {E_{j_1}}^* B_j))$, and then we add the new X_j to the select clause. Trivially, in the where clause we declare all the new E, A and B variables as `isElement`, open and close, respectively. In the end, we remove the original X_i and also duplicate X_j's from the select clause. The where clause is also updated to reflect the `NameTest` requirement for A_j.

following (preceding): For every X_i in the current select clause, we replace it with its 'rights' ('lefts'), as follows: right (left) of X_i is defined as any variable that follows (precedes) the definition of X_i in the pattern clause. If no such variable exists, we simply remove X_i from the select clause. If the right (left) is of form X_j, we just add X_j to the select clause, while for variables of form ${E_j}^{t_j}$, we first replace them with the new pattern ${E_{j_1}}^{t_j}(X_j : A_j {E_{j_2}}^* B_j){E_{j_3}}^{t_j}$, and then we add the new X_j to the select clause. Trivially, in the where clause we declare all the new E, A and B variables as `isElement`, open and close, respectively. In the end, we remove the original X_i and also duplicate X_j's from the select clause. The where clause is also updated to reflect the `NameTest` requirement for A_j.

Adding node filters. In navigational XPath [CM07b], node expressions are used as node filters, with an existential semantic, i.e. $R[N]$ is the subset of nodes satisfying path expression R from which node expression N evaluates to at least one node. Thus, for translating a path expression $R[N]$, we apply the process above to translate R first, then by appending N to R we have another path expression that can be similarly translated into a separate query in K*SQL, which then will be added as a conjunct. When the node expression contains 'not' we first negate the pattern (through its where clause) and then add it a conjunct; Similarly, for node expressions with 'or'/'and', we

use disjunctive/conjunctive sub-queries, accordingly.

For instance, for translating $R[N_1 \; or \; N_2]$ we will have:

SELECT select_clause_for_R

...

WHERE where_clause AND (

EXISTS (K*SQL query for R/N_1)

OR EXISTS (K*SQL query for R/N_2))

3.7.4 XPath for Sequence Queries

XPath is strictly subsumed by K*SQL. Core XPath 2.0 represents a fragment of XPath that is complete for First Order (FO) logic for trees [CM07b]. From Theorem 5 we know that K*SQL is as expressive as VPLs which are equivalent to monadic second order (MSO) logic over nested words [Pit05]. Thus, K*SQL is strictly more expressive than Core XPath 2.0.

Optimization of sequence queries in XPath/XQuery. While there are MSO queries over XML that cannot be expressed in Core XPath 2.0 (e.g., modulo counting [Pot94] such as returning every 4'th tag), and FO queries that cannot be expressed in Core XPath 1.0 (see [CM07b] for an example), in practice, the main deficiency of XQuery and XPath in expressing sequence queries lies in the inevitable complexity of such queries, which compromises their optimization and readability. For instance, consider the following simple sequence query over XML:

Example 7. *For the following stock data xml, find the decreasing sequences of consecutive close prices, with length at least 1.*

```
<Stocks>
    <Stock close="0.98"/>
    <Stock close="0.95"/>
```

```
    ....
</Stocks>
```

Below is a possible way of writing this query, which clearly exemplifies the limited room for optimizations of such complex queries in XPath/XQuery[18]:

```
<results>{
  for $t1 in doc("mydoc.xml")//Stock
  return <result><head>{$t1/@close}{
    for $t4 in $t1/following-sibling::Stock
    let $x:=(for $x in $t1/following-sibling::Stock
                where $x<<$t4 return $x)
    where $t4/@close<=$t1/@close
      and (every $t2 in $x satisfies
              $t2/@close<=$t1/@close and
              $t2/@close>=$t4/@close)
      and (every $t2 in $x, $t3 in for $x in
             $t2/following-sibling::Stock
             where $x<<$t4 return $x
           satisfies $t2/@close>=$t3/@close
                     and $t3/@close>=$t4/@close)
    return <tail>{$t4/@close}</tail>
  }</head></result>}</results>
```

This situation becomes significantly worse if we want to search for several Kstar patterns. However, such queries can be easily represented as a regular expression in K*SQL (see Example 5).

3.7.5 From VPE to K*SQL

Background on Visibly Pushdown Expressions. The class of visibly pushdown lan-

[18]None of the available XQuery engines were able to execute this query on any XML document larger than a few kilobytes.

guages (VPL) has been proposed [AM04] as embeddings of context-free languages that is rich enough to model data with hierarchical relations (such as XML, software analysis, and RNA) and yet is tractable and robust like the class of regular languages. Visibly pushdown automata (VPA) recognize VPLs, where the input symbol determines when the stack should be pushed or popped.

Pitcher [Pit05] generalized the notion of regular expressions for representing VPLs, called Visibly Pushdown Expressions (VPE). VPEs represent another equivalent notion for VPLs: every VPL can be expressed as a VPE, and every VPE can be translated into a monadic second order logic (MSO) over a nested relation, and there exists a VPA that accepts the same language that that VPE expresses. Below is the formal definition of a VPE:

The symbol patterns used in a VPE are defined as follows (where Σ_c, Σ_r and Σ_i are the set of call, return and internal symbols, respectively):

$$
\begin{array}{lcr}
p ::= & a & \text{(symbols, } a \in \Sigma_c \cup \Sigma_r \cup \Sigma_i\text{)} \\
| & p + p & \text{(union)} \\
| & \neg p & \text{(complement)} \\
| & \sim_c & \text{(wildcard for } \Sigma_c\text{)} \\
| & \sim_r & \text{(wildcard for } \Sigma_r\text{)} \\
| & \sim_i & \text{(wildcard for } \Sigma_i\text{)}
\end{array}
$$

In the following, we use the abbreviation $p_1 \& p_2$ to denote $\neg(\neg p_1 + \neg p_2)$. Also, P_c refers to all symbol patterns of the form $\sim_c \& p$, and so on. Thus, a well-matched VPE

(denoted as T) is defined as:

$$
\begin{aligned}
T ::= \quad & \phi && \text{(empty set)} \\
| \quad & () && \text{(empty sequence)} \\
| \quad & p && \text{(symbol pattern where, } p \in P_i\text{)} \\
| \quad & p_1[T]p_2 && \text{(element, } p_1 \in P_c, p_2 \in P_r\text{)} \\
| \quad & T.T && \text{(concatenation)} \\
| \quad & T+T && \text{(union)} \\
| \quad & T\&T && \text{(intersection)} \\
| \quad & A && \text{(VPE variable)} \\
| \quad & T* && \text{(repetition)}
\end{aligned}
$$

And finally, below is the grammar for VPEs:

$$
\begin{aligned}
S ::= \quad & T && \text{(Well-nested VPE)} \\
| \quad & p && \text{(symbol pattern)} \\
| \quad & S.S && \text{(concatenation)} \\
| \quad & S \oplus S && \text{(overlapped concatenation)} \\
| \quad & S+S && \text{(union)} \\
| \quad & S\&S && \text{(intersection)} \\
| \quad & S* && \text{(repetition)}
\end{aligned}
$$

Here, the $\oplus$ operator insists that the last symbol of the first string is the same as the first symbol of the second string, e.g. $a \oplus a.b = a.b$, but $a \oplus (b+c)$ denotes an empty language. Next, we show how our K*SQL3 language can encode any arbitrary VPE.

3.7.5.1 Proof of Theorem 5

Proof. We prove this by induction, with the base case being the expression of the symbol patterns.

Expressing symbol patterns (SP). An arbitrary symbol a in K*SQL3 is a simple pattern A with a predicate $A = a$. The union of two SPs A and B can be written as $A|B$ in the pattern clause with the disjunction of their predicates in the where clause. The complement of p is derived by negating the predicates of the K*SQL3 query for p. Wildcards for calls, returns and internal symbols can be encoded using simple checks, e.g. `A =` c_1 `OR` $\cdots$ `OR A =` c_k for all $c_i \in \Sigma_c$ and so on.

Expressing well-matched VPEs. Empty sets and sequences are trivial. SPs $p \in P_i$ are derived by encoding p inductively, and then adding a conjunctive predicate to enforce that all the symbols are internal. For $p_1[T]p_2$, once we recursively encode p_1, T and p_2, we append their patterns and conjunct their predicates. Note that according to our induction assumption, p_1 and p_2 are guaranteed to be made of open and close tags, i.e. using predicates. Concatenation is encoded by first renaming all the variables such that the two K*SQL3 queries do not share any variables. Then, we append the pattern parts of the queries and conjunct their predicates. Intersection, VPE variables and repetition (a.k.a. Kstar) are directly supported by K*SQL3.

Expressing arbitrary VPEs. T and p can be encoded by our induction assumption. Concatenation, union, intersection and repetition are encoded similarly to their well-matched counterparts. Note that K*SQL3 does not require the pattern to be well-nested, e.g. a check for an open tag does not have to be accompanied by a corresponding check for its close tag. The overlapped concatenation, $S_1 \oplus S_2$, will be encoded as follows. We rename all the variables of the K*SQL3 patterns for S_1 and S_2, to assure that they do not share any variable names. Assume that the first variable of S_2 is v_2 and the last variable of S_1 is v_1. We conjunct the predicates of the K*SQL3 queries for S_1 and S_2, and append their patterns. We then add the following predicate to the resulting K*SQL3 query as a conjunctive term: `last(v1) = first(v2)`. $\square$

3.7.6 Aggregates and Complexity

Similar to other practical query languages, K*SQL also allows certain aggregates to appear in the predicates. For instance, SASE+ allows any associative aggregation operation with an identity element and an NC^1 iterated multiplication algorithm. Once we allow the same set of aggregates in K*SQL and all of the languages discussed in Section 3.2.2, we will achieve similar complexity results.

For instance, the ordered graph reachability problem (called oREACH) can be expressed in a simple SASE+ query (using its 'skip till any match' mode) without using any aggregates [Da08]. Thus, according to Theorem 4, K*SQL2 can also express oREACH which is NSPACE[log n]-complete. However, even after allowing the aforementioned class of aggregates in K*SQL2, the new language will be still contained in NSPACE[log n], since similarly to SASE+, the formulas can be simulated in NC^1, where for the aggregates we perform a partial-prefix computation. Similarly for K*SQL3, after allowing such aggregates, the new language will be still contained in NSPACE[log n].

Thus, in summary, both K*SQL2 and K*SQL3, once enhanced with predicates that have aggregate functions discussed above, can express a subset of NSPACE[log n] including some problems that are complete for NSPACE[log n].

CHAPTER 4

XSeq: High-Performance Complex Event Processing over Hierarchical Data

XPath is an important query language on its own merits and also because it serves as the kernel of other languages used in a wide range of applications, including XQuery, several graph languages [SS09], and OXPath for web information extraction [FGG11]. Much work has also focused on the efficient support for XPath in the diverse computational environments required by these applications. In particular, finite state automata (FSA) have proven to be very effective at supporting XPath queries over XML streams [Koc09], and are also apt at providing superior scalability through the right mix of determinism versus non-determinism. In fact, numerous XML engines have been successfully built for efficient and continuous processing of XML streams [CDZ06, PC03, OKB03, Ba03, JFB05, Fa03, DAF03]. All these systems support full or fragments of XPath or XQuery, and thus, naturally inherit the pros and cons of these languages. The simplicity of XPath and the generality of XQuery have made them very successful and effective for general-purpose applications. However, these languages lack explicit constructs for expressing Kleene-* and sequential patterns—a vital requirement in many CEP applications[1]. As a result, while the existing engines remain

[1]There are several definitions of CEP applications [BGH09, Luc01, WDR06], but they commonly involve three requirements: (i) complex predicates (filtering, correlation), (ii) temporal/order/sequential patterns, and (iii) transforming the event(s) into more complex structures. In this paper we mainly focus on (i) and (ii) while achieving (iii) represents a direction for future research, e.g. by embedding our language (called XSeq) inside XSLT.

very effective in general-purpose applications over XML streams, their usability for CEP applications (that involve complex patterns) becomes highly limited as none of these engines provide any explicit sequencing/Kleene-* constructs over XML.

To better illustrate the difficulty of expressing sequence queries in existing XML engines (that mostly support fragments of XPath/ XQuery), in Fig. 4.1 we have expressed a common query from stock analysis in XPath 2.0, where the user is interested in a sequence of stocks with falling prices[2]. As shown in this example, due to the lack of explicit constructs for sequencing and Kleene-* patterns, the query in XPath/ XQuery is very hard to write and understand for humans and is also difficult to optimize. We will return to this query in Section 4.2, and show that it can be easily expressed using simple sequential constructs (see Example 16 and Query 9). In fact, it is not a surprise that the general-purpose XML engines perform two orders of magnitude slower on these complex sequential queries than the same queries expressed and executed in XSeq (the language and system presented in this chapter), whereby explicit constructs for Kleene-* patterns and effective VPA-based optimizations allow for high-performance execution of CEP queries.

These limitations of XPath are not new, as several extensions of XPath have been previously proposed in the literature [Cat06, CM07b, CM07a]. However, the efficient implementation of even these extensions (often referred to as Regular XPath) remained an open research challenge, which the papers proposing said extensions did not tackle (neither for stored data nor for data streams). In fact, the following was declared to be an important open problem since 2006 [Cat06]: *"Efficient algorithms for computing the transitive closure of XPath path expressions"*.

Fortunately, significant advances have been recently made in automata theory with the introduction of Visibly Pushdown Automata [AM04, AM06]. VPAs strike a bal-

[2] In fact, in practice, stock queries tend to be much more complex, e.g. in a wedge pattern (www.investopedia.com), the user seeks an arbitrary number of falling and rising phases of a stock.

```
<result>{
for $t1 in doc("auction.xml")//Stock[@stock_symbol='DAGM']
return <head>{$t1/@close}{
  for $t4 in $t1/following-sibling::Stock[@stock_symbol='DAGM']
  where $t4/@close<=$t1/@close
    and (every $t2 in
           for $x in
             $t1/following-sibling::Stock[@stock_symbol='DAGM']
           where $x<<$t4
           return $x satisfies $t2/@close<=$t1/@close
             and $t2/@close>=$t4/@close)
    and (every $t2 in
           for $x in
             $t1/following-sibling::Stock[@stock_symbol='DAGM']
           where $x<<$t4 return $x,
           $t3 in for $x in
             $t2/following-sibling::Stock[@stock_symbol='DAGM']
           where $x<<$t4
           return $x satisfies $t2/@close>=$t3/@close
             and $t3/@close>=$t4/@close)
  return <bottom> {$t4/@close} </bottom>
}  </head>
}</result>
```

Figure 4.1: A query in XPath 2.0/XQuery for a sequence of 'falling price' in Nasdaq's XML.

ance between expressiveness and tractability: unlike pushdown automata (PDA), VPAs have all the appealing properties of FSA (a.k.a. word automata). For instance, VPAs enjoy higher expressiveness (than word automata) and more succinctness (than tree automata), while their decision complexity and closure properties are analogous to

word automata, e.g., VPAs are closed under union, intersection, complementation, concatenation, and Kleene-*; their deterministic versions are as expressive as their non-deterministic counterparts; and membership, emptiness, language inclusion and equivalence are all decidable [AM04, AM06]. However unlike word automata, VPAs can model and query any *well-nested* data, such as XML, JSON files, RNA sequences, and software traces [AM06]. What these seemingly diverse set of formats have in common is their dual-structures: (i) they all have a sequential structures (e.g. there is a global order of the tags in a JSON or XML file based on the order that they appear in the document), (ii) they also have a hierarchical structure (when XML elements or JSON objects are enclosed in one another), but (iii) this hierarchical structure is well-nested, e.g. the open tags in the XML documents match with their corresponding close tags. Data with these properties can be formally modeled as *Nested Words* or *Visibly Pushdown Words* [AM04, AM06]. (We have included a brief background on nested words and VPAs in Chapter 2.) Throughout this chapter we refer to such formats as 'XML-like' data, but for the most part we focus on XML[3].

Although these new types of automata can bring major benefits in terms of expressive power, to the best of our knowledge, their optimization and efficient implementation in the context of XPath-based query languages have not been explored before. Furthermore, the recently proposed query language K*SQL (See Chapter 3) used VPAs to achieve good performance and expressivity levels needed to query both relational and XML streams. However, while very natural for relational data, K*SQL is quite procedural and verbose for XML, whereby the equivalents of simple XPath queries are long and complex K*SQL statements. Hence, in this chapter, we introduce the XSeq language which succinctly achieves new levels of expressive power supported by a very efficient implementation technology. XSeq extends XPath with powerful

[3]Using XSeq to query other XML-like data (e.g. JSON, RNA, software traces) is straightforward and only involves introducing domain-specific interfaces on top of XSeq, e.g. see [ZYM13] for a few examples of such interfaces.

constructs that support (i) the specification of and search for complex sequential patterns over XML-like structures, and (ii) efficient implementation using the Kleene-* optimization technology and streaming Visibly Pushdown Automata (VPA).

Being able to compile complex pattern queries into equivalent VPAs has several key benefits. First, it allows for expressing complex queries that are common in CEP applications. Second, it allows for efficient stream processing algorithms. Finally, the closeness of VPAs under *union* operation creates the same opportunities for CEP systems (through combining their corresponding VPAs) that the closeness of NFAs (non-deterministic finite automata) created for publish-subscribe systems [DAF03, VMT07, LZ12], where simultaneous processing of massive number of queries becomes possible through merging the corresponding automata of the individual queries.

Contributions. In summary, we make the following contributions:

1. The design of XSeq, a powerful and user-friendly query language for CEP over XML streams or stored sequences.

2. An efficient implementation for XSeq based on VPA-based query plans, and several compile-time and run-time optimizations.

3. Formal results on the expressiveness of XSeq, and the complexity of its query evaluation and query containment.

4. An extensive empirical evaluation of XSeq system, using several well-known benchmarks, datasets and engines.

5. Our XSeq engine can also be seen as the first optimization and implementation for several of the previously proposed languages that are subsumed in XSeq but

were never implemented (e.g. Regular XPath [Cat06], Regular XPath(W) [CS08] and Regular XPath$^{\approx}$ [CM07b]).

Organization. We present the main constructs of our language in Section 4.1 using simple examples. The generality and versatility of XSeq for expressing CEP queries are illustrated in Section 4.2 where several well-known queries are discussed. Our query execution and optimization techniques are presented in Section 4.3. In order to study the expressiveness and complexity of our language, we first provide formal semantics for XSeq in Section 4.4, which is followed by our formal results in Section 4.5, including the translation of XSeq queries into VPAs, their MSO-completeness and their query evaluation and query containment complexities. Our XSeq engine is empirically evaluated in Section 4.6, which is followed by an overview of the related work in Section 4.7. Finally, we conclude in Section 4.8.

4.1 XSeq Query Language

In this section, we briefly introduce the query language supported by our CEP system, called XSeq. The simplified syntax of XSeq is given in Fig. 4.2 which suffices for the sake of this presentation. Below we explain the semantics of XSeq via simple examples. We defer the formal semantics to Section 4.4.

XSeqQuery ← ['return' Output 'from'] Pattern
    ['where' Condition] ['partition by' Pattern]

Output ← Operand [',' Output]

Pattern ← ['doc()'] PathExpr

PathExpr ← Step | PathExprDefinition
    | PathExpr PathExpr | '(' PathExpr ')' '*'
    | PathExpr 'union' PathExpr
    | PathExpr 'intersect' PathExpr

PathExprDefinition ← '(' Variable ':' PathExpr ')'

Step ← Axis NameTest Predicate *

Axis ← AxisSpecifier '::' | AbbreviatedAxisSpecifier

AxisSpecifier ← 'self' | 'child' | 'parent' | 'descendant'
    | 'ancestor' | 'attribute' | 'following_sibling'
    | 'preceding_sibling'| 'first_child'
    | 'immediate_following_sibling'

AbbreviatedAxisSepcifier ← '.' | '/' | '//' | '@' | '\' | '/\'

NameTest ← QName | '*' | Variable | KindTest

KindTest ← 'node()' | 'text()'

Predicate ← '[' (Pattern | Condition) ']'

Condition ← BoolExpr

Operand ← Constant | Aggregate '(' ArithmeticExpr ')'
    | Alias PlainStep * (AttributeStep | TextStep)

PlainStep ← Axis QName

AttributeStep ← ('attribute' '::' | '@') QName

TextStep ← ('child' '::' | '/') 'text()'

Aggregate ← 'max' | 'min' | 'count' | 'sum' | 'avg'

Alias ← SequenceAlias | PlainAlias

SequenceAlias ← ('prev' | 'first' | 'last') '(' Variable ')'

PlainAlias ← Variable

Figure 4.2: XSeq Syntax (QName, Variable, BoolExpr, Constant, and ArithmeticExpr are defined in the text).

Inherited Constructs from Core XPath. The navigational fragments of XPath 1.0 and 2.0 are called, respectively, Core XPath 1.0 [CM07b] and Core XPath 2.0 [CM07a]. The semantics of these common constructs are similar to XPath (e.g., axes, attributes). Other syntactic constructs of XPath (e.g., the $following$ axis) can be easily expressed in terms of these main constructs (see [CM07a]). In XSeq there are two new axes to express the *immediately following* [4] notion, namely $first_child$ and $immediate_following_sibling$, which are described later on. Some of the axes in XSeq have shorthands:

Axis	Shorthand
$self$	$\bullet$
$child$	/
$descendant$	//
$attribute$	@
$following_sibling$	λ (empty string, i.e. default axis)
$first_child$	/\
$immediate_following_sibling$	\

Conditions. In XSeq, a `Condition` can be any predicate which is a boolean combination of *atomic formulas*. An atomic formula is a *binary operator* applied to two *operand*s. A *binary operator* is one of $=, \neq, <, >, \leq, \geq$. An *operand* is any algebraic combination (using +, -, etc.) and aggregates of string or numerical constants, and the attributes or text contents of variable nodes.

Example 8 (A family tree.). *Our XML document is a family tree where every node has several attributes: Cname (for name), Bdate (for birthdate), Bplace (for the city of birth) and each node can contain an arbitrary number of sub-entities Son and Daughter. Under each node, the siblings are ordered by their Bdate.*

In the following, we use this schema as our running example.

[4] XSeq does not have analogous operators for immediately preceding since backward axes of XPath are rarely used in practice.

Example 9. *Find the birthday of Mary's sons.*

Query 1. //daughter[@Cname='Mary'] /son /@Bdate

Kleene-* and parentheses. Similar to Regular XPath [Cat06] and its dialects [CM07b, CS08], XSeq supports path expressions such as $/\mathtt{a}(/\mathtt{b}/\mathtt{c})^*/\mathtt{d}$, where a Kleene-* expression A^* is defined as the infinite union $\cdot \cup A \cup (A/A) \cup (A/A/A) \cup \cdots$

Example 10. *Find those sons born in 'New York', who had a chain of male descendants in which all the intermediary sons were born in 'Los Angeles' and the last one was again born in 'New York'. For all such chains, return the name of the last son.*[5]

Query 2. // son[@Bplace='NY'] (/son[@Bplace='LA'])* /son[@Bplace='NY'] /@Cname

The parentheses in $()^*$ can be omitted when there is no ambiguity. Also, note the difference between the semantics of $(/\mathtt{son})*$ and $//\mathtt{son}$: the latter only requires a son in the last step rather than the entire path.

Syntactic Alternatives. In XSeq, the node selection conditions can be alternatively moved to an optional `where` clause, in favor of readability. When a condition is moved to the `where` clause, its step should be replaced with a variable (variables in XSeq start with $\$$). Also, similarly to XPath 2.0 and XQuery, the query output in XSeq can be moved to an optional `return` clause. Query 3 below is an alternative way of writing Query 2 in XSeq. Here, $\mathtt{tag}(\$\mathtt{X})$ returns the tag name of variable $\$X$.

Query 3. return $B@Cname
from //son[@Bplace='NY'] (/$A)* /$B[@Bplace='NY']
where tag($A)='son' and $A@Bplace='LA' and tag($B)='son'

For clarity, in this chapter we mainly use this alternative syntax.

[5]*This is an example of a well-known class of XML queries which has been proven [Cat06] as not expressible in Core XPath 1.0.*

Order Semantics, Aggregates. XSeq is a *sequence query language*. Therefore, unlike XPath where the input and output are a set (or binary relation), in XSeq the XML stream is viewed as a pre-order traversal of the XML tree. Thus, both the input and the output of an XSeq query are a *sequence*. The XML nodes are ordered according to[6] their relative position in the XML document.

As a result, besides the traditional aggregates (e.g., sum, max), XSeq also supports sequential aggregates (`SeqAggr` in Fig. 4.2) which are only applied to variables under a Kleene-* For instance, the path expression `/son(/$X)*, last($X) @name` returns the name of the last `X` in the `(/$X)*` sequence. Similarly, `first($X)` returns the first node of the `(/$X)*` and `prev($X)` returns the node before the current node of the sequence. Finally, `$X @Bdate > prev($X) @Bdate` ensures that the nodes that match `(/$X)*` are in increasing order of their birth date.

Siblings. Since XSeq is designed for complex sequential queries, its default axis (i.e. when no explicit axis is given) is the 'following_sibling'. The omission of the 'following_sibling' allows for concise expression of complex horizontal patterns.

Example 11. *Find all the younger brothers of 'Mary'.*

Query 4. return $S@Cname
from //$D[@Cname='Mary'] $S
where tag($D)='daughter' and tag($S)='son'

Here, since no other axes appear between `D` and `S`, they are treated as siblings.

Immediately Following. This is the construct that gives XSeq a clear advantage over all the previous extensions of XPath in terms of expressiveness, succinctness and optimizability. We believe that one of the main shortcomings of the previous XML languages for CEP applications is their lack of explicit constructs for expressing the notion

[6] When a WINDOW is defined over the XML stream, the input nodes can be re-ordered. For simplicity of the discussion, we do not discuss re-ordering.

of '*immediately following*' (see Section 4.2). Thus, to overcome this, XSeq provides two explicit axes, \ and /\, for *immediately following* semantics. For example, `Y\X` will return the *immediately next sibling* of node `Y`, while `Y/\X` will return the *very first child* of node `Y`. Similarly to other constructs, these operators return an empty set if no such node can be found, e.g., when we are at the last sibling or a node with no children.

Example 12. *Find the first two elder siblings of 'Mary'.*

Query 5. return $X@Cname, $Y@Cname
from //daughter[@Cname='Mary'] \$X \$Y

Example 13. *Find the second child of 'Mary'.*

Query 6. return $Y@Cname
from //daughter[@Cname='Mary'] /\$X \$Y

Partition By. Inspired by relational Data Stream Management Systems (DSMS), XSeq supports a partitioning operator that is very essential for many CEP applications. Nodes can be partitioned by their key, so that different groups can be processed in parallel as the XML stream arrives. Although this construct does not add to the expressiveness, it provides a more concise syntax for complex queries and better opportunities for optimization. However, XSeq only allows partitioning by an attribute field and requires that except this attribute, the rest of the path expression in the partitioning clause be a prefix of the path expression in the from clause. This constraint is important for ensuring efficiency and also for avoiding queries with ill semantics.

Example 14. *For each city, find the oldest male born there.*

By knowing the cities that are present in our XML, we could write several queries, one for each city e.g., `min(//son[@Bplace =' LA'] @Bdate)`. However, in streaming applications such information is generally not provided a priori. Moreover, instead of

running several queries over the same stream, an explicit partition by clause allows for simultaneous handling of different key values and is much easier to optimize. For instance:

Query 7. return $X @Bplace, min($X @Bdate)
from //$X
where tag($X) = 'son'
partition by //son @Bplace

Path Complementation. XSeq does not provide explicit constructs for path complementation (e.g., `except` in XPath 2.0). This restriction does not reduce XSeq's expressivity, as it has been shown that path complementation can be expressed using Kleene-* and path intersection [CL09]. The reason behind this restriction in XSeq is that, by forcing the programmer to simulate the negation with other constructs, the resulting query is often more amenable to optimization. For instance, the query of Example 10 could be expressed in XPath 2.0 using their `except` operator as:

```
//son[@Bplace='NY']//son[@Bplace='NY']@Cname
except
//son[@Bplace='NY']//son[@Bplace != 'LA']//son[@Bplace='NY']@Cname
```

However, as shown in Query 2, this query can be expressed in XSeq without using the negation.

Path Variables. In Query 3, we showed how variables in XSeq could replace the NameTest of a Step. Such variables are called *step variables*. In practice, and in fact in all the real world examples of Section 4.2, we hardly need any feature beyond these step variables. However, for more expressive power[7], XSeq also supports the so-called *path variables* that can replace path expressions, as shown in the PathExprDefinition rule of Fig. 4.2.

[7]This particular feature of XSeq is interesting from a theoretical point of view, as it makes the language Monadic Second Order (MSO)-complete, thus, subsuming previous extensions of XPath.

Example 15. *Find daughter followed by a sequence of siblings with alternating genders, namely daughter, son, daughter, son, and so on.*

Query 8. return first($Z) $X @Cname
from // ($Z: $X $Y $Z)
where tag($X) = 'daughter' and tag($Y) = 'son'

This query defines the path variable $\$Z$ as $\$X\ \$Y\ \$Z$ which means $\$Z$ is recursively defined as $\$X\ \$Y$ followed by itself. In this particular example, $(\$Z : \$X\ \$Y\ \$Z)$ is equivalent to $(\$Z : \$X\ \$Y)^*$, but in general not all recessive path variables can be replaced with Kleene-*[8]. Also, note that path variables do not have to be recursive, e.g. $\$Z$ in $(\$Z : \$X\ \$Y)^*$ is a valid path variable too.

The same step variable can appear multiple times in the from clause. However, for path variables we differentiate between their definition and their reference. XSeq requires that path variables be defined only once in the from clause. For instance,

return first($Z) $X @Cname
from // ($Z: $X $Y $Z) $Z
where tag($X) = 'daughter' and tag($Y) = 'son'

is a valid query, but the following query is not allowed:

return first($Z) $X @Cname
from // ($Z: $X $Y $Z) ($Z: $X)
where tag($X) = 'daughter' and tag($Y) = 'son'

as it redefines the path variable $\$Z$.

Moreover, there is also a restriction on how path variables can be referenced in the from clause [9]. Before explaining this restriction, we first need to define the concepts of $yield$ and $nend$ for a path variable.

[8] Recursive path variables are a more powerful form of recursion than than Kleene-*. See Section 4.5.
[9] Later in Section 4.5.1, we define the restriction rule more formally

Definition 2. *For a path variable* $\$X$ *defined as* $(\$X : P)$*, where* P *is a path expression, we define the* $nend(\$X)$ *as all the path variables in* P *which do not appear at the end of a production for* P*. We also recursively define* $yield(\$X) = \bigcup_{\$Y \in P} yield(\$Y) \cup \{\$Y\}$ *where* $\$Y$ *iterates over all the path variables appearing in* P*.*

For instance, for $(\$X : \$X\ \$Y\ \$X)(\$Y : /son/\$Z)$ as the from clause, $nend(\$X) = \{\$X, \$Y\}$ and $yield(\$X) = \{\$X, \$Y, \$Z\}$. Now, we are ready to formally define the restriction on referencing path variables: Path variables in XSeq can appear multiple times in the from clause, as long as the following rule is not violated:

Rule 1. *For every path variable defined as* $(\$X : P)$*,* $\$X \notin yield(\$Y)$ *for* $\forall \$Y \in nend(\$X)$*.*

Intuitively, this rule disallows circular definitions of path variables. The reason behind this restriction is that allowing arbitrary references to a path variable can make the language non-regular, and hence not amenable to efficient implementation[10].

Other Constructs in XSeq. `union` and `intersect` have the same semantics as in XPath. If the user desires an XML output, he can embed the XSeq query in an XQuery or XSLT expression. Formatting the output is out of the scope of this chapter and makes an interesting future direction of research. Instead, in this chapter, we only focus on the query expression and its efficient execution for CEP applications.

In the next section, we will use these basic constructs to express more advanced queries from a wide range of CEP applications.

[10] For example, allowing $(\$X : a\ \$X\ \$Y)(\$Y : b)$ would represent the pattern $a^n b^n$ which is not MSO expressible.

4.2 Advanced Queries from Complex Event Processing

In this section we present more complex examples from several domains and show that XSeq can easily express such queries.

4.2.1 Stock Analysis

Consider an XML stream of stock quotes as defined below. Let us start with the following example.

```
<! DOCTYPE stocks [
<! ELEMENT stocks (transaction*)>
<! ATTLIST transaction company CDATA #REQUIRED>
<! ATTLIST transaction price CDATA #REQUIRED>
<! ATTLIST transaction buyer IDREF #REQUIRED>
<! ATTLIST transaction date CDATA #REQUIRED> ]>
```

Example 16 (**Falling pattern**)**.** *Find those stocks whose prices are decreasing.*

Query 9 (Falling Pattern in XSeq)**.**

```
return last($X)@price
from /stocks /$Z (\$X)*
where tag($Z) = 'transaction' and tag($X) = 'transaction'
  and $X@price < prev($X)@price
partition by /stocks /transaction@company
```

This is in fact the same query as the one we had expressed in XPath 2.0 in Fig. 4.1. Comparing the convoluted query of Fig. 4.1 with Query 9 clearly illustrates the importance of having explicit constructs for sequential and Kleene-* constructs in enabling CEP applications. This clarity and succinctness at the language level provide more

opportunities for optimization which eventually translate to more efficiency, as shown in Sections 4.3 and sec:experiments, respectively. Next, let us consider the 'V'-shape pattern which is a well-known query in stock analysis.

Example 17 (**'V'-shape pattern).** *Find those stocks whose prices have formed a 'V'-shape. That is, the price has been going down to a local minimum, then rising up to a local maximum which was higher than the starting price.*

The 'V'-shape query only exemplifies many important queries from stock analysis [11] that are provably impossible to express in Core XPath 1.0 and Regular XPath, simply both of these languages lack the notion of 'immediately following sibling' in their constructs. XPath 2.0, however, can express these queries through the use of its `for` and quantified variables: using these constructs, XPath 2.0 can 'simulate' the concept of 'immediately following sibling' in XPath 2.0 by double negation, i.e. ensuring that 'for each pair of nodes, there is nothing in between'. But this approach leads to very convoluted XPath expressions which are extremely hard to write/understand and almost impossible to optimize (See Fig. 4.1 and Section 4.6).

On the other hand, XSeq can express this queries with its simple constructs that can be easily translated and optimized as VPA:

Query 10 ('V'-pattern in XSeq)**.**

```
return last($Y)@price
from /stocks /$Z (\$X)* (\$Y)*
where tag($Z) = 'transaction'
  and tag($X) = 'transaction' and tag($Y) = 'transaction'
  and $X@price < prev($X)@price
  and $Y@price > prev($Y)@price
partition by /stocks /transaction@company
```

[11] http://www.chartpattern.com/

A more interesting pattern would be the falling wedge pattern, which shows the power of sequence aggregates in XSeq language.

Example 18 (Falling wedge pattern). *Find those stocks whose price fluctuates as a series of 'V'-shape patterns, where in each 'V' the range of the fluctuation becomes smaller. Fig. 4.3(b) shows a falling wedge pattern.*

Query 11 (Falling wedge pattern in XSeq)**.**

```
return $R @price, last($Y) @price
from /stocks /$R ((\$S)* \$X (\$T)* \$Y)*
where tag($R) = 'transaction' and tag($S) = 'transaction'
  and tag($X) = 'transaction' and tag($T) = 'transaction'
  and tag($Y) = 'transaction'
  and $R @price > first($S) @price
  and prev($S) @price > $S @price
  and last($S) @price > $X @price
  and $X@price < first($T) @price
  and prev($T) @price < $T @price
  and last($T) @price < $Y @price
  and prev($X) @price < $X @price
  and prev($Y) @price > $Y @price
partition by /stocks /transaction @company
```

4.2.2 Social Networks

Twitter provides an API[12] to automatically receive the stream of new tweets in several formats, including XML. Assume the tweets are ordered according to their date timestamp:

```
<! DOCTYPE twitter [
<! ELEMENT twitter ((tweet)*)>
```

[12] http://dev.twitter.com/

```
<! ELEMENT tweet (message)>
<! ELEMENT message (#PCDATA)>
<! ATTLIST tweet tweetid CDATA #REQUIRED>
<! ATTLIST tweet userid CDATA #REQUIRED>
<! ATTLIST tweet date CDATA #REQUIRED> ]>
```

Example 19 (Detecting active users)**.** *In a stream of tweets, report users who have been active over a month. A user is active if he posts at least a tweet every two days.*

This query, if not impossible, would be very difficult to express in XPath 2.0 or Regular XPath. The main reason is that, again due to their lack of 'immediate following', they cannot easily express the concept of "adjacen" tweets.

Query 12 (Detecting active users in XSeq)**.**

```
return first($T) @userid
from /twitter /$Z (\$T)*
where tag($Z) = 'tweet' and tag($T) = 'tweet'
  and $T@date-prev($T)@date < 2
  and last($T)@date-first($T)@date > 30
partition by /twitter /tweet @userid
```

4.2.3 Inventory Management

RFID has become a popular technology to track inventory as it arrives and leaves retail stores. Below is a sample schema of events, where events are ordered by their timestamp:

```
<! DOCTYPE events [
<! ELEMENT events (event*)>
<! ELEMENT event (message)>
<! ELEMENT message (#PCDATA)>
<! ATTLIST event ts CDATA #REQUIRED>
```

```
<! ATTLIST event itemid CDATA #REQUIRED>
<! ATTLIST event eventtype CDATA #REQUIRED> ]>
```

Example 20 (Detecting Item Theft)**.** *Detect when an item is removed from the shelf and then removed from the store without being paid for at a register.*

Query 13 (Detecting item theft in XSeq)**.**

```
return $T@itemid
from /events /$T \$W* \$X
where tag($T) = 'event' and tag($W) = 'event' and tag($X) = 'transaction'
  and $T@eventtype = 'removed from shelf'
  and $X@eventtype = 'removed from store'
  and $W@eventtype != 'paid at register'
partition by /events/event@itemid
```

4.2.4 Directory Search

Consider the following first-order binary relation which is familiar from temporal logic [CM07b]:

$$\phi(x, y) = \text{descendant}(x, y) \wedge q(y) \wedge \\ \forall z(\text{descendant}(x, z) \wedge \text{descendant}(z, y) \rightarrow p(z))$$

For instance, for a directory structure that is represented as XML, by defining q and p predicates as $q(y)$: 'y is a file' and $p(z)$: 'z is a non-hidden folder', the ϕ relation becomes equivalent to the following query:

Example 21. *Retrieve all reachable files from the current folder by repeatedly selecting non-hidden subfolders.*

According to the results from [CM07b], such queries are not expressible in XPath 1.0. This query, however, is expressible in XPath 2.0 but not very efficiently. E.g., `//file except //folder[@hidden='true']//file`

Such queries can be expressed much more elegantly in XSeq (and also in Regular XPath):

Query 14 (ϕ query in XSeq)**.**

```
(/folder[@hidden = 'false'])* /file
```

4.2.5 Genetics

Haemophilia is one of the most common recessive X-chromosome disorders. In genetic testing and counseling, if the fetus has inherited the gene from an affected grandparent the risk to the fetus is 50% [AFR00]. Therefore, the inheritance risk for a person can be estimated by tracing the history of haemophilia among its even-distance ancestors, i.e. its grandparents, its grand-parents' grand-parents, and so on.

Example 22. *Given an ancestry XML which contains the history of haemophilia in the family, identify all family members who are at even-distance from an affected member, and hence, at risk.*

This query cannot be easily expressed without Kleene-* [CL09], but is expressible in XSeq:

Query 15 (Descendants of even-distance from a node)**.**

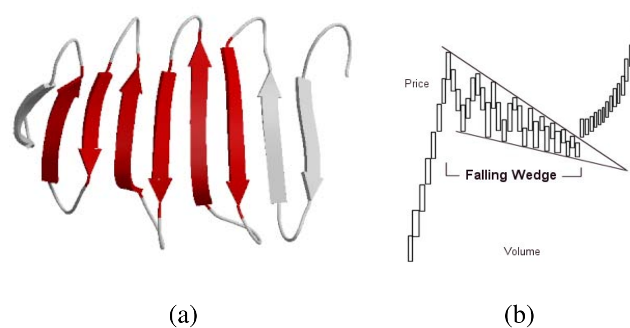
```
return $Z @Cname
from //$X[@haemophilia = 'true'] (/$Y /$Z)*
```

Queries 14 and 15 are not expressible in XPath 1.0, are expressible in XPath 2.0 but not efficiently, and are easily expressible in Regular XPath and XSeq.

4.2.6 Protein, RNA and DNA Databases

The world-wide community of life scientists has access to a large number of public bioinformatics databases and tools. As more and more of the resources offer programmatic web-service interface, XML becomes a widely-used standard data exchange format for basic bioinformatics data. Many public bioinformatics databases provide data in XML format.

Proteins, RNA and DNA are sequences of linear structures, but they are usually with complex secondary or even higher-order structures which play important roles in their functionality. Searching complex patterns in these rich-structured sequences are of great importance in the study of genomics, pharmacy and so on. XSeq provides a powerful declarative query language for access bioinformatics databases, which enables complex pattern searching.



(a) (b)

Figure 4.3: (a) The β-meander motif and (b) the falling wedge pattern

For instance, the *structural motifs* are important supersecondary structures in proteins, which have close relationships with the biological functions of the protein sequences. These motifs are of a large variety of structural patterns, usually very complex, e.g., the β-meander motif is composed of two or more consecutive antiparallel β-strands linked together, as depicted in Fig. 4.3(a)[13], while each β-strand is typically

[13]http://en.wikipedia.org/wiki/Beta_sheet

3 to 10 amino acid. Consider now protein data with a simplified schema as below. Example 16 uses XSeq to detect such motifs.

```
<!DOCTYPE uniprot [
<!ELEMENT uniprot (protein)*>
<!ELEMENT protein (fullName, feature+)>
<!ELEMENT fullName (#PCDATA)>
<!ATTLIST feature type CDATA #REQUIRED> ]>
```

Query 16 (Detecting β-meander motifs)**.**

```
return $N/text()
from //protein[$N] /$F \$G (\$H)*
where tag($N) = 'fullName'
  and tag($F) = 'feature' and $F@type = 'beta-strand'
  and tag($G) = 'feature' and $G@type = 'beta-strand'
  and tag($H) = 'feature' and $H@type = 'beta-strand'
```

4.2.7 Temporal Queries

Expressing temporal queries represents a long-standing research interest. A number of language extentions and ad-hoc solutions have been proposed. Traditional temporal databases use a state-oriented representation, where tuples of a database are time-stamped with their maximal period of validity. This state-based representation requires temporal coalescing and/or temporal joins even for basic query operations (e.g. projection), and are thus prone to inefficient execution. Some recent research work has proposed using XML-based event-oriented representation for transaction-time temporal database, where value updates in database history are recorded as events [AYU00, WZZ08, Zan09b]. For example, below is the DTD of a temporal employee XML, where each employee has a sequence of *salary* and *dept* elements time-stamped by the

tstart, *tend* attributes, representing the update events ordered by their start time in the database's evolution history.

```
<!DOCTYPE employees [
<!ELEMENT employees (employee*)>
<!ELEMENT employee (name (salary | dept)+)>
<!ATTLIST employee id CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT salary (#PCDATA)>
<!ELEMENT dept (#PCDATA)>
<!ATTLIST salary tstart CDATA #REQUIRED>
<!ATTLIST salary tend CDATA #IMPLIED>
<!ATTLIST dept tstart CDATA #REQUIRED>
<!ATTLIST dept tend CDATA #IMPLIED> ]>
```

XSeq is a powerful event-oriented temporal language, which can easily express basic temporal operations (e.g., temporal joins and temporal coalescing), as well as very complex temporal sequence patterns. This can be illustrated by the following examples.

First, let us consider the well-known RISING query which is a famous temporal aggregate introduced by TSQL2 [Sno09].

Example 23 (RISING)**.** *What is the maximum time range during which an employee's salary is rising?*

Query 17.

```
return max(last($X) @tend - first($X) @tstart)
from // $Z* (\$X)*
where tag($X) = 'employee'
  and $X/salary/text() > prev($X)/salary/text()
  and $X@tstart <= prev($X)@tend
partition by //employee @id
```

Example 24. *Find employees who have risen quickly without changing department. More precisely, we want to find employees who*

1. *once hired (with some salary and into some department),*
2. *have gone through one or more salary adjustments, followed by*
3. *a transfer to another department,*
4. *for a final salary that is 40% above the initial one.*

This complex pattern can be expressed succinctly by Query 18.

Query 18.

```
return $X
from //employee[@$X] /$A \$B \$C (\$D)* \$E
where tag($A) = ‘salary’ and tag($B) = ‘dept’
and tag($C) = ‘salary’ and tag($D) = ‘salary’
and tag($E) = ‘dept’ and tag($X) = ‘id’
and $E/text() <> $B/text()
and last($D)/text() > 1.4 * $A/text()
```

4.2.8 Software Trace Analysis

Modern programming languages and software frameworks offer ample support for debugging and monitoring applications. For example, in the .NET framework, the *System.Diagnostics* namespace contains flexible classes which can be easily incorporated into applications to output runtime debug/trace information as XML files. The following XML snippet shows a software trace of a function `fibonacci` that recursively called itself but in the end threw out an exception.

```
<main>
  ...
  <fibonacci @input = ‘500’>
```

```
    <fibonacci @input = '499'>
     ...
      <exception @msg = 'overfolow' />
    </fibonacci>
  </fibanacci>
  ...
</main>
```

Searching and analyzing the patterns in software traces could help debugging. For example, we can easily identify the input to the last iteration of the function `fibonacci` and the depth of the recursive calls by

Query 19.

```
return last($F) @input, count($F)
from //$X (/$F)* /$E
where tag($X) != 'fibonacci'
  and tag($F) = 'fibonacci'
  and tag($E) = 'exception'
```

4.3 XSeq Optimization

The design and choice of operators in XSeq is heavily influenced by whether they can be efficiently evaluated or not. Our criterion for efficiency of an XSeq operator is whether it can be mapped to a Visibly Pushdown Automaton (VPA). The rationale behind choosing VPA as the underlying query execution model is two-fold. First, XSeq is mainly designed for complex patterns and patterns can be intuitively described as transitions in an automaton: fortunately, VPAs are expressive enough to capture all the complex patterns that can be expressed in XSeq. Secondly, VPAs retain many attractive computational properties of finite state automata on words [AM04]. In fact,

by translation into VPAs, we can exploit several existing algorithms for streaming evaluation [MV09] and optimization of VPAs [MZZ10].

In Section 4.3.1, we provide a high-level description of our algorithm for translating the most commonly used operators of XSeq (other operators are covered in Section 4.5) into equivalent VPAs which can faithfully capture the same pattern in the input[14]. Then, in Sections 4.3.2 and 4.3.3, we present several static (compile-time) and run-time optimizations of VPAs in our XSeq implementation. In Section 4.6, we study the effectiveness of these optimizations in practice.

4.3.1 Efficient Query Plans via VPA

In this section, we describe an inductive algorithm to translate the most commonly used features of XSeq into efficient and equivalent VPAs. This algorithm can handle all forward axes, Kleene-*, and step variables (similar fragments to those studied in [GNT11]).

Later, in Section 4.5, we also provide a general algorithm for translating any arbitrary XSeq query Q (including path variables and backward axes) into a VPA (which in general, can be larger and hence, less efficient) accepting all the input trees on which Q returns a non-empty set of results. However, in practice, most of commonly used queries (including all of those of Section 4.2) can be efficiently handled by the algorithm presented below.

Note that although the theoretical notion of VPAs only allows for transitions based on fixed symbols of an alphabet, for efficiency reasons, in our real implementation, we allow the states of the VPA to store values and also to transition when a predicate evaluates to true[15].

[14] Informally, we say that an XSeq query and a VPA are equivalent when every portion of the input XML that produces an output result in the former will be also accepted by the latter and vice versa.

[15] However, in the formal analysis of XSeq's expressiveness in Section 4.5, we will use the theoretical

As described above, compiling XSeq queries into efficient query plans starts by constructing an equivalent VPA for the given query. We construct this VPA by an iterative bottom-up process where we start from a single-state (trivial) VPA and at each forward Step of the XSeq query, we compose the original VPA with a new VPA that is equivalent with the current Step. Next, we show how different forward axes can be mapped into equivalent VPAs. Lastly, we show some of the other constructs of the XSeq query that can be similarly handled.

In the following, whenever connecting the accepting state(s) of a VPA to the starting state(s) of the previous VPA, since VPAs are closed under concatenation, the resulting automaton is still a valid VPA.

Handling /: The /X axis is equivalent to a VPA with two states E and O where E is the starting state at which we invoke the stack on open and closed tags accordingly (see Chapter 2 for the rules regarding stack manipulation in a VPA), and transition to the same state on all input symbols as long as the consumed input in E is well-nested. Upon seeing the appropriate open tag (e.g., ⟨X⟩) we non-deterministically transition to our accepting state O.

Handling @: In the presence of the attribute specifier, @, we add a new state A as the new accepting state which will be transitioned to from our previous accepting state upon seeing any attribute. We remain in state A as long as the input is another attribute, i.e. to account for multiple attributes of the same open tag.

notion of a VPA, i.e. without any storage besides the actual states and without any predicates.

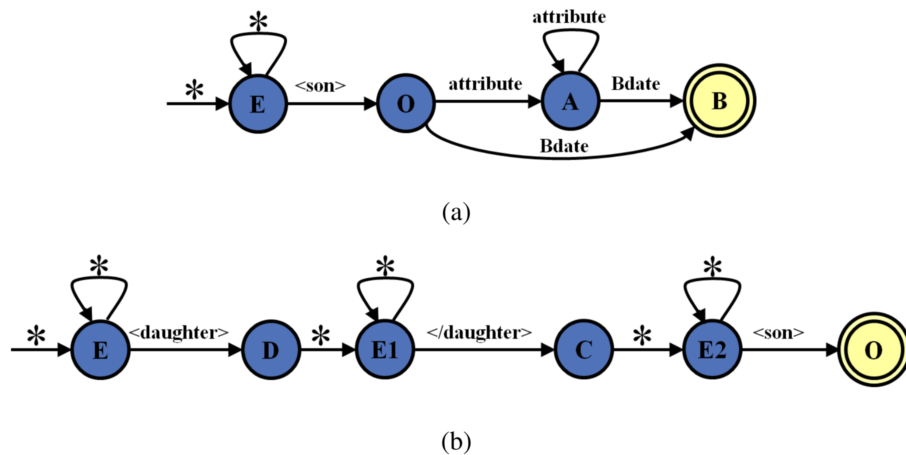


Figure 4.4: VPAs for (a) /`son@Bdate` and (b) /`daughter son`.

Fig. 4.4(a) demonstrates the VPA for /`son@Bdate`. Fig. 4.5 shows the intuitive correspondence of this VPA with the navigation of the XML document, where:

- `E` matches zero or more (well-nested) subtrees in the pre-order traversal of the XML tree,
- `O` matches the open tag for son, i.e. ⟨`son`⟩,
- `A` matches the attribute list of ⟨`son`⟩, namely `O`.

To see the correspondence between this VPA and the XSeq query, note that to find all the direct sons of a daughter, we navigate through the pre-order traversal of the sub-tree under each daughter node, then *non-deterministically* skip an arbitrary number of her children (i.e., $\texttt{E}^*$) until visiting one of her children who is a son (i.e., `O`), and then finally visit all the tokens that correspond to his son's attributes, i.e. $\texttt{A}^*$. The non-determinism assures that we eventually visit all the sons under each daughter.

Handling ()*: Kleene-* expressions in XSeq, such as $(/\texttt{son})^*$, are handled by first constructing a VPA for the part inside the parentheses, say V_1, then adding an ϵ-

transition from the accepting state of V_1 back to its starting state. Since VPAs are closed under Kleene-*, the resulting automaton will still be a VPA.

Handling //: The // axis can also be easily defined as a Kleene-* of the / operator. For instance, the //daughter construct is equivalent to $(/\mathtt{X}) * /\mathtt{daughter}$, where X is a wild card, i.e. matches any open tag. Fig. 4.5 shows the correspondence between the VPA states for // and the familiar traversal of the XML document.

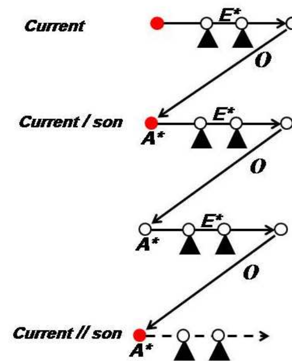


Figure 4.5: Visual correspondence of VPA states and XSeq axes.

Handling siblings: Let V_1 be the VPA that recognizes the query up to node D. The VPA for recognizing the sibling of D, say node S, is constructed by adding four new states (E1, C, E2 and O) to V_1, where:

- We transition from the accepting state(s) of V_1 to E1. E1 invokes the stack on open and closed tags accordingly, and transitions to itself on all input symbols as long as the consumed input in E1 is well-nested.
- Upon seeing a close tag of D, we non-deterministically transition from E1 to C.
- We transition from C to E2 upon any input. Similar to E1, E2 invokes the stack on open and closed tags accordingly, and transitions to itself on all input symbols as long as the consumed input in E2 is well-nested.

- Upon seeing an open tag for the sibling, i.e. ⟨S⟩, we non-deterministically transition from E2 to state O which is marked as the accepting state of the new VPA.

Fig. 4.4(b) shows the VPA for query "/daughter son". The intuition behind this construction is that E1 skips all possible subtrees of the last daughter non-deterministically, while E2 non-deterministically skips all other siblings of the current daughter until it reaches its sibling of type son.

Handling \ : The construct \X is handled according to the last axis that has appeared before it. Let V_1 be the VPA for the XSeq query up to \X. When the previous axis is vertical (e.g. / or //), then we only need to add one new state to the V_1, say O, where from all the accepting states of V_1 we transition to state O upon seeing any open tag of X. The new accepting state will be O.

When the axis before \X is horizontal (e.g. siblings), we add three new states to V_1, say E, C and O, where:

- We transition from the accepting state(s) of V_1 to E. At E, we invoke the stack upon open and closed tags accordingly, and transition to E on all input symbols as long as the consumed input in E is well-nested.

- We non-deterministically transition from E to C upon seeing a close tag of the last (horizontal) axis.

- We transition from C to O upon an open tag for X and fail otherwise. O will be the new accepting state of the VPA.

Handling predicates. In general, arbitrary predicates cannot be handled using the inductive construction described in this section, e.g., when a predicate refers to nodes other than the one being processed. Thus, our construction in this section assumes that

the predicates only refer to attributes of the current node. (In Section 4.5 we consider arbitrary predicates.)

In our real implementation of XSeq, we simply use a few variables (a.k.a. registers) at each state, in order to remember the latest values of the operands in the predicate(s) that need to be evaluated at that state. However, in our complexity analysis in Section 4.5, we use the abstract form of a VPA, namely where a state is duplicated as many as there are unique values for its operands.

Handling partition by. Since the pattern in the 'partition by' clause is the prefix of the pattern in the 'from' clause, the partition by clause can be simply treated as a new predicate on the attribute which is partitioned by. For example, when translating Query 17 into a VPA, assume that the 'partition by' attribute (i.e., ID) has k different values, i.e. $v_1, \cdots, v_k$. Then, we replicate the current VPA k times, each corresponding to a different value of the ID attribute. Once a value of ID is read, say v_i, we transition to the starting state of the VPA that corresponds to v_i and thereon, we simply check that at every state of that sub-automata the current value of the ID attribute is equal to v_i, i.e. otherwise we reject that run of the automata.

Handling other constructs Union, intersection, and, node tests can all be implemented with their corresponding operations on the intermediary VPAs, as VPAs are closed under union, intersection and complementation. The translations are thus straightforward (omitted here for space constraints).

4.3.2 Static VPA Optimization

Cutting the inferrable prefix. When the schema (e.g. DTD) is available, we can always remove the longest prefix of the pattern as long as (i) the prefix has not been referenced in the `return` or the `where` clause, and (ii) the omitted prefix can be always inferred for the remaining suffix. For example, consider the following XSeq query, de-

fined over the SigmodRecord dataset[16]:

`//issue/articles/authors/author[text()='Alan Turing']`

This XSeq query generates a VPA with many states, i.e. 3 states for every step. However, based on the DTD, we infer that `author` nodes always have the same prefix, i.e. `issue/articles/authors/`. Thus, we remove the part of the VPA that corresponds to this common prefix. Due to the sequential nature of VPAs, such simplifications can greatly improve the efficiency by reducing a global pattern search to a more local one.

Reducing non-determinism from the transition table. Our algorithm for translating XSeq queries produces VPAs that are typically non-deterministic. Reducing the degree of non-determinism always improves the execution efficiency by avoiding many unnecessary backtracks. In general, *full determinization* of a VPA is an expensive process, which can increase the number of states from $O(n)$ to $O(2^{n^2})$ [AM04].

However, there are special cases that the degree of non-determinism can be reduced without incurring an exponential cost in memory. Since self-loops in the transition table are one of the main sources of non-determinism, whenever self-loops can only occur a fixed number of times, the XSeq's compile-time optimizer removes such edges from the generated VPA by duplicating their corresponding states accordingly. For instance, consider the XSeq query $//book/year/text()$ and its corresponding VPA in Fig. 4.6. If we know that `book` nodes only contain two subelements, say `title` followed by `year`, the optimizer will replace `E1` with 3 new states (without any self-loops) to explicitly skip the title's open, text and closed tags. The latter expression ($E1$^3) is executed more efficiently as it will be deterministic.

Reducing non-determinism from the states. In order to skip all the intermediate subelements, the automatically generated VPAs contain several states with incoming and outgoing ϵ-transitions. In the presence of the XML schema, many of such states

[16] http://www.cs.washington.edu/research/xmldatasets/

become unnecessary and can be safely removed before evaluating the VPA on the input. We have several rules for such safe omissions. Here, we only provide one example.

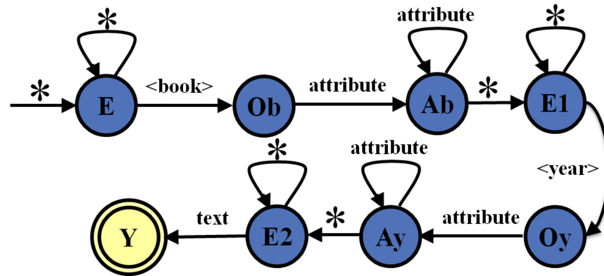


Figure 4.6: `//book/year/text()`

Let us once again consider the query and the VPA of Fig. 4.6 as our example. If according to the schema, we know that the `year` nodes cannot contain any subelements, the optimizer will remove `E2` entirely. Also, if a node, say `year`, does not have any attributes, the optimizer will remove its corresponding state, here Ay.

4.3.3 Run-time VPA Optimization

In the previous sections, we demonstrated how XSeq queries can be translated into equivalent VPAs and presented several techniques for reducing the degree of non-determinism in our VPAs. One of the main advantages of using VPAs as the underlying execution model is that we can take advantage of the rich literature on efficient evaluation of VPAs. In particular we use the one-pass evaluation of the VPAs as described in [MV09] and use the pattern matching optimization of VPAs as described in [MZZ10].

In a straightforward evaluation of a VPA over a data stream, one would consider the prefix starting from every element of the stream as a new input to the VPA. In other words, upon acceptance or rejection of every input, the immediate next starting posi-

tion would be considered. However, for word automata, it is well-known that this naive backtracking strategy can be easily avoided by applying pattern matching techniques such as the KMP [KJP77] algorithm. Recently, a similar pattern matching technique was developed for VPAs, known as VPSearch [MZZ10]. Similar to word automata, VPSearch avoids many unnecessary backtracks and therefore, reduces the number of VPA evaluations. We have implemented VPSearch and its run-time caching techniques in our Java implementation of XSeq. Further details on streaming evaluation of VPAs and the VPSearch algorithm can be found in [MV09] and [MZZ10], respectively. Because of the excellent VPA execution performance achieved by K*SQL [MZZ10], we have used the same run-time engine for XSeq queries once they are compiled into a VPA (see Section 7).

In the next two sections, we define the formal semantics of XSeq and present our results on its expressiveness and complexity.

4.4 Formal Semantics of XSeq

While in the previous section we informally illustrated the semantics of different XSeq operators through intuitive examples, in this section we provide the formal semantics of XSeq which once restricted to its navigational features, will pave the way for a rigorous analysis of the language in Section 4.5. We first define an XML tree.

Definition 3 (XML Tree)**.** *An XML tree Tr is an unranked ordered tree $Tr = (V, L, \downarrow, \rightarrow)$ where V is a set of nodes, $L : V \rightarrow \Sigma$ is a labeling of the nodes to symbols of a finite alphabet Σ, and $R_\downarrow$ and $R_\rightarrow$ are respectively the parent-child and immediately following sibling relationships among the nodes. For leaf nodes v, we define $R_\downarrow(v) = \perp$. Also, for the rightmost child v we define $R_\rightarrow(v) = \perp$. We refer to the root node of Tr as $root(Tr)$.*

Using $R_\downarrow$ and $R_\rightarrow$, we can similarly define R_{ax} where ax is any of the Axes[17] in Fig. 4.2. Next, we define a query, where for simplicity, we ignore the output clause and only consider the 'decision' version of the query, namely query can only return a 'true' if it finds a match, and otherwise returns nothing.

Definition 4 (Query). *We represent an XSeq query of form "return true from doc() P where C" as $Q = (P, C)$ where P is a PathExpr and C is a Condition. When C is absent, we use "true" instead, i.e. $Q = (P, true)$.*

Definition 5 (Normalized Query). *A query $Q = (P, C)$ is normalized if all Predicates in P are patterns.*

Note that we can always normalize any query $Q = (P, C)$ by applying the following steps:

1. For each Condition Predicate cp in P, rewrite cp into disjunctive normal form $cp_1 \vee cp_2 \vee \cdots \vee cp_k$. Then rewrite Q into $Q' = (P_1 \cup P_2 \cup \cdots, P_k, C)$ such that each P_i only contains cp_i. Repeat this process until all Condition Predicates in P_i are in conjunctive form. Let the resulting query be $Q^0 = (P^0, C^0)$.

2. For each conjunctive Condition Predicate $pred$ in Step s of P^0, extract all of its path expressions, say $p_1, p_2, ..., p_j$.

3. By renaming the last Step in p_i with a new Step variable v_i, obtain p'_i. Add a Predicate $[p'_i]$ to Step s.

4. Remove $pred$ from s. Express $pred$ using $\{v_i\}$, say $pred'$. Add $pred' \wedge \{$constraints on $\{v_i\}\}$ to C^0.

[17]For clarity, in this section, we use capitalized words when referring to any of the production rules of Fig. 4.2.

5. If p_i contains Kleene-* and the last Step can be empty (e.g. due to a Kleene-*), rewrite p_i into the union of a set of path expression, whose last Step cannot be empty.

Thus, in the rest of this discussion, we assume all the queries are in their normalized form.

Example 25. *The normalized form of the query* $doc()/a[c/d > e/f]/b$ *is as follows:*

```
doc()/a[c/$X][e/$Y]/b
where tag($X) = 'd' and tag($Y) = 'f'
 and $X > $Y
```

Definition 6. *For a Pattern* p*, we define* $\chi(p)$ = {*all the NameTest's that appear in* p}.

When the same NameTest appears multiple times, we keep all occurrences in χ, e.g. by adding an index.

Example 26. *For* $p = doc()/a/b/\$X/a$*, we have* $\chi(p) = \{a_1, b, a_2, \$X\}$.

Definition 7 (Base of a Predicate)**.** *For any Step of the form "*$ax :: nt\ [p]$*" where* ax *is an Axis,* nt *is a NameTest and* p *is a Pattern, we define the base of* $[p]$ *as* $\beta(p) = nt$.

Example 27. *For the PathExpr* $A/B[C[D]]$ *we have* $\beta(C[D]) = B$ *and* $\beta(D) = C$.

Definition 8 (Flattening a Pattern)**.** *For a Pattern* P*,* $\tilde{P}$ *is the result of removing all the Predicates from* P.

Example 28. *Consider* $p = A/B[C[D]]$*. Then,* $\tilde{p} = A/B$*. Thus,* $\chi(p)$ *may be different from* $\chi(\tilde{p})$.

Definition 9 (Meaning of a Pattern)**.** *Given an XML tree* Tr*, for any Pattern* P *we define its meaning, denoted as* $[[p]]^{Tr}$*, recursively, as follows* [18] *where* $[[p]]^{Tr} \subseteq (N \times (\chi(P) \cup \{\perp\}))^*$ *:*

[18] *For brevity, we assume the tree is fixed and thus, denote* $[[p]]^{Tr}$ *as* $[[p]]$.

- *If* $P = doc()\ p$, *define*
  $[[doc()\ p]] = \{\langle (root(Tr) : \bot), (n_1 : L_1), \cdots (n_k : L_k)\rangle \in [[p]]\}$

- *If* $P = p$ *where* p *is a PathExpr, define*
  $[[p]] = \{\langle (n_1 : L_1), \cdots, (n_k : L_k)\rangle \mid n_i \in N, L_i \in \chi(\tilde{p}) \cup \{\bot\}\}$

- *If* P *is a single Step of form* $nt :: ax$ *where* nt *and* ax *are the NameTest and Axis, respectively, define*
  $[[nt :: ax]] = \{\langle (n : \bot), (m : nt)\rangle \mid n, m \in N, L(m) = nt \text{ and } (n, m) \in R_{ax}\}$

- *If* P *is a single Step of form* $nt :: ax[p]$ *where* nt*,* ax*, and* p *are the NameTest, Axis, and Pattern*[19] *respectively, define*
  $[[nt :: ax[p]]] = \{\langle (n0 : \bot), (n : l)\rangle) \mid \langle (n_0 : \bot), (n : l)\rangle \in [[nt :: ax]], \exists \alpha \text{ s.t. } \langle (n : \bot), \alpha\rangle \in [[p]]\}$

- *If* P *is a path variable* v *with the definition* $(v : p)$ *define* $[[v]] = [[p]]$

- *If* $P = p_1 p_2$*, define*
  $[[p_1 p_2]] = \{\langle \alpha_1, (n : l), \alpha_2\rangle \mid \exists n, l \text{ s.t. } \langle \alpha_1, (n : l)\rangle \in [[p_1]] \text{ and } \langle (n : \bot), \alpha_2\rangle \in [[p_2]]\}$

- *If* $P = (p)*$*, define*
  $[[(p)*]] = \{\langle (n_0 : l_0), \alpha_1, (n_1 : l_1), \alpha_2, (n_2 : l_2), \cdots, (n_{k-1} : l_{k-1}), \alpha_k, (n_k : l_k)\rangle \mid \langle (n_{i-1} : \bot), \alpha_i, (n_i : l_i)\rangle \in [[p]] \text{ for } i = 1, \cdots, k\} \cup \{\epsilon\}$

Example 29. *Consider an XML tree* $Tr = (V, L, \downarrow, \rightarrow)$*, where* $V = \{a_1, b_1, b_2, c_1\}$*,* $L = \{(a_1, a), (b_1, b), (b_2, b), (c_1, c)\}$*,* $R_\downarrow = \{(a_1, b_1), (a_1, b_2), (b_1, c_2)\}$*, and* $R_\rightarrow = \{(b_1, b_2)\}$*. Here, even though:*
$[[doc()/a/b]] = \{\langle (root(Tr) : \bot), (a_1 : a), (b_1 : b)\rangle, \langle (root(Tr) : \bot), (a_1 : a), (b_2 : b)\rangle\}$,

[19]*Note that since the query is normalized, here we do not need to consider Conditions as Predicate.*

$[[doc()/a/b[/c]]]$ *contains only one sequence, i.e.,*

$[[doc()/a/b[/c]]] = \{\langle(root(Tr):\bot),(a_1:a),(b_1:b)\rangle\}$.

$\langle(root(Tr):\bot),(a1:a),(b2:b)\rangle$ *is not in* $[[doc()/a/b[/c]]]$ *because there is no sequence starting with* $(b_2:\bot)$ *in* $[[/c]] = \{\langle(b_1:\bot)(c_1:c)\rangle\}$.

Definition 10 (Environment)**.** *An environment is any mapping* $e_{P,\alpha,n}:\chi(P)\rightarrow N\cup\{\bot\}$ *where* P *is a Pattern,* $\alpha\in[[p]]$*, and* $n\in N\cup\{\bot\}$.

Definition 11 (Valid Environment)**.** *An environment* $e_{P,\alpha,n}$ *is valid iff one of the following conditions holds:*

1. P *is a PathExpr,* $P=\tilde{P}$*, and for all* $l\in\chi(P)$ *we have* $e_{P,\alpha,n}(l)=n'$ *if there exists* n' *such that* $\alpha=\langle(n:\bot),\cdots,(n':l),\cdots\rangle$

2. P *is a PathExpr with top-level Predicates*[20] $p_1,\cdots,p_k$*, and there exist* $\alpha_i\in[[p_i]]$ *for* $1\leq i\leq k$ *such that* $e_{P,\alpha,n}=e_{\tilde{P},\alpha,n}\bigcup\cup_{i=1}^{k}e_{p_i,\alpha_i,e_{\tilde{P},n}(\beta(p_i))}$ *where* $e_{\tilde{P},\alpha,n}$ *and* $e_{p_i,\alpha_i,e_{\tilde{P},n}(\beta(p_i))}$ *are also valid environments.*

3. $P=doc()\,p$ *where* p *is a PathExpr, and there exist* $\alpha'\in[[p]]$ *such that* $e_{p,\alpha',root(Tr)}$ *is a valid environment.*

Example 30. *Given the XML tree defined in Example 29, consider Pattern* $P = doc()/a/\$X[/c]$*. The environment* $e_{P,\alpha,root(Tr)}$ *is valid if* $\alpha=\{\langle(root(Tr):\bot),(a_1:a),(b_1:\$X)\rangle\}$*, and* $e_{P,\alpha,root(Tr)}(\$X)=b_1$.

Definition 12 (Condition Evaluation Under A Valid Environment)**.** *We define when a Condition* C *evaluates to true under a valid environment* e *(which we denote as* $e\models C$*) by defining how to replace different types of Operand with constant values. Once all the Operands in* C *are replaced with their constant values, the entire Condition*

[20] *For instance, for* $p=A/B[C[D]][E]/T[H]$ *the top-level Predicates are* $p_1=C[D]$ *and* $p_2=E$*, and* $p_3=H$.

can be also evaluated by following the conventional rules of arithmetic and boolean expression. There are different types of Operands:

- *Constant is trivial.*
- $seq(X)@attr$, *where* $\alpha = \langle (n_1, l_1), ..., (n_i, l_i), ...(n_m, l_m) \rangle$ *and* $e_{P,\alpha,n}(X) = n_i$, *is replaced with attribute 'attr' of node* $n_j, 1 \leq j \leq m$ *where* $l_j = X$ *and :*
  - *if seq=prev and for* $j + 1 \leq k \leq i - 1$*, we have* $l_k \neq X$*;*
  - *if seq=first, and for* $1 \leq k \leq j - 1$*, we have* $l_k \neq X$*;*
  - *if seq=last, and for* $j + 1 \leq k \leq m$*, we have* $l_k \neq X$*;*

  Otherwise, we replace it with the null value.
- $X@text()$ *is replaced with the text value of node* n_j *where* n_j *is defined as above.*
- $agg(X@attr)$*, where* X *is in* $\tilde{P}$*, a valid environment* $e_{P,\alpha,n}$ *is picked and* $\alpha = \langle (n_1, l_1), ..., (n_i, l_i), ...(n_m, l_m) \rangle$*, is replaced with* $agg(\{n_i | l_i = X, 1 \leq i \leq m\})$*.*

Definition 13 (Query Evaluation)**.** *We say that a query* $Q = (P, C)$ *recognizes the XML tree* Tr*, iff* $[[doc()P]] \neq \emptyset$ *and for all valid environments* $e_{P,root(Tr)}$, $e_{P,root(Tr)} \models C$.

4.5 Expressiveness and Complexity

In Section 4.3, we provided the high-level idea of how most of XSeq queries can be optimized and translated into equivalent VPAs. In this section, we provide our results on the expressiveness of XSeq, and its complexity for query evaluation —two fundamental questions for any query language.

Throughout this section, Σ is the alphabet (i.e., set of unique tokens in the XML document), and MSO is monadic second order logic over trees.

The full language of XSeq is too rich for a rigorous logical analysis, and thus we focus on its navigational features by excluding arithmetics, string manipulations and aggregates. Thus, in Section 4.5.1, we first obtain a more concise language, called CXSeq[21].

We show that, given a CXSeq query Q, the set of input trees for which Q contains a match (we call this the domain of Q) is an MSO definable language. Conversely, for every MSO definable language L, there exists a CXSeq with domain L. The proof of this statement can be found in Section 4.9.

In Section 4.5.3, we use this equivalence result to derive the complexity of query evaluation of CXSeq queries.

4.5.1 CXSeq

In Fig. 4.7, we have provided the syntax of the query language CXSeq.

$$
\begin{aligned}
\text{PathExpr} &::= \text{PathExpr}\,'*' \mid \text{PathExpr}\ '\mathit{intersect}'\ \text{PathExpr} \mid \text{VStep} \\
\text{VStep} &::= \text{Step}\ \text{Variable}* \\
\text{Step} &::= \text{Axis}\ '::'\ \text{NameTest}\ [\text{Variable}] \\
&\mid\ \ \text{Axis}\ '::'\ \text{NameTest} \\
\text{Axis} &::= \odot \mid \downarrow \mid \uparrow \mid \rightarrow \mid \leftarrow
\end{aligned}
$$

Figure 4.7: CXSeq Syntax

A query is a tuple $K = (V, v_0, \rho)$ where V is a finite set of variables, $v_0 \in V$ is the starting variable, and $\rho : V \times P$ is a set of productions where P is the set of elements defined by the grammar in Fig. 4.7 starting with $PathExpr$. In the grammar a Variable is an element of V.

[21]Similar approaches in analyzing XPath 1.0 and 2.0, has led to sub-languages Core XPath 1.0[CM07b] and Core XPath 2.0[CM07a].

The semantics of a query is given with pairs of nodes and is parameterized over variables. The idea is that every XPath query that can be "generated" by the above grammar should be in some sense executed. Consider the query:

$$(X, \uparrow :: aX), (X, \uparrow :: a)$$

Its semantics should be equivalent to $(\uparrow :: a)+$.

Given a variable $v \in V$ we define $S(v) \subseteq N \times N$ as the set of pairs of nodes that satisfy the query v. Informally $S(v)(n, n')$ iff starting in node n we can reach node n' following the query v. Informally every variable v defines a set of pair, but the final result of K is the set of pairs assigned to v_0. We can now proceed inductively and define the semantics as the least fix point of the following relations. $S_{v,\pi} \subseteq N \times N$ (for each v and π) defines the pair of nodes belonging to v when starting with the production π.

1. if $\pi = \uparrow :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, $\text{lab}(x) = s$, and $R_\downarrow(x, z)$, then $S_{v,\pi}(z, y_2)$;
2. if $\pi = \downarrow :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, $\text{lab}(x) = s$, and $R_\downarrow(z, x)$ and does not exists z', $R_\rightarrow(z', x)$, then $S_{v,\pi}(z, y_2)$;
3. if $\pi = \leftarrow :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, $\text{lab}(x) = s$, and $R_\rightarrow(x, z)$, then $S_{v,\pi}(z, y_2)$;
4. if $\pi = \rightarrow :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, $\text{lab}(x) = s$, and $R_\rightarrow(z, x)$, then $S_{v,\pi}(z, y_2)$;
5. if $\pi = \odot :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, and $\text{lab}(x) = s$, then $S_{v,\pi}(x, y_2)$;
6. if $\pi = \pi_1 \cap \pi_2$, $S_{v,\pi} = S_{v,\pi_1} \cap S_{v,\pi_2}$;
7. the other cases are analogous.

Finally $S_v = \bigcup_{(v,\pi)\in\rho} S_{v,\pi}$.

This language allows us to define productions of the form

$$X := \downarrow :: aYZ$$

Without further restrictions this extension would be too expressive. For example the productions

$$X := \downarrow :: aXY, Y := \downarrow :: b$$

would represent the query $(a/)^n(b/)^n$ which is not MSO expressible. In order to reduce the expressiveness we limit the use of recursion. Given a production $p = (v, \pi)$ let $\mathrm{dv}(\pi)$ be the following set of variables:

- $\mathrm{dv}(\pi \cap \pi') = \mathrm{dv}(\pi) \cup \mathrm{dv}(\pi')$;
- $\mathrm{dv}(\pi*) = \mathrm{va}(\pi)$;
- $\mathrm{dv}(d :: a[v]v_1 \dots v_{n+1}) = \{v_1, \dots, v_n\}$;
- $\mathrm{dv}(d :: a[v]) = \{\}$;
- $\mathrm{dv}(d :: av_1 \dots v_{n+1}) = \{v_1, \dots, v_n\}$;
- $\mathrm{dv}(d :: a) = \{\}$.

Similarly we define for a production $p = (v, \pi)$ the set $\mathrm{va}(\pi)$ be the following set of variables:

- $\mathrm{va}(\pi \cap \pi') = \mathrm{dv}(\pi) \cup \mathrm{dv}(\pi')$;
- $\mathrm{va}(\pi*) = \mathrm{va}(\pi)$;
- $\mathrm{va}(d :: a[v]v_1 \dots v_n) = \{v_1, \dots, v_n\}$;

- $\mathrm{va}(d :: a[v]) = \{v\}$;
- $\mathrm{va}(d :: av_1 \ldots v_n) = \{v, v_1, \ldots, v_n\}$;
- $\mathrm{va}(d :: a) = \{\}$.

Now given a variable $v \in V$ we define the sets of variables reachable from v as yi_v as the set satisfying the following equation

$$\mathrm{yi}_v = \{v\} \cup \bigcup_{(v,\pi)\in\rho} \bigcup_{v'\in\mathrm{va}(\pi)} \mathrm{yi}_{v'}$$

We can now formalize the restriction on our grammar.

Definition 14. *A query $K = (V, v_0, \rho)$ is* safe *iff for each $(v, \pi) \in \rho$, for each $v' \in \mathrm{dv}(\pi)$, $v \notin \mathrm{yi}(v')$.*

For the rest of the presentation, we will only consider safe CXSeq queries.

Notice that the $\downarrow$ axis has the meaning of first child of a node instead of child. This language also allows to define the operators $axis*$ using the $*$ operator. We can also extend the language to allow nested stars, and the same translation as before will work. For example the production $(v, ((\uparrow: a)v' * v'')*)$ can be transformed into the following set of productions

$$(v, (\odot : _)t_1); (t_1, (\uparrow: a)t_2t_1); (t_1, (\odot :: _)); (t_2, (\odot : _)t_3v''); (t_3, (\odot : _)v't_3); (t_3, \odot : _)$$

where t_1, t_2, t_3 are fresh names.

To better understand the semantics let's consider the following regular XPath query:

$$\downarrow:: a \downarrow:: b(\downarrow:: c)*$$

This will be encoded in CXSeq with the following productions:

$$(X, \downarrow:: aYZ), (Y, \downarrow:: b), (Z, \odot : _WZ), (Z, \odot : _), (W, \downarrow: c)$$

where X is the first production.

4.5.2 Regularity of CXSeq and Complexity

This section contains the two main results on CXSeq. Given a query K we define its domain as $D_K = \{w | K(w) \neq \emptyset\}$. CXSeq is equivalent to MSO in terms of domain expressiveness and therefore the domain of every CXSeq query can be translated into a VPA.

Theorem 6. *For every CXSeq query $K = (V, v_0, \rho)$, the the domain D_K of K is an MSO definable language. Conversely for every MSO definable language L, there exists a CXSeq query K such that $L = D_K$.*

Theorem 7. *For every CXSeq query $K = (V, v_0, \rho)$, there exists an equivalent VPA A over Σ such that $L(A) = D_K$. A will have $O(r^5 \cdot length(K)^5 \cdot 2^{r \cdot length(K)})$ where $r = |\rho|$.*

The proofs of the above theorems can be found in Section 4.9.

4.5.3 Query Evaluation Complexity

Lemma 8 (Query Evaluation)**.** *Data and query complexities for CXSeq's query evaluation are* PTIME *and* EXPTIME*, respectively.*

Proof. By mapping CXSeq queries into VPAs, the query evaluation of the former corresponds to the language membership decision of the latter. Using the membership algorithm provided in [MV09], we only need space O($s^4 \cdot log\ s \cdot d + s^4 \cdot n \cdot log\ n$) where n is the length of the input, d is the depth of the XML document (thus, $d < n$), and s is the number of the states in the VPA. PTIME data complexity comes from n and the EXPTIME query complexity comes from s which is exponential in the query size (see Theorem 7). $\square$

We conclude this section with a result on containment of query domains.

Lemma 9 (Query Domain Containment)**.** *Given two CXSeq queries K_1 and K_2, it is decidable to check whether the domain of K_1 is contained in the domain of K_2. Moreover, the problem is* 2-EXPTIME*-complete.*

Proof. Once two CXSeq queries are translated into VPAs, their query domain containment problem corresponds to the language inclusion problem for their domain VPAs, say M_1 and M_2. To check $L(M_1) \subseteq L(M_2)$, we check if $L(M_1) \cap \overline{L(M_2)} = \emptyset$. Given M_1 with s_1 states and M_2 with s_2 states, we can determinize [Tan09] and complement the latter to get a VPA for $\overline{L(M_2)}$ of size O($2^{{s_2}^2}$). $L(M_1) \cap \overline{L(M_2)}$ is then of size O($s_1 \cdot 2^{{s_2}^2}$), and emptiness check is polynomial (cubic) in the size of this automaton. Since, s_1 and s_2 are themselves exponential in the size of their CXSeq queries, membership in 2-EXPTIME holds. For completeness of the 2-EXPTIME, note that CXSeq syntactically subsumes Regular XPath$(*, \cap)$ for which the query containment has been shown to be 2-EXPTIME-complete [CL09]. □

4.6 Experiments

In this section we study the amenability of XSeq language to efficient execution. Our implementation of the XSeq language consists of a parser, VPA generator, a compile-time optimizer, and the VPA evaluation and optimization run-time, all coded in Java. We first evaluate the effectiveness of our different compile-time optimization heuristics in isolation. We then compare our XSeq system with the state-of-the-art XML engines for (i) complex sequence queries, (ii) Regular XPath queries, and (iii) simple XPath queries. While these systems are designed for general XML applications, we show that XSeq is far more suited for CEP applications. In fact, XSeq achieves up to two orders of magnitude out-performance on (i) and (ii), and competitive performance on (iii). Finally, we study the overall performance, throughput and memory usage of our

system under different classes of patterns and queries.

All the experiments were conducted on a 1.6GHz Intel Quad-Core Xeon E5310 Processor running Ubuntu 6.06, with 4GB of RAM. We have used several real-world datasets including NASDAQ stocks that contains more than $7.6M$ records[22] since 1970, and also the Treebank dataset[23] that contains English sentences from Wall Street Journal and has with a deep recursive structure (max-depth of 36 and avg-depth of 8). We have also used XMark [Sa02] which is well-known benchmark for XML systems and provides both data and queries. Due to lack of space, for each experiment we only report the results on one dataset. The results and main observations, however, were similar across different datasets.

4.6.1 Effectiveness of Different Optimizations

In this section, we evaluate the effectiveness of the different compile-time optimizations from Section 4.3.2, by measuring their individual contribution to the overall performance[24]. For this purpose, we executed the X2 query from XMark [Sa02] over a wide range of input sizes (generated by XMark, from 50KB to 5MB). The results of this experiment are reported in Fig. 4.8, where we use the following acronyms to refer to different optimization heuristics (see Section 4.3.2):

Opt-1	Cutting the inferrable prefix
Opt-2	Reducing non-determinism from the pattern clause
Opt-3	Reducing non-determinism from the where clause

[22] http://infochimps.org/dataset/stocks_yahoo_NASDAQ

[23] http://www.cs.washington.edu/research/xmldatasets/www/repository.html

[24] The effectiveness of the VPA evaluation and optimization techniques have been previously validated in their respective papers [MV09, MZZ10].

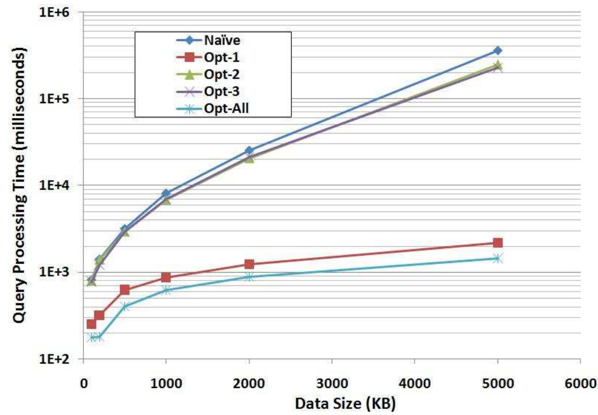


Figure 4.8: Contribution of different optimization techniques.

In this graph, we have also included the naive and combined (Opt-All) versions, namely when, respectively, none and all of the compile-time optimizations are applied. The first observation is that combining all the optimization techniques delivers a dramatic improvement in performance (1-2 orders of magnitude, over the naive one).

Cutting the inferable prefix, Opt-1, leads to fewer states in the final VPA. Like other types of automata, fewer states can significantly reduce the overall degree of non-determinism. The second reason behind the key role of Opt-1 in the overall performance is that it reduces non-determinism from the *beginning* of the pattern: this is particularly important because non-determinism in the starting states of a VPA is usually disastrous as it prevents the VPA from the early detection of unpromising traces of the input. In contrary, reducing non-determinism in the pattern and the `where` clause (Opt-2, Opt-3) has a much more local effect. In other words, the latter techniques only remove the non-determinism from a single state or edge in the automata, while the rest of the automata may still suffer from non-determinism. However local, Opt-2 and Opt-3 can still improve the overall performance when combined with Opt-1. This is because of the extra information that they learn from the DTD file.

4.6.2 Sequence Queries vs. XPath Engines

We compare our system against two[25] of the fastest academic and industrial engines: MonetDB/XQuery[Ba06] and Zorba [Ba09]. First, we used several sequence queries on Nasdaq transactions (embedded in XML tags), including the 'V'-shape pattern (defined in Example 17 and Query 10). By searching for half of a 'V' pattern, we defined another query to find 'decreasing stocks'. Also, by defining two occurrences of a 'V' pattern, we defined what is known as the 'W'-shape pattern [26]. We refer to these queries as S1, S2 and S3. We also defined several Regular XPath queries over the treebank dataset, named R1, R2, R3 and R4 where,

R1: /FILE/EMPTY(/VP)*/NP,

R2: /FILE(/EMPTY)*/S,

R3: /FILE(/EMPTY)*(/S)*/VP,

R4: /FILE(/EMPTY)*/S(/VP)*/NP

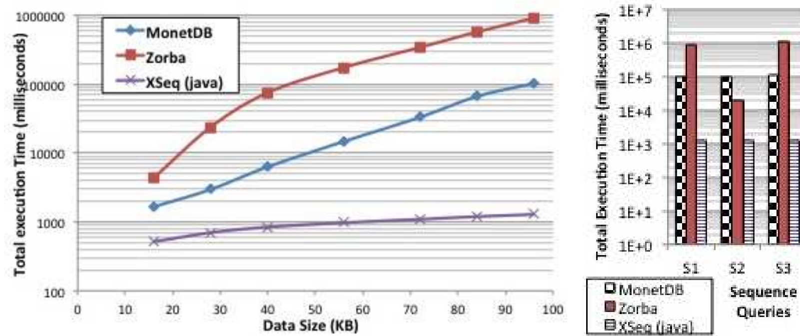
Sequence queries. For expressing these queries (namely S1, S2 and S3) in XQuery, we had to mimic the notion of 'immediately following sibling', i.e. by checking that for each pair of siblings in the sequence, there are no other nodes in between. The XQuery versions of S2 has been given in Fig. 4.1. Due to the similarity of S1 and S3 to S2 here we omit their XQuery version (roughly speaking, S1 and S3 consist of, respectively, two and four repetitions of S2).

Not only were sequence queries difficult to express in XPath/ XQuery but were also extremely inefficient to run. For instance, for the queries at hand, neither of Zorba or MonetDB could handle any input data larger than 7KB. The processing times of these sequence queries, over an input size of 7KB, are reported in Fig. 4.9(b). Note that the Y-axis is in log-scale: *the same sequence queries written in XSeq run between*

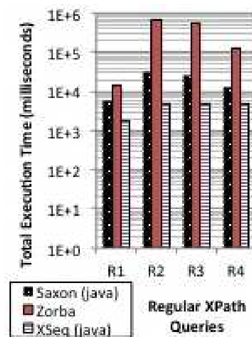
[25] Since the sequence queries of this experiment are not expressible in XPath, we could not use the XSQ [PC03] engine as it does not supports XQuery.

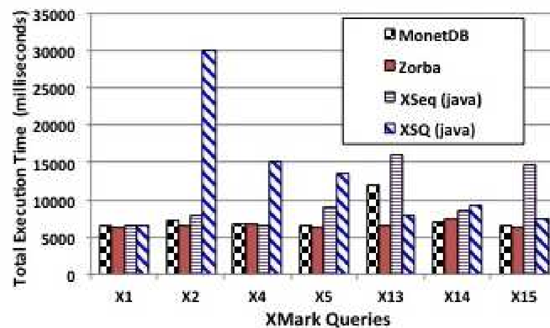
[26] 'W'-pattern (a.k.a. double-bottom) is a well-known query in stock analysis.

1-3 orders of magnitude faster than their XPath/XQuery counterparts do on two of the fastest XML engines. Fig. 4.9(a) shows that gap between XSeq and the other two engines grows with the input size. This is due to the linear-time query processing of XSeq which, in turn, is due to the linear-time algorithm for evaluation of VPAs along with the backtracking optimizations when the VPA rejects an input [MZZ10]. Zorba and MonetDB's processing time for these sequence queries are at least quadratic, due to the nested nature of the queries.



(a) (b)





(c) (d)

Figure 4.9: XSeq vs. XPath/XQuery engines: (a) 'V'-pattern query over Nasdaq stocks, (b) Sequence queries over Nasdaq stocks, (c) Regular XPath queries over XMark data, and (d) conventional XPath queries from XMark.

In summary, the optimized XSeq queries run significantly (1-3 orders of magni-

tude) faster than their equivalent counterparts that are expressed in XQuery. This result indicates that traditional XML languages such as XPath and XQuery (although theoretically expressive enough), due to their lack of explicit constructs for sequencing, are not amenable to effective optimization of complex queries that involve repetition, sequencing, Kleene-*, etc.

Regular XPath queries. As mentioned, despite the many benefits and applications of Regular XPath, currently there are no implementations for this language (to our best knowledge). One of the advantages of XSeq is that it can be also seen as the first implementation of Regular XPath, as the latter is a subset of the former. In order to study the performance of XSeq for Regular XPath queries (e.g., R1, $\cdots$, R4) we compared our system with the only other alternative, namely implementing the Kleene-* operator as a higher-order user-defined functions (UDF) in XQuery. Since MonetDB does not support such UDFs, we used another engine, namely Saxon [Kay08]. The results for 464KB of treebank dataset are presented in Fig. 4.9(c) as Zorba, again, could not handle larger input size. Thus, for Regular XPath queries, similarly to sequence queries, XSeq proves to be 1-2 orders of magnitude faster than Zorba, and between 2-6 times faster than Saxon. Also, note that the relative advantage of Saxon over Zorba is only due to the fact that Saxon loads the entire input file in memory and then performs an in-memory processing of the query [Kay08]. However, this approach is not feasible for streaming or large XML documents[27].

4.6.3 Conventional Queries vs. XPath Engines

As shown in the previous section, complex sequence queries written in XSeq can be executed dramatically faster (from 0.5 to 3 orders of magnitude) than even the fastest

[27]Due to lack of space, we omit the results for the case when the input size cannot fit in the memory. Briefly, unlike XSeq, Saxon results in using the disk swap, and thus, suffers from a poor performance.

of XPath/ XQuery engines. In this section, we continue our comparison of XSeq and native XPath engines by considering simpler XPath queries, i.e. queries without sequencing and Kleene-*. For this purpose, we used the XMark queries which in Fig. 4.9(d) are referred to as X1, X2, and so on[28]. Once again, we executed these queries on MonetDB, Zorba (as state-of-the-art XPath/XQuery engines) and XSQ (as state-of-the-art streaming XPath engine) as well as on our XSeq engine. In this experiment, the XMark data size was 57MB. Note that both Zorba and MonetDB are implemented in C/C++ while XSeq is coded in Java, which generally accounts for an overhead factor of 2X in a fair comparison with C/C++ implementations. The results are summarized in Fig. 4.9(d). The XSeq queries were consistently competitive compared to all the three state-of-the-art XPath/XQuery engines. XSeq is faster than XSQ for most of the tested queries. For some queries, e.g. X2 and X4, XSeq is even 2-4 times faster. Even compared with MonetDB and Zorba, XSeq is giving surprisingly competitive performance, and for some queries, e.g. X4, were even faster. Given that XSeq is coded in Java, this is an outstanding result for XSeq. For instance, once the java factor is taken into account, the only XMark query that runs slower on the XSeq engine is X15, while the rest of the queries will be considered about 2X faster than both MonetDB and Zorba.

In summary, once the maturity of the research on XPath/ XQuery optimization is taken into account, our natural extension of XPath that relies on a simple VPA-based optimization seems very promising: XSeq achieves better or comparable performance on simple queries, and is dramatically faster for more involved queries.

[28] Due to space limit and similarity of the result , here we only report 7 out of the 20 XMark queries.

4.6.4 Throughput for Different Types of Queries

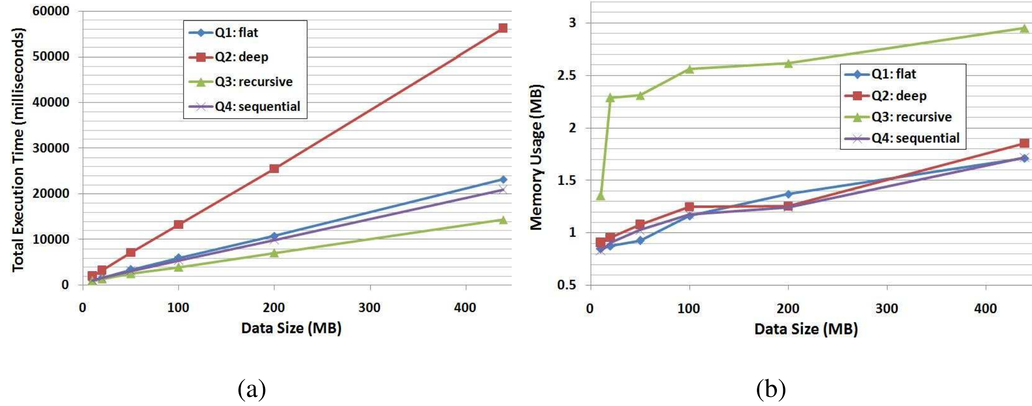
To study the performance of different types of queries in XSeq, we selected four representative queries with different characteristics which, based on our experiments, covered a wide range of different classes of XML queries. To facilitate the discussion, below we label the XML patterns as 'flat', 'deep', 'recursive' and 'monotone':

Q1: flat	/site/people/person[@id = 'person0']/name/text()
Q2: deep	/site/closed_auctions/closed_auction/annotation/ description/parlist/listitem/parlist/listitem/text/ emph/keyword/text()
Q3: recursive	(parlist/listitem)*
Q4: monotonic	//closed_auctions/ (\X[tag(X)='closed_auction' and X@price < prev(X)@price])*

We executed all these queries on XMark's dataset. Also, the first two queries (Q1 and Q2) are directly from XMark benchmark (referred to as Q1 and Q15 in [Sa02]). We refer to them as 'flat' and 'deep' queries, respectively, due to their few and many axes. In XMark's dataset, the `parlist` and `listitem` nodes can contain one another, which when combined with the Kleene-*, is the reason why we have named Q3 'recursive'. The Q4 query, called 'monotonic', searches for all sequences of consecutive closed auctions where the price is strictly decreasing. These queries reveal interesting facts about the nature of XSeq language and provide insight on the types of XSeq queries that are more amenable to efficient execution under the VPA optimizations.

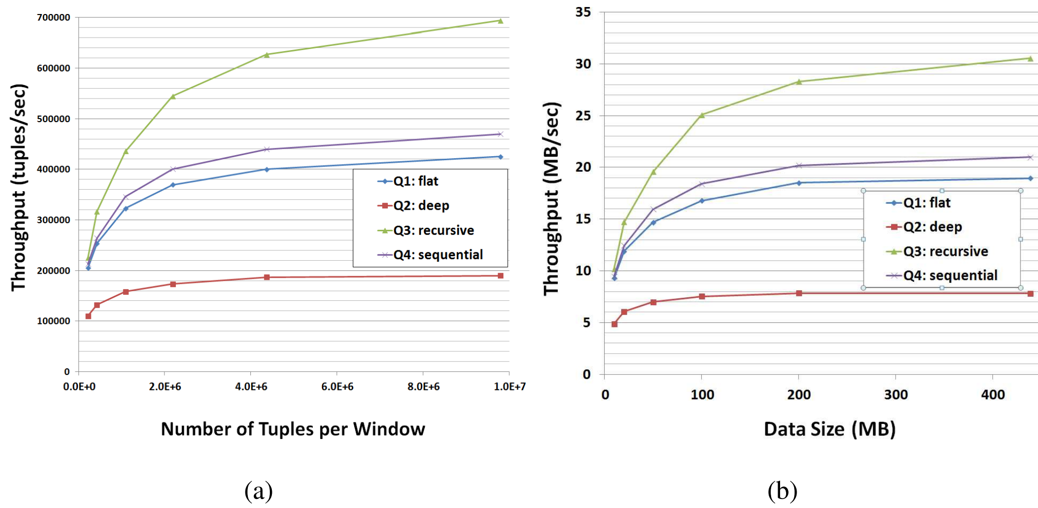
The query processing time is reported in Fig. 4.10(a). The first important observation is that XSeq has allowed for linear scalability in terms of processing time, regardless of the query type. This has enabled our XSeq engine to steadily maintain an impressive throughput of 200,000-700,000 tuples/sec, or equivalently, 8-31 MB/sec even when facing an input size of 450MB. This is shown in Fig. 4.11(a) and 4.11(b)

in which the X-axes are drawn in log-scale. Interestingly, the throughput gradually improves when the window size grows from 200K to 1.1M tuples. This is mainly due to the amortized cost of VPA construction and compilation, and other run-time optimizations such as backtrack matrices [MZZ10] that need to be calculated only once.



(a) (b)

Figure 4.10: Effect of different types of XSeq queries on total execution time (a) and memory usage (b).



(a) (b)

Figure 4.11: The effect of different types of queries on (a) Total query execution time, (b) Throughput in terms of tuple processing, and (c) Throughput in terms of datasize.

Among these queries, the best performance is delivered for Q3 and Q4. This is because they consist of only two XPath steps, and therefore, once translated into VPA, result in fewer states. Q1 comes next, as it contains more steps and thus, a longer pattern clause. Q2 achieves the worst performance. This is again expected, because Q2's deep structure contains many tag names which lead to more states in the final VPA. In summary, this experiment shows that with the help of the compile-time and run-time optimizations, XSeq queries enjoy a linear-time processing. Moreover, the fewer axes (i.e. steps) involved in the query, the better the performance.

4.7 Previous Work

XML Engines. Given the large amount of previous work on supporting XPath/X-Query on stored and streaming data, we only provide a short and incomplete overview, focusing on the streaming ones. Several XPath streaming engines have been proposed over the years, including TwigM [CDZ06], XSQ [PC03], and SPEX [OKB03]; also the processing of regular expressions, which are similar to the XPath queries of XSQ, is discussed in [OKB03] and [Ba03]. XAOS [Ba03] is an XPath processor for XML streams that also supports reverse axes (parent and ancestor), while support for predicates and wildcards is discussed in [JFB05]. Finally, support for XQuery queries on very small XML messages (<100KB) is discussed in [Fa03].

Language extensions. Extending the expressive power of XPath has been the focus of much research [Cat06, CM07b, CM07a, CS08, Mar05]. For instance, *Core XPath 2.0* [CM07a], extended Core XPath 1.0 with path intersection, complementation, and quantified variables. *Conditional XPath* [Mar05], extended XPath with 'until' operators, while the inclusion of a least fixed point operator was proposed in [Cat06]. More modest extensions, that better preserved the intuitive clarity and simplicity of Core XPath 1.0, included *Regular XPath* [Cat06], *Regular XPath*$^{\approx}$ [CM07b] and *Reg-*

ular XPath(W) [CS08]. These allowed expressions such as $/\mathtt{a}(/\mathtt{b}/\mathtt{c})^*/\mathtt{d}$, where a Kleene-* expression A^*, was defined as the infinite union $\cdot \cup A \cup (A/A) \cup (A/A/A) \cup \cdots$ Even for these more modest extensions, however, efficient implementation remained an issue: in 2006, the following open problem was declared as a challenge for the field [Cat06]: *Efficient algorithms for computing the transitive closure of XPath path expressions.*

VPA. Visibly Pushdown Automata (VPA) have been recently proposed for checking Monadic Second Order (MSO) formulas over dual-structured data such as XML [AM04, AM06], and have led to new streaming algorithms for XML processing [MV09, Pit05]. The recently proposed query language K*SQL (See Chapter 3) used VPAs to achieve good performance and expressivity levels needed to query both relational and XML streams. However, while very natural for relational data, K*SQL is quite procedural and verbose for XML, whereby the equivalents of simple XPath queries are long and complex K*SQL statements. At the VPA implementation level, however, the same VPA optimization techniques support both XSeq and K*SQL.

Gauwin and Niehren provided translations for a streamable fragment of forward XPath into nested word automata [GN11]. XSeq on the other hand, is an MSO-complete language (and hence, subsumes XPath) and therefore, our translation to VPAs handles a much larger class of queries.

The current manuscript is an extended version of a conference paper [MZZ12], with new material that were not published in the SIGMOD version, including new applications (Sections 4.2.6, 4.2.8), formal semantics (Section 4.4), and proofs and complexity results (Sections 4.5.1, 4.5.2 and Section 4.9).

4.8 Summary of XSeq

We have described the design and implementation of XSeq, a query language for XML streams that adds powerful extensions to XPath while remaining very amenable to optimization and efficient implementation. We studied the power and efficiency of XSeq both in theory and in practice, and proved that XSeq subsumes Regular XPath and its dialects, and hence, provides the first implementation of these languages as well. Then, we showed that well-known complex queries from diverse applications, can be easily expressed in XSeq, whereas they are difficult or impossible to express in XPath and its dialects. The design and implementation of XSeq leveraged recent advances in VPAs and their online evaluation and optimization techniques. Inasmuch as XPath provides the kernel of several query languages, such as XQuery, we expect that these languages will also benefit from the extensions and implementation techniques described in this chapter.

4.9 Core XSeq Proof of Regularity

In this section, we prove Theorems 6 and 7. We first introduce two simpler query languages, called CXSeqA and CXSeqB. Next, we show how every CXSeq query can be translated into an equivalent CXSeqA query, and every CXSeqA query can be translated into an equivalent CXSeqB query. Then, we show that the domain of a CXSeqB query can be captured by a VPA (MSO equivalent model), and for every Top Down Tree Automata (MSO equivalent model) T, there exists a CXSeq query that has domain equivalent to the language of T. These last two results together prove that every MSO definable language can be expressed as the domain of some CXSeq query and viceversa (Theorem 6). Theorem 7 is a consequence of the complexity of the query transformations.

4.9.1 Core XSeq with Variable Concatenation

In Fig. 4.12 we introduce CXSeqA as a syntactic restriction of CXSeq and show that CXSeq can be compiled to CXSeqA. In the following constructions we will use $\cap$ instead of $'intersect'$.

$$
\begin{array}{rcl}
\text{PathExpr} & ::= & \text{PathExpr } 'intersect' \text{ PathExpr} \mid \text{VStep} \\
\text{VStep} & ::= & \text{Step Variable}* \\
\text{Step} & ::= & \text{Axis } ' ::' \text{ NameTest [Variable]} \\
 & \mid & \text{Axis } ' ::' \text{ NameTest} \\
\text{Axis} & ::= & \odot \mid \downarrow \mid \uparrow \mid \rightarrow \mid \leftarrow
\end{array}
$$

Figure 4.12: CXSeqA

Theorem 10. *Every CXSeq query K can be transformed into an equivalent CXSeqA query K'.*

Proof. Given a production (v, π) we define the following function $\text{st}(\pi)$ that extracts a starred path.

- $\text{st}(\pi \cap \pi') = \text{if st}(\pi) \neq null \text{ then st}(\pi) \text{ else st}(\pi')$;
- $\text{st}(\pi*) = \pi$;
- else $\text{st}(\pi) = null$.

We are given a CXSeq query $K = (V, v_0, \rho)$ such that there exists at least a production $(v, \pi) \in \rho$ such that $\text{st}(\pi) \neq null$.

We rewrite K as follows.

1. Pick a production (v, π) such that $\text{st}(\pi) = \pi'$;
2. Replace π' in π with $\odot :: _fv$ where fv is a fresh variable;

3. Add the productions $(fv, \odot :: _)$ and $(fv, \pi fv)$;

4. Repeat from 1 until there are no more starred productions.

The algorithm terminates since at every step one star is eliminated from the productions. The resulting query is still *safe*. The new query will have $O(|\rho|)$ productions. □

Next we introduce a notion of query size and show that every query of size n is equivalent to some query of size 0. Given a query $K = (V, v_0, \rho)$, the size of K, $size(K)$, is defined as follows. $size(K) = max_{(v,\pi)\in\rho} psize(\pi)$ where

- $psize(\pi \cap \pi') = max(psize(\pi), psize(\pi'))$;
- $psize(d :: a[v]v_1 \dots v_n + 2) = n + 1$;
- $psize(d :: a[v]) = 0$;
- $psize(d :: av_1 \dots v_{n+2}) = n + 1$;
- $psize(d :: a) = 0$.

Lemma 11. *Every CXSeqA query K of size n can be transformed into an equivalent CXSeqA query K' of size smaller or equal than* 1.

Proof. We are given a CXSeqA query $K = (V, v_0, \rho)$ such that there exists at least a production $(v, \pi) \in \rho$ such that $psize(\pi) > 1$.

We pick a production $p = (v, \pi)$ such that $psize(\pi) > 1$ we do the following. We first remove p from ρ. Now we compute the set of productions $\mathrm{pr}_v(\pi)$.

- $\mathrm{pr}_v(\pi \cap \pi') = \mathrm{pr}(\pi) \cup \mathrm{pr}(\pi')$;
- $\mathrm{pr}_v(d :: a[v]v_1 \ \dots \ v_{n+2}) = \{(v_2 \dots v_{n+2}, \odot :: _ v_2 \dots v_{n+2})\}$;

- $\mathrm{pr}_v(d :: av_1 \ \ldots \ v_{n+2}) = \{(v_2 \ldots v_{n+2}, \odot :: _ \, v_2 \ldots v_{n+2})\}$;
- else $\mathrm{pr}_v(\pi) = \{\}$.

and the production $\mathrm{prod}(\pi)$:

- $\mathrm{prod}(\pi \cap \pi') = \mathrm{pr}(\pi) \cap \mathrm{pr}(\pi')$;
- $\mathrm{prod}(d :: a[v]v_1 \ \ldots \ v_{n+2}) = d :: a[v]v_1(v_2 \ldots v_{n+2})$;
- $\mathrm{prod}(d :: av_1 \ \ldots \ v_{n+2}) = d :: a[v]v_1(v_2 \ldots v_{n+2})$;
- else $\mathrm{prod}(\pi) = \pi$.

Therefore we compute $\rho = \rho \cup \mathrm{pr}(\pi) \cup \{(v, \mathrm{prod}(\pi))\}$.

We also compute the following new set of variables $\mathrm{nv}(\pi)$.

- $\mathrm{nv}(\pi \cap \pi') = \mathrm{pr}(\pi) \cup \mathrm{pr}(\pi')$;
- $\mathrm{nv}(d :: a[v]v_1 \ldots v_{n+2}) = \{(v_2 \ldots v_n + 2)\}$;
- $\mathrm{nv}(d :: av_1 \ldots v_{n+2}) = \{(v_2 \ldots v_n + 2)\}$;
- else $\mathrm{nv}(\pi) = \{\}$.

Next, we update the set V as follows: $V := V \cup P_\pi \cup \{(v, \mathrm{prod}(\pi))\}$. The algorithm repeats this step until there are no more productions of size greater than 1. The algorithm terminates since at every step, if a rule of size n is picked the number of productions of size n is reduced by one and only rules of size smaller than n are produced.

We now need to show that every intermediate result is a *safe* query. This is straightforward from the construction.

We define the length of a query $length(K)$ as the sum of the length of all the productions in ρ. $length(L) = \sum_{(v,\pi)\in\rho} plength(\pi)$ where

- $plength(\pi \cap \pi') = plength(\pi) + plength(\pi')$;
- $plength(d :: a[v]v_1 \dots v_{n+1}) = n\}$;
- $plength(d :: av_1 \dots v_{n+1}) = n$;
- else $plength(\pi) = 0$.

The new query will have $O(|\rho| \cdot length(K))$ productions. □

Lemma 12. *Every CXSeqA query K can be transformed into an equivalent query $K' = (V', v'_0, \rho')$ where for each production $(v, \pi)\rho'$, $|\mathrm{dv}(\pi)| = 0$.*

Proof. Using lemma 11 we reduce the query to be of size 1.

Now, we are given a CXSeqA query $K = (V, v_0, \rho)$ such that there exists at least a production $(v, \pi) \in \rho$ such that $psize(\pi) = 1$.

For every production $p = (v, \pi)$ such that $psize(\pi) = 1$ we do the following.

We define the following set of variable pairs that need to be normalized as follows. Given a production (v, π), let $vp(\pi)$ be defined as follows:

- $vp(\pi_1 \cap \pi_2) = vp(\pi_1) \cup vp(\pi_2)$
- $vp(d :: a[v]v_1v_2) = \{(v_1, v_2)\}$;
- $vp(d :: av_1v_2) = \{(v_1, v_2)\}$;
- else $vp(\pi) = \{\}$.

We show how to eliminate a single variable pair from all the productions. The algorithm then iterates. Let (v_1, v_2) be a pair in $vp(\pi)$ for some $(v, \pi) \in \rho$.

Let $X = \mathrm{nv}(v_1)$ and let $\rho(X) = \bigcup_{x \in X} \rho(x)$.

Create a copy of P' of $P = \rho(X)$, where each variable occurrence $x \in X$ that is not a predicate is replaced by a new variable x'. The set of production P' is used to generate the concatenation effect in the production p. Compute $P'' = \{(v, h_{v_2}(\pi)) | (v, \pi) \in P'\}$ using the following function h.

- $h_{v_2}(\pi_1 \cap \pi_2) = h_{v_2}(\pi_1) \cap h_{v_2}(\pi_2)$
- $h_{v_2}(d :: a[v]) = d :: a[v]v_2$;
- $h_{v_2}(d :: a) = d :: av_2$;
- else $h_{v_2}(\pi) = \pi$.

Add P'' to the set of productions ρ.

Now, for each production $(v, \pi) \in \rho$ containing v_1, v_2 replace it with $(v, g_{v_1 v_2}(\pi))$ where

- $g_{v_1 v_2}(\pi_1 \cap \pi_2) = g(\pi_1) \cap g(\pi_2)$
- $g_{v_1 v_2}(d :: a[v]v_1 v_2) = d :: a[v]v_1'$;
- $g_{v_1 v_2}(d :: av_1 v_2) = d :: av_1'$;
- else $g(\pi) = \pi$.

Repeat until there are no more production of size 1.

The termination is guaranteed by the fact that whenever a pair (a, b) is eliminated, the same pair cannot appear in the derivation of a, therefore we can always identifying a total order in variables $v_1, \ldots, v_n$, such that whenever we eliminate a pair for a production v_i the number of productions of size 1 out of v_i decreases and only the number of productions for v_j, $j > i$ can increase. The correctness of the algorithm is guaranteed by the fact that every step preserves safety. $\square$

4.9.2 Core XSeq Basic

Fig. 4.13 presents the syntax of CXSeqB where queries are only allowed to have size 0. We then show that CXSeqA, and therefore CXSeq, have the same expressiveness as CXSeqB. Finally we show that all the languages we introduced can be compiled into equivalent VPA and are all MSO complete.

$$
\begin{array}{rcl}
\text{PathExpr} & ::= & \text{PathExpr } '\mathit{intersect}' \text{ PathExpr } \mid \text{VStep} \\
\text{VStep} & ::= & \text{Step Variable} \mid \text{Step} \\
\text{Step} & ::= & \text{Axis } '::' \text{ NameTest [Variable]} \\
 & \mid & \text{Axis } '::' \text{ NameTest} \\
\text{Axis} & ::= & \odot \mid \downarrow \mid \uparrow \mid \rightarrow \mid \leftarrow
\end{array}
$$

Figure 4.13: CXSeqB

Using Lemma 11 and 12 we prove the following theorem.

Theorem 13. *Every CXSeqA query K can be transformed into an equivalent CXSeqB query K'.*

4.9.2.1 Translating CXSeqB into VPA

In the following we show that a CXSeqB query can be encoded as an equivalent visibly pushdown automata.

Theorem 14. *For every query $K = (V, v_0, \rho)$ in CXSeqB there exists an equivalent VPA A over Σ such that a word w is accepted by A iff $K(w) \neq \emptyset$. A has $O(r^5 \cdot 2^r)$ states where $r = |\rho|$.*

Proof. Given a query $K = (V, v_0, \rho)$ we construct an equivalent nondeterministic VPA $A = (Q, Q_0, \Gamma, \delta, F)$. The main idea of the construction is that each variable v in V

corresponds to a query and at every step the automaton keeps in state the information on which queries are consistent with the neighboring nodes. Whenever the state contains the query v_0, a bit is set to 1 to remember that the starting query has been answered.

The state of A will encode which productions in the grammar the current node and its neighbors have been traversed with. The construction will be nondeterministic since some query results will depend on stretches of the input that have not been read yet. Assuming the query has been executed over the input using a set of productions, the run of the automaton will guess which productions have been used to process each element of the input.

We define a notion of consistency that will allow us to show the correctness of the construction. For $v \in V$, let $\rho(v) = \{(v, \pi) \mid (v, \pi) \in \rho\}$, be the set of productions in ρ with first element v. We introduce the function $f_\pi : (V, (\Sigma \cup \{_\})^5$, that given a path π, computes the set of queries that must answer the current nodes and its neighbors nodes. We define f inductively as follows:

- $f(\odot :: s[v_1]v_2) = ((\{v_1, v_2\}, s), (\{\}, _), (\{\}, _), (\{\}, _), (\{\}, _));$
- $f(\odot :: s[v_1]) = ((\{v_1\}, s), (\{\}, _), (\{\}, _), (\{\}, _), (\{\}, _));$
- $f(\odot :: s) = ((\{\}, s), (\{\}, _), (\{\}, _), (\{\}, _), (\{\}, _));$
- $f(\downarrow :: s[v_1]v_2) = ((\{\}, _), (\{v_1, v_2\}, s), (\{\}, _), (\{\}, _), (\{\}, _));$
- $f(\uparrow :: s[v_1]v_2) = ((\{\}, _), (\{\}, _), (\{v_1, v_2\}, s), (\{\}, _), (\{\}, _));$
- $f(\uparrow :: s[v_1]v_2) = ((\{\}, _), (\{\}, _), (\{\}, _), (\{v_1, v_2\}, s), (\{\}, _));$
- $f(\uparrow :: s[v_1]v_2) = ((\{\}, _), (\{\}, _), (\{\}, _), (\{\}, _), (\{v_1, v_2\}, s));$
- if $f(\pi_1) = ((C_1, c_1), (FC_1, fc_1), (P_1, p_1), (NS_1, ns_1), (PS_1, ps_1)),$

and $f(\pi_2) = ((C_2, c_2), (FC_2, fc_2), (P_2, p_2), (NS_2, ns_2), (PS_2, ps_2))$, then $f(\pi_1 \cap \pi_2) = ((C_1 \cup C_2, c_1 \diamondsuit c_2), (FC_1 \cup FC_2, fc_1 \diamondsuit fc_2), (P_1 \cup P_2, p_1 \diamondsuit p_2), (NS_1 \cup NS_2, ns_1 \diamondsuit ns_2), (PS_1 \cup PS_2, ps_1 \diamondsuit ps_2))$, where if $a \in \Sigma \cup \{_\}$, then $\diamondsuit(a, _) = \diamondsuit(_, a) = \diamondsuit(a, a) = a$, and it is undefined in any other case (when undefined the path is also inconsistent, so it should not belong to the productions);

- the omitted cases are similar.

Each state of A is an element of the relation $((2^{\rho} \times \Sigma) \cup \{\bot\})^5 \times \{0, 1\}$, such that before reading the node n, A is in state $((PC, a), (PFC, b), (PP, c), (PNS, d), (PPS, e), i)$ iff

- for each $(v, \pi) \in PC$ there exists a derivation of v starting in n with the production (v, π) and the label of n is a,
- for each $(v, \pi) \in PFC$ there exists a derivation of v starting from the first child n' of n with the production (v, π) and the label of n' is b,
- for each $(v, \pi) \in PP$ there exists a derivation of v starting from the parent n' of n with the production (v, π) and the label of n' is c,
- for each $(v, \pi) \in PNS$ there exists a derivation of v starting from the next sibling n' of n with the production (v, π) and the label of n' is d,
- for each $(v, \pi) \in PPS$ there exists a derivation of v starting from the previous sibling n' of n with the production (v, π) and the label of n' is e,
- $i = 1$ iff a state such that $(v_0, \pi) \in C$, for some π, has been already traversed.

Whenever a component of the state is $\bot$ it means that that neighbor is not defined. For example if the second component is $\bot$ it means that the node does not have a parent.

We need to restrict the states to not violate the semantics of the production we defined before (f). Given a set of productions Π, we define $\nu(\Pi) = \{v \mid \exists(v, \pi) \in \Pi\}$. Let T be the following set:

$$\{((C, a), t_1, t_2, t_3, t_4) \mid (v, \pi) \in C \wedge f_v(\pi) = ((C, a), t_1, t_2, t_3, t_4)\}$$

A state $((PC', a'), (PFC', b'), (PP', c'), (PNS', d'), (PPS', e'), i)$ is consistent with respect to ρ if the following holds:
for each $((C, a), (FC, b), (P, c), (NS, d), (PS, e)) \in T$, $\diamondsuit(a, a')$, $\diamondsuit(b, b')$, $\diamondsuit(c, c')$, $\diamondsuit(d, d')$, $\diamondsuit(e, e')$ are defined and $C \subseteq \nu(PC')$, $FC \subseteq \nu(PFC')$, $P \subseteq \nu(PP')$, $NS \subseteq \nu(PNS')$, and $PS \subseteq \nu(PPS')$. If one of the component is $\perp$ the corresponding pair in the tuple should be $(\{\}, _)$.

The set of states $Q \subseteq ((2^\rho \times \Sigma \cup \{\perp\})^5 \times \{0, 1\}$ contains all the tuples consistent with respect to ρ. The set of stack states Γ is the same as Q. The set of initial states is $Q_0 = (2^\rho \cup \{\perp\}) \times (2^\rho \cup \{\perp\}) \times \{\perp\} \times \{\perp\} \times \{\perp\} \times \{0, 1\} \cap Q$. The set of final states is $F = \{\perp\} \times \{\perp\} \times \{\perp\} \times \{\perp\} \times (2^\rho \cup \{\perp\}) \times \{0, 1\} \cap Q$. where only the previous sibling has replied to some queries (maybe the empty set if the root is not in any). We assume that A only accepts well-matched words.

We can now give the transition function δ of A. We need to maintain the state invariant defined before.

- for every S_1, S_2 such that $(FC, S_1, (PC, a), S_2, \perp, i) \in Q$,
  $(FC, S_1, (PC, a), S_2, \perp, i)$,
  $((PC, a), FC, P, NS, PS, i)) \in ((PC, a), FC, P, NS, PS, i)(\langle a)$;
- for every S_1, S_2 such that $(FC, S_1, (PC, a), S_2, \perp, 1) \in Q$ and $v_0 \in \nu(PC)$,
  $(FC, S_1, (PC, a), S_2, \perp, i)$,
  $((PC, a), FC, P, NS, PS, 1)) \in ((PC, a), FC, P, NS, PS, 0)(\langle a)$;

- for every S_1, S_2 such that $(NS, S_1, P, S_2, C, max(i,j)) \in Q$,
  $(NS, S_1, P, S_2, C, max(i,j)) \in (\bot, \bot, P', PS', \bot, i)(a\rangle, (C, FC, P, NS, PS, j)$.

This concludes the proof.

We can actually show that A does not need to have Σ as a state component, but can instead just keep a set $\Sigma' \cup \{_\}$ where Σ' contains only symbols that actually appear in the query, and $_$ is a place holder for all the other symbols.

The automaton will have $O(r^5 \cdot 2^r)$ states where $r = |\rho|$. □

4.9.2.2 Translating Top-down Tree Automata into CXSeqB

To show MSO completeness we translate nondeterministic Top-Down Tree Automata (TA) into CXSeqB. A TA A over binary trees, is a tuple (Q, q_0, δ) where Q is a set of states and $q_0 \in Q$ is the initial states. The productions in δ are of the form: $q(b(x,y)) \rightarrow q_1(x), q_2(y)$ or $q(z)$ where z and b are respectively a leaf and an internal node. The set of tree accepted by A starting in state q (L_q) is defined inductively as follows: 1) $a(x,y) \in L_q$ if $q(a(x,y)) \rightarrow q_1(x), q_2(y) \in \delta$ and $x \in L_{q_1}$ and $y \in L_{q_2}$, 2) $b \in L_q$ if $q(b) \in \delta$.

We consider an encoding of unranked trees into binary trees where for each node $n = a(x,y)$, x, n has label a, x is the first child of n and y is the next sibling of n.

Given a TA $A = (Q, q_0, \delta)$ we construct an equivalent query (V, v_0, ρ), where $V = Q$, $v_0 = q_0$ and ρ is defined as follows. For every rule $q(z) \in \delta$ the production $(q, z :: \odot)$ will belong to ρ. For every rule $q(b(x,y)) \rightarrow q_1(x), q_2(y) \in \delta$ the production $(q, (\odot :: b) \cap (\downarrow :: _[q_1]) \cap (\rightarrow :: _[q_2]))$ will belong to ρ.

Regularity of CXSeq and Complexity. Theorem 6 and Theorem 7 follow from the transformation of Section 4.9.2.2 and Theorem 14.

CHAPTER 5

Trinity.RDF: A Distributed Graph Engine for Web Scale Graphs

RDF data is becoming increasingly more available: The semantic web movement towards a web 3.0 world is proliferating a huge amount of RDF data. Commercial search engines including Google and Bing are pushing web sites to use RDFa to explicitly express the semantics of their web contents. Large public knowledge bases, such as DBpedia [ABK07] and Probase [WLW12] contain billions of facts in RDF format. Web content management systems, which model data in RDF, mushroom in various communities all around the world.

Challenges. RDF data management systems are facing two challenges: namely, systems' scalability and generality. The challenge of scalability is particularly urgent. Tremendous efforts have been devoted to building high performance RDF systems and SPARQL engines [ACK01, BKH02, jen, WSK03, CDE05, AMM09, WKB08, NW10]. Still, scalability remains the biggest hurdle. Essentially, RDF data is highly connected graph data, and SPARQL queries are like subgraph matching queries. But most approaches model RDF data as a set of triples, and use RDBMS for storing, indexing, and query processing. These approaches do not scale as processing a query often involves a large number of join operations that produce large intermediate results. Furthermore, many systems, including SW-Store [AMM09], Hexastore [WKB08], and RDF-3x [NW10] are single-machine systems. As the size of RDF data keeps soaring,

it is not realistic for single-machine approaches to provide good performance. Recently, several distributed RDF systems, such as SHARD [RS10], YARS2 [HUH07], Virtuoso [EM09], and [HAR11], have been introduced. However, they still model RDF data as a set of triples. The cost incurred by excessive join operations is further exacerbated by network communication overhead. Some distributed solutions try to overcome this limitation by brute-force replication of data [HAR11]. However, this approach simply fails in the face of complex SPARQL queries (e.g., queries with a multi-hop chain), and has a considerable space overhead (usually exponential).

The second challenge lies in the generality of RDF systems. State-of-the-art systems are not able to support general purpose queries on RDF data. In fact, most of them are optimized for SPARQL only, but a wide range of meaningful queries and operations on RDF data cannot be expressed in SPARQL. Consider an RDF dataset that represents an entity/relationship graph. One basic query on such a graph is reachability, that is, checking whether a path exists between two given entities in the RDF data. Many other queries (e.g., community detection) on entity/relationship data rely on graph operations. For example, random walks on the graph can be used to calculate the similarity between two entities. All of the above queries and operations require some form of graph-based analytics [WHY06, NG04, LYT05, SWW12]. Unfortunately, none of these can be supported in current RDF systems, and one of the reasons is that they manage RDF data in some foreign forms (e.g., relational tables or bitmap matrices) instead of its native graph form.

Overview of Our Approach. We introduce Trinity.RDF, a distributed in-memory RDF system that is capable of handling web scale RDF data (billion or even trillion triples). Unlike existing systems that use relational tables (triple stores) or bitmap matrices to manage RDF, Trinity.RDF builds on top of a memory cloud, and models RDF data in its native graph form (i.e., representing entities as graph nodes, and

relationships as graph edges). We argue that such a memory-based architecture that logically and physically models RDF in native graphs opens up a new paradigm for RDF management. It not only leads to new optimization opportunities for SPARQL query processing, but also supports more advanced graph analytics on RDF data.

To see this, we must first understand that most graph operations do not have locality [LGH07, SWX12], and rely exclusively on random accesses. As a result, storing RDF graphs in disk-based triple stores is not a feasible solution since random accesses on hard disks are notoriously slow. Although sophisticated indices can be created to speed up query processing, they introduce excessive join operations, which become a major cost for SPARQL query processing.

Trinity.RDF models RDF data as an in-memory graph. Naturally, it supports fast random accesses on the RDF graph. But in order to process SPARQL queries efficiently, we still need to address the issues of how to reduce the number of join operations, and how to reduce the size of intermediary results. In this chapter, we develop novel techniques that use efficient in-memory graph exploration instead of join operations for SPARQL processing. Specifically, we decompose a SPARQL query into a set of triple patterns, and conduct a sequence of graph explorations to generate bindings for each of the triple pattern. The exploration-based approach uses the binding information of the explored subgraphs to prune candidate matches in a greedy manner. In contrast, previous approaches isolate individual triple patterns, that is, they generate bindings for them separately, and make excessive use of costly join operations to combine those bindings, which inevitably results in large intermediate results. Our new query paradigm greatly reduces the amount of intermediate results, boosts the query performance in a distributed environment, and makes the system scale. We show in experiments that even without a smart graph partitioning scheme, Trinity.RDF achieves several orders of magnitude speed-up on web scale RDF data over state-of-the-art RDF

systems.

We also note that since Trinity.RDF models data as a native graph, we enable a large range of advanced graph analytics on RDF data. For example, random walks, regular expression queries, reachability queries, distance oracles, community searches can be performed on web scale RDF data directly. Even large scale vertex-based analytical tasks on graph platforms such as Pregel [MAB10] can be easily supported in our system. We refer interested readers to the Trinity system [SWL13, tri] for detailed information.

Contributions. We summarize the novelty and advantages of our work as follows.

1. We introduce a novel graph-based scheme for managing RDF data. Trinity.RDF has the potential to support efficient graph-based queries, as well as advanced graph analytics, on RDF.

2. We leverage graph exploration for SPARQL processing. The new query paradigm greatly reduces the volume of intermediate results, which in turn boosts query performance and system scalability.

3. We introduce a new cost model, novel cardinality estimation techniques, and optimization algorithms for distributed query plan generation. These approaches ensure excellent performance on web scale RDF data.

Outline. The rest of the chapter is organized as follows. Section 5.1 describes the difference between join operations and graph exploration. Section 5.2 presents the architecture of the Trinity.RDF system. Section 5.3 describes how we model RDF data as native graphs. Section 5.4 describes SPARQL query processing techniques. Section 5.5 shows experimental results. We conclude in Section 5.7.

5.1 Join vs. Graph Exploration

Joins are the major operator in SPARQL query processing. Trinity.RDF outperforms existing systems by orders of magnitude because it replaces expensive join operations by efficient graph exploration. In this section, we discuss the performance implications of the two different approaches.

5.1.1 RDF and SPARQL

Before we discuss join operations vs. graph exploration, we first introduce RDF and SPARQL query processing on RDF data. An RDF data set consists of statements in the form of (*subject*, *predicate*, *object*). Each statement, also known as as a triple, is about a fact, which can be interpreted as *subject* has a *predicate* property whose value is *object*. For example, a movie knowledge base may contain the following triples about the movie "Titanic":

```
(Titanic, has_award, Best_Picture)
(Titanic, casts, L_DiCaprio),
(J_Cameron, directs, Titanic)
(J_Cameron, wins, Oscar_Award)
...
```

An RDF dataset can be considered as representing a directed graph, with entities (i.e. *subjects* and *objects*) as nodes, and relationships (i.e. *predicates*) as directed edges. SPARQL is the standard query language for retrieving data stored in RDF format. The core syntax of SPARQL is a conjunctive set of triple patterns called a *basic graph pattern*. A *triple pattern* is similar to an RDF triple except that any component in the triple pattern can be a variable. A basic graph pattern describes a subgraph which a user wants to match against the RDF data. Thus, SPARQL query processing is

essentially a subgraph matching problem. For example, we can retrieve the cast of an award-winning movie directed by an award-winning director using the following query:

Example 31.

```
SELECT ?movie, ?actor WHERE{
 ?director wins ?award .
 ?director directs ?movie .
 ?movie has_award ?movie_award .
 ?movie casts ?actor .}
```

SPARQL also contains other language constructs that support disjunctive queries and filtering.

5.1.2 Using Join Operations

Many state-of-the-art RDF systems store RDF data as a set of triples in relational tables, and therefore, they rely excessively on join operations for processing SPARQL queries. In general, query processing consists of two phases [NW08]: The first phase is known as the *scan* phase. It decomposes a SPARQL query into a set of triple patterns. For the query in Example 1, the triple patterns are *?director wins ?award*, *?director directs ?movie*, *?movie has_award ?movie_award*, and *?movie casts ?actor*. For each triple pattern, we scan tables or indices to generate bindings. Assuming we are processing the query against the RDF graph in Figure 5.1. The base tables that contain the bindings are shown in Table 5.1. The second phase is the *join* phase. The base tables are joined to produce the final answer to the query.

?director	**?award**
J_Cameron	*Oscar_Award*
G_Lucas	*Saturn_Award*

?director	**?movie**
P_Haggis	*Crash*
J_Cameron	*Titanic*
J_Cameron	*Avatar*

?movie	**?movie_award**
Titanic	*Best_Picture*
Crash	*Best_Picture*

?movie	**?actor**
Crash	*D_Cheadle*
Titanic	*L_Dicaprio*
Avatar	*S_Worthington*
Star War VI	*M_Hamill*

Table 5.1: Base tables and bound variables.

Sophisticated techniques have been used to optimize the order of joins to improve query performance. Still, the approach has inherent limitations: (1) It uses many costly join operations. (2) The scan-join process produces large redundant intermediary results. From Table 5.1, we can see that most intermediary join results will be produced in vain. After all, only *Titanic* directed by *J_Cameron* matches the query. Moreover, useless intermediary resutls may only be detected in later stages of the join process. For example, if we choose to join *?director directs ?movie* and *?movie casts ?actor* first, we will not know that the resulting rows related to *Avatar* and *Crash* are useless until joining with *?director wins ?award* and *?movie has_award ?movie_award*. Sideways Information Passing (SIP) [NW09] was proposed to alleviate this problem. SIP is a dynamic optimization technique for pipelined execution plans. It introduces *filters* on subject, predicate, or object identifiers, and passes these filters to joins and scans in other parts of the query that need to process similar identifiers.

5.1.3 Using Graph Explorations

In this chapter, we adopt a new approach that greatly improves the performance of SPARQL query processing. The idea is to use *graph exploration* instead of joins.

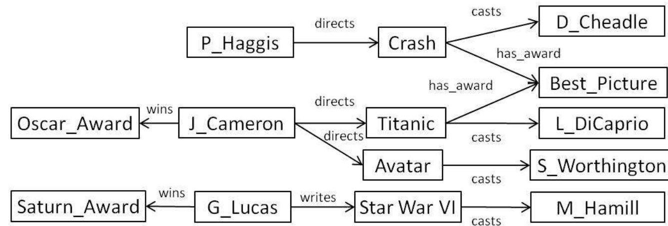


Figure 5.1: An example RDF graph

The intuition can be illustrated by an example. Assume we perform the query in Example 31 over the RDF graph in Figure 5.1 starting with the pattern: *?director wins ?award*. After exploring the neighbors of *?award* connected via the *wins* edge, we find that the possible binding for *?director* is *J_Cameron* and *G_Lucas*. Then, we explore the graph further from node *J_Cameron* and *G_Lucas* via edge *directs*, and we generate bindings for *?director directs ?movie*. In the above exploration, we prune *G_Lucas* because it does not have a *directs* edge. Also, we do not produce useless bindings as those shown in Table 5.1, such as the binding (*P_Haggis*, *Crash*). Thus, we are able to prune unnecessary intermediate results efficiently.

The above intuition is only valid *if graph exploration can be implemented more efficiently than join.* This is not true for existing RDF systems. If the RDF graph is managed by relational tables, triple stores, or disk-based key-value stores, then we need to use join operations to implement graph exploration, which means graph exploration cannot be more efficient than join: With an index, it usually requires an $O(\log N)$ operation to access the triples relating to a *subject/object*[1]. In our work, we use native graphs to model RDF data, which enables us to perform the same operation in $O(1)$ time. With the support of the underlying architecture, we make graph exploration extremely efficient. In fact, Trinity.RDF can explore as many as 2.3 million nodes on a graph distributed over an 8-server cluster within one tenth of a second [SWL13]. This

[1] N is the total number of RDF triples

lays the foundation for exploration-based SPARQL query processing.

We need to point out that the order of exploration is important. Starting with the highly selective pattern *?movie has_award ?movie_award*, we can prune a lot of candidate bindings of other patterns. If we explore the graph in a different order, e.g., exploring *?movie cast ?actor* followed by *?director directs ?movie*, then we will still generate useless intermediate results. Thus, query plans need to be carefully optimized to pick the optimal exploration order, which is not trivial. We will discuss our algorithm for optimal graph exploration plan generation in Section 5.4.

Note that graph exploration (following the links) is to certain extent similar to index-nested-loops join. However, index-nested-loops join is costly for RDBMS or disk-based data, because it needs a random access for each index lookup. Hence, in previous approaches, scan-joins, which perform sequential reads on sorted data, are preferred. Our approach further extends the random access approach in a distributed environment and minimizes the size of intermediate join results.

5.2 System Architecture

In this section, we give an overall description of the data model and the architecture of Trinity.RDF. We model and store RDF data as a directed graph. Each node in the graph represents a unique entity, which may appear as a *subject* and/or an *object* in an RDF statement. Each RDF statement corresponds to an edge in the graph. Edges are directed, pointing from *subjects* to *objects*. Furthermore, edges are labeled with the *predicates*. We will present the data structure for nodes and edges in more detail in Section 5.3.

To ensure fast random data access in graph exploration, we store RDF graphs in memory. A web scale RDF graph may contain billions of entities (nodes) and trillions

of triples. It is unlikely that a web scale RDF graph can fit in the RAM of a single machine. Trinity.RDF is based on Trinity [SWL13], which is a *distributed* in-memory key-value store. Trinity.RDF builds a graph interface on top of the key-value store. It randomly partitions an RDF graph across a cluster of commodity machines by hashing on the nodes. Thus, each machine holds a disjoint part of the graph. Given a SPARQL query, we perform search in parallel on each machine. During query processing, machines may need to exchange data as a query pattern may span multiple partitions.

Figure 5.2 shows the high level architecture of Trinity.RDF. A user submits a query to a proxy. The proxy generates a query plan and delivers the plan to all the Trinity machines, which hold the RDF data. Then, each machine executes the query plan under the coordination of the proxy. When the bindings for all the variables are resolved, all Trinity machines send back the bindings (answers) to the proxy where the final result is assembled and sent back to the user. As we can see, the proxy plays an important role in the architecture. Specifically, it performs the following tasks. First, it generates a query plan based on available statistics and indices. Second, it keeps track of the status of each Trinity machine in query processing by, for example, synchronizing the execution of each query step. However, each Trinity machine not only communicates with the proxy. They also communicate among themselves during query execution to exchange intermediary results. All communications are handled by a message passing mechanism built in Trinity.

Besides the proxy and the Trinity machines, we also employ a string indexing server. We replace all literals in RDF triples by their ids. The string indexing server implements a literal-to-id mapping that translates literals in a SPARQL query into ids, and an id-to-literal mapping that maps ids in the output back to literals for the user. The mapping can be either implemented by a separate Trinity in-memory key-value store for efficiency, or by a persistent key-value store if memory space is a concern.

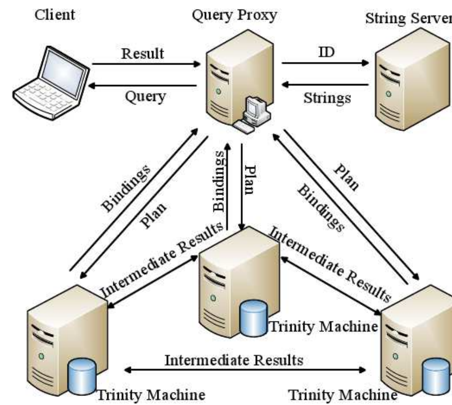


Figure 5.2: Distributed query processing framework

Usually the cost of mapping is negligible compared to that of query processing.

5.3 Data Modeling

To support graph-based operations including SPARQL queries on RDF data more effectively, we store RDF data in its native graph form. In this section, we describe how we model and manipulate RDF data as distributed graphs.

5.3.1 Modeling Graphs

Trinity.RDF is based on Trinity, which is a key-value store in a memory cloud. We then create a graph model on top of the key-value store. Specifically, we represent each RDF entity as a graph node with a unique id, and store it as a key-value pair in the Trinity memory cloud:

$$(\textit{node-id}, \langle \textit{in-adjacency-list}, \textit{out-adjacency-list} \rangle) \tag{5.1}$$

The key-value pair consists of the the *node-id* as the key, and the node's adjacency list as the value. The adjacency list is divided into two lists, one for neighbors with incoming edges and the other for neighbors with outgoing edges. Each element in the

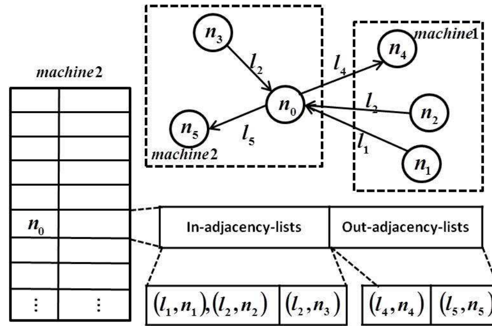


Figure 5.3: An example of model (5.1)

adjacency lists is a (*predicate*, *node-id*) pair, which records the id of the neighbor, and the predicate on the edge.

Thus, we have created a graph model on top of the key-value store. Given any node, we can find the node-id of any of its neighbors, and the underlying Trinity memory cloud will retrieve the key-value pair for that node-id. This enables us to explore the graph from any given node by accessing its adjacency lists. Figure 5.3 shows an example of the data structure.

5.3.2 Graph Partitioning

We distribute an RDF graph across multiple machines, and this is achieved by the underlying memory cloud, which partitions the key-value pairs in a cluster. However, due to the characteristics of graphs, we need to look into how the graph is partitioned in order to ensure best performance.

Two factors may have impact on network overhead when we explore a graph. The first factor is how the graph is partitioned. In our system, sharding is supported by the underlying key-value store, and the default sharding mechanism is hashing on node-id. In other words, the graph is randomly partitioned. Certainly, sophisticated graph partitioning methods can be adopted for sharding. However, graph partitioning is beyond

the scope of this discussion.

The second factor is how we model graphs on top of the key-value store. The model given by (5.1) may have potential problems for real-life large graphs. Many real-life RDF graphs are scale-free graphs whose node degrees follow the power law distribution. In DBpedia [ABK07], for example, over 90% nodes have less than 5 neighbors, while some top nodes have more than 100,000 neighbors. The model may incur a large amount of network traffic when we explore the graph from a top node x. For simplicity, let us assume none of x's neighbors resides on the same machine as x does. To visit x's neighbors, we need to send the node-ids of its neighbors to other machines. The total amount of information we need to send across the network is exactly the entire set of node-ids in x's adjacency list. For the DBpedia data, in the worst case, whenever we encounter a top node in graph exploration, we need to send $800K$ data (each node-id is 64 bit) across the network. This is a huge cost in graph exploration.

We take the power law distribution into consideration in modeling RDF data. Specifically, we model a node x by the following key-value pair:

$$(\textit{node-id}, \langle \textit{in}_1, \cdots, \textit{in}_k, \textit{out}_1, \cdots, \textit{out}_k \rangle) \tag{5.2}$$

where in_i and out_i are keys to some other key-value pairs:

$$(in_i, \textit{in-adjacency-list}_i) \qquad (out_i, \textit{out-adjacency-list}_i) \tag{5.3}$$

The essense of this model is the following: The key-value pair $(in_i, \textit{in-adjacency-list}_i)$ and the nodes in *in-adjacency-list*$_i$ are stored on the same machine i. In other words, we partition the adjacency lists in model (5.1) by machine.

The benefits of this design is obvious. No matter how many neighbors x has, we will send no more than k *nids* (in_i and out_i) over the network since each machine i,

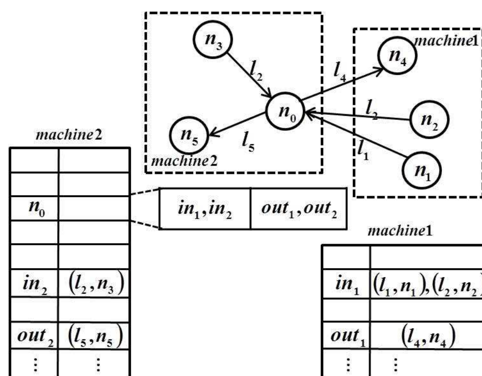


Figure 5.4: An example of model (5.2)

upon receiving nid$_i$, can retrieve x's neighbors that reside on machine i without incurring any network communication. However, for nodes with few neighbors, model (5.2) is more costly than model (5.1). In our work, we use a threshold t to decide which model to use. If a node has more than t neighbors, we use model (5.2) to map it to the key-value store; otherwise, we use model (5.1). Figure 5.4 gives an example with $t = 1$. Furthermore, in our design, all triples are stored decentralized at its subject and object. Thus, update has little cost as it only affects a few nodes. However, update is out of the scope of this chapter and we omit detailed discussion here.

5.3.3 Indexing Predicates

Graph exploration relies on retrieving nodes connected by an edge of a given predicate. We use two additional indices for this purpose.

Local Predicate Indexing. We create a local predicate index for each node x. We sort all (*predicate*, *node-id*) pairs in x's adjacency lists first by *predicate* then by *node-id*. This corresponds to the SPO or OPS index in traditional RDF approaches. In addition, we also create an aggregate index to enable us to quickly decide whether a node has a given *predicate* and the number of its neighbors connected by the predicate.

Global Predicate Indexing. The global predicate index enables us to find all nodes that have incoming or outgoing neighbors labeled by a given predicate. This corresponds to the PSO or POS index in traditional approaces. Specifically, for each predicate, machine i stores a key-value pair

$$(\textit{predicate}, \langle \textit{subject-list}_i, \textit{object-list}_i \rangle)$$

where *subject-list*$_i$ (*object-list*$_i$) consists of all unique subjects (objects) with that predicate on machine i.

5.3.4 Basic Graph Operators

We provide the following three graph operators with which we implement graph exploration:

1. $LoadNodes(predicate, dir)$: Return nodes that have an $incoming$ or $outgoing$ edge labeled as $predicate$.
2. $LoadNeighborsOnMachine(node, dir, i)$: For a given $node$, return its $incoming$ or $outgoing$ neighbors that reside on machine i.
3. $SelectByPredicate(nid, predicate)$: From a given partial adjacency list specified by nid, return nodes that are labeled with the given $predicate$.

Here, dir is a parameter that specifies whether the predicate is an $incoming$ or an $outgoing$ edge. $LoadNodes()$ is straightforward to understand. When it is called, it uses the *global predicate index* on each machine to find nodes that have at least one incoming or outgoing edge that is labeled as $predicate$.

The next two operators together find specific neighbors for a given node. In specific, $LoadNeighborsOnMachine()$ finds a node's incoming or outgoing neighbors

on a given machine. But, instead of returning all the neighbors, it simply returns the in_i or out_i id as given in (5.2). Then, given the in_i or out_i id, $SelectByPredicate()$ finds nodes in the adjacency list that is associated with the given $predicate$. Certainly, if the node has less than t neighbors, then its adjacency list is not distributed, and the two functions simply operate on the local adjacency list.

We now use some examples to illustrate the use of the above 3 operators on the RDF graph shown in Figure 5.4. $LoadNodes(l_2, out)$ finds n_2 on machine 1, and n_3 on machine 2. $LoadNeighborsOnMachine(n_0, in, 1)$ returns the partial adjacency list's id in_1, and $SelectByPredicate(in_1, l_2)$ returns n_2.

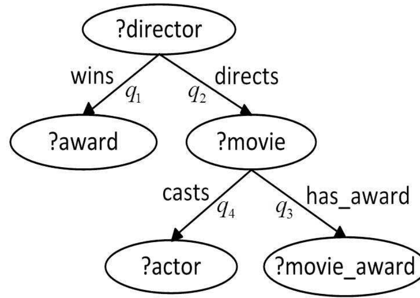
5.4 Query Processing

In this section, we present our exploration-based approach for SPARQL query processing.

5.4.1 Overview

We represent a SPARQL query Q by a *query graph* $\mathcal{G}$. Nodes in $\mathcal{G}$ denote subjects and objects in Q, and directed edges in $\mathcal{G}$ denote predicates. Figure 5.5 shows the query graph corresponding to the query in Example 31, and lists the 4 triple patterns in the query as q_1 to q_4.

With $\mathcal{G}$ defined, the problem of SPARQL query processing can be transformed to the problem of subgraph matching. However, as we pointed out in Section 2, existing RDF query processing and subgraph matching algorithms rely excessively on costly joins, which cannot scale to RDF data of billion or even trillion triples. Instead, we use efficient graph exploration in an in-memory key-value store to support fast query processing. The exploration is conducted as follows: We first decompose Q into an



- q_1: (*?director wins ?award*)
- q_2: (*?director directs ?movie*)
- q_3: (*?movie has_award ?movie_award*)
- q_4: (*?movie casts ?actor*)

Figure 5.5: The query graph of Example 31

ordered sequence of triple patterns: $q_1, \cdots, q_n$. Then, we find matches for each q_i, and from each match, we explore the graph to find matches for q_{i+1}. Thus, to a large extent, graph exploration acts as joins. Furthermore, the exploration is carried out on all distributed machines in parallel. In the final step, we gather the matchings for all individual triple patterns to the centralized query proxy, and combine them together to produce the final results.

5.4.2 Single Triple Pattern Matching

We start with matching a single triple pattern. For a triple pattern q, our goal is to find all its matches $R(q)$. Let P denote the predicate in q, V denote the variables in q, and $B(V)$ denote the binding of V. If V is a free variable (not bound), we also use $B(V)$ to denote all possible values V can take. We regard a constant as a special variable with only *one* binding.

We use graph exploration to find matches for q. There are two ways of exploration: from subject to object (We first try to find matches for the subject in q, and then for each match, we find matches for the object in q. We denote this exploration as $\overrightarrow{q}$) and from object to subject (We denote this exploration as $\overleftarrow{q}$). We use src and tgt to refer to the source and target of an exploration (i.e., in $\overrightarrow{q}$ the src is the subject, while in $\overleftarrow{q}$ the src is the object).

Algorithm 1 MatchPattern(e)

obtain src, tgt, and predicate p from e ($e = \overrightarrow{q}$ or $e = \overleftarrow{q}$)

// On the src side:
if src is a free variable **then**
  $B(src) = \bigcup_{\forall p \in B(P)} LoadNodes(p, dir)$
set $M_i = \emptyset$ for all i // initialize messages to machine i
for each s in $B(src)$ **do**
  for each machine i **do**
    $nid_i = LoadNeighborsOnMachine(s, dir, i)$
    $M_i = M_i \cup (s, nid_i)$
batch send messages M to all machines

// On the tgt side:
for each (s, nid) in M **do**
  for each p in $B(P)$ **do**
    $N = SelectByPredicate(nid, p)$
    for each n in $N \cap B(tgt)$ **do**
      $R = R \cup (s, p, n)$
return R

Algorithm 1 outlines the matching procedure using the basic operators introduced in Section 5.3.4. If src is a constant, we only need to explore from *one* node. If src is a variable, we initialize its bindings by calling $LoadNodes$, which searches the global predicate index to find the matches for src. Note that if the predicate itself is a free variable, then we have to load nodes for every predicate. After src is bound, for each node that matches src and for each machine i, we call $LoadNeighborsOnMachine()$ to find the key nid_i. The node's neighbors on machine i are stored in the key-value pair with nid_i as the key. We then send nid_i to machine i.

Each machine, on receiving the message, starts the matching on the tgt side. For each eligible predicate p in $B(P)$, we filter neighbors in the adjacency list by p by calling $SelectByPredicate()$. If tgt is a free variable, any neighbor is eligible as a binding, so we add (s, p, n) as a match for every neighbor n. If tgt is a constant, however, only the constant node is eligible. As we treat a constant as a special variable with only one binding, we can uniformly handle these two cases: we match a new edge only if its target is in $B(tgt)$.

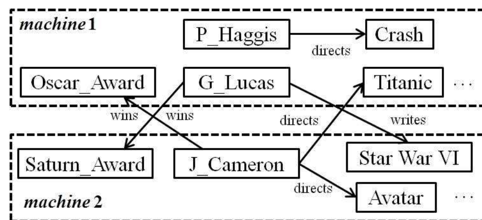


Figure 5.6: Distribution of the RDF graph in Figure 5.1

We use an example to demonstrate how $MatchPattern$ works. Assume the RDF graph is distributed on two machines as shown in Figure 5.6. Suppose we want to find matches for $\overleftarrow{q_1}$ where q_1 is "*?director wins ?award*". In this case, src is *?award*. We first call $LoadNodes(wins, in)$ to find $B(?award)$, which are nodes having an incoming $wins$ edge. This results in $Oscar_Award$ on machine 1, and $Saturn_Award$

on machine 2. Next, on the target $?director$ side, machine 1 gets the key of the adjacency list sent by $Saturn_Award$, and after calling $SelectByPredicate()$, it gets G_Lucas. Since the target $?director$ is a free variable, any edge labeled with win will be matched. We add matching edge (G_Lucas, $wins$, $Saturn_Award$) to R. Similarly on machine 2, we get ($J_Cameron$, $wins$, $Oscar_Award$).

As Algorithm 1 shows, given a triple q, each machine performs $MatchPattern()$ independently, and obtains and stores the results on the target side, that is, on machines where the target is matched. For the example in Figure 5.6, matches for $\overleftarrow{q_1}$ where q_1 is "*?director wins ?award*" are stored on machine 1, where the target G_Lucas is located. Table 5.2 shows the results on both machines for q_1. We use $R^i(q)$ to denote matches for of q on machine i. Note that the constant column $wins$ is not stored.

(a) $R^1(q_1)$

?director	**?award**
G_Lucas	*Saturn_Award*

(b) $R^2(q_1)$

?director	**?award**
J_Cameron	*Oscar_Award*

Table 5.2: Individual matching result of q_1

5.4.3 Multiple Pattern Matching by Exploration

A query consists of multiple triple patterns. Traditional approaches match each pattern individually and join them afterwards. A single pattern may generate a large number of results, and this leads to large intermediary join results and costly joins. For the example of Figure 5.6, suppose we generate the matchings for pattern q_1, q_2 separately. The results are Table 5.2 for q_1 and Table 5.3 for q_2. We can see although P_Haggis has not won an award, we still generate ($Crash$, P_Haggis) in $R(q_2)$.

Instead of matching single patterns independently, we treat the query as a sequence of patterns. The matching of the current pattern is based on the matches of the previous

(a) $R^1(q_2)$

?movie	?director
Titanic	*J_Cameron*
Crash	*P_Haggis*

(b) $R^2(q_2)$

?movie	?director
Avatar	*J_Cameron*

Table 5.3: Individual matching result of q_2

patterns, i.e., we "explore" the RDF graph from the matches of the previous patterns to find matches for the current pattern. In other words, we eargerly prune invalid matchings by exploration to avoid the cost of joining large sets of results later.

(a) $R^1(q_2)$

?movie	?director
Titanic	*J_Cameron*

(b) $R^2(q_2)$

?movie	?director
Avatar	*J_Cameron*

Table 5.4: Matching result of q_2 after matching q_1

We now use an example to illustrate the exploration and pruning process. Assume we explore the graph in Figure 5.1 in the order of $\overrightarrow{q_1}$, $\overrightarrow{q_2}$, $\overleftarrow{q_3}$, $\overrightarrow{q_4}$. Clearly, how the triple patterns are ordered may have a big impact on the intermediate results size. We discuss query plan optimization in Section 5.4.5.

There are two different cases in exploration and pruning, and they are examplified by matching $\overrightarrow{q_2}$ after $\overrightarrow{q_1}$, and by matching $\overleftarrow{q_3}$ after $\overrightarrow{q_2}$, repsectively. We describe them separately. *In the first case, the source of exploration is bound.* Exploring q_2 after q_1 belongs to this case, as the source $?director$ is bound by q_1. So, instead of using $LoadNodes()$ to find all possible directors, we start the exploration from existing bindings ($J_Cameron$ and G_Lucas), so we won't generate movies not directed by award-winning directors. Moreover, note that in Figure 5.1, G_Lucas does not have a $directs$ edge, so exploring from G_Lucas will not produce any matching triple. It means we can prune G_Lucas safely: There is no need to send the key to its adjacency-

list across the network. The results are in Table 5.4, which contains fewer tuples than Table 5.3.

In the second case, the target of exploration is bound. Exploring q_3 after q_2 belongs to this case, as $?movie$ is bound to $\{Titanic, Avatar\}$ by $\overrightarrow{q_2}$. We only add results in this binding set to the matching results, namely ($Best_Picture$, $Titanic$). Independently, ($Best_Picture$, $Crash$) also satiesfies the pattern, but $Crash$ is not in the binding set, so it is pruned. Furthermore, since the previous binding of $Avatar$ does not match any triple in this round, it is also safely pruned from $?movie$'s binding. Finally, we incorporate the matches of q_3 into the result. As shown in Table 5.5, it now has three bound variables $?movie$, $?director$, and $?movie_award$, and contains one row ($Titanic$, $J_Cameron$, $Best_Picture$) on machine 1 where $Titanic$ is located.

?movie	**?director**	**?movie_award**
Titanic	*J_Cameron*	*Best_Picture*

Table 5.5: Results after incorporating q_2 and q_3

5.4.4 Final Join after Exploration

We used two mechanisms to prune intermediate results: a binding is pruned if it cannot reach any bound target, or it cannot be reached from any bound source. Furthermore, once we prune the target (source), we also prune corresponding values from the source (target). This greatly reduces the size of the intermediary results, and does not incur much additional communication, as shown in the previous example.

However, invalid intermediary results may still remain after the pruning. This is because the pruning of q's intermediary results only affects the bindings of q and the immediate neighbors of q. Bindings of other patterns are not considered because otherwise we need to carry all historical bindings in exploration, which incurs big com-

munication cost.

After the exploration finishes, we obtain all the matches in R. Since R is distributed and may contain invalid results, we gather these results to a centralized proxy and perform a final join to assemble the final answer. As we have eagerly prune most of the invalid results in the exploration phase, our join phase is light-weight compared with traditional RDF systems that intensively rely on joins, and we simply adopt the left-deep join for this purpose.

5.4.5 Exploration Plan Optimization

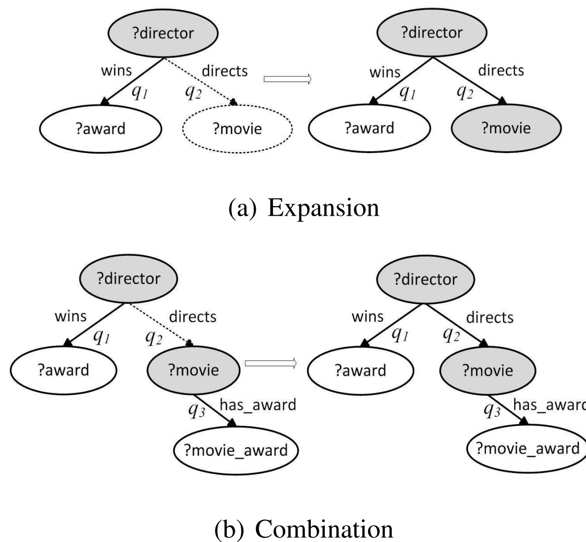
Section 5.4.3 described the query process for an ordered sequence of triple patterns. The order has significant impact on the query performance. We now describe a cost-based approach for finding the optimal exploration plan.

We define an exploration plan as a graph traversal plan, and we denote it as a sequence $\langle e_1, \cdots, e_n \rangle$, where each e_i denotes a directed exploration of a predicate q_i in the query graph, that is, $e_i = \overrightarrow{q_i}$ or $e_i = \overleftarrow{q_i}$. The cost of the plan is $\Sigma_i cost(e_i)$, where $cost(e_i)$, the cost of matching $\overrightarrow{q_i}$ or $\overleftarrow{q_i}$, is roughly proportional to the size of q_i's results (Section 5.4.6 will describe cost estimation in more depth). Clearly, the size of q_i's results depends on the matching of some $q_j, j < i$. Thus, the total cost depends on the order of e_i's in the sequence.

Naive query plan optimization is costly. There are $n!$ different orders for a query graph with n edges, and for each q_i, there are two directions of exploration. It is also temping to adopt the join ordering method in the relational query optimizer. However, there is a fundamental difference between our scenario and theirs. In the relational optimizer, later joins depend on previous intermediary join results, while for us, later explorations depend on previous intermediary bindings. The intermediary join results do not depend on the order of join, while the intermediary bindings do depend on the

order of exploration. For example, consider two plans (1) $\{\overrightarrow{q_1}, \overrightarrow{q_2}, \overleftarrow{q_3}, \overrightarrow{q_4}\}$ and (2) $\{\overrightarrow{q_2}, \overrightarrow{q_3}, \overleftarrow{q_1}, \overrightarrow{q_4}\}$, where the first 3 elements are q_1, q_2, and q_3, but in different order. For the relational optimizer (ignore the direction of each q_i), the join results q_1, q_2, and q_3 are the same no matter how they are ordered. But in our case, plan (1) produces $\{$*Titanic*$\}$ and plan (2) produces $\{$*Titanic*, *Crash*$\}$ for $B(?movie)$, as shown in Table 5.5. The redundant *Crash* will makes $\overrightarrow{q_4}$ in plan (2) more costly than plan (1).

We now introduce our approach for exploration order optimization. For a query graph, we find exploration plans for its subgraphs (starting with single nodes), and expand/combine the plans until we derive the plan for the entire query graph. There are two ways to grow a subgraph: expansion and combination. Figure 5.7(a) depicts an example of expansion: we explore to a free variable or a constant and add an edge to the subgraph. The subgraph $\{q_1\}$ is expanded to a larger graph $\{q_1, q_2\}$. Another way to grow a subgraph is that we combine two disjoint subgraphs by exploring an edge starting from one subgraph to the other. Figure 5.7(b) shows such an example: we combine the subgraph with one edge q_1 with the subplan of q_3 by exploring $\overleftarrow{q_2}$. This way, we construct a larger subgraph from two smaller subgraphs.



(a) Expansion

(b) Combination

Figure 5.7: Expansion and combination examples

Now, we introduce heuristics for exploration optimization. Let $\mathcal{E}$ denote a subgraph, $R(\mathcal{E})$ denote its intermediary join results, and $B(\mathcal{E})$ denote the bindings of variables in $\mathcal{E}$. Note that in our exploration, we compute $B(\mathcal{E})$ only, but not $R(\mathcal{E})$. Furthermore, bindings for some variables in $\mathcal{E}$ may contain redundant values. We define a variable $?c$ as an *exploration point* if it satisfies $B(c) = \Pi_c R(\mathcal{E})$. Intuitively, node $?c$ is an exploration point if it does not contain any redundant value, in other words, each of its values must appear in the intermediary join results $R(\mathcal{E})$. We then adopt the following heuristics in subgraph expansion/combination.

Heuristic 1. *We expand a subgraph from its exploration point. We combine two subgraphs by connecting their exploration points.*

The reason we want to expand/combine at the exploration point is because the exploration points do not contain redundant values. Hence, they introduce fewer redundant values for other variables in the exploration.

After the expansion/combination, we need to determine the exploration points of the resulting graph. Heuristic 1 leads to the following property:

Property 1. *We expand a subgraph or combine two subgraphs through an edge. The two nodes on both ends of the edge are valid exploration points in the new graph.*

Proof. For expansion from subgraph $\mathcal{E}$, we start from an exploration point c that satisfies $B(c) = \Pi_c R(\mathcal{E})$ and explore a new predicate $q = c \rightsquigarrow c'$. Based on our algorithm, we have $\Pi_c R(e) \subseteq \Pi_c R(\mathcal{E})$. Since $q \notin \mathcal{E}$ and $c' \notin \mathcal{E}$, we get $R(\mathcal{E} \cup q) = R(\mathcal{E}) \ltimes_c R(q)$. Thus:

$$\begin{aligned} \Pi_{c'} R(\mathcal{E} \cup q) &= \Pi_{c'}(R(\mathcal{E}) \ltimes_c R(q)) \\ &= \Pi_{c'} R(q) = B(c') \end{aligned}$$

which means c' is an exploration point of $\mathcal{E} \cup q$. After $B(c')$ is obtained, the algorithm uses it to prune $B(c)$ so that c's new binding satisfies $B(c) = \Pi_c R(q)$. Thus:

$$\begin{aligned}\Pi_c R(\mathcal{E} \cup q) &= \Pi_c(R(\mathcal{E}) \bowtie_c R(q)) \\ &= \Pi_c R(\mathcal{E}) \bowtie_c \Pi_c R(q) = \Pi_c R(q) = B(c)\end{aligned}$$

which means c is a valid exploration point of $\mathcal{E} \cup q$. Similarly, we can show Property 1 holds in subgraph combination. □

We use dynamic programming (DP) for exploration optimization. We use $(\mathcal{E}, c)$ to denote a state in DP. We start with subgraphs of size 1, that is, subgraphs of a single edge $q = u \rightsquigarrow v$. The states are $(\{q\}, u)$ and $(\{q\}, v)$. For their cost, we consider both explorations $\overleftarrow{q}$ and $\overrightarrow{q}$ to obtain the minimal cost of reaching the state.

After computing cost for subgraphs of size k, we perform expansion and combination to derive subgraphs of size $\geq k + 1$. Specifically, assuming we are expanding $(\mathcal{E}, c)$ through edge $q = c \rightsquigarrow v$, we reach two states:

$$(\mathcal{E} \cup \{q\}, v) \quad \text{and} \quad (\mathcal{E} \cup \{q\}, c) \tag{5.4}$$

Let C denote the cost of the state before expansion, and C' the cost of the state after expansion. We have:

$$C' = \min\{C', C + cost(\overrightarrow{q})\} \tag{5.5}$$

Note that: i) We may reach the expanded state in different ways, and we record the minimal cost of reaching the state; ii) C is the cost of state of size $\leq k$, which is determined in previous iterations; iii) If q is in the other direction, i.e., $q = v \rightsquigarrow c$, then $cost(\overrightarrow{q})$ above becomes $cost(\overleftarrow{q})$.

For combining two states $(\mathcal{E}_1, c_1)$ and $(\mathcal{E}_2, c_2)$ where $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$ through edge $q = c_1 \rightsquigarrow c_2$, we reach two states:

$$(\mathcal{E}_1 \cup \mathcal{E}_2 \cup q, c_1) \quad \text{and} \quad (\mathcal{E}_1 \cup \mathcal{E}_2 \cup q, c_2) \tag{5.6}$$

Let C_1 and C_2 denote the cost of the two states before combination. We update the cost of the combined state to be:

$$C' = \min\{C', C_1 + C_2 + cost(\overrightarrow{q})\} \tag{5.7}$$

We now show the complexity of the DP:

Theorem 15. *For a query graph $\mathcal{G}(V, E)$, the DP has time complexity $O(n \cdot |V| \cdot |E|)$ where n is the number of connected subgraphs in $\mathcal{G}$.*

Here is a brief sketch-proof: There are $n \cdot |V|$ states in the DP process (each subgraph $\mathcal{E}$ can have $|\mathcal{E}| \leq |V|$ nodes), and each update can take at most $O(|E|)$ time.

Theorem 16. *Any acyclic query Q with query graph $\mathcal{G}$ is guaranteed to have an exploration plan.*

We give a brief sketch-proof. Our optimizer resembles the idea of semi-joins although we do not perform join. Bernstein et al. proved [BC81] that for any relation in an acyclic query, there exists a semi-join program that can fully reduce the relation by evaluating each join condition only once. By mapping each node in $\mathcal{G}$ to a relation, and an edge in $\mathcal{G}$ to a join condition, we can see that our algorithm can find an exploration plan that evaluates each pattern exactly once.

Discussion. There are two cases we have not considered formally: i) $\mathcal{G}$ is cyclic, and ii) $\mathcal{G}$ contains a join on predicates. For the first case, our algorithm may not be able to find an exploration plan. However, we can break a cycle in $\mathcal{G}$ by duplicating some variable in the cycle. For example, one heuristic to pick the break point is that we break a cycle at node u if it has the smallest cost when we explore u's adjacent edges uv_1 and uv_2 from u; and in the case of many cycles, we repeatedly apply this process. The resulting query graph $\mathcal{G}'$ is acyclic. We can apply our algorithm to search for an approximate plan. For the second case, consider a join on predicate ($?s$ $?p$ $?u$), ($?x$

$?p$ $?y$). Here, we cannot explore from the first pattern from bound variables $?s$ or $?u$ because they are not connected with the second pattern. To handle this case, after we explore an edge with a variable predicate, we iterate through all unvisited patterns sharing the same predicate variable $?p$, i.e. ($?x$ $?p$ $?y$), and use $LoadNodes$ to create an initial binding for $?x$ and $?y$. This enable us to contine the exploration.

5.4.6 Cost Estimation

SPARQL selectivity estimation is a challenging task. Stocker et al. [SSB08] assumes subject, predicate and object are independent and the selectivity of each triple is the product of the three. The result is far from optimal. RDF-3X [NW08] uses two approaches: One assumes independence between triples and relies on traditional join estimation techniques. The other mines frequent join pathes for large joins and maintains statistics for these pathes, which is very costly and unfeasible for web-scale RDF data.

We propose a novel estimation method that captures the correlation between triples but requires little extra statistics and data preprocessing. Specifically, we estimate $cost(e)$ where $e = \overrightarrow{q}$ or $\overleftarrow{q}$. In the following, we estimate $cost(\overrightarrow{q})$ only, and the estimation of $cost(\overleftarrow{q})$ can be obtained in the same way. Also, we use src and tgt to denote the source and target nodes in e. The computation cost of matching q is estimated as the size of the results, namely $|R(q)|$. Since we operate in a distributed environment, we model communication cost as well. During exploration, we send bindings and ids of adjacency lists across network, so we measure communication cost as the binding size of the source node of the exploration, i.e. $|B(src)|$. The final $cost(\overrightarrow{q})$ is a linear combination of $|R(q)|$ and $|B(src)|$.

Now, if we know $|B(src)|$, we can estimate $|R(q)|$ and $|B(tgt)|$ as

$$|R(q)| = |B(src)|\frac{C_p}{C_p(src)}, |B(tgt)| = |B(src)|\frac{C_p(tgt)}{C_p(src)}$$

where $C_q, C_q(src), C_q(tgt)$ are the number of triples and connected subject/object with predicate p, which can be obtained from a single global predicate index look-up. If the predicate of q is unknown, we consider the average case for all possible predicates. For the case where the source or target of q is constant, we use the local predicate index to get a more accurate estimation.

We then derive $|B(src)|$. For a standalone $\overrightarrow{q}$, we can derive $|B(src)|$ from the global predicate index. When $\overrightarrow{q}$ is not standalone, the binding size of src is affected by related patterns already explored. To capture this correlation, we maintain a two-dimensional $predicate \times predicate$ matrix[2]. Each cell (i, j) stores four statistics: the number of unique nodes with predicates p_i, p_j as its incoming/outgoing edges (4 combinations). When no confusion shall arise, we simply use $C_{p_ip_j}$ to denote the correlation.

As shown in Section 5.4.5, the query optimizer handles two cases: expansion and combination. In the first case, assume we expand through a new edge p_2 from variable x which is already connected with p_1. Assume the original binding size of x is N_x. We have the new binding size N'_x as

$$N'_x = N_x \frac{C_{p_1p_2}}{C_{p_1}} \tag{5.8}$$

The second case is combining two edges p_1 and p_2 on x. Assume the original binding sizes of x with predicate p_1 and predicate p_2 are $N_{x,1}$ and $N_{x,2}$ respectively. We have the new binding size N'_x as

$$N'_x = N_{x,1}N_{x,2}\frac{C_{p_1p_2}}{C_{p_1}C_{p_2}} \tag{5.9}$$

[2] In many RDF datasets, there is a special predicate *rdf:type* which characterizes the types of entities. Since the number of entities associated with a certain type varies greatly, we treat each type as a different *predicate*.

For more complex cases in expansion and combination during exploration, e.g. expanding a new pattern from a subgraph, or joining two subgraphs, we simply pick the most selective pair from all pairs of involved predicates.

5.5 Experiments

We evaluate Trinity.RDF on both real-life and synthetic datasets, and compare it against the state-of-the-art centralized and distributed RDF systems. The results show that Trinity.RDF is a highly scalable, highly parallel RDF engine.

Systems. We implement Trinity.RDF in C#, and deploy it on a cluster, wherein each machine has 96 GB DDR3 RAM, two 2.67 GHz Intel Xeon E5650 CPUs, each with 6 cores and 12 threads, and one 40Gb/s InfiniBand Network adaptor. The OS is 64-bit Windows Server 2008 R2 Enterprise with service pack 1.

We compare Trinity.RDF with centralized RDF-3X [NW10] and BitMat [ACZ10], as well as distributed MapReduce-RDF-3X (a Hadoop-based RDF-3X solution [HAR11]). We deploy the three systems on machines running 64 bit Linux 2.6.32 using the same hardware configuration as used by Trinity.RDF. Just like Trinity.RDF, all of the competitor systems map literals to IDs in query processing. But BitMat relies on manual mapping. For a fair comparison, we measure the query execution time by excluding the cost of literal/ID mapping. Since all of these three systems are disk-based, we report both their warm-cache and cold-cache time.

Datasets. We use two real-life and one synthetic datasets. The real-life datasets are the Billion Triple Challenge 2010 dataset (BTC-10) [btc] and DBpedia's SPARQL Benchmark (DBPSB) [dbp]. The synthetic dataset is the Lehigh University Benchmark (LUBM) [GPH05]. We generated 6 datasets of different sizes using the LUBM data generator v1.7. We summarize the statistics of the data and some exemplary queries

Dataset	#Triples	#S/O
BTC-10	3,171,793,030	279,082,615
DBPSB	15,373,833	5,514,599
LUBM-40	5,309,056	1,309,072
LUBM-160	21,347,999	5,259,588
LUBM-640	85,420,588	21,037,012
LUBM-2560	341,888,947	84,202,729
LUBM-10240	1,367,122,031	336,711,191
LUBM-100000	9,956,527,583	2,452,700,932

Table 5.6: Statistics of datasets used in experiments

BTC-10	S1	S2	S3	S4	S5	S6	S7	
# of joins	7	5	9	12	6	9	7	
DBPSB	D1	D2	D3	D4	D5	D6	D7	D8
# of joins	1	1	2	3	3	4	4	5
LUBM	L1	L2	L3	L4	L5	L6	L7	
# of joins	6	1	6	4	1	3	6	

Table 5.7: Statistics of queries used in experiments

(LUBM queries are also published in [ACZ10]) in Table 5.6 and Table 5.7. All of the queries used in our experiments can be found online[3].

	L1	L2	L3	L4	L5	L6	L7	Geo. mean
Trinity.RDF	**281**	132	110	**5**	**4**	**9**	**630**	**46**
RDF-3X (In Memory)	34179	**88**	485	7	5	18	1310	143
BitMat (In Memory)	1224	4176	**49**	6381	6	51	2168	376
RDF-3X (Cold Cache)	35739	653	1196	735	367	340	2089	1271
BitMat (Cold Cache)	1584	4526	286	6924	57	194	2334	866

Table 5.8: Query run-time in milliseconds on the LUBM-160 dataset (21 million triples)

[3]http://research.microsoft.com/trinity/Trinity.RDF.aspx

	D1	D2	D3	D4	D5	D6	D7	D8	Geo. mean
Trinity.RDF	**7**	220	**5**	**7**	**8**	**21**	**13**	**28**	**15**
RDF-3X (In Memory)	15	**79**	14	18	22	34	68	35	29
BitMat (In Memory)	335	1375	209	113	431	619	617	593	425
RDF-3X (Cold Cache)	522	493	394	498	366	524	458	658	482
BitMat (Cold Cache)	392	1605	326	279	770	890	813	872	639

Table 5.9: Query run-time in milliseconds on the DBPSB dataset (15 million triples)

Join vs. Exploration. We compare graph exploration (Trinity.RDF) with scan-join (RDF-3X and BitMat) on DBPSB and LUBM-160 datasets. The experiment results show that Trinity.RDF outperforms RDF-3X and BitMat; and more importantly, its superiority does not just come from its in-memory architecture, but from the fact that graph exploration itself is more efficient than join.

For a fair comparison, we set up Trinity.RDF on a single machine, so we have the same computation infrastructure for all three systems. Specifically, to compare the in-memory performance, we set up a 20 GB *tmpfs* (an in-memory file system supported by Linux kernel from version 2.4), and deploy the database images of RDF-3X and BitMat in the in-memory file system.

The first observation is that managing RDF data in graph form is space-efficient. The database images of LUBM-160 and DBPSB in Trinity.RDF are of 1.6G and 1.9G respectively, which are smaller or comparable to RDF-3X (2GB and 1.4GB respectively), and are much more efficient than BitMat (3.6GB and 19GB respectively even without literal/ID mapping).

The results on LUBM-160 and DBPSB are shown in Table 5.8 and 5.9. For RDF-3X and BitMat, both in-memory and on-disk (cold-cache) performances are reported. Trinity.RDF outperforms the on-disk performances of RDF-3X and BitMat by a large margin for all queries: For most queries, Trinity.RDF has 1 to 2 orders of magnitude

performance gain; for some queries, it has 3 orders of magnitude speed-up. The results from the in-memory performance comparison are more interesting. Here, since all systems are memory-based, the comparison is solely about graph exploration versus scan-join. We can see that the improvement is easily 2-5 fold, and for L4, Trinity.RDF has 3 orders of magnitude speed-up. This also shows that, although SIP and semi-join are proposed to overcome the shortcomings of the scan-join approach, they are not always effective, as shown by L1, L2, L4, D1, D7, etc. Moreover, we vary the complexity of DBPSB queries from 1 join to 5 joins, where Trinity.RDF achieves very stable performance gain. It proves that our query algorithm can effectively find the optimal exploration order even for complex queries with many patterns.

We also show that in-memory RDF-3X or BitMat runs slightly better than Trinity.RDF on L2, L3 and D2. This is because L2, D2 have very simple structures and few intermediate results, and Trinity has the overhead due to its C# implementation.

Performance on Large Datasets. We experiment on three datasets, LUBM-10240, LUBM-100000 and BTC-10, to study the performance of Trinity.RDF on billion scale datasets, and compare it against both centralized and distributed RDF systems. The results are shown in Table 5.10, 5.11 and 5.12. As distributed systems, Trinity.RDF and MapReduce-RDF-3X are deployed on a 5-server cluster for LUBM-10240, a 8-server cluster for LUBM-100000 and a 5-server cluster for BTC-10. And we implement the directed 2-hop guarantee partition for MapReduce-RDF-3X.

BitMat fails to run on BTC-10 as it generates terabytes of data for just a single SPO index. Similar issues happen on LUBM-100000. For some datasets and queries, BitMat and RDF-3X fail to return answers in a reasonable time (denoted as "aborted" in our experiment results).

On LUBM-10240 and LUBM-100000, Trinity.RDF gets similar performance gain over RDF-3X and BitMat as on LUBM-160. Even compared with MapReduce-RDF-

	L1	L2	L3	L4	L5	L6	L7	Geo. mean
Trinity.RDF	**12648**	6018	8735	**5**	**4**	**9**	31214	**450**
RDF-3X (Warm Cache)	36m47s	14194	27245	8	8	65	69560	2197
BitMat (Warm Cache)	33097	209146	**2538**	aborted	407	1057	aborted	5966
RDF-3X (Cold Cache)	39m2s	18158	34241	1177	1017	993	98846	15003
BitMat (Cold Cache)	39716	225640	9114	aborted	494	2151	aborted	9721
MapReduce-RDF-3X (Warm Cache)	17188	**3164**	16932	14	10	720	**8868**	973
MapReduce-RDF-3X (Cold Cache)	32511	7371	19328	675	770	1834	19968	5087

Table 5.10: Query run-times in milliseconds for the LUBM-10240 dataset (1.36 billion triples)

	L1	L2	L3	L4	L5	L6	L7	Geo. mean
Trinity.RDF	176	21	119	**0.005**	**0.006**	**0.010**	126	**1.494**
RDF-3X (Warm Cache)	aborted	96	363	0.011	0.006	0.021	548	1.726
RDF-3X (Cold Cache)	aborted	186	1005	874	578	981	700	633.842
MapReduce-RDF-3X (Warm Cache)	**102**	**19**	**113**	0.022	0.016	0.226	**51.98**	2.645
MapReduce-RDF-3X (Cold Cache)	171	32	151	1.113	0.749	1.428	89	13.633

Table 5.11: Query run-times in seconds for the LUBM-100000 dataset (9.96 billion triples)

	S1	S2	S3	S4	S5	S6	S7	Geo. mean
Trinity.RDF	**12**	10	**31**	**21**	**23**	**33**	**27**	**21**
RDF-3X (Warm Cache)	108	8407	27428	62846	32	260	238	1175
RDF-3X (Cold Cache)	5265	23881	41819	91140	1041	3065	1497	8101
MapReduce-RDF-3X (Warm Cache w/o MapReduce)	132	**8**	4833	6059	24	1931	2732	453
MapReduce-RDF-3X (Cold Cache w/o MapReduce)	2617	661	13755	18712	801	4347	7950	3841
MapReduce-RDF-3X (MapReduce)	N/A	N/A	39928	39782	N/A	33699	33703	36649

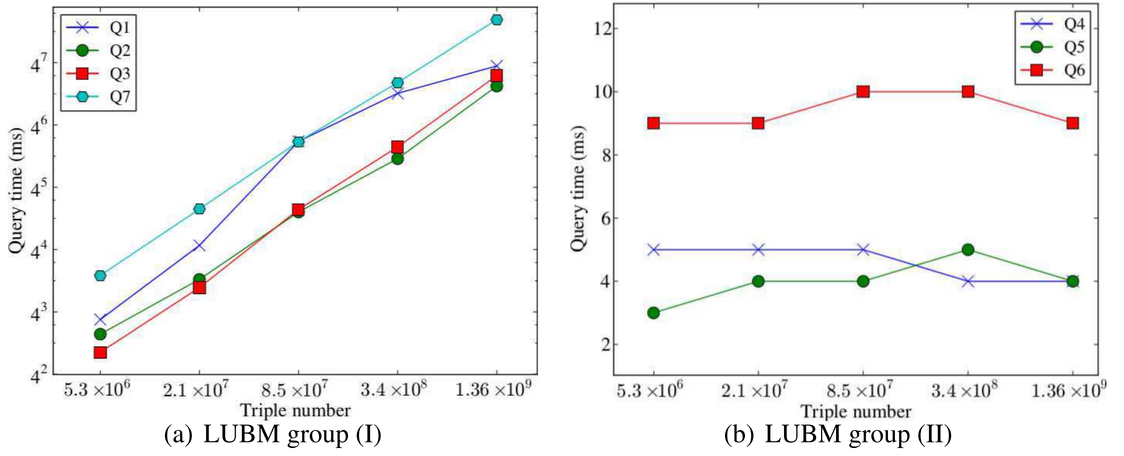
Table 5.12: Query run-times in milliseconds for BTC-10 dataset (3.17 billion triples)

3X, Trinity.RDF gives surprisingly competitive performance, and for some queries, e.g. L4-6, Trinity.RDF is even faster. These results become more remarkable if we note that all the LUBM queries are with simple structures, and MapReduce-RDF-3X specially partitions the data so that these queries can be answered fully in parallel with zero network communication. In comparison, Trinity.RDF randomly partitions the data, and has a network overhead. However, data partitioning is orthogonal to our algorithm and can be easily applied to reduce the network overhead. This is also evidenced by the results of L4-6. L4-6 only explore a small set of triples (as shown in Table 5.14) and incur little network overhead. Thus, Trinity.RDF outperforms even MapReduce-RDF-3X. Moreover, MapReduce-RDF-3X's partition algorithm incurs great space overhead. As shown in Table 5.13, MapReduce-RDF-3X indexes twice as many as triples than RDF-3X and Trinity.RDF do.

	LUBM-10240	LUBM-100000	BTC-10
#triple	2,459,450,365	20,318,973,699	6,322,986,673
Overhead	1.80X	2.04X	1.99X

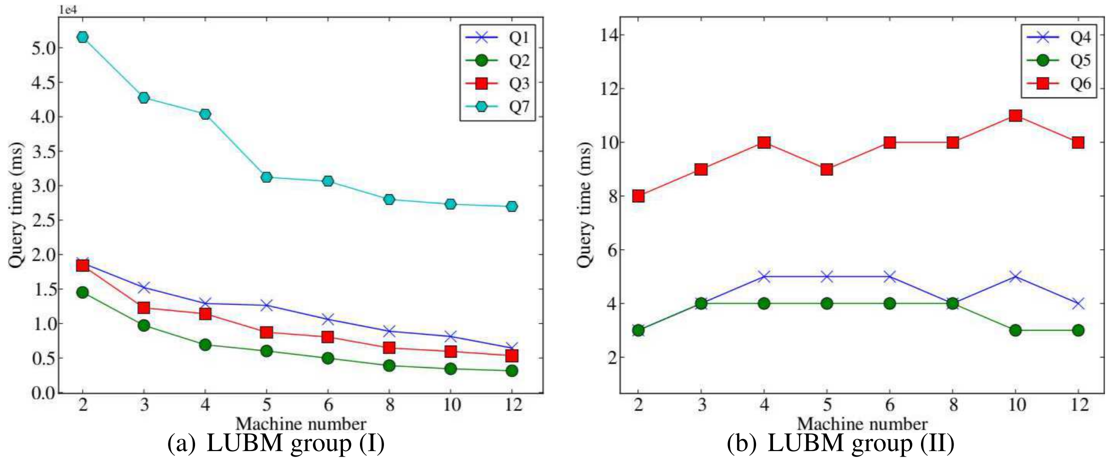
Table 5.13: The space overhead of MapReduce-RDF-3X compared with the original datasets

The BTC-10 benchmark has more complex queries, some with up to 13 patterns. In specific, S3, S4, S6 and S7 are not parallelizable without communication in MapReduce-RDF-3X, and additional MapReduce jobs are invoked to answer the queries. In Table 5.12, we list separately the time of RDF-3X jobs and MapReduce jobs for MapReduce-RDF-3X. Interestingly, Trinity.RDF shows up to 2 orders of magnitude speed-up even over the RDF-3X jobs of MapReduce-RDF-3X. This is probably because MapReduce-RDF-3X divides a query into multiple subqueries and each subquery produces a much larger result set. This result again proves the performance impact of exploiting the correlations between patterns in a query, which is the key idea behind graph exploration.



(a) LUBM group (I) (b) LUBM group (II)

Figure 5.8: Data scalability



(a) LUBM group (I) (b) LUBM group (II)

Figure 5.9: Machine scalability

	L1	L2	L3	L4	L5	L6	L7
LUBM-160	397	173040	0	10	10	125	7125
LUBM-10240	2502	11016920	0	10	10	125	450721

Table 5.14: The result sizes of LUBM queries

Scalability. To evaluate the scalability of our systems, we carry out two experiments by (1) scaling the data while fixing the number of servers, and (2) scaling the number of servers while fixing the data. We group LUBM queries into two categories according

to the sizes of their results, as shown in Table 5.14: (I) Q1, Q2, Q3, Q7. The results of these queries increase as the size of the dataset increases. Note that although Q3 produces an empty result set, it is more similar to queries in group (I) as its intermediate result set increases when the input dataset increases. (II) Q4, Q5, Q6. These queries are very selective, and produce results of constant size as the size of dataset increases.

Varying size of data: We test Trinity.RDF running on a 3-server cluster on 5 datasets LUBM-40 to LUBM-10240 of increasing sizes. The results are shown in Figure 5.8 (a) and (b). Trinity.RDF utilizes selective patterns to do efficient pruning. Therefore, Trinity.RDF achieves constant size of intermediate results and stable performance for group (II) regardless of the increasing data size. For group (I), Trinity.RDF scales linearly as the size of the dataset increases, which shows that the network overhead is alleviated by the efficient pruning of intermediate results in graph exploration.

Varying number of machines: We deploy Trinity.RDF in clusters with varying number of machines, and test its performance on dataset LUBM-10240. The results are shown in Figure 5.9 (a) and (b). For group (I), the query time of Trinity.RDF decrease reciprocally w.r.t. the number of machines. which testifies that Trinity.RDF can efficiently utilize the parallelism of a distributed system. Moreover, although more partitions increase the amount of intermediate data delivered across network, our storage model effectively bounds this overhead. For group (II), due to selective query patterns, the intermediate results are relatively small. Using more machines does not improve the performance, but again the performance is very stable and is not impacted by the extra network overhead.

5.6 Related Work

Tremendous efforts have been devoted to building high performance RDF management systems [BKH02, WSK03, CDE05, AMM09, WKB08, NW10, NW09, ACZ10, AG05, HUH07]. State-of-the-art approaches can be classified into two categories:

Relational Solutions. Most existing RDF systems use a relational model to manage RDF data, i.e. they store RDF triples in relational tables, and use RDBMS indexing to tune query processing, which aim solely at answering SPARQL queries. SW-Store [AMM09] exploits the fact that RDF data has a small number of predicates. Therefore, it vertically partitions RDF data (by predicates) into a set of property tables, maps them onto a column-oriented database, and builds subject-object index on each property table; Hexastore [WKB08] and RDF-3x [NW10] manage all triples in a giant triple table, and build indices of all six combinations (SPO, SOP, etc.).

The relational model decides that SPARQL queries are processed as large join queries, and most prior systems rely on SQL join optimization techniques for query processing. RDF-3x [NW10], which is considered the fastest existing system, proposed sophisticated bushy-join planning and fast merge join for query answering. However, this approach requires scanning large fraction of indexes even for very selective queries. Such redundancy overhead quickly becomes a bottleneck for billion triple datasets and/or complex queries. Several join optimization techniques are proposed. SIP (*sideways information passing*) is a dynamic optimization technique for pipelined execution plans [NW09]. It introduces filters on subject, predicate, or object identifiers, and passes these filters to other joins and scans in different parts of the operator tree that need to process similar identifiers. This introduces opportunities to avoid some unnecessary index scans. BitMat [ACZ10] uses a matrix of bitmaps to compress the indexes, and use lightweight semi-join operations on compressed data to reduce the intermediate result before actually joining. However, these optimizations do not solve

the fundamental problem of the join approach. In comparison, our exploration-based approach is radically different from the join approach.

Graph-based Solutions. Another direction of research investigated the possibility of storing RDF data as graphs [HG04, AG05, BHS03]. Many argued that graph primitives besides pattern matching (SPARQL queries) should be incorporated into RDF languages, and several graph models for advanced applications on RDF data have been proposed [HG04, AG05]. There are several non-distributed implementations, including one that builds an in-memory graph model for RDF data using Jena, and another that stores RDF as a graph in an object-oriented database [BHS03]. However, both of them are single-machine solutions with limited scalability. A related research area is subgraph matching [CYD08, ZCO09, HS08, ZQL11] but most solutions rely on complex indexing techniques that are often very costly, and do not have the scalability to process web scale RDF graphs.

Recently, several distributed RDF systems [HUH07, EM09, RS10, HAR11, HMM11] have been proposed. YARS2 [HUH07], Virtuoso [EM09] and SHARD [RS10] hash partition triples across multiple machines and parallelize the query processing. Their solutions are limited to simple index loop queries and do not support advanced SPARQL queries, because of the need to ship data around. Huang et al. [HAR11] deploy single-node RDF systems on multiple machines, and use the MapReduce framework to synchronize query execution. It partitions and aggressively replicates the data in order to reduce network communication. However, for complex SPARQL queries, it has high time and space overhead, because it needs additional MapReduce jobs and data replication. Furthermore, Husain et at [HMM11] developed a batch system solely relying on MapReduce for SPARQL queries. It does not provide real-time query support. Yang et al. [YYZ12] recently proposed a graph partition management strategy for fast graph query processing, and demonstrate their system on answering SPARQL queries.

However, their work focuses on partition optimization but not on developing scalable graph query engines. Further, the partitioning strategy is orthogonal to our solution and Trinity.RDF can apply their algorithm on data partitioning to achieve better performance.

5.7 Summary of Trinity.RDF

We proposed a scalable solution for managing RDF data as graphs in a distributed in-memory key-value store. Our query processing and optimization techniques support SPARQL queries without relying on join operations, and we reported performance numbers of querying against RDF datasets of billions of triples. Besides scalability, our approach also has the potential to support queries and analytical tasks that are far more advanced than SPARQL queries, as RDF data is stored as graphs. In addition, our solution only utilizes basic (distributed) key-value store functions and thus can be ported to any in-memory key-value store.

Part II

APPROXIMATION OPTIMIZATION

CHAPTER 6

EARL: Early Accurate Results for Advanced Analytics on MapReduce

In today's fast-paced business environment, obtaining results quickly represents a key desideratum for 'Big Data Analytics' [HLL11]. For most applications on large datasets, performing careful sampling and computing early results from such samples provide a fast and effective way to obtain approximate results within the prescribed level of accuracy. Although the need for approximation techniques obviously grow with the size of the data sets, general methods and techniques for handling complex tasks are still lacking in both MapReduce systems and parallel databases although these claim 'big data' as their forte. Therefore in this chapter, we focus on providing this much needed functionality. To achieve our goal, we explore and apply the powerful bootstrap method [ET93] developed in statistics to estimate results and the accuracy obtained from sampled data. We propose a method and a system that optimize the work-flow computation on massive data-sets to achieve the desired accuracy while minimizing the time and the resources required. Our approach is effective for analytical applications of arbitrary complexity (e.g., complex data mining tasks), and is supported by an Early Accurate Result Library (*EARL*) that we developed for Hadoop, which will be released for experimentation and non-commercial usage [rel]. The early approximation techniques presented here are also important for fault-tolerance, when some nodes fail and error estimation is required to determine if node recovery is nec-

essary.

The importance of EARL follows from the fact that real-life applications often have to deal with a tremendous amount of data. Performing analytics and delivering exact query results on such large volumes of stored data can be a lengthy process, which can be entirely unsatisfactory to a user. In general, overloaded systems and high delays are incompatible with a good user experience; moreover approximate answers that are accurate enough and generated quickly are often of much greater value to users than tardy exact results. The first line of research work pursuing similar objectives is that of Hellerstein et al. [HHW97b], where early results for simple aggregates are returned. In EARL however, we seek an approach that is applicable to complex analytics and dovetails with a MapReduce framework.

When computing some analytical function in EARL, a uniform sample, s, of stored data is taken, and the resulting error is estimated using the sample. Using sampling allows for a reduced computation and/or I/O costs. If the error is too high, then another iteration is invoked where the sample size is expanded and the error is recomputed. This process is repeated until the computed error is below the user-defined threshold. The error for arbitrary analytical functions can be estimated via the bootstrapping technique described in [ET93]. This technique relies on resampling methods, where a number of samples are drawn from s. The function of interest is then computed on each sample producing a distribution used for estimating various accuracy measures of interest. Sampling in the bootstrapping technique is done *with replacement*, and therefore an element in the resample may appear more than once.

Hadoop is a natural candidate for implementing EARL. In fact, while our early result approximation approach is general, it benefits from the fundamental Hadoop infrastructure. Hadoop employs a data *re-balancer* which distributes HDFS [had] data uniformly across the DataNodes in the cluster. Furthermore, in a MapReduce frame-

work there are a set of (key, value) pairs which map to a particular reducer. This set of pairs can be distributed uniformly using random hashing and by choosing a subset of the keys at random, a uniform sample can be generated quickly. These two features make Hadoop a desirable foundation for *EARL*, while Hadoop's popularity maximizes the potential for practical applications of this new technology.

Thus, as an underlying query processing engine we chose Hadoop [had]. Hadoop, and more generally the MapReduce framework, was originally designed as a batch-oriented system, however it is often used in an interactive setting where a user waits for her task to complete before proceeding with the next step in the data-analysis workflow. With the introduction of high-level languages such as Pig [ORS08], Sawzall [PDG05] and Hive [TSJ09], this trend had accelerated. Due to its batch oriented computation mode, traditional Hadoop provides a poor support for interactive analysis. To overcome this limitation, Hadoop Online Prototype (HOP) [CCA09] introduces a pipelined Hadoop variation in which a user is able to refine results interactively. In HOP however, the user is left with the responsibility of devising and implementing the accuracy estimation and improvement protocols. Furthermore in HOP, there is no feedback mechanism from the reducer back to the mapper, which is needed to effectively control the dynamically expanding sample.

Because *EARL* can deliver approximate results, it is also able to provide fault-tolerance in situations where there are node failures. Fault-tolerance is addressed in Hadoop via data-replication and task-restarts upon node failures, however with *EARL* it is possible to provide a result and an approximation guarantee despite node failures without task restarts.

Our approach, therefore, addresses the most pressing problem with Hadoop and with MapReduce framework in general: a high latency when processing large datasets. Moreover, the problem of reserving too many resources to ensure fault-tolerance

can also be mitigated by our approach, and is discussed in Section 6.2.4.

Contributions. The chapter makes the following three contributions:

1. A general early-approximation method is introduced to compute accurate approximate results with reliable error-bound estimation for arbitrary functions. The method can be used for processing large data-sets on many systems including Hadoop, Teradata, and others. An Early Accurate Result Library (EARL) was implemented for Hadood and used for the experimental validation of the method.

2. An improved resampling technique was introduced for error estimation; the new technique uses delta maintenance to achieve much better performance.

3. A new sampling strategy is introduced that assures a more efficient drawing of random samples from a distributed file system.

Organization. In section 6.1 we describe the architecture of our library as it is implemented on Hadoop. Section 6.2 describes the statistical techniques used for early result approximation. Section 6.3 presents the resampling optimizations. In Sections 6.5 and 6.6 we empirically validate our findings and discuss related works that inspired some of our ideas. Finally, Section 6.7 draws conclusions about our work.

6.1 Architecture

This section describes the overall EARL architecture and gives a background on the underlying system. For a list of all symbols used refer to Table 6.1. EARL consists of (1) a sampling stage, (2) a user's task, and (3) an accuracy estimation stage which are presented in Figure 6.1. The sampling stage draws a uniform sample s of size n from the original data set S of size N where $n << N$. In Section 6.2.3 we discuss how this

sampling is implemented using tuple-based and key-based sampling for MapReduce. After the initial sample s is drawn from the original data-set, B bootstrap resamples are taken from s. These resamples are used in the work phase (user's task) to generate B results, which are then used to derive a result distribution [DM01] in the accuracy estimation phase. The sample result distribution is used for estimating the accuracy. If the accuracy obtained is unsatisfactory, the above process is repeated by drawing another sample Δs which is aggregated with the previous sample s to make a larger sample s' for higher accuracy. The final result is returned when a desired accuracy is reached.

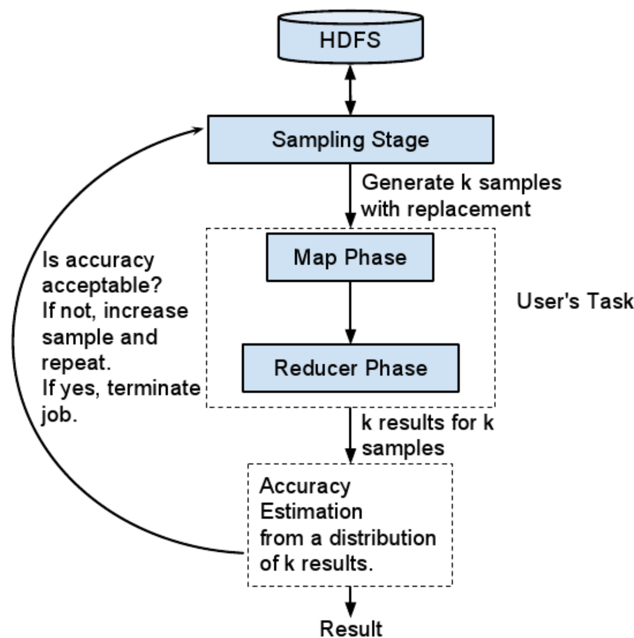


Figure 6.1: A simplified EARL architecture

6.1.1 Extending MapReduce

The MapReduce (MR) model is becoming increasingly popular for tasks involving large data processing. The programming model adopted by MapReduce was originally

Symbol	Description
B	Number of bootstraps
b	A particular bootstrap sample
n	Sample size
s	Array containing the sample
p	Percentage of the data contained in a sample
N	Total data size
S	Original data-set
F_i	File split i
c_v	Coefficient of variation
f	Statistic of interest
σ	User desired error bound
τ	Error accuracy
X_i	A particular data-item i
D	Total amount of data processed

Table 6.1: Symbols used

inspired by functional programming. In the MR model two main *stages*, map and reduce, are defined with the following signatures:

$$map : (k_1, v_1) \rightarrow (k_2, list(v_2))$$

$$reduce : (k_2, list(v_2)) \rightarrow (k_3, v_3)$$

The map function is applied on every tuple (k_1, v_1) and produces a list of intermediate (k_2, v_2) pairs. The reduce function is applied to all intermediate tuples with the same key producing (k_3, v_3) as output. The MapReduce model makes it simple to parallelize *EARL*'s approximation technique introduced in Section 6.2.1.

Hadoop, an open source implementation of the MapReduce framework, leverages Hadoop Distributed File System (HDFS) for distributed storage. HDFS stores file system metadata and application data separately. HDFS stores metadata on a dedicated

node, termed the *NameNode* (other systems, such as the Google File System (GFS) [GGL03] do likewise). The application data is stored on servers termed *DataNode*s. All communication between the servers is done via TCP protocols. The file block-partitioning, the replication, and the logical data-splitting provided by HDFS simplify *EARL*'s sampling technique, as discussed in Section 6.2.3.

For implementing the underlying execution engine, we evaluated 3 alternatives, (1) Hadoop, (2) HaLoop [BHB10] and (3) Hadoop online [CCA09]. Although HaLoop would allow us to easily expand the sample size on each iteration, it would be slow for non-iterative MR jobs due to the extra overhead introduced by HaLoop. With Hadoop online, we would get the benefit of pipelining, however further modifications would be needed to allow the mapper to adjust the current sample size. Since both Hadoop Online and HaLoop do not exactly fit our requirements, we therefore decided to make a relatively simple change to Hadoop that would allow dynamic input size expansion required by out approach. Thus *EARL* adds a simple extension to Hadoop to support dynamic input and efficient resampling. An interesting future direction is to combine *EARL*'s extensions with those of HaLoop and HOP to make a comprehensive data-mining platform for analyzing massive data-sets. In summary, with the goals of seeking *EARL* fast and requiring the least amount of changes to the core Hadoop implementation we decided to use the default version of Hadoop instead of using Hadoop extensions such as Hadoop online or HaLoop .

To achieve dynamic input expansion we modify Hadoop in three ways: (1) to allow the reducers to process input before mappers finish, (2) to keep mappers active until explicitly terminated, and (3) to provide a communication layer between the mappers and reducers for checking the termination condition. While the first goal is similar to that of pipelining implemented in Hadoop Online Prototype (HOP) [CCA09], *EARL* is different from HOP in that in *EARL* the mapper is actively, rather than a passively,

transfers the input to the reducer. In other words, the mapper actively monitors the sample error and actively expands the current sample size. The second goal is to minimize the overall execution time, thus instead of restarting a mapper every time sample size expands, we reuse an already active mapper. Finally, each mapper monitors the current approximation error and is terminated when the required accuracy is reached.

We also modify the reduce phase in Hadoop to support efficient incremental computation of the user's job. We extend the MapReduce framework with a finer-grained reduce function, to implement incremental processing via four methods: (i) *initialize*, (ii) *update*, (iii) *finalize* and (iv) *correct*. The *initialize* function reduces a set of data values into a *state*, i.e., $(k, v_1), (k, v_2), \cdots \rightarrow (k, state)$. A state is a representation of a user's function f after processing s on f. Each resample will produce a state. Saving states instead of the original data requires much less memory as needed for fast in-memory processing. The *update* function updates the state with a new input which can be another state or a $(key, value)$ pair. The *finalize* function computes the current error and outputs the final result. The *correct* function takes the output of the *finalize* function, and corrects the final result. When computed from a subset of the original data, some user's tasks need to be adjusted in order to get the right answer. For example, consider a SUM query which sums all the input values. If we only use p of the input data, we need to scale the result by $1/p$. As the system is unaware of the internal semantics of user's MR task, we allow our users to specify their own correction logic in *correct* with a system provided parameter p which is the percentage of the data used in computation.

Hadoop's limited two stage model makes it difficult to design advanced data-mining applications for which reason high level languages such as PIG [ORS08] were introduced. *EARL* does not change the logic of the user's MapReduce programs and achieves the early result approximation functionality with minimal modifications to

the user's MR job (see Figure 6.4). Next the accuracy estimation stage is described.

6.2 Estimating Accuracy

In EARL, error estimation of an arbitrary function can be done via resampling. By re-computing a function of interest many times, a result distribution is derived from which both the approximate answer and the corresponding error are retrieved. EARL uses a clever delta maintenance strategy that dramatically decreases the overhead of computation. As a measurement of error, in our experiments, we use a coefficient of variation (c_v) which is a ratio between the standard deviation and the mean. Our approach is independent of the error measure and is applicable to other errors (e.g., bias, variance). Next a traditional approach to error estimation is presented, after which our technique is discussed.

A crucial step in statistical analysis is to use the given data to estimate the accuracy measure, such as the bias, of a given statistic. In a traditional approach, the accuracy measure is computed via an empirical analog of the explicit theoretical formula derived from a postulated model [ET93]. Using variance as an illustration let $X_1, ..., X_n$ denote the data set of n independent and identically distributed (i.i.d.) data-items from an unknown distribution F and let $f_n(X_1, ..., X_n)$ be the function of interest we want to compute. The variance of f_n is then:

$$var(f_n) = \int [f_n(x) - \int f_n(y) d \prod_{i=1}^{n} F(y_i)]^2 d \prod_{i=1}^{n} F(x_i) \tag{6.1}$$

where $x = (x_1, ..., x_n)$ and $y = (y_1, ..., y_n)$. Given a simple f_n we can obtain an equation of $var(f_n)$ as a function of some unknown quantities and then substitute the estimates of the unknown quantities to estimate the $var(f_n)$. In the case of the sample mean, where $f_n = \bar{X}_n = n^{-1} \sum_{i=1}^{n} X_i$, $var(\bar{X}_n) = n^{-1} var(X)$. We can therefore estimate $var(\bar{X}_n)$ by estimating $var(X)$ which is usually estimated by the

sample variance $(n-1)^{-1}\sum_{i=1}^{n}\left(X_i-\bar{X}_n\right)^2$. The use of Equation 6.1 to estimate the variance is computationally feasible only for simple functions, such as the mean. Next we discuss the bootstrap method used to estimate the variance of arbitrary functions.

Bootstrap provides an accuracy estimation for general functions, which does not require a theoretical formula to produce the error estimate of a function. Bootstrap can estimate the sampling distribution of almost any statistic, by simply using repeated computation of the function of interest on different resamples [ET93]. The estimate of the variance of the result, can then be determined from the sampling distribution, i.e., from the repeated computation we can get $var(f_n) = E_F(f_n - E_F(f_n))^2$.

To compute an *exact* bootstrap variance estimate $\binom{2n-1}{n-1}$ resamples are required, which for $n = 15$ is already equal to 77×10^6, therefore an approximation is necessary to make the bootstrap technique feasible. The *Monte-Carlo simulation* [ET93] is the standard approximation technique used for resampling methods including the bootstrap that requires less than n resamples. It works by taking B resamples resulting in variance estimate of $v\hat{a}r_B = \frac{1}{B}\sum_{i=1}^{B}(f_n^i - f_n^*)^2$ where f_n^* is the average of f_n^i's. The theory suggests that B should be set to $\frac{1}{2}\epsilon_0^{-2}$, where ϵ_0 corresponds to the desired error of the Monte Carlo approximation with respect to the the original bootstrap estimator. Experiments, however, show that a much better value of B can be used in practical applications, therefore in Section 6.2.2 we develop an algorithm to empirically determine a good value of B.

6.2.1 Accuracy Estimation Stage

The accuracy estimation stage (*AES*) uses the bootstrap resampling technique outlined in the previous subsection to estimate the standard error c_v of the statistic f computed from sample s.

In many applications, the number of bootstrap samples required to estimate c_v to

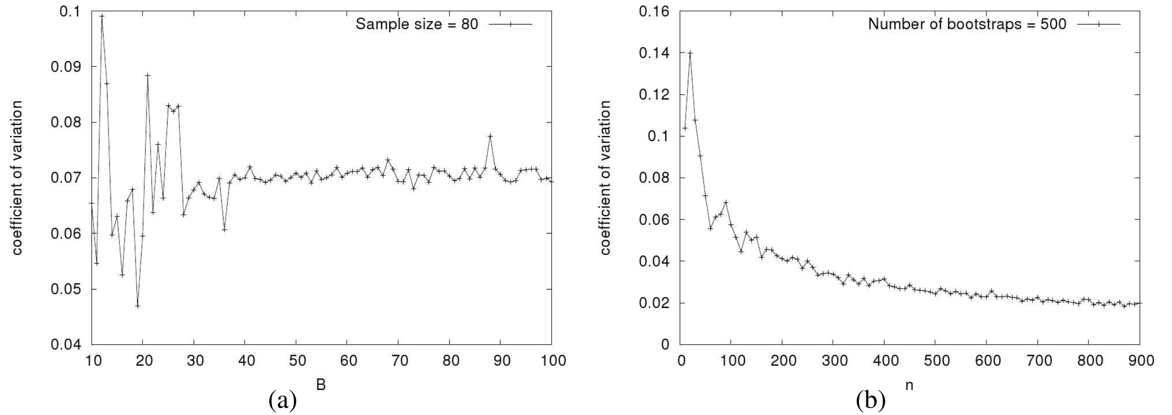


Figure 6.2: (a) Effect of B on c_v, (b) Effect of n on c_v

within a desired accuracy τ can be substantial. τ is defined as $\tau = (c_{v_i} - c_{v_{i+1}})$ which measures the stability of the error. Before performing the approximation, we estimate the required B and n to compute f with $c_v \leq \sigma$. If $B \times n \geq N$, then EARL informs the user that an early estimation with the specified accuracy is not faster than computing f over N and instead the computation over the entire data-set is performed. AES allows for error estimation of general MR-Jobs (mining algorithms, complex functions etc).

For completeness, we will first discuss how B and n impact the error individually, and then in Section 6.2.2 we present an algorithm to pick B and n that empirically minimizes the product $B \times n$. Figure 6.2 (left) shows how B affects c_v experimentally. Normally roughly 30 bootstraps are required to provide a confident estimate of the error. The sample size, n, given a fixed B has a similar effect on c_v as shown in Figure 6.2 (right). A larger n results in a lower error. Depending on the desired accuracy, n can be chosen appropriately as described next.

6.2.2 Sample Size and Number of Bootstraps

To perform resampling efficiently (i.e., without processing more data than is required) we need to minimize the sample size (n) and the number of resamples performed

(B). A straightforward sample size adjustment might work as follows: pick an initial sample size s of size n which theoretically achieves the desired error σ and compute f on s. If the resulting error $\hat{\sigma}$ is greater than σ then the sample size is increased (e.g., doubled). A similar naïve strategy may be applicable when estimating the minimum B. This naïve solution however may result in an overestimate of the sample size and the number of resamples. Instead, following [CDS04] we propose a two phase algorithm to estimate the final early approximate result satisfying the desired error bound while empirically minimizing $B \times n$. As shown later, our algorithm requires only a single iteration.

Sample Size And Bootstrap Estimation (SSABE) algorithm performs the following operations: (1) In the first phase, it estimates the minimum B and n and then (2) in the second phase, it evaluates the function of interest, f, B times on s of size n. To estimate the required B, the first phase an initially small n, a fraction p of N, is picked. In practice we found that $p = 0.01$ gives robust results. Given a fixed n, a sample s is picked. The function f is then computed for different candidate values of B $(\{2, ..., \frac{1}{\tau}\})$. The execution terminates when the difference $|c_{v_i} - c_{v_{i-1}}| < \tau$. The B value so determined is used as the estimated number of bootstraps. In practice the value of B so calculated is much smaller than the theoretically predicted $\frac{1}{2}\epsilon_0^{-2}$.

To estimate the required sample size n, first the initial sample size $\frac{1}{\tau}$ is picked. The initial sample is split into l smaller subsamples s_i each of size n_i where $n_i = \frac{n}{2^{l-i}}$ and $1 \leq i \leq l$. In our experiments we found it to be sufficient to set $l = 5$. For each s_i we compute the c_v using B resamples. When computing f on s_i we perform delta maintenance discussed in Section 6.3. The result will be a set of points $A[s_i] = c_v$. For these set of points, the best fitting curve is constructed. The curve fitting is done using the standard method of least squares. The best fitted curve yields an s_i that satisfies the given σ. Finally, once the estimate for B and n is complete, the second phase is

invoked where the actual user job is executed using s of size n and B.

The initial n is picked to be small, therefore the sample size and the number of bootstraps estimation can be performed on a single machine prior to MR job start-up. Thus, when performing the estimation for n and B, we run the user's MR job in a local mode without launching a separate JVM. Using the local-mode we avoid running the mapper and the reducer as separate JVM tasks and instead a single JVM is used which allows for a fast estimation of the required parameters needed to start the job.

6.2.3 Sampling

In order to provide a uniformly random subset of the original data-set, *EARL* performs sampling. While sampling over memory-resident, and even disk resident, data had been studied extensively, sampling over a distributed file system, such as HDFS, has not been fully addressed [OR90]. Therefore, we provide two sampling techniques: (1) pre-map sampling and (2) post-map sampling. Each of the techniques has its own strengths and weaknesses as discussed next.

In HDFS, a file is divided into a set of blocks, each block is typically 64MB. When running an MR job, these blocks can be further subdivided into "Input Splits" which are used as input to the mappers. Given such an architecture, a naïve sampling solution is to pick a set of blocks B_i at random, possibly splitting B_i into smaller splits, to satisfy the required sample size. This strategy however will not produce a uniformly random sample because each of the B_i and each of the splits can contain dependencies (e.g., consider the case where data is clustered on a particular attribute resulting in clustered items to be placed next to each other on disk due to spatial locality). Another naïve solution is to use a reservoir smapling algorithm to select n random items from the original data-set. This approach produces a uniformly random sample, but it suffers from slow loading times because the entire dataset needs to be read, and possibly re-

read when further samples are required. We thus seek a sampling algorithm that avoids such problems.

In a MapReduce environment, sampling can be done before or while sending the input to the Mapper (*pre-map* and *post-map* sampling respectively). Pre-map sampling significantly reduces the load times, however the sample produced may be an inaccurate representation of the total (k, v) pairs present in the input. Post-map sampling first reads the data and then outputs a uniformly random sample of desired size. Post-map sampling also avoids the problem of inaccurate (k, v) counts.

Post-map sampling works by reading and parsing the data before sending the selected (k, v) pairs to the reducer. Each (k, v) pair is stored by using random hashing that generates a pre-determined set of keys, of size proportional to the required sample size. We store all (k, v) pairs on the mapper locally, and when all data had been received, we randomly pick p (k, v) pairs that satisfy the sample size and send them to the reducer. Because sampling is done without replacement, the (k, v) pairs already sent are removed from the hashmap. Post-map sampling is shown in Algorithm 2.

Unlike *post-map* sampling, which first reads the entire dataset and then randomly chooses the required subset to process, *pre-map* sampling works by sampling a portion p of the initial dataset before it gets passed into the mapper. Therefore, because sampling is done prior to data loading stage, the response time is greatly improved, with a potential downside of a slightly less accurate result. The reason for this is because when sampling from HDFS directly, we can efficiently only do so by sampling lines[1]. Each line however may contain a variable number of (k, v) pairs so that when producing a 1$\%$ sample of the (k, v) pairs, we may produce a larger or a lesser sample. Therefore, for f which needs correction, we would be unable to do so accurately without additional information from the user. For majority of the cases however correcting

[1] A default file format in Hadoop is a line delimitted by a new-line character. Another format can be specified via the *RecordReader* class in Hadoop.

Algorithm 2 Post-map sampling

```
    hash ← initialize the hash
 2: timestamp ← initialize the timestamp
    while input ! = null do
 4:   key ← get random key for input
      value ← get value for input
 6:   hash[key] ← value
    sendSample(hash(rand()%hash_size)
 8: while true do
      if get_new_error_average (timestamp) > required then
10:     sendSample(hash(rand()%hash_size))
      else
12:     return
```

the final result is not necessary, and even for cases when correction is required, the estimate of the number of the (k, v) pairs produced by the *pre-map* sampling approach is good enough in practice. Nevertheless the user has the flexibility to use *post-map* sampling if an accurate correction to the final result is desired.

We assume, w.l.o.g., that the input is delimited by *new-lines*, as opposed to commas or other delimiters. A set of logical splits is first generated from the original file which will be used for sampling. For each split F_i, we maintain a bit-vector representing the *start* byte locations of the lines we had already included in our sample. Therefore until the required sample size is met, we continue picking a random F_i and a random *start* location which will be used to include a line from a file. To avoid the problem of picking a file *start* location which is not a beginning of a line, we use Hadoop's *LineRecordReader* to backtrack to the beginning of a line. Using *pre-map* sampling we avoid sending an overly large amount of data to the mapper which improves response

time as seen in experiment in Section 6.5.1. In rare cases where a larger sample size is required for an in-progress task, a new split is generated and the corresponding map task is restarted in the *TaskInProgress* Hadoop class. Algorithm 3 presents the HDFS sampling algorithm used in pre-map sampling.

Algorithm 3 HDFS sampling algorithm used in pre-map sampling

$start \leftarrow split.getStart()$

$end \leftarrow start + split.getLength()$

$sample \leftarrow \emptyset$

while $|sample| < n$ **do**

  $start \leftarrow$ pick a random start position

  if start $! =$ beginning of a line **then**

    skipFirstLine $\leftarrow$ true

    fileIn.seek(start)

  in = new LineReader(fileIn, job)

  if skipFirstLine **then**

    start += in.readLine(new Text(),

    0, (int)Math.min((long)Integer.MAX_VALUE,

    end - start))

  sample $\leftarrow$ includeLineInSample()

  skipFirstLine $\leftarrow$ false

Therefore, while pre-map is fast and works well for most cases, post-map is still very useful for applications where a correction function relies on an accurate estimate of the total *key, value* pairs. Experiments highlighting the difference between the two sampling methods are presented in experiment of Section 6.5.5.

In both the post-map and the pre-map sampling, every reducer writes its computed error together with a time-stamp onto HDFS. These files are then read by the mappers

to compute the overall average error. Because both the mappers and the reducers share the same JobID, it is straight forward to list all files generated by the reducers within the current job. The mapper stores a time-stamp that corresponds to the last successful read attempt of the reducer output. The mapper collects all errors, and computes the average error. The average error, incurred by all the reducers, is used to decide if sample size expansion is required. Lines 9-15 in Algorithm 2 demonstrate this for pos-map sampling.

Note that in a MapReduce framework independence is assumed between (k, v) pairs. In addition to being natural in a MapReduce environment, the independence assumption also makes sampling applicable to algorithms relying on capturing data-structure such as correlation analysis.

6.2.4 Fault Tolerance

Most clusters that use Hadoop and the MapReduce frameworks utilize commodity hardware and therefore node failure are a part of every-day cluster maintenance. Node failure is handled in the Hadoop framework with the help of data-replication and task-restarts upon failures. Such practices however can be avoided if the user is only interested in an approximate result. Authors in [SG07] show that in the real world, over 3% of hard-disks fail per year, which means that in a server farm with 1,000,000 storage devices, over 83 will fail every day. Currently, the failed nodes have to be manually replaced, and the failed tasks have to be restarted. Given a user specified approximation bound however, even when most of the nodes have been lost, a reasonable result can still be provided. Using the ideas from *AES* stage the error bound of the result can still be computed with a reasonable confidence. Using our simple framework, a system can therefore be made more robust against node failures by delivering results with an estimated accuracy despite node failures.

6.3 Optimizations

The most computationally intensive part of *EARL*, aside from the user's job j, is the re-execution of j on an increasingly larger sample sizes, during both the main job execution and during initial sample size estimation. One important observation is that this intensive computation can reuse its results from the previous iterations. By utilizing this incremental processing, performing large-scale computations can be dramatically improved. We first take a more detailed look at the processing of two consecutive bootstrap iterations and then we discuss the optimization of the bootstrapping (resampling) procedure so that when recomputing f on a new resample s' we can perform delta maintenance using a previous resample s.

6.3.1 Inter-Iteration Optimization

Let s denote the sample of size n used in the i-th iteration, and $\{b_i, 1 \le i \le B\}$ denote the B bootstrap resamples drawn from s. The user's job j is repeated on all b_i's. In the $(i+1)$-th iteration, we enlarge sample s with another sample Δs. s and Δs are combined to get a new sample s' of size n'. B bootstrapping resamples $\{b'_i, 1 \le i \le B\}$ are drawn from s', and the user's job j is repeated on all b'_i's. Each resample b'_i can be decomposed into two parts: (1) the set of data-items randomly sampled from s, denoted by $b'_{i,s}$, and (2) the set of data-items randomly sampled from Δs, denoted by $b'_{i,\Delta s}$.

Therefore, in the $(i+1)$-th iteration, instead of drawing a completely new $\{b'_i\}$ from s', we can reuse the resamples $\{b_i\}$ generated in the i-th iteration. The idea is to generate $b'_{i,s}$ by updating b_i, and to generate $b'_{i,\Delta s}$ by randomly sampling from Δs. This incremental technique has two benefits, in that we can save a part of: (1) the cost of bootstrapping resampling $\{b'_i\}$, and (2) the computation cost of repeating the user's

job j on $\{b'_i\}$.

The process of generating $b'_{i,s}$ from b_i is not trivial, due to the following observation. Each data item in b'_i is drawn from b_i with probability $\frac{n}{n'}$, and from Δs with probability $1 - \frac{n}{n'}$. We have the following equation modeling the size of $b'_{i,s}$ by a binomial distribution.

$$P(|b'_{i,s}| = k) = \binom{n'}{k} \left(\frac{n}{n'}\right)^k \left(1 - \frac{n}{n'}\right)^{n'-k} \quad (6.2)$$

This means that we may need to randomly delete data-items from b_i, or add data-items randomly drawn from s to b_i. We first present a naive algorithm which maintains a resample b'_i from s' by updating the resample b_i form s in three steps: (1) randomly generate $|b'_{i,s}|$ according to Equation 6.2. (2) if $|b'_{i,s}| < n$, then randomly delete $(n - |b'_{i,s}|)$ data-items from b_i; if $|b'_{i,s}| > n$, then randomly sample $(|b'_{i,s}| - n)$ data-items from s and combine them with b_i. (3) generate $(n' - |b'_{i,s}|)$ random sample from Δs and combine them with b_i.

The above process requires us to record all the data-items of s and b_i, which is a huge amount of data that cannot reside in memory. Therefore, s and b_i must be stored on the HDFS file system. Because this data will be accessed frequently, the disk I/O cost can be a major performance bottleneck.

Next, we present our optimization algorithm with a cache mechanism that supports fast incremental maintenance. Our approach is based on an interesting observation from Equation 6.2. With n' very large and n/n' fixed, which is usually the case in massive MapReduce tasks, Equation 6.2 can be approximated by the Gaussian distribution

$$N\left(n, n\left(1 - \frac{n}{n'}\right)\right) \quad (6.3)$$

For a Gaussian distribution, by the famous *3-sigma rule*, most data concentrate around the mean value, to be specific, within 3 standard deviations of the mean. As an exam-

ple, for the distribution 6.3 with its standard deviation denoted by $\sigma_0 = \sqrt{n\left(1 - \frac{n}{n'}\right)}$, over 99.7% data lie within the range $(n - 3\sigma_0, n + 3\sigma_0)$; over 99.9999% data lie within the range $(n - 5\sigma_0, n + 5\sigma_0)$. Note that $\sigma_0 < \sqrt{n}$.

Next we explain our optimized algorithm in more detail. For the i-th iteration, we define the delta sample added to the previous sample as Δs_i. For the first iteration, we can treat the initial sample as a delta sample added to an empty set. Therefore we can denote it by Δs_1. The size of Δs_i is n_i. After the i-th iteration, a bootstrapping resample b can be partitioned into $\{b_{\Delta s_k}, k < i\}$, where $b_{\Delta s_k}$ represents the data-items in b drawn from Δs_k. We build a two-layer memory-disk structure of b. Instead of simply storing b on a hard-disk, we build two pieces of information of it: (i) memory-layer information (a sketch structure) and (ii) disk-layer information (the whole data set). A *sketch* of data set of size n is $c\sqrt{n}$ data items randomly drawn without replacement from it where c is a chosen constant. Determining an appropriate c is a trade-off between memory space and the computation time. A larger c will cost more memory space but will introduce less randomized update latency. The sketch structure contains $\{sketch(b_{\Delta s_k})\}$ and $\{sketch(\Delta s_k)\}$.

During updating, instead of accessing s and b directly, we always access the sketches first. Specifically, for step 2 in our algorithm, if we need to randomly delete data-items from $b_{\Delta s_k}$, we sequentially pick the data-items from $sketch(b_{\Delta s_k})$ for deletion; if we need to add data-items randomly drawn from Δs_k, we sequentially pick the data-items from $sketch(\Delta s_k)$ for addition. For already picked data-items, we mark them as *used*. At the end of each iteration, we will randomly substitute some of the unused data items in $sketch(b_{\Delta s_k})$ with the used data items in $sketch(\Delta s_k)$ by following a reservoir sampling approach, in order to maintain $sketch(b_{\Delta s_k})$ as a random sketch of $b_{\Delta s_k}$. If we use up all the data-items in a sketch, we access the copy stored in HDFS, applying two operations: (1) committing the changes on the sketch, and (2) resampling a new sketch

from the data.

6.3.2 Intra-Iteration Optimization

When performing a resample, at times a large portion of the new sample is identical to the previous sample in which case effective delta maintenance can be performed. Our main observation is shown in Equation 6.4. The equation represents the probability that a fraction y of a resample is identical to that of another resample. Therefore, for example if $n = 29$ and $y = 0.3$, that means that 35% of the time, resamples will contain 30% of identical data. In other words, for roughly 1 in 3 resamples, 30% of each resample will be identical to one-another. Because of the relatively high level of similarity among samples, an intra iteration delta maintenance can be performed where the part that is shared between the resamples is reused.

$$P(X = y) = \frac{n!}{(n - y * n)! \times n^{y*n}} \tag{6.4}$$

Using Equation 6.4 we can find the optimal y, given n, that minimizes the overall work performed by the bootstrapping stage. The overall work saved can be stated as $P(X = y) * y$. Figure 6.3 shows how the overall work saved varies with n for different y. The optimal y for given n can be found using a simple binary search. Overall we found that on average we save over 20% of work using our Intra Iteration Optimization procedure when compared to the standard bootstrapping method.

While the optimization techniques presented in this section greatly increase the performance of the standard bootstrap procedure there is still more research to be done with regards to delta maintenance and sampling techniques. Our optimization techniques are best suited for small sample sizes, which is reasonable for a distributed system where both response time and data-movement must be minimized. Next, we outline several challenges that we faced during the implementation of EARL.

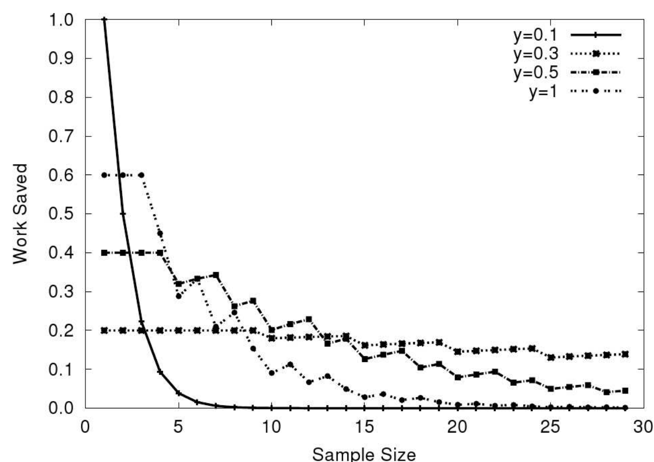


Figure 6.3: Work saved using our intra iteration optimization vs. sample size

6.4 Current Implementation

We have used Hadoop version 0.20.2-dev, to implement our extension and run the experiments on a small cluster of machines of size 5 containing Intel Core duo (CPU E8400 @ 3.00GHz), 320MB of RAM and Ubuntu 11.0 32bit. Each of the parts shown in Figure 6.1 are implemented as separate modules which can seamlessly integrate with user's Map-Reduce jobs. The sampler is implemented by modifying the *RecordReader* class to implement the *Pre-Map sampling* and extending the *map* class to implement the *Post-Map sampling*. The resampling and update strategies are implemented by extending the *Reduce* class. The results generated from resamples are used for result and accuracy estimation in the *AES* phase. The *AES* phase computes the coefficient of variation (c_v) and outputs the result to HDFS which is read by the main Map-Reduce job where the termination condition is checked. Because the number of required resamples and the required sample size are estimated via regression, a single iteration is usually required. Figure 6.4 shows an example of an MR program written using EARL's API. As can be observed from the figure, the implementation allows for a generic *user_job* to take advantage of our early approximation library.

```
public static void main(String[] args) throws Exception {
  // Initialization of local variables ...
  Sampler s = new Sampler();

  while (error > sigma) {
    s.Init(path_string); // path_string is the initial DataSet
    // num_resamples of resamples of size sample_size is generated.
    // Both of these variables are determined empirically.
    s.GenerateSamples(sample_size, num_resamples);
    JobConf aes_job = new JobConf(AES.class);
    JobConf user_job;

    // For each sample we execute user_job
    for (int i = 0; i < num_resamples; i++) {
      user_job = new JobConf(MeanBootstrap.class); // Init ...
      JobClient.runJob(user_job);
    }

    // AES uses the input from user_job to compute
    // the approximation error.
    // Init of the aes_job ...
    // The aes job also updates the err.
    JobClient.runJob(aes_job);
    // In cases where early approximation is not possible,
    // sample_size and num_resamples will be set to N and 1,
    // respectively
    UpdateSampleSizeAndNumResamples();
  }
}
```

Figure 6.4: An example of how a user_job would work with the EARL framework

The biggest implementation challenge with *EARL* was reducing the overhead of the *AES* phase and of the sample generation phase. If implemented naively, (i.e., making both the sampler and the *AES* phase its separate job) then the execution time would be inferior to that of the standard Hadoop especially for small data-sets and light aggregates where *EARL*'s early approximation framework has little impact to begin with. We wanted to make *EARL* light-weight so that even for light tasks, *EARL* would not add additional overhead and the execution time in the worst case would be comparable to that of the standard Hadoop.

The potential overhead of our system is due to three factors: (1) creating a new MR job for each iteration used for sample size expansion (2) generating a sample of the original dataset and (3) creating numerous resamples to compute a result distribution that will be used for error estimation. The first overhead factor is addressed with the help of pipelining, similar to that of Hadoop Online, however in our case the mappers also communicate with reducers to receive events that signal sample size expansion or termination. With the help of pipelining and efficient inter task communication, we are able to reuse Hadoop tasks while refining the sample size. The second challenge is addressed via the added feature of the mappers to directly ask for more splits to be assigned, in the case of pre-map sampling, when a larger sample size is required. Alternatively a sample can be generated using post-map sampling as discussed in Section 6.2.3. Post and pre-map sampling work flawlessly with the Hadoop infrastructure to deliver high quality random sample of the input data. Finally the last challenge is addressed via a resampling algorithm and its optimizations presented in Section 6.2.3. Re-sampling is actually implemented within a reduce phase, to minimize any overhead due to job restarts. Due to delta maintenance, introduced in Section 6.3, resampling becomes efficient and its overhead is tremendously decreased making our approach not only feasible but to deliver an impressive speed-up over standard Hadoop. Next key experiments are presented which showcase the performance of *EARL*.

6.5 Experiments

This section is aimed at testing the efficiency of our approach. A set of experiments measuring the efficiency of the accuracy estimation and sampling stages are presented in the following sections. To measure the asymptotic behavior of our approach a synthetically generated data-set is used. The synthetic dataset allows us to easily validate the accuracy measure produced by EARL. More experiments with advanced data-minng algorithms in real-world workflows is currently under way. The normalized error used for the approximate early answer delivery is 0.05 (i.e. our results are accurate to within 5% of the true answer). The experiments were executed on simple, single phase MR tasks to give concrete evidence of applicability of EARL, and more elaborate experiments on a wider range of mining algorithms is part of our future work.

6.5.1 A Strong Case for EARL

In this experiment, we implemented a simple MR task computing the mean, and tested it on both standard Hadoop and EARL. Figure 6.5(a) shows the performance comparison between these two. It shows that when the data-set size is relatively large ($> 100GB$), our solution provides an impressive performance gain (4x speed-up) over standard Hadoop even for a simple function such as the mean. In the worst case scenario, where our framework cannot provide early approximate results ($< 1GB$), our platform intelligently switch back to the original work flow which runs on the entire data-set without incurring a big overhead. It demonstrates clearly that EARL greatly outperforms the standard Hadoop implementation even for light-weight functions. Figure 6.5(a) also shows that a standard Hadoop data loading approach is much less efficient than the proposed pre-map sampling technique.

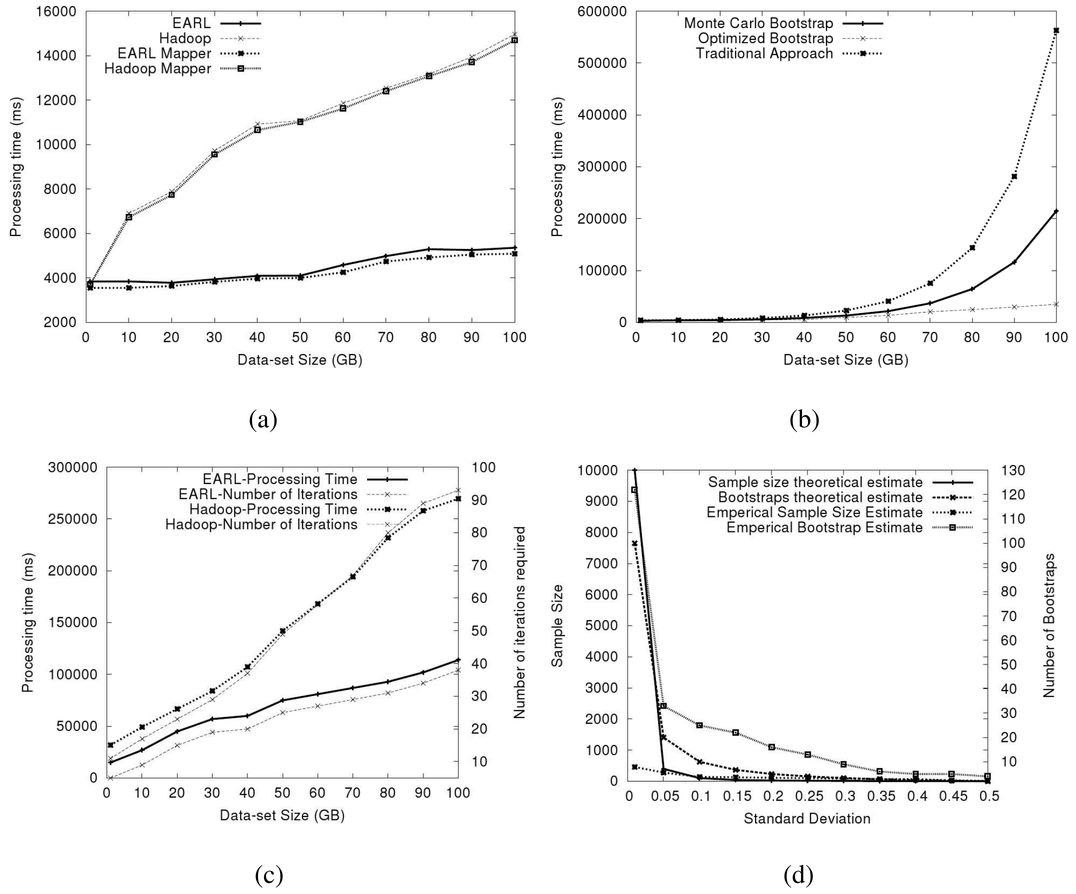


Figure 6.5: (a) Computation of average using EARL and stock Hadoop, (b) Computation of median using EARL and stock Hadoop, (c) Computation of K-Means using EARL, (d) Empirical sample size and number of bootstraps estimates vs. a theoretical prediction

6.5.2 Approximate Median Computation

In this experiment, we did a break-down study, to measure how much a user defined MR task can benefit from resampling techniques and our optimization techniques. We used the computation of a median as an example, and tested it using three different implementations: (1) standard Hadoop, (2) original resampling algorithm, and (3) optimized resampling algorithm. Figure 6.5(b) shows that: (1) With a naïve Monte Carlo

bootstrap, we can provide a reliable estimate for median with a 3-fold speed-up, compared to the standard Hadoop, due to a much smaller sample size requirement. (2) Our optimized algorithm provides another 4x speed-up over the original resampling algorithm.

6.5.3 EARL and Advanced Mining Algorithms

EARL can be used to provide early approximation for advanced mining algorithms, and this experiment provides a performance study when using *EARL* to approximate *K-Means*.

It is well known that *K-Means* algorithm converges to a local optima and is also sensitive to the initial centroids. For these reasons the algorithm is typically restarted from many initial positions. There are various techniques used to speed up K-Means, including parallelization [ZMH09]. Our approach, compliments previous techniques by speeding up K-Means without changing the underlying algorithm.

Figure 6.5(c) shows the results of running *K-Means* with EARL and stock Hadoop. Our approach leads to a speed up due to two reasons: (1) K-Means is executed over a small sample of the original data and (2) K-Means converges more quickly for smaller data-sets. Because of a synthetic data-set, we were also able to validate that EARL finds centroids that are within 5% of the optimal.

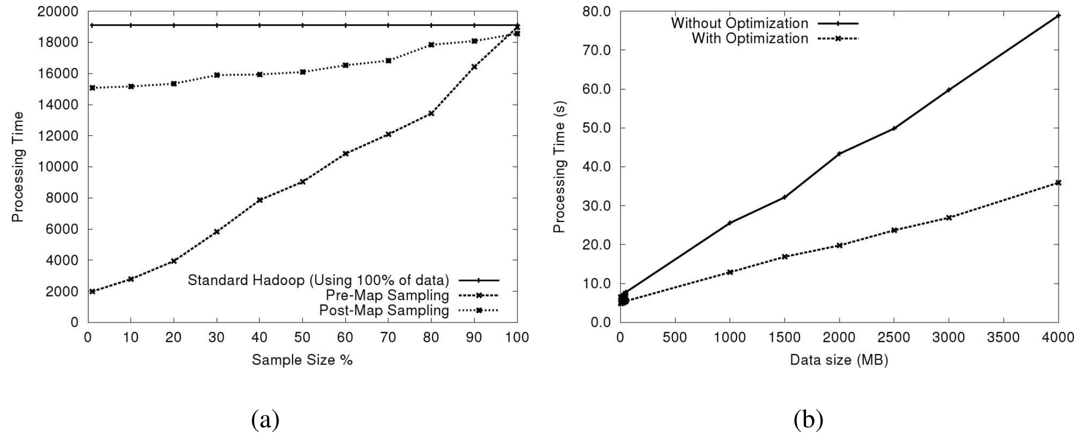
6.5.4 Sample Size and Number of Bootstraps

In this experiment we measure how the theoretical sample size and the theoretical number of bootstraps prediction compare to our empirical technique of estimating the sample size and the number of bootstraps. We use a sample mean as the function of interest. Frequently, theoretical prediction for sample size is over estimated given a low

error tolerance and is under-estimated for a relatively high error tolerance. Furthermore, theoretical bootstrap prediction frequently under-estimates the required number of bootstraps. In other empirical tests we have observed cases where theoretical bootstrap prediction is much higher than the practical requirement. This makes a clear case for the necessity of an empirical way to determine the required sample size and the number of bootstraps to deliver the user-desired error bound. In the case of the sample mean, we found that for a 5% error threshold, a 1% uniform sample and 30 bootstraps are required.

6.5.5 Pre-map and Post-map Sampling

In this experiment we determine the efficiency of pre-map and post-map sampling as described in Section 6.2.3 when applied to computation of the mean. Recall that pre-map sampling is done before sending any input to the mapper thus significantly decreasing the load-times and improving response time. The down-side of pre-map sampler is a potential decrease in accuracy of estimating the number of *key, value* pairs which may be required for correcting the final output. In post-map sampling, the sampling is done per-key, which increases the load-times but potentially improves accuracy of estimating the number of *key, value* pairs. As presented in Figure 6.6(a) the pre-map sampling is faster than post-map sampling in terms of total processing time. Furthermore, our empirical evidence suggests that for a large sample size, pre-map sampler is as accurate in terms of the number of *key, value* prediction as the post-map sampler. Therefore, to decrease the load-times, and to produce a reasonable estimate for functions that require result correction, the *pre-map* sampler should be used. The *post-map* sampler should be used when load-times are of low concern and a fast as well as accurate estimates of a function on a relatively small sample size are required.



(a) (b)

Figure 6.6: (a) Processing times of pre-map and post-map sampling, and (b) Processing time with the update procedure

6.5.6 Update Overhead

This experiment measures the benefit that our incremental processing strategies (inter-iteration and intra-iteration) achieve. Recall that in order to produce samples of larger sizes and perform resampling efficiently, we rely on delta maintenance as described in Section 6.3. Figure 6.6(b) shows the total processing time of computing the mean function with and without the delta maintenance optimization. The data-size represents the total data that the function was to process. The without optimization strategy refers to executing the function of interest on the entire dataset and with optimization strategy refers to execution the function on half of the data and merging the results with the previously saved state as described in Section 6.6(b). The optimized strategy clearly outperforms the non-optimized version. The optimized strategy introduced achieved a speedup of close to 300% for processing a 4GB data-set as compared to the standard method.

6.6 Related Work

Sampling techniques for Hadoop were studied previously in [GC12] where authors introduce an approach of providing Hadoop job with an incrementally larger sample size. The authors propose an *Input Provider* which provides the *JobClient* with the initial subset of the input splits. The subset of splits used for input are chosen randomly. More splits are provided as input until the JobTracker declares that enough data had been processed as indicated by the required sample size. The claim that the resulting input represents a uniformly random sample of the original input is not well validated. Furthermore the assumption that each of the initial splits represents a random sample of the data, which would justify the claim that the overall resulting sample is a uniform sample of the original data, is not well justified. Finally the authors do not provide an error estimation framework which would make it useful for the case where only a small sample of the original data is used. Overall, however, [GC12] provides a very nice overview of a useful sampling framework for Hadoop that is simple to understand and use.

Random sampling over database files is closely related to random sampling over HDFS and the authors in [OR90] provide a very nice summary of various file sampling techniques. The technique discussed in [OR90] that closely resembles our sampling approach is known as a *2-file technique combined with an ARHASH method*. In the method, a set of blocks, F_1, are put into main memory, and the rest of the blocks, F_2, reside on disk. When seeking a random sample, a 2-stage sampling process is performed where F_1 or F_2 is first picked randomly, and then depending on the choice, a random sample is drawn from memory or from disk. The expected number of disk seeks under this approach is clearly much less than if only the disk was used for random sampling. The method described, however, is not directly applicable to our environment and therefore must be extended to support a distributed filesystem.

Authors in [CDS04, DBS06] explore another efficient sampling approach, termed *block sampling*. Block-sampling suffers, however, from a problem that it no longer is a uniform sample of the data. The approximation error derived from a block-level sampling depends on the layout of the data on disk (i.e., the way that the tuples are arranged into blocks). When the layout is random, then the block-sample will be just as good as a uniform tuple sample, however if there is a statistical dependence between the values in a block (e.g., if the data is clustered on some attribute), the resulting statistic will be inaccurate when compared to that constructed from a uniform-random sample. In practice most data layouts fall somewhere between the clustered and the random versions [CDS04]. Authors in [CDS04] present a solution to this problem where the number of blocks to include in a sample, are varied to achieve a uniformly random distribution.

Approximation techniques for data-mining were also extensively studied, however the approximation techniques introduced are not general (e.g., specific to association rule mining [CHS02]). The authors in [CHS02] propose a two phase sampling approach, termed *FAST*, where a large sample S is first taken and then a smaller sample is derived from S. This tailored sampling technique works well, giving speedups up to a factor of 10. Although the proposed approach is highly specialized, *EARL* can still obtain comparable results.

Hellerstein et al. [CCA10] presented a framework for *Online Aggregation* which is able to return early approximate results when querying aggregates on large stored data sets [HHW97b]. This work is based on relational database systems, and is limited to simple single aggregations, which restricts it to `AVG`, `VAR`, and `STDDEV`. Work in [PBJ11b] provides online support for large map-reduce jobs but is again limited to simple aggregates. Similarly, work by [PJ05b] provides an approximation technique using bootstrapping, however the optimization presented was only studied in the con-

text of simple aggregates. Later, B. Li et. al [LMD11] and Condie et al. [CCA10] built systems on top of MapReduce to support continuous query answering. However, these systems do not provide estimation of the accuracy of the result.

EARL can potentially plug into other massively parallel systems such as Hyracks [BCG11]. Hyracks is a new partitioned-parallel software platform that is roughly in the same space as the Hadoop open source platform. Hyracks is designed to run data-intensive computations on large shared-nothing clusters of computers. Hyracks allows users to express a computation as a DAG of data operators and connectors. A Possible future direction would be to use *EARL* on Hyracks' DAG.

EARL relies on dynamic input support in order to incrementally increase sample size as required. While Hadoop extensions such as HaLoop [BHB10] can support incremental data processing, this support is mainly aimed at the iterative execution model of data-mining algorithms (e.g., K-Means). HaLoop dramatically improves the iterative job execution efficiency by making the task scheduler loop-aware and by adding various caching mechanisms. However, due to its batch-oriented overhead, HaLoop, is not suitable for tasks that require dynamically expanding input.

6.7 Summary of EARL

A key part of big data analytics is the need to collect, maintain and analyze enormous amounts of data efficiently. With such application needs, frameworks like MapReduce are used for processing large data-sets using a cluster of machines. Current systems however are not able to provide accurate estimates of incremental results. In this chapter we presented *EARL*, a non-parametric extension of Hadoop that delivers early results with a reliable accuracy estimates. Our approach can be applied to improve efficiency of fault-tolerance techniques by avoiding the restart of failed nodes if the de-

sired accuracy is reached. Our approach builds on resampling techniques from statistics. To further improve the performance, EARL supports various optimizations to the resampling methods which makes the framework even more attractive. The chapter also introduced sampling techniques that are suitable for a distributed file-system. Experimental results suggest that impressive speed-ups can be achieved for a broad range of applications. The experimental results also indicate that this is a promising method for providing the interactive support that many users seek when working with large data-sets.

CHAPTER 7

The Analytical Bootstrap: a New Method for Fast Error Estimation in Approximate Query Processing

Data-driven activities in business, science, and engineering are rapidly growing in terms of both data size and significance. This situation has brought even more attention to the already-active area of Approximate Query Processing (AQP), and in particular to sampling approaches as a critical and general technique for coping with the ever-growing size of big data. Sampling techniques are widely used in databases [AGP99b, BCD03, CDN07, HHW97a, JAP07, OBE09], stream processors [BDM04, MZ10], and even Map-Reduce systems [AMP13, LZZ12].

This most commonly used technique consists in evaluating the queries on a small random *sample* of the original database. Of course, the approximate query answers obtained this way are of very limited utility unless they are accompanied by some *accuracy guarantees*. For instance, in estimating income from a small sample of the population, a statistician seeks assurance that the derived answer falls within a certain interval of the exact answer computed on the whole population with high confidence (e.g., within $\pm 1\%$ of the correct answer with probability $\geq 95\%$). This enables the users to decide whether the current approximation is "good enough" for their purpose. Thus, assessing the quality (i.e., error estimation) of approximate answers is a fundamental aspect of AQP.

The extensive work on error estimation in the past two decades can be categorized

into two main approaches. The first approach [AMP13, CCM00, CDN07, HHW97a, HSS09, JAP07, JJ09, PBJ11a, WOT10] analytically derives closed-form error estimates for common aggregate functions in a database, such as SUM, AVG, etc. Although computationally appealing, analytic error quantification is restricted to a very limited set of queries. Thus, for every new type of queries, a new closed-form formula must be derived. This derivation is a manual process that is ad-hoc and often impractical for complex queries [PJ05a].[1]

To address this problem, a second approach, called *bootstrap*, has emerged as a more general method for routine estimation of errors [KTA13, LZZ12, PJ05a]. Bootstrap [ET93] is essentially a Monte Carlo procedure, which for a given initial sample, (i) repeatedly forms simulated datasets by resampling tuples i.i.d. (identically and independently) from the given sample, (ii) recomputes the query on each of the simulated datasets, and (iii) assesses the answer quality on the basis of the empirical distribution of the produced query answers. The wide applicability and automaticity of bootstrap is confirmed both in theory [BF81, VW00] and practice [KTA13, LZZ12, PJ05a]. Unfortunately, bootstrap tends to suffer from its high computational overhead, since hundreds or even thousands of bootstrap trials are typically needed to obtain reliable estimates [KTA13, PJ05a].

In this chapter, we introduce a new technique, called the *Analytical Bootstrap Method* (ABM), which is both computationally efficient and automatically applicable to a large class of SQL queries, and thus combines the benefits of these two approaches. Thus, our main contribution is *a probabilistic relational model for the bootstrap process that allows for automatic error quantification of a large class of SQL queries (defined in Section 7.1.2) under sampling, but without performing the actual Monte Carlo simulation.* We show that our error estimates are provably equivalent to those

[1] This is evidenced by the difficulties faced by previous analytic approaches in supporting approximation for queries that are more complex than simple group-by aggregate queries.

produced by the simulation-based bootstrap (Theorems 17 and 22).

The basic idea of ABM is to annotate each tuple of the sampled database with an integer-valued random variable that represents the possible multiplicities with which this tuple would appear in the simulated datasets generated by bootstrap. This small annotated database is called a *probabilistic multiset database* (PMDB). This PMDB succinctly models all possible simulated datasets that could be generated by bootstrap trials. Then, we extend relational operators to manipulate these random variables. Thus executing the query on the PMDB generates an annotated relation which encodes the distribution of all possible answers that would be produced *if* we actually performed bootstrap on the sampled database. In particular, using a single-round query evaluation, ABM accurately estimates the empirical distribution of the query answers that would be produced by hundreds or thousands of bootstrap trials.[2]

We have evaluated ABM through extensive experiments on the TPC-H benchmark and on the actual queries and datasets used by leading customers of Vertica Inc. [ver]. Our results show that ABM is an accurate prediction of the simulation-based bootstrap. Additionally, it is 3–4 orders of magnitude faster than the state-of-the-art parallel implementations of bootstrap [KTS12].

Therefore, ABM promises to be a technique of considerable practical significance: The immediate implication of this new technique is that, the quality assessment module of any AQP system that currently relies on bootstrap (e.g., [KTA13, LZZ12, PJ05a]) can now be replaced by a process that uses 3–4 orders of magnitude fewer resources (resp. lower latency) if it currently uses a parallel (resp. sequential) implementation of bootstrap. We envision that by removing bootstrap's computational overhead,

[2]Note that we do not claim to provide low approximation error for arbitrary queries. For instance, random sampling is known to be futile for certain queries (e.g., joins, `MIN`, `MAX`). However, we guarantee the same empirical distribution that would be produced by thousands of bootstrap trials (which is the state-of-the-art error quantification technique for AQP of complex SQL analytics), whether or not sampling leads to low-error or unbiased answers.

ABM would also significantly broaden the application of AQP to areas which require *interactive* and *complex* analytics expressible in SQL, such as root cause analysis and A/B testing [AMP13], real-time data mining, and exploratory data analytics.

The chapter is organized as follows. Section 7.1 exemplifies bootstrap and the query evaluation problem considered in this chapter. Section 7.2 provides the necessary theoretical background. In Section 7.3, we present our probabilistic multiset relational model. We explain our efficient query evaluation technique in Sections 7.4 and 7.5. In Section 7.6, we discuss several extensions of our technique. We report the experimental study in Section 7.7, followed by the related work in Section 7.9. We conclude in Section 7.10.

7.1 Problem Statement

In this section, we formally state the problem addressed by this chapter. We then provide a small example of bootstrap in Section 7.1.1, and a high-level overview of our approach in Section 7.1.2.

Given a database $\mathfrak{D}$ and a query $\mathfrak{q}$, let $\mathfrak{q}(\mathfrak{D})$ denote the exact answer of evaluating $\mathfrak{q}$ on $\mathfrak{D}$. An approximate answer can be obtained by (i) extracting from $\mathfrak{D}$ a random sample D, (ii) evaluating a potentially modified version of $\mathfrak{q}$ (say q) on D, and (iii) using $q(D)$ as an approximation of $\mathfrak{q}(\mathfrak{D})$. In some cases, such as AVG, q is the same as $\mathfrak{q}$, but in other cases, q can be a modified query that produces *better* results. For instance, when evaluating SUM on a sample of size $\frac{1}{f}|\mathfrak{D}|$, one will choose $q = f\mathfrak{q}$ to scale up the sample sum by a factor of f. The particular selection of q for a given $\mathfrak{q}$ is outside the scope of our discussion.[3] Instead, we focus on estimating the quality of $q(D)$ for a given q, as described next.

[3]Similar to the original bootstrap [KTA13, LZZ12, PJ05a], we assume that q is given. Deriving q for a given $\mathfrak{q}$ has been discussed in [MZ10].

Let $D_1, \cdots, D_N$ be all possible sample instantiations of $\mathfrak{D}$. Then, $q(D)$ could be any of the $\{q(D_i), i = 1, \cdots, N\}$ values. Therefore, to assess the quality of $q(D)$, one needs to (conceptually) consider all possible query answers $\{q(D_i)\}$, and then compute some user-specified measure of quality, denoted by $\xi(\mathfrak{q}(\mathfrak{D}), \{q(D_i)\})$. For instance, when approximating an `AVG` query, ξ could be the variance, or the 99% confidence interval for the $\{q(D_i)\}$ values. However, since computing $\xi(\mathfrak{q}(\mathfrak{D}), \{q(D_i)\})$ directly is typically infeasible, a technique called *bootstrap* is often used to approximate this value.

Bootstrap. Bootstrap [ET93] is a powerful technique for approximating unknown distributions. Bootstrap consists in a simple Monte Carlo procedure: it repeatedly carries out a sub-routine, called a *trial*. Each trial generates a simulated database, say $\hat{D}_j$, which is the same size as D (by sampling $|D|$ tuples i.i.d. from D *with replacement*), and then computes query q on $\hat{D}_j$. The collection $\{q(\hat{D}_j)\}$ from all the bootstrap trials forms an empirical distribution, based on which $\xi(q(D), \{q(\hat{D}_j)\})$ is computed and returned as an approximation of $\xi(\mathfrak{q}(\mathfrak{D}), \{q(D_i)\})$. Bootstrap is effective and robust across a wide range of practical situations [KTA13, LZZ12, PJ05a].

Problem Statement. Our goal is to devise an efficient algorithm for computing the empirical distribution $\{q(\hat{D}_j)\}$ produced by bootstrap, but *without* executing the actual bootstrap trials. In particular, we are interested in the marginal distribution of each individual result tuple, i.e., the probability of each tuple appearing in any $q(\hat{D}_j)$.[4] This marginal distribution enables us to compute the commonly used quality measures ξ (e.g., mean, variance, standard deviation, and quantiles as used in [KTA13, LZZ12, PJ05a]). Next, we demonstrate this in an example.

[4]The set of all possible query answers, $\{q(\hat{D}_j)\}$, may have up to $O(2^{|D|})$ elements. Thus, due to its extremely large size, returning this set to the user is impractical.

7.1.1 An Example of Bootstrap

Bootstrapping a Database. A single bootstrap trial on a relation R produces a multiset relation, since a tuple may be drawn more than once. Thus, different trials could result in different multiset relations. This set of multiset relations can be modeled as a single *probabilistic* multiset relation, where each tuple has a random multiplicity. Let R^r denote the probabilistic relation that results from bootstrapping R. Likewise, let D^r denote the probabilistic database obtained by bootstrapping the relations of a database sample D. For instance, consider D in Figure 7.1(a), which has a single relation R (named *stock*) containing three tuples. Figure 7.1(b) shows a possible instance $\hat{R}$ of R^r, produced by a single bootstrap trial, where tuple t_2 and t_3 are drawn twice and once, respectively, while t_1 is not selected. For brevity, we denote this resample as $\{(t_2, 2), (t_3, 1)\}$.

stock

$R =$	**Part**	**Type**	**Qty**
t_1	p01	a	4
t_2	p02	b	5
t_3	p03	a	3

(a)

stock

$\hat{R} =$	**Part**	**Type**	**Qty**
t_2	p02	b	5
t_2	p02	b	5
t_3	p03	a	3

(b)

Figure 7.1: (a) An example of a database sample D with one relation R (named *stock*), and (b) a resample instance of R^r

Bootstrapping this particular database sample D generates 10 possible multiset relations. We refer to these as the *possible multiset worlds* of D^r, denoted as $pmw(D^r)$. Figure 7.2 shows all the ten possible instances with their probabilities of being generated.

Queries on the Resampled Database. Consider the “Important Stock Types” query q in Example 32, which finds the stock types having a quantity $> 30\%$ of the total

multiset world	prob.
$D_1 = \{(t_1, 1), (t_2, 1), (t_3, 1)\}$	$2/9$
$D_2 = \{(t_1, 2), (t_2, 1)\}$	$1/9$
$D_3 = \{(t_1, 2), (t_3, 1)\}$	$1/9$
$D_4 = \{(t_2, 2), (t_1, 1)\}$	$1/9$
$D_5 = \{(t_2, 2), (t_3, 1)\}$	$1/9$
$D_6 = \{(t_3, 2), (t_1, 1)\}$	$1/9$
$D_7 = \{(t_3, 2), (t_2, 1)\}$	$1/9$
$D_8 = \{(t_1, 3)\}$	$1/27$
$D_9 = \{(t_2, 3)\}$	$1/27$
$D_{10} = \{(t_3, 3)\}$	$1/27$

Figure 7.2: The possible multiset worlds of D^r

quantity. To answer q on D^r, one needs to evaluate q against each world in $pmw(D^r)$, and add up the probabilities of all the worlds returning the same answer. For instance, since $q(D_1) = q(D_2) = \{t_a, t_b\}$ (as in Figure 7.3(a)), then $\Pr(\{t_a, t_b\}) = \frac{2}{9} + \frac{1}{9} = \frac{1}{3}$. All possible answers from evaluating q on D^r, denoted by $q(D^r)$, are shown in Figure 7.3(b).

Example 32 (Important Stock Types)**.**

```
SELECT distinct Type FROM stock
WHERE Qty > 0.3 * (SELECT SUM(Qty) FROM stock)
```

In general, when R has n tuples, $q(D^r)$ can have up to $O(2^n)$ possible answers, which makes it impractical to deliver the distribution on all possible answers. Instead, for each possible tuple in the query result (t_a and t_b in this case) we compute its marginal probability of appearing in the query result, as shown in Figure 7.3(c). For example, t_a may appear in the results $\{t_a, t_b\}$ or $\{t_a\}$. Thus, the marginal probability of t_a appearing in any result is $\frac{1}{3} + \frac{8}{27} = \frac{17}{27}$. We denote this marginal distribution as $q^{mrg}(D^r)$.

	Type
t_a	a
t_b	b

(a)

answer	prob.
$\{t_a, t_b\}$	$1/3$
$\{t_a\}$	$8/27$
$\{t_b\}$	$10/27$

(b)

answer	prob.
t_a	$17/27$
t_b	$19/27$

(c)

Figure 7.3: (a) The result of $q(D_1)$ and $q(D_2)$, (b) all possible answers of q, and (c) their marginal probabilities

In real life, instead of our three-tuple example, we have tables with millions or billions of tuples, where enumerating all possible worlds is impossible. Thus, bootstrap uses Monte Carlo simulation to approximate the above process, which requires hundreds or thousands of trials to achieve an accurate estimate for $q^{mrg}(D^r)$.

With $q^{mrg}(D^r)$, we can now measure the quality of $q(D)$. For instance, the average false negative rate of tuples in $q(D)$, i.e., the average probability of missing t_a or t_b, can be computed by $\frac{1}{2}\{(1 - \Pr(t_a)) + (1 - \Pr(t_b))\} = \frac{1}{3}$.

7.1.2 Scope of Our Approach

In this chapter, we propose a new approach, called the Analytical Bootstrap Method (ABM), which avoids the computational overhead of bootstrap. We study the set of conjunctive queries with aggregates, i.e., queries expressed in the following relational algebra: σ (selection), Π (projection), $\bowtie$ (join), δ (deduplication), and γ (aggregate). $\gamma_{A,\alpha(B)}$ denotes applying aggregate α on B grouped by A. We focus on queries without nested aggregates, i.e., B is not the output of another aggregate. To simplify presentation, we first focus on aggregates `SUM`, `COUNT`, and `AVG`, and defer the extension of ABM to more general aggregates to Section 7.6. Since Bootstrap only works for "smooth" queries,[5] we also restrict ourselves to such queries. In particular, any of

[5] A discussion of smoothness can be found in [ET93, PJ05a].

the following constructs can easily lead to non-smooth queries: (1) extrema aggregates (MIN/MAX) and (2) equality checks on aggregate results (e.g., projection/group-by/equi-join on aggregate results). Thus, we only discuss queries without these constructs.

Among the studied queries, we identify *eligible queries*, for which we provide an efficient extensional evaluation. Furthermore, our query evaluation techniques enjoy a DBPTIME complexity for a large subset of eligible queries (see Section 7.4.2).

To provide an intuitive sense of how general/restrictive these classes of queries are in practice, in Table 7.1 we summarize the syntactic constraints imposed by different error estimation techniques along with some statistics. Table 7.1 compares the set of queries supported by our formal semantics, our ABM (eligible queries and those with DBPTIME complexity), previous analytical techniques (which are strictly subsumed by ABM), and bootstrap (which strictly subsumes ABM). We have analyzed 22 TPC-H benchmark queries, as well as a real-life query log from Conviva Inc.[con] consisting of 6660 queries.[6] Table 7.1 reports the fraction of queries from TPC-H and the Conviva log that is supported by different techniques, i.e., queries that satisfy the constraints imposed by each technique.

The set of queries supported by ABM strictly subsumes those of previous analytical approaches, and it also constitutes a majority subset of those supported by bootstrap. For instance, ABM supports $19/22$ TPC-H queries, and 98.6% of the Conviva ones, covering most of the queries supported by bootstrap (i.e., $19/22$ in TPC-H and 99.1% in the Conviva log). Previous analytical approaches only support $9/22$ of TPC-H and 36.9% of the Conviva queries. Also, note that 81.0% of the Conviva queries are supported by ABM while enjoying a guaranteed DBPTIME complexity.

[6] Due to proprietary reasons, we had restricted access to Conviva queries; we were only allowed to run a customized parser on the query log to compute the breakdown of different query types. Thus, for performance evaluations, we use Vertica and TPC-H queries.

Note that User Defined Aggregate Functions (UDAFs) can only be handled by (simulation-based) bootstrap. Fortunately, UDAFs are quite rare in day-to-day data analysis, as most users find it more convenient to write pure SQL queries. For instance, *none* of the 6660 queries in the Conviva log contained a UDAF. [7]

7.2 Background

In this section, we provide background on semirings and their connection with relational operators, followed by a brief overview of semiring random variables. These concepts will be used in our probabilistic relational model and query evaluation technique.

7.2.1 Semirings and Relational Operators

Here, we provide an overview on semirings as well as how they can be used to compute queries on a database.

A **monoid** is a triplet $(S, +, 0)$, where:

- S is a set closed under $+$, i.e., $\forall s_1, s_2 \in S$, $s_1 + s_2 \in S$
- $+$ is an associative binary operator, i.e., $\forall s_1, s_2, s_3 \in S$, $(s_1 + s_2) + s_3 = s_1 + (s_2 + s_3)$
- 0 is the identity element of $+$, i.e., $\forall s \in S$,$s + 0 = 0 + s = s$

For example, $(\mathbb{N}, +, 0)$ is a monoid, where $\mathbb{N}$ is the set of natural numbers, and 0 and $+$ are the numerical zero and addition.

[7] UDAFs should not be confused with User Defined Functions (UDFs) which are quite common in practice (e.g., 42% of Conviva queries contain UDFs); UDFs operate on a single tuple and return a single value, while UDAFs operate on multiple tuples and return a single value. UDFs are usually used to transform or extract data from each tuple. By treating UDFs as extra columns in the table, they can be easily supported by both analytic approach and ABM.

Technique	**Constraints**	**in TPCH**	**in Conviva Log**
Analytic Approach	Simple group-by aggregate (no `MIN`/`MAX`) queries [AMP13, CCM00, CDN07, HHW97a, HSS09, JAP07, JJ09, PBJ11a, WOT10]	9/22	36.9%
ABM Semantics	Conjunctive queries with (a) no nested aggregates, (b) no `MIN`/`MAX` and (c) no equality checks on aggregate results	19/22	99.1%
ABM Eligible	Queries that (a) satisfy all ABM semantics constraints, and (b) have an eligible query plan (see Definition 24)	19/22	98.6%
ABM Eligible with DBP-TIME Query Evaluation	ABM eligible queries that (a) have DBPTIME-eligible plans or (b) can be optimized by the containment join optimization (see Section 7.4.2)	15/22	81.0%
Bootstrap	"Smooth" queries	19/22	99.1%

Table 7.1: Classes of SQL queries supported by different techniques, and their coverage of TPC-H and Conviva queries

A **semiring** is a quintuplet $(S, +, \cdot, 0, 1)$ which follows the four axioms below:

- $(S, +, 0)$ is a monoid where $+$ is commutative, i.e., $\forall s_1, s_2 \in S$, $s_1 + s_2 = s_2 + s_1$ (a.k.a. a **commutative monoid**)
- $(S, \cdot, 1)$ is a monoid
- $\cdot$ is left and right distributive over $+$, i.e., $\forall s_1, s_2, s_3 \in S$, $s_1 \cdot (s_2 + s_3) = s_1 \cdot s_2 + s_1 \cdot s_3$ and $(s_2 + s_3) \cdot s_1 = s_2 \cdot s_1 + s_3 \cdot s_1$
- 0 annihilates S, i.e. $\forall s \in S$, $0 \cdot s = s \cdot 0 = 0$

A **commutative semiring** is a semiring in which $(S, \cdot, 1)$ is also a commutative monoid. In this chapter, all semirings are commutative semirings. E.g., $S_{\mathbb{N}} = (\mathbb{N}, +, \cdot, 0, 1)$ is a commutative semiring.

Green et al. [GKT07] showed that many different extensions of relational algebra can be formulated by annotating database tuples with semiring elements and propagating these annotations during query processing. Consider a multiset database as an example. Tuples in a multiset relation are annotated with natural numbers $\mathbb{N}$, representing their multiplicities in the database. Formally, a multiset relation R with the tuple domain U is described by an annotation function $\pi_R : U \to \mathbb{N}$, where a tuple $t \in R \Leftrightarrow \pi_R(t) \neq 0$.

During query processing, the relational algebra is extended with the $+$ and $\cdot$ operators in semiring $S_{\mathbb{N}}$, which manipulate the annotated multiplicities: for projection, we add the multiplicities of all input tuples that are projected to the same result tuple, while for join, we multiply the multiplicities of the joined tuples. That is, inductively we have:

- **Selection** $\sigma_c(R)$. $\pi_{\sigma_c(R)}(t) = \pi_R(t) \cdot \mathbb{1}(c(t))$, where $\mathbb{1}(c(t))$ returns 1 if $c(t)$ is true and 0 otherwise.
- **Projection** $\Pi_A(R)$. $\pi_{\Pi_A(R)}(t) = \sum_{t'[A]=t} \pi_R(t')$ where $t'[A]$ is the projection of t' on A.

- **Join** $R_1 \bowtie R_2$. $\pi_{R_1 \bowtie R_2}(t) = \pi_{R_1}(t_1) \cdot \pi_{R_2}(t_2)$, where t_i is t on U_i.

Green et al. showed that by annotating tuples with elements from $S_{\mathbb{N}}$ and using the extended relational algebra defined above, we obtain the relational algebra with multiset semantics. However, this extension is not applicable to bootstrap, because the multiset relation generated by bootstrap is probabilistic (shown in Section 7.1.1). Thus, we introduce our *probabilistic multiset relational model* in Section 7.3 by using *semiring random variables* (described next).

7.2.2 Semiring Random Variables

A random variable that takes values from the elements of a semiring S is called a *semiring random variable*, denoted by S-rv. Analogous to operators $+$ and $\cdot$ on semiring elements, there are corresponding operators that operate on semiring random variables, called *convolutions*. Below we define the $+$ convolution (denoted as $\oplus$), and the $\cdot$ convolution (denoted as $\odot$).

Definition 15 (Convolution)**.** *Let r_1 and r_2 be two S-rvs. The convolution $\oplus$ (resp. $\odot$) is a binary operator defined on monoid $(S, +, 0)$ (resp. $(S, \cdot, 0)$), such that $r_1 \oplus r_2$ (resp. $r_1 \odot r_2$) is a S-rv, where $\forall s \in S$,*

$$\Pr(r_1 \oplus r_2 = s) = \sum \{\Pr(r_1 = x \wedge r_2 = y) | \forall x, y \in S, x + y = s\}$$
$$\Pr(r_1 \odot r_2 = s) = \sum \{\Pr(r_1 = x \wedge r_2 = y) | \forall x, y \in S, x \cdot y = s\}$$

Similar to conventional random variables, we define the *disjointness*, *independence* and *entailment* between S-rvs, but with some special treatment for $0 \in S$.

Definition 16. *Let r_1 and r_2 be two S-rvs.*

- *r_1 and r_2 are* ***disjoint****, if* $\Pr(r_1 \neq 0 \wedge r_2 \neq 0) = 0$.

- *r_1 and r_2 are* ***independent****, if* $\forall s_1, s_2 \in S$,

$$\Pr(r_1 = s_1 \wedge r_2 = s_2) = \Pr(r_1 = s_1) \cdot \Pr(r_2 = s_2)$$

- *r_1* ***entails*** *r_2, if* $\Pr(r_1 \neq 0 | r_2 \neq 0) = 1$.

The marginal distribution of a S-rv r can be represented by a vector $\mathbf{p}^r$ indexed by S, namely $\forall s \in S$, $\mathbf{p}^r[s] = \Pr(r = s)$.

In general, using Definition 15 to compute convolutions of S-rvs can be quite inefficient, especially when the semiring S is large. However, we make the observation that under certain conditions, the convolution of S-rvs can be quickly computed by simply manipulating their probability vectors, as stated next.[8]

Proposition 1 (Efficient Convolution)**.** *Let r_1 and r_2 be two S-rvs on semiring $(S, +, \cdot, 0, 1)$:*

- *If r_1 and r_2 are disjoint, then $\mathbf{p}^{r_1 \oplus r_2} = \mathbf{p}^{r_1} \uplus \mathbf{p}^{r_2}$, where*

$$(\mathbf{p}^{r_1} \uplus \mathbf{p}^{r_2})[s] \stackrel{\text{def}}{=\!=} \begin{cases} \mathbf{p}^{r_1}[s] + \mathbf{p}^{r_2}[s] & \textit{if } s \neq 0 \\ \mathbf{p}^{r_1}[0] + \mathbf{p}^{r_2}[0] - 1 & \textit{if } s = 0 \end{cases}$$

- *If r_1 entails r_2 where r_2 can only take value 0 or 1, then $\mathbf{p}^{r_1 \odot r_2} = \mathbf{p}^{r_1}$.*

7.3 Semantics & Query Evaluation

Next, we introduce our probabilistic multiset relational model, which annotates each tuple with a $S_{\mathbb{N}}$-rv representing its nondeterministic multiplicity and extends the relational algebra to propagate these annotations during query processing. Table 7.2 lists the notations we use.

[8] All the proofs can be found in Section 7.11.

R/D	deterministic relation/database
R^r/D^r	probabilistic relation/database from bootstrap
$\hat{R}$/$\hat{D}$	a resample instance from bootstrap
$m_{\hat{R}}(t)$	multiplicity of t in $\hat{R}$
U	tuple domain of a relation
$head(\cdot)$	set of attributes in the output of a query
$rels(\cdot)$	set of relations occurring in a query
π,ϖ	annotation functions for PMRs
0/**1**	Constant random variables with value 0/1

Table 7.2: Summary of Notations

7.3.1 Formal Semantics

Probabilistic Multiset Database Semantics. A *probabilistic multiset relation* (PMR) is a multiset relation whose tuples have nondeterministic multiplicities. Formally, the functional representation of a PMR R^r is an annotation function $\pi_R : U \rightarrow \{S_{\mathbb{N}}\text{-rv}\}$, where $\pi_R(t) = \pi_t$ when tuple t occurs in R^r with a random multiplicity π_t; otherwise, $\pi_R(t) = \mathbf{0}$.

Specifically, the PMR R^r resulting from bootstrapping R (of size n) is modeled thus: for all tuples $\{t_i | i = 1, \cdots, n\}$ in R, $(\pi(t_1), \cdots, \pi(t_n))$ jointly follow a multinomial distribution $M(n, [\frac{1}{n}, \cdots, \frac{1}{n}])$. We can also model a deterministic relation R as a special PMR, where $\forall t \in R$, $\pi_R(t) = \mathbf{1}$.

Definition 17 (PMR Semantics)**.** *The semantics of a PMR R^r annotated by $\pi_R : U \rightarrow \{S_{\mathbb{N}}\text{-}rv\}$ is a set of possible multiset worlds $pmw(R^r)$, where the probability of each world (i.e., resample instance) $\hat{R}$ is*

$$\Pr(\hat{R}) = \Pr\left(\bigwedge\{\pi_R(t) = m_{\hat{R}}(t) \mid t \in R\}\right)$$

A *probabilistic multiset database* (PMDB) D^r is a database with PMRs.

Definition 18 (PMDB Semantics)**.** *The semantics of D^r which consists of k PMRs $(R_1^r, R_2^r, \cdots, R_k)$ is a set of possible multiset worlds $pmw(D^r)$ defined as:*

$$pmw(D^r) = \{(\hat{R}_1, \hat{R}_2, \cdots, \hat{R}_k) \mid \hat{R}_i \in pmw(R_i^r), \forall i\}$$

The probability of each world $\hat{D}$ is $\Pr(\hat{D}) = \prod_{i=1}^{k} \Pr(\hat{R}_i)$.

Query Semantics. Evaluating a query q on a PMDB D^r is equivalent to evaluating q on every possible multiset world in $pmw(D^r)$.

Definition 19 (Query Semantics)**.** *The semantics of evaluating a query q on a PMDB D^r is a set of possible answers $q^{pmw}(D^r)$, where each possible answer's probability is:*

$$\forall \hat{R}_q \subseteq U_q: \ \Pr(\hat{R}_q) = \sum \{\Pr(\hat{D}) \mid \hat{D} \in pmw(D^r) \wedge q(\hat{D}) = \hat{R}_q\}$$

Since computing $q^{pmw}(D^r)$ is infeasible (see Section 7.1.1), we return a marginal summary $q^{mrg}(D^r)$. i.e., for each tuple $t \in U_q$, we return a probability vector $\mathbf{p}$, where $\mathbf{p}[m]$ is the probability of t appearing in any possible answer m times, i.e.,

$$\mathbf{p}[m] = \sum \{\Pr(\hat{R}_q) | \hat{R}_q \subseteq U_q, \pi_{\hat{R}_q}(t) = m\}$$

7.3.2 Intensional Query Evaluation

A prohibitive number of worlds makes direct application of the possible worlds semantics impractical. Thus, this section introduces our *intensional evaluation* and its semantics, which lay the theoretical foundation of the evaluation technique we introduce in Section 7.4. Intensional evaluation extends relational algebra to propagate the annotations symbolically throughout query execution.

Extending Relational Algebra. Analogous to the multiset semantics introduced in Section 7.2, we extend relational algebra with $\oplus$ and $\odot$ operators from the semiring

$(S_{\mathbb{N}}\text{-rv}, \oplus, \odot, \mathbf{0}, \mathbf{1})$ to manipulate the random multiplicities of each tuple, so that query evaluation using these operators will produce the correct results (with respect to the possible multiset worlds semantics). We define this extended relational algebra as follows. (We first discuss how aggregates manipulate tuple multiplicities, postponing discussion of how aggregates compute values later.)

Definition 20 (Intensional Evaluation)**.** *Intensional evaluation is defined inductively on a query Plan P:*

- *If $P = \sigma_c(P_1)$, then $\pi_P(t) = \pi_{P_1}(t) \odot \mathbb{1}(c(t))$.*
- *If $P = \Pi_A(P_1)$, then $\pi_P(t) = \bigoplus_{t'[A]=t} \pi_{P_1}(t')$.*
- *If $P = P_1 \bowtie P_2$, then $\pi_P(t) = \pi_{P_1}(t_1) \odot \pi_{P_2}(t_2)$, where $t_i = t[head(P_i)]$.*
- *If $P = \delta(P_1)$, then $\pi_P(t) = \mathbb{1}(\pi_{P_1}(t))$.*
- *If $P = \gamma_{A,\alpha(B)}(P_1)$, then $\pi_P(t) = \pi_{\delta(\Pi_A(P_1))}(t)$.*

where $\mathbb{1}(r)$ maps a random variable r to another random variable, i.e. if $r \neq 0$, then $\mathbb{1}(r) = 1$, and $\mathbb{1}(r) = 0$ otherwise. When r is deterministic, $\mathbb{1}(r)$ degenerates to $\mathbf{0}$ or $\mathbf{1}$.

Aggregates. Besides manipulating π, an aggregate produces values absent from the input relations; an extra semiring annotation (denoted as ϖ) is thus required to define its computation.

Next, we define the intensional evaluation of SUM. Since manipulating π is defined in Definition 20, we focus on the manipulation of ϖ. (COUNT is a special case of SUM and AVG can be defined as an arithmetic function of SUM and COUNT.)

Definition 21 (Intensional Evaluation of $\gamma_{A,\text{SUM}(B)}$)**.** *The Intensional evaluation of $\gamma_{A,\text{SUM}(B)}(P)$ is defined as follows:*[9]

[9] *W.l.o.g, in the following discussion, we assume $dom(B) = \mathbb{R}$.*

1. *Annotate* P *with* $\varpi : U \to \{S_{\mathbb{R}}$*-rv*$\}$ *where* $S_{\mathbb{R}} = (\mathbb{R}, +, \cdot, 0, 1)$*, i.e.,* $\varpi(t) = t[B] \odot \pi_P(t)$*, where* $t[B]$ *is treated as a degenerated* $S_{\mathbb{R}}$*-rv taking a constant value.*
2. *Compute* $\text{SUM}(B) = \varpi_{\gamma_{A,\text{SUM}(B)}(P)}(t) = \bigoplus_{t'[A]=t} \varpi_P(t')$*.*

Example 33. *Consider the database* D *in Figure 7.1(a) and the query* q *in Example 32. Figure 7.4 shows the intensional evaluation of* q *on* D^r *using the query plan shown in Figure 7.4(a). Figure 7.4(g) shows the truth table for tuple* t_a*. From this, we can determine the probability of* t_a *appearing in the result as* $\frac{17}{27}$*.*

Intensional evaluation of q on D^r produces a new PMR, denoted by $q^i(D^r)$, whose semantics is defined as follows.

Definition 22 (Intensional Semantics)**.** *The semantics of* $q^i(D^r)$ *is a set of possible multiset worlds,* $pmw(q^i(D^r))$*, where any assignment of the annotations of each tuple* t*, namely* $\pi_q(t)$ *and* $\varpi_q(t)$*, yields a possible world instance* $\hat{R}_q$*, such that:*

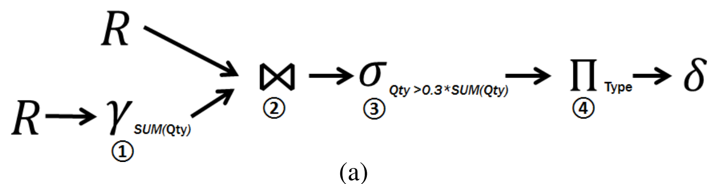
$$\Pr(\hat{R}_q) = \Pr\left(\bigwedge\{\pi_q(t) = m_{\hat{R}_q}(t) \wedge \varpi_q(t) \models t \mid t \in U_q\}\right)$$

Here, $\varpi_q(t) \models t$ *means that* $\varpi_q(t)$ *takes the corresponding aggregate value in* t*.*

Because of the commutative and distributive properties of semirings, $q^i(D^r)$ is independent of the plan chosen for q [GKT07]. Furthermore, this semantics is equivalent to the possible multiset worlds semantics $q^{pmw}(D^r)$, as stated next.

Theorem 17. *We have* $pmw(q^i(D^r)) \equiv q^{pmw}(D^r)$ *for every conjunctive query* q *with aggregates and every PMDB* D^r*, as long as the projected, group-by, and aggregated columns are not the output of another aggregate.*

Theorem 17 proves the correctness of intensional evaluation. However, intensional evaluation is quite inefficient, since in the worst case the size of each annotation can



(a)

tuple	ϖ_{SUM}	π
t_1	$4\pi_1$	π_1
t_2	$5\pi_2$	π_2
t_3	$3\pi_3$	π_3

(b)

ϖ_{SUM}	π
$4\pi_1 \oplus 5\pi_2 \oplus 3\pi_3$	$\mathbb{1}(\pi_1 \oplus \pi_2 \oplus \pi_3)$

(c) Step ① in (a)

tuple	ϖ_{SUM}	π
t_1	$4\pi_1 \oplus 5\pi_2 \oplus 3\pi_3$	$\pi_1 \odot \mathbb{1}(\pi_1 \oplus \pi_2 \oplus \pi_3)$
t_2	$4\pi_1 \oplus 5\pi_2 \oplus 3\pi_3$	$\pi_2 \odot \mathbb{1}(\pi_1 \oplus \pi_2 \oplus \pi_3)$
t_3	$4\pi_1 \oplus 5\pi_2 \oplus 3\pi_3$	$\pi_3 \odot \mathbb{1}(\pi_1 \oplus \pi_2 \oplus \pi_3)$

(d) Step ② in (a)

tuple	π
t_1	$\pi'_1 = \mathbb{1}(4\pi_1 \oplus 5\pi_2 \oplus 3\pi_3 < 13.3) \odot \pi_1 \odot \mathbb{1}(\pi_1 \oplus \pi_2 \oplus \pi_3)$
t_2	$\pi'_2 = \mathbb{1}(4\pi_1 \oplus 5\pi_2 \oplus 3\pi_3 < 16.7) \odot \pi_2 \odot \mathbb{1}(\pi_1 \oplus \pi_2 \oplus \pi_3)$
t_3	$\pi'_3 = \mathbb{1}(4\pi_1 \oplus 5\pi_2 \oplus 3\pi_3 < 10) \odot \pi_3 \odot \mathbb{1}(\pi_1 \oplus \pi_2 \oplus \pi_3)$

(e) Step ③ in (a)

	Type	π
t_a	a	$\pi'_1 \oplus \pi'_3$
t_b	b	π'_2

(f) Step ④ in (a)

π_1	π_2	π_3	prob.
1	1	1	$2/9$
2	1	0	$1/9$
2	0	1	$1/9$
1	0	2	$1/9$
3	0	0	$1/27$
0	0	3	$1/27$

(g)

Figure 7.4: (a) Query plan of Example 32, (b) initial annotation of R^r, (c-d) intensional evaluation steps, and (f) truth table of $\pi_q(t_a) = 1$

grow to the same order of magnitude as the database, and computing the distribution requires enumerating all possible annotation values. Thus, our next section develops an efficient evaluation technique.

7.4 Extensional Query Evaluation

This section presents an *extensional evaluation* technique that increases efficiency over its intensional counterpart by manipulating succinct multinomial representations of the annotations rather than the annotations themselves,

For simplicity's sake, we limit discussion to queries with a single (re)sampled relation and without any self-joins of the (re)sampled relation. (Section 7.6 discusses extensions to general queries.) Next, we introduce our multinomial representation, and then describe extensional evaluation for queries without and with aggregates.

7.4.1 The Multinomial Representation

This observation prompts our proposed multinomial representation of the annotations:

Observation 1. *A bootstrap trial on a relation R of n tuples comprises n i.i.d. experiments. The i-th experiment picks a single tuple at random, hence producing a probabilistic relation R_i^r. Formally, each R_i^r is annotated by $\rho_i : U \to \{S_{\mathbb{N}}\text{-}rv\}$, such that $\rho_i(t) = 1$ when tuple t is selected and $\rho_i(t) = 0$ otherwise. The relation R^r resulting from bootstrap is the union of $\{R_i^r\}$. One can easily verify that $\pi(t) = \bigoplus_{i=1}^{n} \rho_i(t)$.*

Based on Observation 1, we define *atoms* of an annotation $\pi(t)$ as the set of annotations comprising $\pi(t)$, which are generated from each experiment, i.e., $atom(\pi(t)) = \{\rho_i(t) \mid i = 1, \cdots, n\}$. The atoms hold these properties:

- Within an experiment i, $\rho_i(t)$ and $\rho_i(t')$ are disjoint for any two tuples t, t';
- Across different experiments i and j, $\rho_i(\cdot)$ and $\rho_j(\cdot)$ are independent;
- $\{\rho_i(t) \mid i = 1, \cdots, n\}$ are i.i.d. S-rvs with the same marginal probability vector $\mathbf{p}$, where $\mathbf{p}[0,1] = [\frac{n-1}{n}, \frac{1}{n}]$.

These properties allow us to uniquely and succinctly represent $\pi(t)$ by a pair $[n, \mathbf{p}]$, namely a *multinomial representation*, reinterpretable as *the convolution sum of* n *i.i.d. semiring random variables with the probability vector* $\mathbf{p}$. The probability distribution of $\pi(t)$ can be easily reconstructed from its multinomial representation, using the probability mass function of the Multinomial distribution $M(n, \mathbf{p})$.

Moreover, when $atom(\pi(t))$ satisfies certain properties, $\oplus$ and $\odot$ can be directly computed on its multinomial representation, i.e.,

Proposition 2. *Let* $\pi_1 = [n, \mathbf{p}_1]$ *and* $\pi_2 = [n, \mathbf{p}_2]$.

- *If* $atom(\pi_1) \cap atom(\pi_2) = \emptyset$, *then* $\pi_1 \oplus \pi_2 = [n, \mathbf{p}_1 \uplus \mathbf{p}_2]$.
- *If* $atom(\pi_1) \subseteq atom(\pi_2)$, *then* $\pi_1 \odot \mathbb{1}(\pi_2) = [n, \mathbf{p}_1]$.

Next, we show how our multinomial representation and its properties enable us to devise the extensional evaluation.

7.4.2 Queries without Aggregates

This section first formally defines the set of queries *eligible* for efficient evaluation, then presents our extensional evaluation.

Preliminaries. Let us denote the sampled relation as R^f. To simplify discussion, we base it on *canonical query plans*, obtainable by repeating the procedure below on an arbitrary query plan (using relational algebra's rewriting rules [GUW09]):

1. Distinguish different occurrences of R^f by renaming them.
2. Push σ below Π and δ, e.g., $\sigma_c(\Pi_A(R)) \equiv \Pi_A(\sigma_c(R))$.

3. Eliminate duplicate δ, e.g., $\delta(\Pi_A(\delta(R))) \equiv \delta(\Pi_A(R))$.
4. Pull σ and Π above $\bowtie$, i.e., $R_1 \bowtie \sigma_c(R_2) \equiv \sigma_c(R_1 \bowtie R_2)$ and $R_1 \bowtie \Pi_A(R_2) \equiv \Pi_{head(R_1),A}(R_1 \bowtie R_2)$.
5. If both subqueries of $\bowtie$ are deduplicated, pull one δ above $\bowtie$, i.e. $\delta(R_1) \bowtie \delta(R_2) \equiv \delta(R_1 \bowtie \delta(R_2))$.

Note that in the canonical form δ is always the last operator of a join subtree. Since we do not consider self-joins of R^f, w.l.o.g. we use $\bowtie \delta$ instead of $\bowtie$, e.g., $q_1 \bowtie \delta(q_2)$ means q_1 joined with the deduplicated q_2.

We also define the *induced functional dependencies* of query q, denoted by $\Gamma(q)$ as in [DS07]:

- Any functional dependency of $rels(q)$ is in $\Gamma(q)$.
- $R^f.\pi \rightarrow head(R^f)$ and $head(R^f) \rightarrow R^f.\pi$ are in $\Gamma(q)$, i.e., each tuple in R^f has a unique annotation.
- For every join predicate $R_i.A = R_j.B$, both $R_i.A \rightarrow R_j.B$ and $R_j.B \rightarrow R_i.A$ are in $\Gamma(q)$;
- For every selection $R_i.A = constant$, $\emptyset \rightarrow R_i.A$ is in $\Gamma(q)$.

Following the convention from previous work [CWW00], we recursively define the *lineage* of $\pi_P(t)$, denoted by $L(\pi_P(t))$, as

Definition 23 (Lineage)**.** *The lineage of $\pi_P(t)$ is defined inductively on a query plan P:*

- *If $P = \sigma_c(P_1)$, then $L(\pi_P(t)) = L(\pi_{P_1}(t))$.*
- *If $P = \Pi_A(P_1)$, then $L(\pi_P(t)) = \bigcup_{t'[A]=t} L(\pi_{P_1}(t'))$.*
- *If $P = P_1 \bowtie \delta(P_2)$, then $L(\pi_P(t)) = L(\pi_{P_1(t)})$.*
- *If $P = \delta(P_1)$, then $L(\pi_P(t)) = L(\pi_{P_1}(t))$.*
- *If P is a relation, $L(\pi_{R^f}(t)) = \{t\}$, $L(\pi_R(t)) = \emptyset$.*

It is easy to see that for any subplan P, the lineage $L(\pi_P(t))$ for any tuple t consists of tuples from the same base relation. We denote the relation as R_{L_P}.

Eligible Plan. Queries can be efficiently computed by the extensional evaluation with an eligible query plan. Given $\Gamma(q)$, we define an *eligible plan* thus.

Definition 24 (Eligible Plan)**.** *A canonical plan P is eligible if all its operators are eligible:*

- *Operators σ, δ and $\bowtie$ are always eligible*
- *Operator $\Pi_A(P_1)$ is eligible, if $\Gamma(P_1)$ implies $\langle A, R_{L_{P_1}}.\pi\rangle$ $\rightarrow head(P_1)$.*

A query's eligibility can be efficiently checked at compile time. Next, we introduce the extensional evaluation restricting ourselves to a strict subset of eligible queries (i.e., those with simple joins) before generalizing to all eligible queries.

Extensional Evaluation of Queries with Simple Joins. A join $P_1 \bowtie \delta(P_2)$ is a *simple join* if $R^f \notin rels(P_2)$. Given an eligible plan P where all joins are simple joins and standalone deduplication is the last operator of P, one can evaluate P via the following extensional evaluation procedure, which directly manipulates the multinomial representations of the annotations.

Definition 25 (Extensional Evaluation (part 1))**.**
Extensional evaluation is defined inductively on a query plan P. Let $\pi_{P_1}(t) = [n, \mathbf{p}_t]$, then:

- *If $P = \sigma_c(P_1)$, then $\pi_P(t) = [n, \mathbf{p}_t]$ if $c(t)$ is true, and $[n, \mathbf{0}]$ otherwise.*
- *If $P = \Pi_A(P_1)$, then $\pi_P(t) = [n, \biguplus_{t'[A]=t} \mathbf{p}_{t'}]$.*
- *If $P = P_1 \bowtie \delta(P_2)$, then $\pi_P(t) = [n, \mathbf{p}_{t_1}]$ where $t_1 = t[head(P_1)]$.*
- *If $P = \delta(P_1)$, then $\pi_P(t) = [1, [\mathbf{p}_t[0]^n, 1 - \mathbf{p}_t[0]^n]]$.*

To reconstruct $q^{mrg}(D^r)$, we compute the probability distribution of $\pi_P(t) = [n, \mathbf{p}]$ by $\mathbf{p}^{\pi_P(t)}[k] = \binom{n}{k}\mathbf{p}[0]^{n-k}\mathbf{p}[1]^k$.

Intuitively, an eligible plan ensures that the tuple annotations are still disjoint after a projection. We formalize this intuition by tracking the lineage of tuples.

Lemma 18. *For any subplans P_1 and P_2,*

1. *$\forall t \in P_1$, $\pi_{P_1}(t) = \bigoplus_{t \in L(\pi_{P_1}(t))} \pi_{R^f}(t)$.*
2. *$\forall t_1, t_2 \in P_1$, $L(\pi_{P_1}(t_1)) \cap L(\pi_{P_1}(t_2)) = \emptyset$.*

Lemma 18 ensures that the extensional evaluation correctly estimates $q^{mrg}(D^r)$ on any D^r by following Proposition 2.

Extensional Evaluation of General Queries. The extensional evaluation from Definition 25 does not apply to eligible queries with general joins. E.g., $P_1 \bowtie \delta(P_2)$ may produce different annotations: if $\pi_{P_2}(t_2) \neq 0$, $\pi_{P_1 \bowtie \delta(P_2)}(t) = \pi_{P_1}(t_1)$, and otherwise $\pi_{P_1 \bowtie \delta(P_2)}(t) = \mathbf{0}$. To resolve this, we enumerate all possible π_{P_2} values; since for each π_{P_2} value, $P_1 \bowtie \delta(P_2)$ is deterministic, we can apply the extensional evaluation from Definition 25.

To aid the enumeration, we represent $\pi(t)$ as a set of triplets $\{(\mathfrak{c}_i, \pi_i, \mathfrak{L}_i)\}$, where

- $\mathfrak{c}_i$ is a set of conjunctive conditions $\{c_j(\pi_{i,j}), \forall j\}$ on annotations $\{\pi_{i,j}, \forall j\}$,
- $\mathfrak{L}_i$ is a set of lineages $\{L(\pi_{i,j}), \forall j\} \cup \{L(\pi_i)\}$.

A triplet $(\mathfrak{c}_i, \pi_i, \mathfrak{L}_i)$ is interpreted as follows: *if all $c_j(\pi_{i,j}) \in \mathfrak{c}_i$ are true, $\pi(t)$ takes value π_i, and the lineages of $\{\pi_{i,j}, \forall j\}$ and π_i are stored in $\mathfrak{L}_i$*. The set $\{\mathfrak{c}_i, \forall i\}$ has two properties: (1) any $\mathfrak{c}_i$ and $\mathfrak{c}_j$ are disjoint, and (2) $\{\mathfrak{c}_i, \forall i\}$ enumerates all possible conditions on $\{\pi_{i,j}, \forall j\}$. Thus, the distribution of $\pi(t)$ can be reconstructed by Equation 7.1, which computes a weighted sum of π_i's distributions under all conditions. Since $\mathfrak{L}_i$ captures the correlation between $\mathfrak{c}_i$ and π_i, Equation 7.1 can easily be computed. (For

details, see Section 7.12.)

$$f(\pi(t)) = \sum_{\forall i} \Pr(\mathfrak{c}_i) f(\pi_i \mid \mathfrak{c}_i) \tag{7.1}$$

At query time, we extend the extensional evaluation to manipulate $\mathfrak{c}_i$, π_i, and $\mathfrak{L}_i$. When combining two sets of conditions $\mathfrak{c}_i$ and $\mathfrak{c}_j$, we also modify π_i and π_j following Definition 25 and modify $\mathfrak{L}_i$ and $\mathfrak{L}_j$ following Definition 23. Below, we show the extended extensional evaluation. For the sake of simplicity, we only show the manipulation of $\mathfrak{c}_i$, omitting that of π_i and $\mathfrak{L}_i$. Furthermore, lineage maintenance has been well addressed in database provenance literature (e.g.,[GA09]). We denote the set $\{\mathfrak{c}_i, \forall i\}$ of $\pi_q(t)$ by $\mathbb{C}_q(t)$. Here, $\times$ denotes the set Cartesian product.

Definition 26 (Extensional Evaluation (part 2))**.**
Extensional evaluation is defined inductively on a query plan P:

- *If $P = \sigma_c(P_1)$, then $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t)$.*
- *If $P = \Pi_A(P_1)$, then $\mathbb{C}_P(t) = \times_{t'[A]=t} \mathbb{C}_{P_1}(t')$.*
- *If $P = P_1 \bowtie \delta(P_2)$, then $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t_1) \times \mathbb{C}_{P_2}(t_2) \times \{\{\pi_{P_2}(t_2) \neq 0\}, \{\pi_{P_2}(t_2) = 0\}\}$ where $t_i = t[head(P_i)]$.*
- *If $P = \delta(P_1)$, then $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t) \times \{\{\pi_{P_1}(t) \neq 0\},$
$\{\pi_{P_1}(t) = 0\}\}$.*

This extensional evaluation works for any eligible query q. For certain queries, one can theoretically create a worst-case database that makes $|\mathbb{C}_q(t)|$ grow exponentially in the size of the database. Next, we identify situations where $|\mathbb{C}_q(t)| = O(1)$, i.e., when DBPTIME evaluation is guaranteed.[10]

[10] However, some queries may still be evaluated efficiently on real-life databases despite being DBPTIME-ineligible, e.g., queries Q17, Q18, Q20, Q22, V3, and V4 in Section 7.7.

Definition 27 (DBPTIME-Eligible Projection)**.**

$\Pi_A(P_1)$ is DBPTIME-eligible, if for every join $P_2 \bowtie \delta(P_3)$ or $\delta(P_3)$ in P_1, $\Gamma(P_1)$ implies $A \rightarrow head(P_3)$.

If $\Pi_A(P_1)$ is DBPTIME-eligible, then $\mathbb{C}_{\pi_{A(P_1)}}(t) = \mathbb{C}_{P_1}(t')$ for any $t'[A] = t$, giving us the following lemma:

Lemma 19. *If every Π_A in an eligible plan P is DBPTIME-eligible, then $|\mathbb{C}_P(t)| = O(1)$.*

We can optimize the extensional evaluation for eligible queries (both DBPTIME-eligible and DBPTIME-ineligible queries) by detecting containment joins. A join $P_1 \bowtie \delta(P_2)$ is a *containment join* if (1) $\Gamma(P_1 \bowtie \delta(P_2))$ implies $head(P_1) \rightarrow head(P_2)$ and (2) $Q = \delta(\Pi_{head(P_1)}(P_1 \bowtie \delta(P_2)))$ contains $\delta(P_1)$, i.e., $\delta(P_1) \subseteq Q$ for any input database. Whether a join is a containment join can be decided in polynomial time for a large class of conjunctive queries [KV98]. A containment join ensures that joined tuples' annotations satisfy entailment, i.e.,

Lemma 20. *For any tuple t generated by a containment join $P_1 \bowtie \delta(P_2)$, $L(\pi_{P_1}(t_1)) \subseteq L(\pi_{P_2}(t_2))$ where $t_i = t[head(P_i)]$.*

Therefore, we have the following optimization:

- If $P = P_1 \bowtie \delta(P_2)$ is a containment join, then $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t_1)$ where $t_i = t[head(P_i)]$.

Note that the size of $\mathbb{C}_P(t)$ is reduced by pruning the set $\mathbb{C}_{P_2}(t_2) \times \{\{\pi_{P_2}(t_2) \neq 0\}, \{\pi_{P_2}(t_2) = 0\}\}$. Furthermore, the corresponding lineage can be dropped from the annotation. Specifically, if every join in plan P either satisfies Definition 27 or is a containment join, P can be evaluated in DBPTIME. The following is an example of the extensional evaluation for general queries.

Example 34. *Consider the query in Example 32 and its evaluation steps in Figures 7.4(c) and 7.4(d). Let* $\pi_i = [3, \mathbf{p}_i]$ *for* $i = 1, 2, 3$*, where* $\mathbf{p}_i[0, 1] = [2/3, 1/3]$.

- $\pi_1 \oplus \pi_2 \oplus \pi_3 = [3, \biguplus_{i=1}^{3} \mathbf{p}_i] = [3, \mathbf{p}]$ *where* $\mathbf{p}[0, 1] = [0, 1]$.
- $\pi_i \odot \mathbb{1}(\pi_1 \oplus \pi_2 \oplus \pi_3) = [3, \mathbf{p}_i]$*, for* $i = 1, 2, 3$ *as Step ② is a containment join.*

7.4.3 Queries with Aggregates

This section extends our extensional evaluation to queries with aggregates. We first consider aggregates in the return clause of a query, then aggregates in predicates.

Handling Aggregates in Return Clauses. Similar to Section 7.4.1, we can represent the annotation ϖ of aggregate $\gamma_{A,\alpha(B)}$ in a multinomial representation, such that Proposition 2 still applies to ϖ. Taking $\gamma_{A,\text{SUM}(B)}(P)$ as an example, let $\pi_p(t) = [n, \mathbf{p}]$. We can represent the annotation $\varpi(t)$ as $[n, \mathbf{p}']$ where $\mathbf{p}'[0] = \mathbf{p}[0]$ and $\mathbf{p}'[t[B]] = \mathbf{p}[1]$.

To check the eligibility of plan P of a query with aggregates, one can simply substitute every $\gamma_{A,\alpha(B)}$ in P with $\delta(\Pi_A)$, and check modified plan P^*'s eligibility. P is eligible if and only if P^* is eligible. Extensional evaluation of γ is very similar to Π (as in Definition 25 and 26), and is omitted. (When maintaining the multinomial representation $[n, \mathbf{p}]$ of ϖ, $\mathbf{p}$ could grow to the database size. Space-efficient approximation is introduced in Section 7.5.)

Handling Aggregates in Predicates. In operator σ_c, if the predicate c contains aggregate $\gamma(q)$ and $R^f \in rel(q)$ (e.g., see step ③ in Figure 7.4(a)), then $c(t)$ is uncertain since γ is random. Thus, we must enumerate all possible value ranges of γ. We modify Definition 26 thus:

Definition 28 (Extensional Evaluation (part 3))**.**
We extend Definition 26 by modifying the rule of σ *as follows:*

- *If* $P = \sigma_c(P_1)$*, then* $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t) \times enum_c(t)$

where $enum_c(t)$ *enumerates all possible valuations of predicate* c*, that is, if* c *is deterministic,* $enum_c(t) = \{\mathbf{T}\}$ *where* $\mathbf{T}$ *is the event true; otherwise,* $enum_c(t) = \{\{c(t)\}, \{\neg c(t)\}\}$.

Similar to Section 7.4.2, the size of $\mathbb{C}_q(t)$ may grow exponentially with the size of the database due to projection and aggregation. Next, we propose an optimization technique to prune conditions with 0 probability, greatly reducing $\mathbb{C}_q(t)$'s size. Then, we again identify cases where DBPTIME evaluation is guaranteed.

Observation 2. *Consider two sets of conditions* $\{\gamma < 4, \gamma \geq 4\}$ *and* $\{\gamma < 3, \gamma \geq 3\}$*. A Cartesian product of these two sets produces* 4 *conditions. However, clearly* $\gamma \geq 4 \wedge \gamma < 3$ *is false. In fact,* 3 *and* 4 *partition the domain of* γ *into three parts:* $(-\infty, 3)$*,* $[3, 4)$ *and* $[4, \infty)$*. Now a Cartesian product of these two sets produces exactly* 3 *valid conditions corresponding to the three partitions, i.e.,* $\gamma \in (-\infty, 3)$*,* $\gamma \in [3, 4)$ *and* $\gamma \in [4, \infty)$*.*

We can generalize this observation into the following pruning rule, reducing the size of the Cartesian product of m condition-sets from $O(2^m)$ to $O(m)$.

Proposition 3. *Let* $c_0 = -\infty < c_1 < \cdots < c_m < c_{m+1} = \infty$ *and* $\mathbb{C}_i = \{\{\gamma < c_i\}, \{\gamma \geq c_i\}\}$*,* $i = 1, \cdots, m$*. Then,*

$$\times_{i=1}^{m} \mathbb{C}_i = \{\{\gamma \in (c_j, c_{j+1}]\} \mid j = 0, \cdots, m\}$$

Next, we identify situations with efficient condition enumeration.

Lemma 21. *After substituting every* $\gamma_{A,\alpha(B)}$ *with* $\delta(\Pi_A)$ *in an eligible plan* P*, if all the* Π *are DBPTIME-eligible, then* $|\mathbb{C}_P(t)| = O(n^{|P|})$*, where* n *is the size of the database.*

7.4.4 Correctness and Complexity

Let $q^e(D^r)$ be the result of extensional evaluation of query q on D^r. The next theorem ensures that extensional evaluation gives (1) the correct $q^{mrg}(D^r)$ for eligible queries on any D^r, and (2) an efficient evaluation time for DBPTIME-eligible queries.

Theorem 22. *For any eligible query q and any PMDB D^r, $q^e(D^r) \equiv q^{mrg}(D^r)$. For any DBPTIME-eligible query q, $q^e(D^r)$ can be computed in DBPTIME.*

The extensional evaluation efficiently computes the annotations in multinomial representation. Nevertheless, reconstructing the exact distributions of aggregates from their annotations can be computationally expensive. For example, the multinomial representation of `SUM` could be of size $O(n)$ where n is the size of the database. This requires enumerating $O(2^n)$ possible cases to compute the distribution of `SUM`. However, many real-world users willingly trade accuracy for efficiency. Thus, Section 7.5 introduces a fast approximation technique for reconstructing the distribution from the annotations.

7.5 Efficient Approximation

We now describe our solution to the problem introduced in Section 7.4.4. By approximating multinomial representations as asymptotically Gaussian distributions, we efficiently reconstruct the required distributions from extensional evaluation outputs. Further, we approximate the multinomial representation by maintaining a few moments (i.e., mean, variance, and covariance) instead of the probability vector itself, which reduces the space overhead.

Given an extended multinomial representation $(\mathfrak{c}, \varpi^*, \mathfrak{L})$, Theorem 23 states that we can model $\{\varpi^*\} \cup \{\varpi_j \in \mathfrak{c}, \forall j\}$ as an asymptotically multivariate normal dis-

tribution. Here, we slightly abuse the notation to refer to a random variable by its probability vector.

Theorem 23. *Let $\varpi_i, i = 1, \cdots, k$ be k semiring random variables, whose multinomial representations are $[n, \mathbf{p}_i]$, respectively. When $n \to \infty$, $\vec{\varpi} = [\varpi_1, \cdots, \varpi_k]$ jointly follows asymptotically a Gaussian distribution:*

$$\vec{\varpi} \sim \mathcal{N}\left(n\vec{\mu}, n\Sigma\right)$$

where $\vec{\mu}_i = \mu_{\mathbf{p}_i}$ is the mean of $\mathbf{p}_i$, $\Sigma_{i,i} = \sigma^2_{\mathbf{p}_i}$ is the variance of $\mathbf{p}_i$, and $\Sigma_{i,j} = \sigma_{\mathbf{p}_i,\mathbf{p}_j}$ is the covariance of $\mathbf{p}_i, \mathbf{p}_j$. Furthermore, applying any differentiable function ψ on $\vec{\varpi}$ still yields an asymptotic Gaussian distribution:

$$\psi(\vec{\varpi}) \sim \mathcal{N}\left(\psi(n\vec{\mu}), n\nabla\psi(n\vec{\mu})^T\Sigma\nabla\psi(n\vec{\mu})\right)$$

where $\nabla\psi$ is the gradient of ψ.

Theorem 23 follows straightforwardly the Central Limit Theorem and Cramér's Theorem [Fer96]. Thus we omit the proofs. This theorem provides accurate approximation whenever the size n of the fact relation is sufficiently large; this is typically the case in large-scale real-life applications. Also, various rules of thumb [BHH05] can be used to check the quality of approximation. Next, we explain how to compute the distribution of AVG, and the conditional probability/distributions in Equation 7.1.

Computing AVG. AVG can be expressed as SUM/COUNT. Thus, to compute $\gamma_{A,\mathrm{AVG}(B)}$, we rewrite it as two aggregates $\gamma_{A,\mathrm{SUM}(B)}$ and $\gamma_{A,\mathrm{COUNT}(B)}$, and evaluate them instead of the original AVG. Let ϖ be the AVG, and ϖ_1, ϖ_2 be the corresponding SUM and COUNT. We have $\varpi = \psi(\varpi_1, \varpi_2)$ where $\psi(x, y) = x/y$, for which we can directly apply Theorem 23 to compute the distribution.

Computing Conditions. As discussed in Section 7.4.3, there are two types of conditions: (1) comparing an aggregate with a constant, and (2) comparing two aggregates. As discussed in Section 7.1.2, we focus on comparison operators $\{<, >, \leq, \geq\}$.

W.l.o.g., consider extensional evaluation output $(\mathfrak{c}, \varpi^*, \mathfrak{L})$ where conditions $\mathfrak{c} = \{\varpi_i < 0 \mid i = 1, \cdots, m\} \cup \{\varpi'_i < \varpi''_i \mid i = 1, \cdots, n\}$. Let $\vec{\varpi}$, $\vec{\varpi}'$ and $\vec{\varpi}''$ denote the vectors $[\varpi_i]$, $[\varpi'_i]$ and $[\varpi''_i]$ respectively. We have the following corollary:

Corollary 24. *Let* $[\varpi^*, \vec{\varpi}, \vec{\varpi}', \vec{\varpi}''] \sim \mathcal{N}(n\mu, n\Sigma)$. *Then*

$$[\varpi^*, \vec{\varpi}, \vec{\varpi}' - \vec{\varpi}''] \sim \mathcal{N}(nA\mu, nA^T\Sigma A) = f$$

where $A = [1, \mathbf{0}_{1(m+2n)}; \mathbf{0}_{m1}, I_m, \mathbf{0}_{m(2n)}; \mathbf{0}_{n(m+1)}, I_n, -I_n]$. *Let* $\vec{l} = -\vec{\infty}$ *and* $\vec{u} = [\infty; \vec{0}]$. *Thus, (1)* $f(\pi \mid \mathfrak{c})$ *is the marginal distribution of* ϖ *when* f *is truncated by the range* $[\vec{l}, \vec{u}]$ *and (2)* $\Pr(\mathfrak{c})$ *is the cumulative probability in the range* $[\vec{l}, \vec{u}]$.

Computing the truncated probability/distribution in Corollary 24 is well-studied in statistics. Efficient solutions can be found in [WM10].

Moreover, Corollary 24 gives us a pruning method for further reducingthe number of conditions we need to enumerate, as in Corollary 25, which we can use to quickly prune conditions with extremely low probability.

Corollary 25. *Let* $\mathfrak{c}$ *be a set of conditions.* $\Pr(\mathfrak{c})$ *is computed by the cumulative probability of* $\mathcal{N}(n\vec{\mu}, n\Sigma)$ *in the range* $[\vec{l}, \vec{u}]$,

$$\Pr(\mathfrak{c}) \leq \Phi(\frac{\vec{u}_i - n\vec{\mu}_i}{\sqrt{n\Sigma_{i,i}}}) - \Phi(\frac{\vec{l}_i - n\vec{\mu}_i}{\sqrt{n\Sigma_{i,i}}}), \forall i$$

Sketching Multinomial Representations. As discussed above, our approximation technique only requires mean, variance, and covariance to reconstruct the distributions. We can use these moments to sketch the multinomial representations without maintaining the actual probability vector; this greatly reduces the storage overhead of our algorithms. During extensional evaluation, we directly maintain these moments using the following equations.

$$\mu_{\mathbf{p}_1 \uplus \mathbf{p}_2} = \mu_{\mathbf{p}_1} + \mu_{\mathbf{p}_2}, \sigma^2_{\mathbf{p}_1 \uplus \mathbf{p}_2} = \sigma^2_{\mathbf{p}_1} + \sigma^2_{\mathbf{p}_2} - 2\mu_{\mathbf{p}_1}\mu_{\mathbf{p}_2}$$

$$\sigma_{\mathbf{p}_1 \uplus \mathbf{p}'_1, \mathbf{p}_2} = \sigma_{\mathbf{p}_1, \mathbf{p}_2} + \sigma_{\mathbf{p}'_1, \mathbf{p}_2}$$

Note that, in the special case of simple group-by aggregates, our extensional evaluation becomes equivalent to previous analytical approaches. This is because in these cases extensional evaluation using the approximate multinomial representation simply consists of summing up the annotations of the corresponding tuples, thereby computing the basic statistics in the same way as previous analytical approaches. However, as we show in Table 7.1, our technique can handle a much larger class of queries.

7.6 Extensions of ABM

In this section, we discuss several extensions of ABM.

General Aggregates. Many common aggregates can be written as functions of simple aggregates. E.g., VAR and STDEV can be written as functions of 1st and 2nd moments. Other aggregates such as MEDIAN, QUANTILE can also be computed using our annotation by a different CLT approximation [SB69].

Multiple Sampled Relations. Uniform sampling on multiple relations participating in a join yields non-uniform and undesirably sparse results [AGP99a]. Join synopses [AGP99a] are usually used to solve this problem, where the basic idea is to pre-compute the join (including self-joins), and uniformly sample from the join results. ABM can easily support join synopses by simply treating the join synopsis as the input sampled relation.

Using Stratified Samples. Although uniform samples are widely used in practice, in some cases stratified samples enable better approximations (e.g., for skewed data) where each stratum is itself a uniform sample, but with a different sampling rate than other strata [AMP13, CDN07]. Bootstrap can also work on stratified samples, by bootstrapping each stratum and combining the resamples from all strata as the simulated

dataset [ET93].

ABM can be easily extended to support stratified samples. Instead of using a single pair $[n, \mathbf{p}]$ to represent each annotation, we extend the multinomial representation to a set of pairs $\{[n_i, \mathbf{p}_i] \mid i = 1, \cdots, h\}$, one for each stratum. We can manipulate this generalized multinomial representation following a similar procedure, as described in Section 7.4. ABM can even handle the case where some strata are sampled while other small strata are not. In Section 7.7.4, we study ABM's accuracy on stratified samples.

7.7 Experiments

To evaluate the effectiveness and efficiency of ABM, we conduct experiments on both synthetic and real-world workloads, and compare the results against both sequential and parallel/distributed implementations of bootstrap.

We use MonetDB (v11.15.19) [mon] for implementing both bootstrap and ABM. We do not modify the internals of the relational engine, but rather implement a middle layer in Java to re-write the SQL queries to support our extensional evaluation. These modified queries are then executed by the relational engine. The returned results are fed into a post-processing module (implemented in R [r] to compute the probabilities/distributions. All pre- and post-processing times are included in ABM's execution times.

7.7.1 Experiment Setup

Parallel/distributed experiments are performed on a cluster of 15 machines, each with two 2.20 GHz Intel Xeon E5-2430 CPU cores and 96GB RAM. Sequential experiments use only one of these machines. We report experiments on three workloads: (1) TPC-H benchmark [tpc], (2) skewed TPC-H benchmark [RPJ13] and (3) a real-world

dataset and query log from the biggest customers of Vertica Inc. [ver] (referred to as Vertica).

TPC-H. We use a 100 GB benchmark (scale factor of 100). We use 17 queries out of the 22 TPC-H queries, namely: Q1, Q3, Q5-Q12, Q14, Q16-Q20, and Q22.[11] The other queries contain aggregates MIN/MAX, or otherwise do not satisfy our eligible query conditions. We (re)sample the largest relation *lineitem*. For queries without *lineitem*, we (re)sample the second largest relations, i.e., *customer* and *partsupp*.

Skewed TPC-H. We generate a 1 GB micro-benchmark (scale factor 1) using the SSB benchmark [RPJ13] (a star schema variation of TPC-H). All the numeric columns follow Zeta distribution with parameter $s \in [2.1, 2.3]$. We use 13 out of the above 17 TPC-H queries after modifying them according to the SSB schema; we leave out Q11, Q16, Q20 and Q22, which are inconsistent with the SSB schema. Again, we (re)sample the largest relation *lineorder*.

Vertica. The Vertica benchmark consists of 52 GB and 310 relations. We have chosen the 6 most complex queries (denoted as V1-V6) from the query logs, which have similar query structures as TPC-H queries Q1, Q11, Q18, Q22. Again, for each query, we (re)sample the largest relation. All (re)sampled relations have 9.2 million tuples each, and are 37.8 GB in total.

7.7.2 Error Quantification Accuracy

In this section, we evaluate the accuracy of our ABM. For each workload and each query q, we conduct three sets of experiments:

1. Ground Truth (GT). Similar to [KTA13], we take an $x\%$ ($x = 1, 2, 5, 10$) random sample from a single relation, leave the other relations intact, and compute q on

[11] As some of queries produce undesirably sparse results under sampling, we keep the query structures but modify the very selective WHERE predicates and/or GROUP BY clauses.

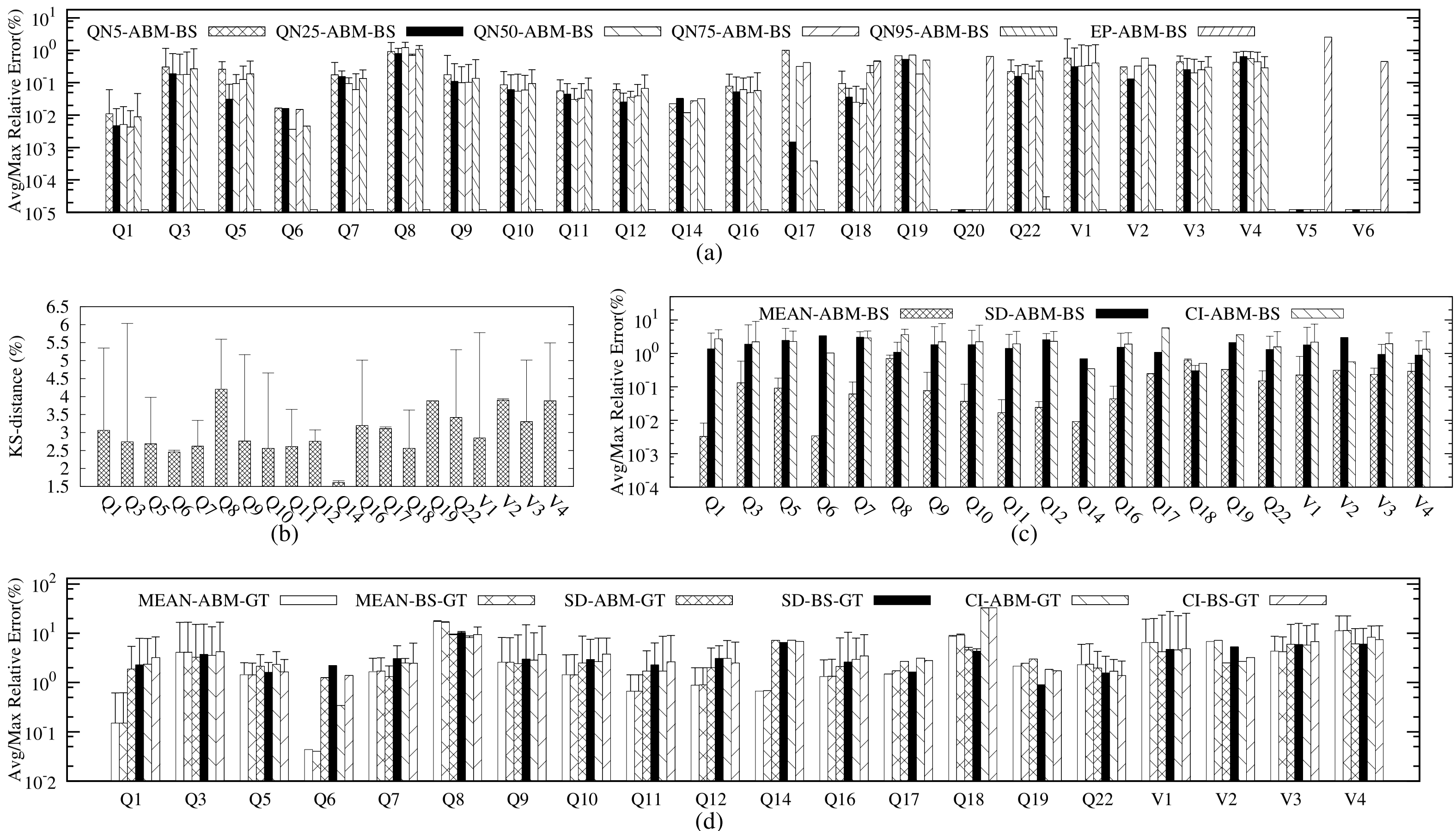


Figure 7.5: Comparing the distributions given by ABM and bootstrap on (a) Quantiles & existence probabilities, (b) KS distance and (c) User-defined quality measures; (d) Comparing user-defined quality measures given by ABMand bootstrap to ground truth

them. We repeat this procedure n times to collect the empirical distribution of all the n results.

2. Bootstrap (BS). We take an $x\%$ random sample from a single relation, we bootstrap this sample, and compute q on each resample and the other intact relations. We repeat the resampling process n times to collect the empirical distribution of all the n results.

3. ABM. We take an $x\%$ random sample from a single relation, and apply extensional evaluation to compute q on the sample and the other intact relations. For comparison purposes, we use the same random sample as bootstrap.

To compare the predicted distribution of query results given by ABM with the empirical distributions given by the ground truth and bootstrap, we measure various distribution statistics, including mean (`MEAN`), standard deviation (`SD`), quantiles (`QN`), 5%-95% confidence interval (`CI`), and existence probability (`EP`) (the probability of each tuple appearing in the query's output). For ground truth and bootstrap, we compute these statistics based on the empirical distribution of the collected results, whereas for ABM, we compute these statistics directly on the estimated distribution of the query results. We report the relative error of these statistics given by different methods. In the figures, we use the notation `S-A-B` to denote the relative error of the statistic `S` given by method `A` compared to the same statistic given by method `B`. For example, the relative error of our mean prediction μ_{ABM} to the empirical mean produced by Ground Truth μ_{GT} is defined as follows $\texttt{MEAN-ABM-GT} = |\frac{\mu_{\text{ABM}}-\mu_{\text{GT}}}{\mu_{\text{GT}}}|$. Note that some test queries (e.g., Q20 in TPC-H, V5 and V6 in Vertica) do not return aggregate values, for which we only report the existence probability.

When a query returns more than one column, we compute both the average and maximum relative error of all the columns in the query. Both errors are shown in our figures, where the histograms represent average error, and the T-shaped error bars represent the maximum error.

1. Predicting the Empirical Distribution of Bootstrap. In the first set of experiments, we study whether the approximate distribution produced by ABM is an accurate prediction of the empirical distribution given by bootstrap. For this purpose, we compare the two distributions in terms of: (1) `EP-ABM-BS` and `QN`k`-ABM-BS` for (k = 5% to 95%) as shown in Figure 7.5(a), and (2) the Kolmogorov-Smirnov (KS) distribution distance as shown in Figure 7.5(b).[12]

We perform the experiments on both TPC-H and Vertica benchmarks with different sample rates ($x = 1, 2, 5, 10$) and $n = 1000$ bootstrap trials. Due to space limitations, we only report the results on the smallest sample size x=1%. The results on larger sample sizes have better accuracy, and thus are omitted. ABM provides highly accurate predictions of bootstrap across all different metrics and queries: On all quantiles, ABM's largest average relative error is less than 2%, while most of the average relative errors are even below 0.1%. Also, the maximum relative error is always below 5% across all quantiles. For most queries, both distributions predict the same existence probabilities, i.e., `EP-ABM-BS` is 0. The average KS distance is about 5%, which is relatively small for 1% sampling rate. In summary, these results show that ABM produces highly accurate predictions of the empirical distribution of bootstrap.

2. Predicting User-defined Quality Measures. In this set of experiments, we study the accuracy of various user-defined quality measures predicted by ABM. We take 1% samples and conduct 1000 sampling/bootstrap trials. The comparison results between ABM and bootstrap on TPC-H and Vertica workloads are reported in Figures 7.5(c). Again, ABM provides highly accurate predictions of bootstrap: On all statistics, ABM's maximum relative error is less than 10%, while most of the relative errors are even below 2%. We also report the results of comparing both ABM and bootstrap against the ground truth in Figure 7.5(d) on TPC-H and Vertica workloads.

[12] http://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test

More interestingly, ABM can also serve as an accurate predication of the ground truth. As shown, comparing to the ground truth, most of ABM's average relative errors are below 5%. Noticeably, ABM is very consistent with bootstrap, i.e., when bootstrap approximates the ground truth well, ABM makes very accurate predictions; when bootstrap has a relatively large error, so does ABM. This evidences that ABM is equivalent to bootstrap.

3. Evaluating on skewed TPC-H benchmark. To study how ABM performs when encounters skewed data, we conduct a micro-benchmark study using the skewed TPC-H workload. We directly run 1000 bootstrap trials on the whole dataset and compare the user-defined quality measures predicted by ABM and bootstrap. The results are shown in Figure 7.6(a). Over all queries and all the compared statistics, ABM's maximum relative error is less than 10%, while most of the relative errors are even below 2%, which shows that the data skewness does not impact the accuracy of ABM much.

4. Varying Number of Bootstrap Trials & Sample Size. We also study the effect of the sample size and the number of bootstrap trials on the prediction accuracy of ABM. We compare ABM with bootstrap on both the distance measures used in Experiment 1 and the user-defined measures used in Experiment 2. Due to space limitations, we only report some representative measures on TPC-H and take the average and maximum of their relative errors across all queries. The other measures and the results on Vertica workload are similar, and are thus omitted.

To study the effect of the number of bootstrap trials, we fix the sampling rate to 1%, but vary the number of bootstrap trails ($n = 100$ to 1000). For each n, we compute the relative error of the statistics given by ABM against those given by bootstrap. As shown in Figure 7.6(b), the relative error between ABM and bootstrap decreases as the number of trials increases, which clearly shows that bootstrap suffers from accuracy loss with a finite number of trials, whereas ABM's analytical modeling of all possible

worlds overcomes this limitation. Moreover, ABM saves the user from error-prone parameter tuning required by bootstrap.

To study the effect of sample size, we conduct the experiments using 1000 bootstrap trials, but varying the sampling rates ($x = 1, 2, 5, 10$). As shown in Figure 7.6(c), ABM performs stably for `CI`, `SD`, and `KS`, while for more linear statistics (i.e., mean and quantiles) its error further decreases with higher sampling rates. Nonetheless, the average relative error of all statistics consistently stays below 3%, even for the smallest sample size ($x = 1\%$).

7.7.3 Error Quantification Performance

This section demonstrates the superior speed of ABM by comparing it against both sequential and parallel/distributed state-of-the-art bootstrap implementations, as well as CLT-based analytical approach. We show that ABM is 5 orders of magnitude faster than the naïve bootstrap and 2-4 orders of magnitude faster than highly optimized variants of bootstrap.

5. Comparing Time Performance. In the first experiment, we compare ABM against three algorithms: (1) naïve bootstrap, (2) *On-Demand Materialization*(ODM) [PJ05a], and (3) Bag of Little Bootstrap (BLB) [KTS12]. To the best of our knowledge, ODM is the best sequential bootstrap algorithm reported. However, it is limited to simple group-by aggregate queries. Bag of Little Bootstrap (BLB) is a bootstrap variant optimized for distributed and parallel platforms. We deploy the sequential algorithms (naïve bootstrap and ODM) and ABM on a single machine, while deploying the parallel/distributed algorithm (BLB) on 10 machines. We compare ABM with all the three counterparts on TPC-H ($x = 10\%$). All the counterpart bootstrap algorithms use 1000 trials. Figure 7.6(d) reports the running time. ABM is 5 orders of magnitude faster than the naïve bootstrap, and more than 2-4 orders of magnitude faster than ODM.

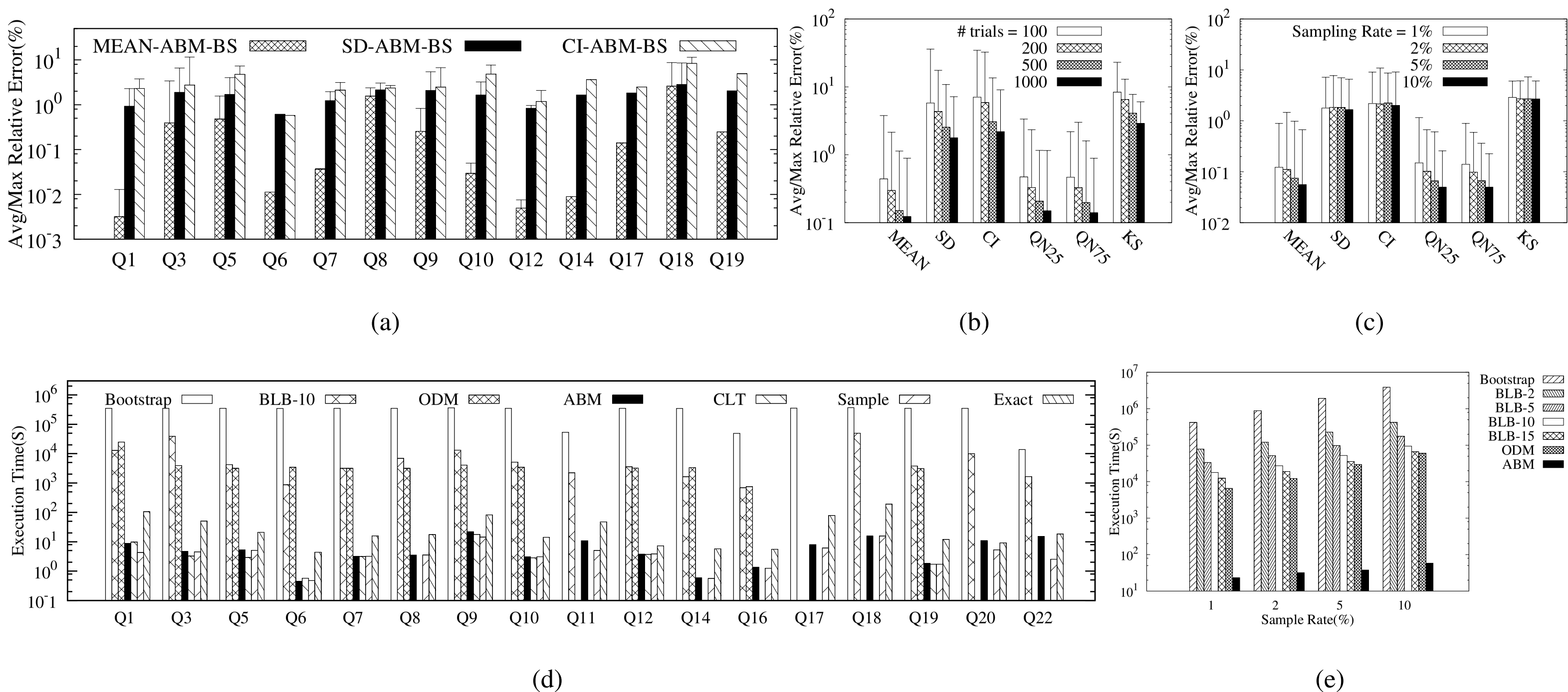


Figure 7.6: (a) ABM vs. bootstrap on user-defined quality measures for Skewed TPC-H; effect of varying (b) number of bootstrap trails, and (c) sampling rate; comparing time performance of ABM & various techniques (d) under 10% sampling rate, (e) under different sampling rates

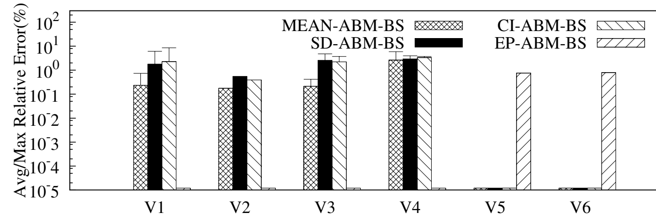


Figure 7.7: ABM vs. bootstrap under stratified sampling

Compared with BLB, ABM is 2-4 orders of magnitude faster although BLB is using 10 times more computation resources.

We also compare ABM with (1) CLT-based analytical approach (which is only applicable to simple group-by aggregates), (2) executing the approximate query on the sample (Sample), and (3) executing the exact query on the original DB (Exact). This comparison clearly demonstrates that ABM achieves almost identical running time as CLT and Sample, incurring little overhead. Since ABM is computed on 10% sample, ABM achieves 10X speed up on almost every query compared with Exact.

6. Varying the Sample Size. Furthermore, as shown in Figure 7.6(e), the execution time of naïve bootstrap and ODM increases with the sample size. On the contrary, our ABM does not vary much in terms of execution time as the sample size increases, since a large portion of ABM's time is spent on query evaluation, which can be highly optimized by modern database engines.

7.7.4 Using Stratified Samples

We study the accuracy of ABM when applied to stratified samples using the Vertica dataset. We apply different stratification on the 4 relations used in the experiments, consisting of 74 and up to 360 strata. The dataset is skewed such that the smallest stratum contains $1/100000$ of the corresponding relation, while the largest stratum contains 63%. We apply the same stratification configuration described in [AMP13].

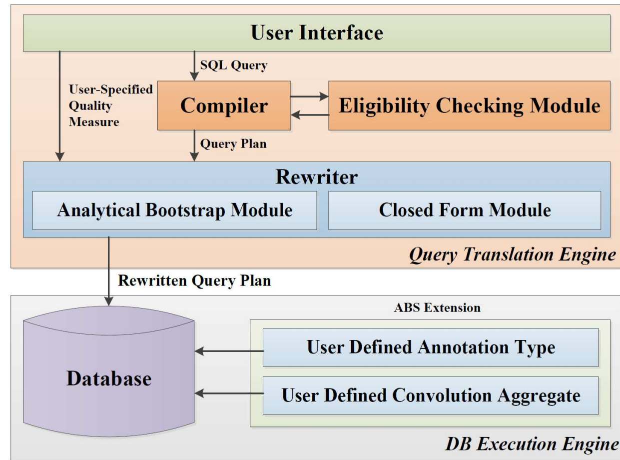


Figure 7.8: ABS Architecture

In particular, we take a stratified sample sized at 1% of the original relation equally from each stratum, while the overly small strata are not sampled. We report the relative error of the predictions given by ABM and bootstrap in Figure 7.7. As shown, all relative errors are below 5%, which evidences that ABM can be extended to support stratified sampling with high accuracy.

7.8 The ABS System

We implemented the analytical bootstrap in a prototype system — Analytical Bootstrap System (ABS). Figure 7.8 shows the high-level architecture of the ABS system, which can be divided into two main components: (1) *Query Translation Engine*: transparently compiling, checking and rewriting the query to support error estimation. (2) *DB Execution Engine*: evaluating the query augmented with error estimation operations, and delivering the accuracy measures in user-specified metrics. The two components are decoupled by the rewritten query plans:

Query Translation Engine Taking a SQL query, the compiler generates a query plan

expressed in relational algebra. Then, the compiler passes the execution plan along with the basic settings of the input database to the eligibility checking module. Based on the eligibility rules defined in Section 7.4, the eligibility checking module verifies if the input query plan is suitable to perform analytical bootstrap.

If an eligible plan is found, the compiler passes the plan to the rewriter. The rewriter takes into consideration the user-specified quality measures, and rewrites the plan into a new query plan with annotation enhanced operations in order to support error estimation, i.e., with additional annotations, and functions that propagate the annotations according to the extensional evaluation. The rewritten query plan preserves the result of the original query, and adds the desired error measures specified by the user.

DB Execution Engine The rewritten query plan is then submitted to the execution engine. Through user-defined type and user-defined aggregates/functions, the extensional evaluation of the ABS sytstem can be easily integrated into common database engines as an extension module (ABS extension as shown in Figure 7.8). Specifically, the ABS system expresses the annotations in the form of user-defined types, and the convolution operations of the annotations as user-defined aggregates/functions. Currently, we implement the ABS system on top of Hive [hiv], an open source distributed data warehouse that supports efficient query evaluation on massive data sets. Furthermore, since our implementation is built as an extension module of Hive, it is easy to deploy ABS on other query engines (e.g., Shark [sha]).

7.9 Related Work

There has been a large body of research on using sampling to provide quick answers to database queries, on database systems [AMP13, CCM00, CDN07, HHW97a, HSS09, JAP07, JJ09, PBJ11a, WOT10], and data stream systems [BDM04, MZ10]. Approx-

imate aggregate processing has been the focus of many of these works, which study randomized joins [JAP07], optimal sample construction [AMP13, CDN07], sample reusing [WOT10], and sampling plan in a stream setting [BDM04, MZ10]. Most of them use statistical inequalities and the central limit theorem to model the confidence interval or variance of the approximate aggregate answers [AMP13, CDN07, HHW97a, HSS09, JAP07, WOT10]. Recently, Pansare et al. [PBJ11a] develop a very sophisticated Bayesian framework to infer the confidence bounds of approximate aggregate answers. However, this approach is limited to simple group-by aggregate queries and does not provide a systematic way of quantifying approximation quality.

Many other works have focused on specific types of queries. For example, Charikar et al. [CCM00] study distinct value estimation from a sample; Joshi and Jermaine [JJ09] propose an EM algorithm to quantify aggregate queries with subset testing.

The bootstrap has become increasingly popular in statistics during the last two decades. Various theoretical [BF81, ET93, VW00] and experimental works [AMK14, KTA13, LZZ12, PJ05a] have proven its effectiveness as a powerful quality assessment tool. Recent works [LZZ12, PJ05a] have used bootstrap in a database setting, in order to quantify the quality of approximate query answers. Nevertheless, all these works focus on improving the Monte-Carlo process of the bootstrap. Thus, Pol et al. [PJ05a] focus on efficiently generating bootstrap samples in a relational database setting, while Laptev et al. [LZZ12] target MapReduce platforms and study how to overlap computation across different bootstrap trials or bootstrap samples. A diagnostic procedure is proposed in [AMK14, KTA13] to determine when bootstrap's error estimation is reliable. This procedure applies bootstrap to multiple sample sizes. Since ABM is equivalent to bootstrap, it can be seamlessly used in this diagnostic procedure, as long as the input query is supported by ABM (e.g., no UDAFs).

Another line of related work is approximate query processing in probabilistic databases.

Much existing work in this area [AJK08, DS07, RS09, SDG10, WMG08] uses possible world semantics to model uncertain data and its query evaluation. Tuples in a probabilistic database have binary uncertainty, i.e., they either exist or not with a certain probability. Specifically, [DS07, RS09] use semirings for modeling and querying probabilistic databases, focusing on conjunctive queries with `HAVING` clauses. On the contrary, we focus on the bootstrap process and model resampled data, using a possible multiset world semantics where database tuples have uncertain multiplicities. Furthermore, bootstrap is fundamentally different from probabilistic databases, since tuples in a resampled relation are always correlated, whereas many probabilistic databases assume that tuples are independent [AJK08, DS07, RS09, SDG10], or propose new query evaluation methods to handle particular correlations. For instance, [TDS13, TPD12] propose Gaussian models to process continuous uncertainty data. Our work is instead based on the bootstrap, which is naturally characterized by discrete distributions, rather than the continuous distributions required by previous techniques.

7.10 Summary of ABM

In this chapter, we developed a probabilistic model for the statistical bootstrap process and showed how it can be used for automatically deriving error estimates for complex database queries. First, we provided a rigorous semantics and a unified analytical model for bootstrap-based error quantification; then we developed an efficient query evaluation technique for a general class of analytical SQL queries. Evaluation using the new method is 2–4 orders of magnitude faster than the state-of-the-art bootstrap implementations. Extensive experiments on a variety of synthetic and real-world datasets and queries confirm the effectiveness and superior performance of our approach. We finally developed an Analytical Bootstrap System (ABS) for parallel and distributed computing platforms. ABS is applicable to most relational database queries and deliv-

ers very accurate estimates at speeds that outperforms the traditional bootstrap method by orders of magnitude.

7.11 Correctness of Intensional & Extensional Evaluation

7.11.1 Background

Proof of Proposition 1. 1°. Consider two S-rvs r_1 and r_2, where r_1 and r_2 are disjoint.

- For $s \in S, s \neq 0$, as $\Pr(r_1 \neq 0 \wedge r_2 \neq 0) = 0$, then

$$\begin{aligned}\Pr(r_1 \oplus r_2 = s) &= \sum_{\substack{\forall x,y \in S,\\ x+y=s}} \Pr(r_1 = x \wedge r_2 = y) \\ &= \Pr(r_1 = 0 \wedge r_2 = s) + \Pr(r_1 = s \wedge r_2 = 0) \\ &= \Pr(r_2 = s) + \Pr(r_1 = s)\end{aligned}$$

- For $s \in S, s = 0$, then

$$\begin{aligned}\Pr(r_1 \oplus r_2 = 0) &= 1 - \sum_{s \neq 0} \Pr(r_1 \oplus r_2 = s) \\ &= 1 - \sum_{s \neq 0} \Pr(r_1 = s) - \sum_{s \neq 0} \Pr(r_2 = s) \\ &= \Pr(r_1 = 0) + \Pr(r_2 = 0) - 1\end{aligned}$$

2°. Consider two S-rvs r_1 and r_2, where r_1 entails r_2 where r_2 can only take value 0 or 1.

- For $s \in S, s \neq 0$, then

$$\Pr(r_1 \odot r_2 = s) = \sum_{\substack{\forall x,y \in S,\\ x \cdot y=s}} \Pr(r_1 = x \wedge r_2 = y) = \Pr(r_1 = s \wedge r_2 = 1) = \Pr(r_1 = s)$$

- For $s \in S, s = 0$, then

$$\Pr(r_1 \odot r_2 = 0) = 1 - \sum_{s \neq 0} \Pr(r_1 \odot r_2 = s) = 1 - \sum_{s \neq 0} \Pr(r_1 = s) = \Pr(r_1 = 0)$$

□

7.11.2 Semantics & Query Evaluation

Proof of Theorem 17. Given a world W (i.e., a deterministic multiset database W). Let $pmw(q^i(D^r))[W]$ represent the possibility given by the intensional semantics, and $q^{pmw}(D^r)[W]$ represent the possible multiset worlds semantics.

Before proving the theorem, let us first note that given any query $q = q_2(q_1)$,

$$q^{pmw}(D^r)[W] = \sum\{\Pr(\hat{D}) \mid \hat{D} \in pmw(D^r), q(\hat{D}) = W\}$$
$$= \sum\{\Pr(\hat{D}) \mid \hat{D} \in pmw(D^r), q_2(q_1(\hat{D})) = W\}$$
$$= \sum\{\Pr(\hat{D}) \mid \hat{D} \in pmw(D^r), q_1(\hat{D}) = W', q_2(W') = W\}$$
$$= \sum\{q_1^{pmw}(D^r)[W'] \mid q_2(W') = W\}$$

We prove inductively on the size of q. If q is a single relation, this holds trivially. Otherwise, q is one of the following 4 cases. We only show the proof for the annotation function π, but omitting the annotation function ϖ as it follows similar proofs (the proof of aggregation γ is similar to that of projection Π).

1° $q = \sigma_c(q_1)$.

$$q^{pmw}(D^r)[W] = \sum\{q_1^{pmw}(D^r)[W'] \mid \sigma_c(W') = W\}$$
$$= \sum\{pmw(q_1^i(D^r))[W'] \mid \sigma_c(W') = W\}$$
$$= \sum\{\Pr(\bigwedge\{\pi_{q_1}(t') = m_{W'}(t') \mid t' \in U_{q_1}\}) \mid \sigma_c(W') = W\}$$
$$= \Pr(\bigwedge\{\pi_{q_1}(t') \odot \mathbb{1}(c(t)) = m_W(t') \mid t' \in U_{q_1}\})$$
$$= \Pr(\bigwedge\{\pi_q(t) = m_W(t) \mid t \in U_q\}) = pmw(q^i(D^r))[W]$$

2° $q = \Pi_A(q_1)$.

$$
\begin{aligned}
q^{pmw}(D^r)[W] &= \sum\{q_1^{pmw}(D^r)[W'] \mid \Pi_A(W') = W\} \\
&= \sum\{pmw(q_1^i(D^r))[W'] \mid \Pi_A(W') = W\} \\
&= \sum\{\Pr(\bigwedge\{\pi_{q_1}(t') = m_{W'}(t') \mid t' \in U_{q_1}\}) \mid \Pi_A(W') = W\} \\
&= \Pr(\bigwedge\{\bigoplus_{t'[A]=t} \pi_{q_1}(t') = m_W(t) \mid t' \in U_{q_1}\}) \\
&= \Pr(\bigwedge\{\pi_q(t) = m_W(t) \mid t \in U_q\}) = pmw(q^i(D^r))[W]
\end{aligned}
$$

3° $q = q_1 \bowtie q_2$. We use (q_1, q_2) to emphasize the correlation between q_1 and q_2.

$$
\begin{aligned}
q^{pmw}(D^r)[W] &= \sum\{(q_1, q_2)^{pmw}(D^r)[W_1, W_2] \mid W_1 \bowtie W_2 = W\} \\
&= \sum\{pmw((q_1^i, q_2^i)(D^r))[W_1, W_2] \mid W_1 \bowtie W_2 = W\} \\
&= \sum\{\Pr(\bigwedge\{\pi_{q_1}(t_1) = m_{W_1}(t_1) \wedge \pi_{q_2}(t_2) = m_{W_2}(t_2) \mid t_1 \in U_{q_1}, t_2 \in U_{q_2}\}) \\
&\quad \mid W_1 \bowtie W_2 = W\} \\
&= \Pr(\bigwedge\{\pi_{q_1}(t_1) \odot \pi_{q_2}(t_2) = m_W(t_1, t_2) \mid t_1 \in U_{q_1}, t_2 \in U_{q_2}, t_1 \text{ can join with } t_2\}) \\
&= \Pr(\bigwedge\{\pi_q(t) = m_W(t) \mid t \in U_q\}) = pmw(q^i(D^r))[W]
\end{aligned}
$$

4° $q = \delta(q_1)$.

$$
\begin{aligned}
q^{pmw}(D^r)[W] &= \sum\{q_1^{pmw}(D^r)[W'] \mid \delta(W') = W\} \\
&= \sum\{pmw(q_1^i(D^r))[W'] \mid \delta(W') = W\} \\
&= \sum\{\Pr(\bigwedge\{\pi_{q_1}(t') = m_{W'}(t') \mid t' \in U_{q_1}\}) \mid \delta(W') = W\} \\
&= \Pr(\bigwedge\{\mathbb{1}(\pi_{q_1}(t')) = m_W(t') \mid t' \in U_{q_1}\}) \\
&= \Pr(\bigwedge\{\pi_q(t) = m_W(t) \mid t \in U_q\}) = pmw(q^i(D^r))[W]
\end{aligned}
$$

Thus, by induction, $q^{pmw}(D^r) = pmw(q^i(D^r))$. □

7.11.3 Extensional Query Evaluation

Proof of Proposition 2. Consider $\pi_i = \bigoplus_{j=1}^{n} \rho_j^{(i)}$ for $i = 1, 2$.

1° $\pi_1 \oplus \pi_2 = \bigoplus_{j=1}^{n} \rho_j^{(1)} \oplus \bigoplus_{j=1}^{n} \rho_j^{(2)} = \bigoplus_{j=1}^{n} \rho_j^{(1)} \oplus \rho_j^{(2)}$. As $atom(\pi_1) \cap atom(\pi_2) = \emptyset$ implies that $\rho_j^{(1)}$ and $\rho_j^{(2)}$ are disjoint for $\forall j$, and thus $\mathbf{p}^{\rho_j^{(1)} \oplus \rho_j^{(2)}} = \mathbf{p}^{\rho_j^{(1)}} \uplus \mathbf{p}^{\rho_j^{(2)}}$ by Proposition 1. Also note that $\rho_j^{(1)} \oplus \rho_j^{(2)}$ are i.i.d. for $\forall j$. Thus, $\mathbf{p}^{\pi_1 \oplus \pi_2} = [n, \mathbf{p}_1 \uplus \mathbf{p}_2]$.

2° $\pi_1 \odot \mathbb{1}(\pi_2) = \bigoplus_{j=1}^{n} \rho_j^{(1)} \odot \mathbb{1}(\bigoplus_{j=1}^{n} \rho_j^{(2)}) = \bigoplus_{j=1}^{n} \rho_j^{(1)} \oplus \mathbb{1}(\rho_j^{(2)} \oplus \bigoplus_{k \neq j} \rho_k^{(2)})$. As $atom(\pi_1) \subset atom(\pi_2)$ implies that π_1 entails π_2, and thus π_1 entails $\mathbb{1}(\pi_2)$. By Proposition 1, $\mathbf{p}^{\pi_1 \odot \mathbb{1}(\pi_2)} = [n, \mathbf{p}_1]$. □

Proof of Lemma 18. We prove by induction on the size of the query plan P. If P is a single relation, the lemma holds trivially. Otherwise, assume that for subquery P_1 (P_2) of P, the lemma holds.

1° $P = \sigma_c(P_1)$ or $P = \delta(P_1)$ or $P = P_1 \bowtie \delta(P_2)$ (We do not discuss rule 1 for $P = \delta(P_1)$). If $P = \sigma_c(P_1)$ and $c(t) = false$, $\pi_P(t) = \mathbf{0}$, where the lemma hold trivially. Otherwise, $\pi_P(t) = \pi_{P_1}(t) = \bigoplus_{t \in L(\pi_{P_1}(t))} \pi_{R^f}(t)$. Thus, $L(\pi_P(t)) = L(\pi_{P_1}(t))$. The lemma holds for P.

2° $P = \Pi_A(P_1)$. $\pi_P(t) = \bigoplus_{t'[A]=t} \pi_{P_1}(t') = \bigoplus_{t'' \in L(\pi_{P_1}(t')) | \forall t', t'[A]=t} \pi_{R^f}(t'')$. And thus $L(\pi_P(t)) = \bigcup_{\forall t', t'[A]=t} L(\pi_{P_1}(t'))$. Obviously, since for $\forall t_1, t_2 \in P_1$, $L(\pi_{P_1}(t_1)) \cap L(\pi_{P_1}(t_2)) = \emptyset$, we have for $\forall t, t' \in P$, $L(\pi_P(t)) \cap L(\pi_P(t')) = \emptyset$. Therefore, the lemma holds for P. □

Proof of Lemma 19. We prove by induction on the size of plan P. Assume for subquery P_1 (P_2) of P, $\mathbb{C}_{P_1}(t)$ ($\mathbb{C}_{P_2}(t)$) has at most 1) conditions.

1. $P = \sigma_c(P_1)$. Since $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t)$, then $|\mathbb{C}_P(t)| = |\mathbb{C}_{P_1}(t)| = 1$. Lemma 19 holds for P.
2. $P = \Pi_A(P_1)$. Since Π_A is *DBPTIME*-eligible, thus $\mathbb{C}_{\pi_A(P_1)}(t) = \mathbb{C}_{P_1}(t')$. $|\mathbb{C}_P(t)| = |\mathbb{C}_{P_1}(t')| = 1$. Lemma 19 holds for P.
3. $P = P_1 \bowtie \delta(P_2)$. Note that $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t_1) \times \mathbb{C}_{P_2}(t_2) \times \{\{\pi_{P_2}(t_2) \neq 0\}, \{\pi_{P_2}(t_2) = 0\}\}$. But only $\{\pi_{P_2}(t_2) \neq 0\}$ may produce a valid result. Thus

$|\mathbb{C}_P(t)| = |\mathbb{C}_{P_1}(t_1)||\mathbb{C}_{P_2}(t_2)| = 1$. Lemma 19 holds for P.

4. $P = \delta(P_1)$. Note that $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t) \times \{\{\pi_{P_1}(t) \neq 0\}, \{\pi_{P_1}(t) = 0\}\}$. But only $\{\pi_{P_1}(t) \neq 0\}$ may produce a valid result. Thus $|\mathbb{C}_P(t)| = |\mathbb{C}_{P_1}(t)| = 1$. Lemma 19 holds for P.

□

Proof of Lemma 20. We prove by contradiction. Assume $L(\pi_{P_1}(t_1)) \not\subseteq L(\pi_{P_2}(t_2))$, i.e. there exists such a tuple $t \in R^f$, such that $t \in L(\pi_{P_1}(t_1))$, but $t \notin L(\pi_{P_2}(t_2))$. Consider an R^f only containing this tuple t. Clearly, $\delta(P_2)$ is empty but $\delta(P_1)$ is not. That is, $\delta(P_1)$ is not contained by $\delta(\Pi_{head(P_1)}(P_1 \bowtie \delta(P_2)))$. We reach a contradiction.

□

Proof of Proposition 3. Consider any pair $c_i < c_j$. The combination of $\gamma < c_i$ and $\gamma > c_j$ cannot hold. Proposition 3 directly follows this observation. □

Proof of Lemma 21. We prove by induction on the size of plan P. Assume for subqueries P_1 (P_2) of P, $\mathbb{C}_{P_1}(t)$ ($\mathbb{C}_{P_2}(t)$) has at most $O(n^{|P_1|})$ ($O(n^{|P_2|})$) conditions.

1. $P = \sigma_c(P_1)$. Since $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t) \times enum_c(t)$ and $enum_c(t) = \{c(t), \neg c(t)\}$, then $|\mathbb{C}_P(t)| = 2|\mathbb{C}_{P_1}(t)| = 2O(n^{|P_1|}) = O(n^{|P|})$. Lemma 21 holds for P.
2. $P = \Pi_A(P_1)$. Since $\mathbb{C}_P(t) = \times_{t'[A]=t}\mathbb{C}_{P_1}(t')$. Since $A \to A'$ for any $\gamma_{A',\alpha'(B')}$ appearing the predicates in q', thus for any tuple $t \in P$, $\mathbb{C}_P(t)$ has at most $|P|$ different aggregate instance (Here we call $\gamma_A \mid_{A=a}$ as a single aggregate instance). There are 3 different cases of predicates involving aggregates: (a) comparing an aggregate with another aggregate, (b) comparing an aggregate with a constant, and (c) comparing an aggregate with an attribute. For case (a) and (b), it is trivial to see that each aggregate instance can introduce at most 2 different conditions. For case (c), assume we compare aggregate γ with attribute C in P_1. According

to Proposition 3, each aggregate instance can introduce at most $O(|P_1|)$ conditions. As implied by the disjointness property in Lemma 18, $|P_1| \leq n$. Thus, we prove that $|\mathbb{C}_q(t)| \leq O(n^{|q|})$. Lemma 21 holds for P.

3. $P = \gamma_{A,\alpha(B)}(P_1)$. The proof follows similarly to Case 2, and thus is omitted.
4. $P = P_1 \bowtie \delta(P_2)$. Since $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t_1) \times \mathbb{C}_{P_2}(t_2) \times \{\{\pi_{P_2}(t_2) \neq 0\}, \{\pi_{P_2}(t_2) = 0\}\}$, then $|\mathbb{C}_P(t)| = 2|\mathbb{C}_{P_1}(t_1)||\mathbb{C}_{P_2}(t_2)| = 2O(n^{|P_1|}) \cdot O(n^{|P_2|}) = 2O(n^{|P_1|+|P_2|}) = O(n^{|P|})$. Lemma 21 holds for P.
5. $P = \delta(P_1)$. Since $\mathbb{C}_P(t) = \mathbb{C}_{P_1}(t) \times \{\{\pi_{P_1}(t) \neq 0\}, \{\pi_{P_1}(t) = 0\}\}$, then $|\mathbb{C}_P(t)| = 2|\mathbb{C}_{P_1}(t)| = 2O(n^{|P_1|}) = O(n^{|P|})$. Lemma 21 holds for P.

□

Proof of Theorem 22. The conclusion about correctness directly follows Theorem 17 and the discussion in Section 7.4. The conclusion about complexity directly follows Lemma 19 and 21. Thus, the proof is omitted. □

7.12 Constructing Distributions for General Queries without Aggregates

Given a pair of $(\mathfrak{c}, \pi)$, we study the problem of computing $\Pr(\mathfrak{c}) f(\pi \mid \mathfrak{c})$.

In general, $\mathfrak{c}$ can be divided into two sets $\mathfrak{c}_0$ and $\mathfrak{c}_1$, where $\mathfrak{c}_0$ is the set of conditions which are in the form of $\pi_{0,i} = 0$, and $\mathfrak{c}_1$ is the set of conditions which are in the form of $\pi_{1,i} \neq 0$. We use $\bar{\mathfrak{c}}_1$ to denote the condition that *at least one condition from* $\mathfrak{c}_1$ *is false*. Then $\Pr(\mathfrak{c})$ and $f(\pi \mid \mathfrak{c})$ can be computed by the following equations:

$$\Pr(\mathfrak{c}) = 1 - \Pr(\mathfrak{c}_0 \wedge \bar{\mathfrak{c}}_1), \quad f(\pi \mid \mathfrak{c}) = f(\pi \mid \mathfrak{c}_0) - f(\pi \mid \mathfrak{c}_0 \wedge \bar{\mathfrak{c}}_1)$$

The probability (distribution) involving $\bar{\mathfrak{c}}_1$ can be computed by using the Inclusion-Exclusion Principle[Fel68].

CHAPTER 8

Conclusion and Future Work

In this dissertation, we tackled the problem of efficient pattern searching and data analytics on massive data bases and data streams. This is a problem of great practical significance for modern data-intensive applications, and we made significant progress towards the solution.

In particular, we proposed K*SQL and XSeq, which extend SQL and XSeq with powerful Kleene-* constructs that allow for the expression of complex patterns over both streaming and stored data. We then provided an efficient execution model and an optimization framework that allows for high-performance execution. Defining domain-specific generalization of K*SQL and XSeq for specialized application areas, and scalable processing of massive number of K*SQL and XSeq patterns in publish-subscribe systems represent interesting problems for future research.

We developed a novel scalable solution for managing and searching web-scale graph data in distributed memory cloud. Our graph exploration and optimization techniques provide a new query processing paradigm for graph pattern matching queries, which achieves superior performance improvement compared with traditional approaches. Besides scalability, our approach opens the potential to support both graph pattern searching and graph analytics in a single system.

We presented the EARL system, which represents one of the first research efforts in supporting approximation as a first-class citizen in big data platforms. Our ap-

proach builds on bootstrap — a novel error estimation technique — and exploits delta-maintenance optimizations to achieve fast query response time and minimal resource usage for general data analytics tasks.

Finally, we developed the Analytical Bootstrap Method, a novel error estimation technique which accurately models the standard bootstrap for a general class of SQL queries, and achieves orders of magnitude better performance than both sequential and parallel/distributed state-of-the-art implementations of bootstrap. For the benefit of users, the techniques of our EARL and ABM systems should be integrated into a single system that intelligently chooses the optimal error estimation method, and this represents an interesting problem for future research

REFERENCES

[Aa90] J.P. Abrashams and et. al. "Prediction of RNA secondary structure, including pseudoknotting." *Nucleic Acids Research*, **18**(10):3035, 1990.

[Aa09] Mohamed H. Ali and et. al. "Microsoft CEP Server and Online Behavioral Targeting." *PVLDB*, 2009.

[ABK07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. "DBpedia: A Nucleus for a Web of Open Data." In *ISWC/ASWC*, pp. 722–735, 2007.

[ACK01] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. "The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases." In *SemWeb*, 2001.

[ACZ10] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. "Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data." In *WWW*, pp. 41–50, 2010.

[AFR00] M. Alexander, J. Fawcett, and P. Runciman. *Nursing practice: hospital and home : the adult*. Churchill Livingstone; 2nd edition, 2000.

[AG05] Renzo Angles and Claudio Gutiérrez. "Querying RDF Data from a Graph Database Perspective." In *ESWC*, pp. 346–360, 2005.

[AGP99a] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. "Join Synopses for Approximate Query Answering." In *SIGMOD*, pp. 275–286, 1999.

[AGP99b] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. "The Aqua Approximate Query Answering System." In *SIGMOD*, pp. 574–576, 1999.

[AJK08] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. "Fast and Simple Relational Processing of Uncertain Data." In *ICDE*, pp. 983–992, 2008.

[Alu07] Rajeev Alur. "Marrying Words and Trees." In *PODS*, 2007.

[AM04] Rajeev Alur and P. Madhusudan. "Visibly pushdown languages." In *STOC*, 2004.

[AM06] Rajeev Alur and P. Madhusudan. "Adding Nesting Structure to Words." In *Developments in Language Theory*, 2006.

[AMK14] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Barzan Mozafari, Michael Jordan, Samuel Madden, and Ion Stoica. “Knowing When You’re Wrong: Building Fast and Reliable Approximate Query Processing Systems.” In *SIGMOD*, 2014.

[AMM09] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. “SW-Store: a vertically partitioned DBMS for Semantic Web data management.” *VLDB J.*, **18**(2):385–406, 2009.

[AMP13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. “BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data.” In *EuroSys*, pp. 29–42, 2013.

[AYU00] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. “A Data Model for Temporal XML Documents.” In *DEXA*, pp. 334–344, 2000.

[Ba03] Charles Barton and et. al. “Streaming XPath Processing with Forward and Backward Axes.” In *ICDE*, 2003.

[Ba06] P. Boncz and et. al. “MonetDB/XQuery: a fast XQuery processor powered by a relational engine.” In *SIGMOD*, 2006.

[Ba07a] Yijian Bai and et. al. “RFID Data Processing with a Data Stream Query Language.” In *ICDE*, 2007.

[Ba07b] Roger S. Barga and et. al. “Consistent Streaming Through Time: A Vision for Event Stream Processing.” In *CIDR*, 2007.

[Ba09] R. Bamford and et. al. “XQuery reloaded.” *VLDB*, 2009.

[BC81] Philip A. Bernstein and Dah-Ming W. Chiu. “Using Semi-Joins to Solve Relational Queries.” *J. ACM*, **28**(1):25–40, 1981.

[BCD03] Brian Babcock, Surajit Chaudhuri, and Gautam Das. “Dynamic Sample Selection for Approximate Query Processing.” In *SIGMOD*, pp. 539–550, 2003.

[BCG11] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. “Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing.” In *ICDE*, pp. 1151 –1162, 2011.

[BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. “Load Shedding for Aggregation Queries over Data Streams.” In *ICDE*, p. 350, 2004.

[BF81] Peter J. Bickel and David A. Freedman. “Some Asymptotic Theory for the Bootstrap.” *The Annals of Statistics*, **9**(6):1196–1217, 1981.

[BGH09] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. “Distributed event stream processing with non-deterministic finite automata.” In *DEBS*, 2009.

[BHB10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. “HaLoop: Efficient Iterative Data Processing on Large Clusters.” In *VLDB*, pp. 285–296, 2010.

[BHH05] George Box, J. Stuart Hunter, and William Hunter. *Statistics for Experimenters: Design, Innovation, Discovery*. Wiley-Interscience, 2005.

[BHS03] Valerie Bönström, Annika Hinze, and Heinz Schweppe. “Storing RDF as a Graph.” In *LA-WEB*, pp. 27–36, 2003.

[BKH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema.” In *ISWC*, 2002.

[btc] “Billion Triple Challenge.” http://challenge.semanticweb.org/.

[Cat06] Balder ten Cate. “The expressivity of XPath with transitive closure.” In *PODS*, 2006.

[CCA09] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. “MapReduce Online.” Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, 2009.

[CCA10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. “Online aggregation and continuous query support in MapReduce.” In *SIGMOD*, pp. 1115–1118. ACM, 2010.

[CCM00] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. “Towards Estimation Error Guarantees for Distinct Values.” In *PODS*, pp. 268–279, 2000.

[CDE05] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. “An Efficient SQL-based RDF Querying Scheme.” In *VLDB*, 2005.

[CDN07] Surajit Chaudhuri, Gautam Das, and Vivek R. Narasayya. "Optimized stratified sampling for approximate query processing." *TODS*, **32**(2):9, 2007.

[CDS04] Surajit Chaudhuri, Gautam Das, and Utkarsh Srivastava. "Effective use of block-level sampling in statistics estimation." In *SIGMOD*, pp. 287–298. ACM, 2004.

[CDZ06] Yi Chen, Susan B. Davidson, and Yifeng Zheng. "An Efficient XPath Query Processor for XML Streams." In *ICDE*, 2006.

[CHS02] Bin Chen, Peter Haas, and Peter Scheuermann. "A new two-phase sampling based algorithm for discovering association rules." In *SIGKDD*, pp. 462–468. ACM, 2002.

[CL09] Balder ten Cate and Carsten Lutz. "The complexity of query containment in expressive fragments of XPath 2.0." *J. ACM*, **56**(6), 2009.

[CM07a] Balder ten Cate and Maarten Marx. "Axiomatizing the Logical Core of XPath 2.0." In *ICDT*, 2007.

[CM07b] Balder ten Cate and Maarten Marx. "Navigational XPath: calculus and algebra." *SIGMOD Record*, **36**(2), 2007.

[con] "Conviva Inc." http://www.conviva.com/.

[CS08] Balder ten Cate and Luc Segoufin. "XPath, transitive closure logic, and nested tree walking automata." In *PODS*, 2008.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. "Tracing the lineage of view data in a warehousing environment." *TODS*, **25**(2):179–227, 2000.

[CYD08] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. "Fast Graph Pattern Matching." In *ICDE*, pp. 913–922, 2008.

[Da07] Alan J. Demers and et. al. "Cayuga: A General Purpose Event Monitoring System." In *CIDR*, 2007.

[Da08] Y. Diao and et. al. "SASE+: An Agile Language for Kleene Closure over Event Streams." Technical report, University of Massachusetts, Amherst, 2008.

[Da09] N. Dindar and et. al. "DejaVu: declarative pattern matching over live and archived streams of events." In *SIGMOD*, 2009.

[DAF03] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. “Path sharing and predicate evaluation for high-performance XML filtering.” *TODS*, **28**(4), 2003.

[dbp] “DBpedia SPARQL Benchmark (DBPSB).” http://aksw.org/Projects/DBPSB.

[DBS06] Arnaud Doucet, Mark Briers, and Stphane Sncal. “Efficient Block Sampling Strategies for Sequential Monte Carlo Methods.” *Journal of Computational and Graphical Statistics*, **15**(3):693–711, 2006.

[DGG86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. “GAMMA - A High Performance Dataflow Database Machine.” In *VLDB*, pp. 228–237, 1986.

[DGH06] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. “Towards Expressive Publish/Subscribe Systems.” In *EDBT*, 2006.

[DM01] Russell Davidson and James G. MacKinnon. “Bootstrap Tests: How Many Bootstraps?” Working Papers 1036, Queen’s University, Department of Economics, 2001.

[DS07] Nilesh N. Dalvi and Dan Suciu. “Efficient query evaluation on probabilistic databases.” *VLDBJ*, **16**:523–544, 2007.

[EM09] Orri Erling and Ivan Mikhailov. “Virtuoso: RDF Support in a Native RDBMS.” In *Semantic Web Information Management*, pp. 501–519. Springer, 2009.

[ET93] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, New York, 1993.

[Fa03] Daniela Florescu and et. al. “The BEA/XQRL Streaming XQuery Processor.” In *VLDB*, 2003.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, January 1968.

[Fer96] Thomas S. Ferguson. *A Course in Large Sample Theory*. Chapman and Hall, 1996.

[FGG11] Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Jon Sellers. “OXPath: A Language for Scalable, Memory-efficient Data Extraction from Web Applications.” *PVLDB*, **4**(11), 2011.

[GA09] Boris Glavic and Gustavo Alonso. "Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting." In *ICDE*, pp. 174–185, 2009.

[GAD08] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. "On Supporting Kleene Closure over Event Streams." In *ICDE*, 2008.

[GC12] Raman Grover and Michael Carey. "Extending Map-Reduce for Efficient Predicate-Based Sampling." In *ICDE*, 2012.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." In *SOSP*, pp. 29–43. ACM, 2003.

[GKT07] Todd J. Green, Gregory Karvounarakis, and Val Tannen. "Provenance semirings." In *PODS*, pp. 31–40, 2007.

[GN11] Olivier Gauwin and Joachim Niehren. "Streamable Fragments of Forward XPath." In *CIAA*, pp. 3–15, 2011.

[GNT11] Olivier Gauwin, Joachim Niehren, and Sophie Tison. "Queries on Xml streams with bounded delay and concurrency." *Inf. Comput.*, **209**(3):409–442, March 2011.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems." *Journal of Web Semantics*, **3**(2-3):158–182, 2005.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.).* Pearson Education, 2009.

[had] "Apache Hadoop Project." http://hadoop.apache.org/.

[HAR11] Jiewen Huang, Daniel J. Abadi, and Kun Ren. "Scalable SPARQL Querying of Large RDF Graphs." *PVLDB*, **4**(11), 2011.

[HG04] Jonathan Hayes and Claudio Gutierrez. "Bipartite Graphs as Intermediate Model for RDF." In *ISWC*, 2004.

[HH05] Lilian Harada and Yuuji Hotta. "Order checking in a CPOE using event analyzer." In *CIKM*, 2005.

[HHW97a] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. "Online Aggregation." In *SIGMOD*, pp. 171–182, 1997.

[HHW97b] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. "Online Aggregation." In *SIGMOD*, pp. 171–182. ACM Press, 1997.

[hiv] "Apache Hive Project." https://hive.apache.org/.

[HLL11] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. "Starfish: A Self-tuning System for Big Data Analytics." In *CIDR*, pp. 261–272, 2011.

[HMM11] Mohammad Farhan Husain, James P. McGlothlin, Mohammad M. Masud, Latifur R. Khan, and Bhavani M. Thuraisingham. "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing." *IEEE Trans. Knowl. Data Eng.*, **23**(9):1312–1327, 2011.

[HS08] Huahai He and Ambuj K. Singh. "Graphs-at-a-time: query language and access methods for graph databases." In *SIGMOD*, 2008.

[HSS09] Ying Hu, Seema Sundara, and Jagannathan Srinivasan. "Estimating Aggregates in Time-Constrained Approximate Queries in Oracle." In *EDBT*, pp. 1104–1107, 2009.

[HUH07] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. "YARS2: A Federated Repository for Querying Graph Structured Data from the Web." In *ISWC/ASWC*, pp. 211–224, 2007.

[JAP07] Christopher Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. "Scalable Approximate Query Processing with DBO Engine." In *SIGMOD*, pp. 1–54, 2007.

[jen] "Jena." http://jena.sourceforge.net.

[JFB05] Vanja Josifovski, Marcus Fontoura, and Attila Barta. "Querying XML streams." *VLDB Journal*, **14**(2), 2005.

[JJ09] Shantanu Joshi and Christopher Jermaine. "Sampling-Based Estimators for Subset-Based Queries." *VLDB J.*, **18**(1):181–202, 2009.

[JS08] C. S. Jensen and R. T. Snodgrass. "Temporal Query Languages." In *Temporal Database Entries for the Springer Encyclpedia of Database Systems*, volume TR-90, 2008.

[Kay08] Michael Kay. "Ten Reasons Why Saxon XQuery is Fast." *IEEE Data Eng. Bull.*, **31**(4), 2008.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. "Fast Pattern Matching in Strings." *SIAM J. Comput.*, **6**(2), 1977.

[KMS08] Leila Kaghazian, Dennis McLeod, and Reza Sadri. "Scalable complex pattern search in sequential data." In *CIKM*, 2008.

[Koc09] Christoph Koch. “XML Stream Processing.” In *Encyclopedia of Database Systems*. 2009.

[KTA13] Ariel Kleiner, Ammet Talwalkar, Sammeer Agarwal, Ion Stoica, and Michael Jordan. “A General Bootstrap Performance Diagnostic.” In *KDD*, pp. 419–427, 2013.

[KTS12] Ariel Kleiner, Ameet Talwalkar, Purnamrita Sarkar, and Michael I. Jordan. “The Big Data Bootstrap.” In *ICML*, 2012.

[KV98] Phokion G. Kolaitis and Moshe Y. Vardi. “Conjunctive-Query Containment and Constraint Satisfaction.” In *PODS*, pp. 205–213, 1998.

[La10] M. Liu and et al. “E-Cube: Multi-Dimensional Event Sequence Processing Using Concept and Pattern Hierarchies.” In *ICDE*, 2010.

[LGH07] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. “Challenges in Parallel Graph Processing.” *Parallel Processing Letters*, **17**(1):5–20, 2007.

[LMD11] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant J. Shenoy. “A platform for scalable one-pass analytics using MapReduce.” In *SIGMOD*. ACM Press, 2011.

[Luc01] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.

[LYT05] Jing Lu, Yong Yu, Kewei Tu, Chenxi Lin, and Lei Zhang. “An Approach to RDF(S) Query, Manipulation and Inference on Databases.” In *WAIM*, pp. 172–183, 2005.

[LZ12] Nikolay Laptev and Carlo Zaniolo. “Optimization of Massive Pattern Queries by Dynamic Configuration Morphing.” In *ICDE*, pp. 917–928, 2012.

[LZZ12] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. “Early Accurate Results for Advanced Analytics on MapReduce.” *PVLDB*, **5**(10):1028–1039, 2012.

[MAB10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. “Pregel: a system for large-scale graph processing.” In *SIGMOD*, 2010.

[Mar05] Maarten Marx. “Conditional XPath.” *TODS*, **30**(4), 2005.

[mon] "MonetDB." http://www.monetdb.org/Home.

[MV09] P. Madhusudan and M. Viswanathan. "Query Automata for Nested Words." In *MFCS*, 2009.

[MZ10] Barzan Mozafari and Carlo Zaniolo. "Optimal Load Shedding with Aggregates and Mining Queries." In *ICDE*, pp. 76–88, 2010.

[MZZ10] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. "From Regular Expressions to Nested Words: Unifying Languages and Query Execution for Relational and XML Sequences." *PVLDB*, **3**(1), 2010.

[MZZ12] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. "High-performance complex event processing over XML streams." In *SIGMOD Conference*, pp. 253–264, 2012.

[NG04] M.E.J. Newman and M. Girvan. "Finding and evaluating community structure in networks." *Physical review E*, **69**(2):026113, 2004.

[NW08] Thomas Neumann and Gerhard Weikum. "RDF-3X: a RISC-style engine for RDF." *PVLDB*, **1**(1), 2008.

[NW09] Thomas Neumann and Gerhard Weikum. "Scalable Join Processing on Very Large RDF Graphs." In *SIGMOD*, 2009.

[NW10] Thomas Neumann and Gerhard Weikum. "The RDF-3X engine for scalable management of RDF data." *VLDB J.*, **19**(1):91–113, 2010.

[OBE09] Christopher Olston, Edward Bortnikov, Khaled Elmeleegy, Flavio Junqueira, and Benjamin Reed. "Interactive Analysis of Web-Scale Data." In *CIDR*, 2009.

[OKB03] Dan Olteanu, Tobias Kiesling, and François Bry. "An Evaluation of Regular Path Expressions with Qualifiers against XML Streams." In *ICDE*, 2003.

[OR90] Frank Olken and Doron Rotem. "Random Sampling from Database Files: A Survey." In *SSDBM*, pp. 92–111, 1990.

[ORS08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig latin: a not-so-foreign language for data processing." In *SIGMOD*, pp. 1099–1110. ACM, 2008.

[PBJ11a] Niketan Pansare, Vinayak R. Borkar, Chris Jermaine, and Tyson Condie. "Online Aggregation for Large MapReduce Jobs." *PVLDB*, **4**(11):1135–1145, 2011.

[PBJ11b] Niketan Pansare, Vinayak R. Borkar, Chris Jermaine, and Tyson Condie. "Online Aggregation for Large MapReduce Jobs." *PVLDB*, **4**(11):1135–1145, 2011.

[PC03] Feng Peng and Sudarshan S. Chawathe. "XPath Queries on Streaming Data." In *SIGMOD Conference*, 2003.

[PDG05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. "Interpreting the data: Parallel analysis with Sawzall." *Sci. Program.*, **13**(4):277–298, 2005.

[Pit05] Corin Pitcher. "Visibly pushdown expression effects for XML stream processing." In *PLAN-X*, 2005.

[PJ05a] Abhijit Pol and Chris Jermaine. "Relational Confidence Bounds Are Easy With The Bootstrap." In *SIGMOD*, pp. 587–598, 2005.

[PJ05b] Abhijit Pol and Christopher Jermaine. "Relational confidence bounds are easy with the bootstrap." In *SIGMOD*, pp. 587–598. ACM, 2005.

[Pot94] Andreas Potthoff. "Modulo-counting quantifiers over finite trees." *Theor. Comput. Sci.*, **126**(1), 1994.

[r] "The R Project." http://www.r-project.org/.

[rel] "EARL Project." http://yellowstone.cs.ucla.edu/wis/.

[RPJ13] Tilmann Rabl, Meikel Poess, Hans-Arno Jacobsen, Patrick O'Neil, and Elizabeth O'Neil. "Variations of the Star Schema Benchmark to Test the Effects of Data Skew on Query Performance." In *SPEC*, pp. 361–372, 2013.

[RS09] Christopher Ré and Dan Suciu. "The Trichotomy of HAVING Queries on a Probabilistic Database." *VLDBJ*, **18**(5):1091–1116, 2009.

[RS10] Kurt Rohloff and Richard E. Schantz. "High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store." In *PSI EtA*, 2010.

[Sa01] Reza Sadri and et. al. "A Sequential Pattern Query Language for Supporting Instant Data Mining for e-Services." In *VLDB*, 2001.

[Sa02] Albrecht Schmidt and et. al. "XMark: a benchmark for XML data management." In *VLDB*, 2002.

[SB69] M. M. Siddiqui and Calvin Butler. "Asymptotic Joint Distribution of Linear Systematic Statistics from Multivariate Distributions." *Journal of the American Statistical Association*, **64**(325):300–305, 1969.

[SDG10] Prithviraj Sen, Amol Deshpande, and Lise Getoor. "Read-Once Functions and Query Evaluation in Probabilistic Databases." *PVLDB*, **3**(1):1068–1079, 2010.

[SG07] Bianca Schroeder and Garth A. Gibson. "Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you?" In *USENIX FAST*, pp. 1–12. USENIX Association, 2007.

[sha] "Shark Project." http://shark.cs.berkeley.edu/.

[Sno09] Richard T. Snodgrass. "TSQL2." In *Encyclopedia of Database Systems*. 2009.

[SS09] Lena Strömbäck and Stefan Schmidt. "An Extension of XQuery for Graph Analysis of Biological Pathways." In *DBKDA*, 2009.

[SSB08] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. "SPARQL basic graph pattern optimization using selectivity estimation." In *WWW*, 2008.

[SWL13] Bin Shao, Haixun Wang, and Yatao Li. "Trinity: A Distributed Graph Engine on a Memory Cloud." In *SIGMOD Conference*, 2013.

[SWW12] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. "Efficient subgraph matching on billion node graphs." *Proceedings of the VLDB Endowment*, **5**(9):788–799, 2012.

[SWX12] B. Shao, H. Wang, and Y. Xiao. "Managing and mining large graphs: Systems and implementations." In *SIGMOD*, 2012.

[SZZ01] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. "Optimization of Sequence Queries in Database Systems." In *PODS*, 2001.

[SZZ04] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. "Expressing and optimizing sequence queries in database systems." *TODS*, **29**(2):282–318, 2004.

[Tan09] Nguyen Van Tang. "A Tighter Bound for the Determinization of Visibly Pushdown Automata." In *INFINITY*, 2009.

[TDS13] Thanh T. L. Tran, Yanlei Diao, Charles Sutton, and Anna Liu. "Supporting User-Defined Functions on Uncertain Data." *PVLDB*, **6**(6):469–480, 2013.

[ter] "Teradata Corp." http://www.teradata.com/?LangType=1033.

[tpc] "TPC-H Benchmark." http://www.tpc.org/tpch/.

[TPD12] Thanh T. L. Tran, Liping Peng, Yanlei Diao, Andrew McGregor, and Anna Liu. "CLARO: Modeling and Processing Uncertain Data Streams." *VLDBJ*, **21**(5):651–676, 2012.

[tri] "Trinity." http://research.microsoft.com/en-us/projects/trinity/.

[TSJ09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive- A Warehousing Solution Over a Map-Reduce Framework." In *VLDB*, pp. 1626–1629, 2009.

[ver] "Vertica Inc." http://www.vertica.com/.

[VMT07] Zografoula Vagena, Mirella Moura Moro, and V. J. Tsotras. "RoXSum: Leveraging Data Aggregation and Batch Processing for XML Routing." In *ICDE*, 2007.

[VW00] Aad van der Vaart and Jon Wellner. *Weak Convergence and Empirical Processes*. Springer, corrected edition, November 2000.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-performance complex event processing over streams." In *SIGMOD*, 2006.

[WHY06] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. "Dual Labeling: Answering Graph Reachability Queries in Constant Time." In *ICDE*, p. 75, 2006.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. "Hexastore: sextuple indexing for semantic web data management." *PVLDB*, **1**(1):1008–1019, 2008.

[WLW12] W. Wu, H. Li, H. Wang, and K.Q. Zhu. "Probase: A probabilistic taxonomy for text understanding." In *SIGMOD*, 2012.

[WM10] Stefan Wilhelm and B. G. Manjunath. "tmvtnorm: A Package for the Truncated Multivariate Normal Distribution." *The R Journal*, **2**(1):25–29, June 2010.

[WMG08] Daisy Zhe Wang, Eirinaios Michelakis, Minos N. Garofalakis, and Joseph M. Hellerstein. "BayesStore: Managing Large, Uncertain Data Repositories with Probabilistic Graphical Models." *PVLDB*, **1**(1):340–351, 2008.

[WOT10] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. "Continuous Sampling for Online Aggregation over Multiple Queries." In *SIGMOD*, pp. 651–662, 2010.

[WSK03] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. "Efficient RDF Storage and Retrieval in Jena2." In *SWDB*, pp. 131–150, 2003.

[WZZ08] Fusheng Wang, Carlo Zaniolo, and Xin Zhou. "ArchIS: an XML-based approach to transaction-time temporal database systems." *VLDB J.*, **17**(6):1445–1463, 2008.

[YYZ12] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. "Towards effective partition management for large graphs." In *SIGMOD Conference*, pp. 517–528, 2012.

[Za06] Xin Zhou and et. al. "Unifying the Processing of XML Streams and Relational Data Streams." In *ICDE*, 2006.

[Zan09a] Carlo Zaniolo. "Event-Oriented Data Models and Query Languages in Transaction-Time Databases." In *TIME*, 2009.

[Zan09b] Carlo Zaniolo. "Event-Oriented Data Models and Temporal Queries in Transaction-Time Databases." In *TIME*, pp. 47–53, 2009.

[ZCO09] Lei Zou, Lei Chen, and M. Tamer Özsu. "DistanceJoin: Pattern Match Query In a Large Graph Database." *PVLDB*, **2**(1):886–897, 2009.

[ZMH09] Weizhong Zhao, Huifang Ma, and Qing He. "Parallel K-Means Clustering Based on MapReduce." *Cloud Computing*, **5931**:674–679, 2009.

[ZQL11] Feida Zhu, Qiang Qu, David Lo, Xifeng Yan, Jiawei Han, and Philip S. Yu. "Mining Top-K Large Structural Patterns in a Massive Network." *PVLDB*, **4**(11):807–818, 2011.

[ZWC07] Fred Zemke, Andrew Witkowski, Mitch Cherniak, and Latha Colby. "Pattern matching in sequences of rows." 2007.

[ZYM13] Kai Zeng, Mohan Yang, Barzan Mozafari, and Carlo Zaniolo. "Complex Pattern Matching in Complex Structures: the XSeq Approach." In *ICDE Demo*, 2013.