UNIVERSITY OF CALIFORNIA

Los Angeles

Declarative Frameworks and Optimization Techniques for Developing Scalable Advanced Analytics over Databases and Data Streams

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Ariyam Das

2019

© Copyright by

Ariyam Das

2019

Abstract of the Dissertation

Declarative Frameworks and Optimization Techniques for Developing Scalable Advanced Analytics over Databases and Data Streams

by

Ariyam Das
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Carlo Zaniolo, Chair

In the past, the semantic issues raised by the non-monotonic nature of aggregates often prevented their use in the recursive statements of logic programs and deductive databases. However, the recently introduced notion of Pre-Mappability ($\mathscr{P}reM$) has shown that, in key applications of interest, aggregates can be used in recursion to optimize the perfect-model semantics of aggregate-stratified programs. Therefore, we can preserve the declarative formal semantics of such programs, while achieving a highly efficient operational semantics that is conducive to scalable implementations on parallel and distributed platforms.

In this work, we show that using $\mathscr{P}reM$, a wide spectrum of classical algorithms, ranging from graph analytics and dynamic programming based optimization problems to data mining, machine learning and online streaming applications can be concisely expressed in declarative languages by using aggregates in recursion. We present a concise analysis of this very general property and characterize its different manifestations for different constraints and rules.

Next, we prove that $\mathscr{P}reM$-optimized plans are easily parallelizable and produce the same results as the single executor programs. Thus, $\mathscr{P}reM$ can be trivially assimilated into the data-parallel computation plans of different distributed systems, irrespective of whether these follow bulk synchronous parallel (BSP) or asynchronous computing models. This makes possible many advanced BigData applications to be now expressed declaratively in logic-based languages, includ-

ing Datalog, Prolog, and even SQL, while enabling their execution with superior performance and scalability as compared to other specialized systems. Furthermore, we show that under $\mathscr{P}reM$ non-linear recursive queries can be evaluated using a hybrid stale synchronous parallel (SSP) model with relaxed synchronization on distributed environments. We present empirical evidence of its benefits. We also compare the usability, expressivity and performance of $\mathscr{P}reM$-optimized queries with queries written in quasi-declarative programming methodologies inspired by procedural languages like XY-stratification to showcase the different trade-offs and ramifications associated with each.

Lastly, we present robust online optimization techniques using two popular case studies, namely online lossless frequent pattern mining and online decision tree construction, to show how compact representations and statistical approximations can deliver superior performances in real-time for several streaming data mining and machine learning applications.

The dissertation of Ariyam Das is approved.

Junghoo Cho

Wei Wang

Yingnian Wu

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2019

To my parents

Table of Contents

1 Introduction **1**

2 Declarative Recursive Computation with $\mathscr{P}reM$ **5**

2.1 Pre-Mappability 6

2.2 Motivation 10

2.3 An Overview of Parallel Bottom-Up Evaluation 12

2.4 Parallel Evaluation with $\mathscr{P}reM$ 15

2.5 A Case for Relaxed Synchronization 17

2.6 Bottom-up Evaluation with SSP Processing 21

2.6.1 SSP Evaluation of Queries without $\mathscr{P}reM$ Constraint 24

2.6.2 Experimental Results 25

2.7 Conclusion 28

3 Efficiently Computable Subclass of Stable Models **29**

3.1 Introduction 30

3.2 $XY-$Stratification 32

3.3 Comparison of $XY-$Stratification with $\mathscr{P}reM$ 35

3.3.1 Temporal Coalescing 35

3.3.2 Floyd-Warshall Algorithm 39

3.4 Stable Model Computation 41
3.5 When $\mathscr{P}reM$ is Inapplicable 43
3.6 Conclusion 45

4 Expressing BigData Applications with $\mathscr{P}reM$ 46
4.1 Evolution of Declarative Semantics 46
4.2 Different Manifestations of $\mathscr{P}reM$ 47
4.3 Dynamic Programming based Optimization Problem 49
4.4 K-Nearest Neighbors Classifier 50
4.5 Iterative-Convergent Machine Learning Models 52
4.6 $\mathscr{P}reM$ for Continuous Queries on Data Streams 53
4.6.1 Streamlog: Formal Semantics 55
4.6.2 Streaming Applications with $\mathscr{P}reM$ 55
4.7 Conclusion 58

5 Fast Frequent Itemset Mining from Data Streams 60
5.1 Introduction 60
5.2 Related Work 61
5.3 Contributions 63
5.4 Preliminaries 64
5.5 Crucial Patterns 65
5.6 Properties of Crucial Patterns 68
5.7 Set Enumeration Tree Construction 71
5.8 Crucial Pattern Mining Algorithm 72
5.8.1 Initialization 72

5.8.2 Delta Maintenance of Prefix Tree. 72
5.8.3 Update of Crucial Patterns. 75
5.9 Experiments 76
5.9.1 Setup 76
5.9.2 Datasets 76
5.9.3 Results 77
5.10 Conclusion 80

6 Scalable Construction of Online Decision Trees 82
6.1 Introduction 82
6.2 Related Work 86
6.2.1 Incremental Decision Tree Learning 86
6.2.2 Resampling Methods 87
6.3 Methodology 87
6.3.1 Preliminary 87
6.3.2 Non-parametric Bootstrap Driven Split 88
6.3.3 Memory-Efficient Bootstrap Simulation (Mem-ES) 91
6.4 Evaluation 92
6.4.1 Experimental Setup 92
6.4.2 Results and Discussion 94
6.5 Conclusion 95

7 Conclusion and Future Work 101

References 103

List of Figures

2.1 A toy graph distributed across two workers. 20

2.2 BSP vs. SSP model for evaluating *all pairs shortest path* query on two workers. . . 20

2.3 SSP based bottom-up evaluation plan executed by worker *i* for computing all pairs shortest path. 21

3.1 Temporal projections 36

3.2 Comparing $XY-$stratified query evaluation with BSP model vs. $\mathscr{P}reM$ enabled query evaluation with SSP model for Floyd-Warshall algorithm. 42

3.3 Optimal time to get part E. 44

5.1 A sliding window example with *window size=4* and *slide size=2*. 66

5.2 All frequent patterns represented with their respective supports. 66

5.3 Mining frequent itemsets with associated branch ids from FP-tree. 69

5.4 Algorithm for computing crucial patterns. 73

5.5 Maintaining a close-to-optimal prefix tree. 75

5.6 Mining time comparison for lossless extraction methods. 78

5.7 Delta maintenance time comparison for lossless extraction methods. 79

5.8 Running time comparison for lossless extraction methods. 80

5.9 Compression ratio comparison for lossless representations. 81

6.1 Online decision tree construction 83

6.2 Observations over decision tree node split 84

6.3 Algorithm for constructing online decision tree 89

6.4 Algorithm for Mem-ES 97

6.5 Comparison with state-of-the-art methods: Error Rate 98

6.6 Tree growth and memory consumption: Mem-ES vs. VFDT and *VFDT+IG* 99

6.7 Tree growth and memory consumption: Mem-ES vs. EFDT 100

List of Tables

2.1 Comparing BSP vs. SSP model for all pairs shortest path query containing aggregates in recursion (with $\mathscr{P}reM$). 27

2.2 Comparing BSP vs. SSP model for transitive closure query containing *no* aggregates in recursion (without $\mathscr{P}reM$). 27

ACKNOWLEDGMENTS

Let me begin by first thanking my advisor, Carlo Zaniolo, for his guidance throughout my doctoral studies. I am immensely grateful to him for giving me an opportunity to pursue a PhD and also offering me great flexibility to work on different research topics. He has been a tremendous role model and has always provided me with excellent advice and support throughout my studies.

I would also like to express my deep gratitude to my committee, Junghoo Cho, Wei Wang and Yingnian Wu, for their time, help and wonderful suggestions over the years. During my time at UCLA and my summers at SRI International and Google, I also had the opportunity to work closely with many other amazing people. In particular, I would like to thank Ameet Talwalkar, Shalini Ghosh, Akhilesh Shirbhate, Wantanee Viriyasitavat and Xiaolong Long for mentoring me and making this journey even more enjoyable and rewarding.

I am also grateful to all my friends and fellow students at UCLA for helping me along the way. This list includes but is not limited to Muhao Chen, Xuelu Chen, Hsuan Chiu, Ling Ding, Sahil Gandhi, Shi Gao, Jiaqi Gu, Bo-Jhang Ho, Matteo Interlandi, Tushar Sudhakar Jee, Arul Jeyaraj, Chelsea J-T Ju, Seungbae Kim, Kannan KT, Mahati Kumar, Jae Lee, Mingda Li, Ruirui Li, Yi Li, Youfu Li, Giuseppe Mazzeo, Tonmoy Monsoor, Uneeb Rathore, Manoj Reddy, Alexander Shkapsky, Ishan Upadhyaya, Jin Wang, Zijun, Xue, Mohan Yang and Wenchao Yu. I am also very thankful to our department staffs, especially Juliana Alvarez, Steve Arbuckle and Joseph Brown, who always promptly responded whenever I needed help.

Last, but not least, I am infinitely grateful to my parents, Sarmistha Das and Abhijit Das, for their unconditional love and unwavering support. Without their constant encouragement, I would not have been able to embark on this wonderful journey. They are my greatest teachers, and whatever I may have accomplished is a credit to them. Thank you, Ma and Baba.

VITA

2011	Bachelor of Computer Science & Engineering, Jadavpur University, India.
2011-2014	Senior Software Developer, Yahoo!, Bangalore, India.
2014-2015	Graduate Student Researcher, Department of Computer Science, UCLA.
2015-2016	Teaching Assistant, Department of Computer Science, UCLA.
2016	Research Intern, SRI International, Menlo Park, CA.
2016-2017	Teaching Associate, Department of Computer Science, UCLA.
2017	Master of Science in Computer Science, UCLA.
2017	SWE Intern, Google, Mountain View, CA.
2018	SWE Intern, Google, Los Angeles, CA.
2017-2019	Teaching Fellow, Department of Computer Science, UCLA.

PUBLICATIONS

- Ariyam Das, Carlo Zaniolo. A Case for Stale Synchronous Distributed Model for Declarative Recursive Computation. *TPLP*, 2019.
- Ariyam Das, Jin Wang, Sahil M. Gandhi, Jae Lee, Wei Wang, Carlo Zaniolo. Learn Smart with Less: Building Better Online Decision Trees with Fewer Training Examples. *IJCAI*, 2019.
- Ariyam Das, Y. Li, J. Wang, M. Li, Carlo Zaniolo. BigData Applications from Graph Analytics to Machine Learning by Aggregates in Recursion. *ICLP Technical Communications*, 2019.
- Tyson Condie, Ariyam Das, Matteo Interlandi, A. Shkapsky, Mohan Yang, Carlo Zaniolo. Scaling-Up Reasoning and Advanced Analytics on BigData. *TPLP*, 2018.
- Ariyam Das, Sahil M. Gandhi, Carlo Zaniolo. ASTRO: A Datalog System for Advanced Stream Reasoning. *CIKM*, 2018.
- Manoj Dareddy[*], Ariyam Das[*], Junghoo Cho, Carlo Zaniolo. How Much Are You Willing to Share? A “Poker-Styled” Selective Privacy Preserving Framework for Recommender Systems. *Arxiv Preprint*, 2018 ([*] denotes equal contribution).
- Carlo Zaniolo, Mohan Yang, M. Interlandi, Ariyam Das, A. Shkapsky, T. Condie. Declarative BigData Algorithms via Aggregates and Relational Database Dependencies. *AMW*, 2018.
- Ariyam Das, Ishan Upadhyaya, Xiangrui Meng, Ameet Talwalkar. Collaborative Filtering as a Case-Study for Model Parallelism on Bulk Synchronous Systems. *CIKM*, 2017.
- Shalini Ghosh, Ariyam Das, Phillip Porras, Vinod Yegneswaran, Ashish Gehani. Automated Categorization of Onion Sites for Analyzing the Darkweb Ecosystem. *KDD*, 2017.
- Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, A. Shkapsky, T. Condie. Fixpoint Semantics and Optimization of Recursive Datalog Programs with Aggregates. *TPLP*, 2017.
- Ariyam Das, Carlo Zaniolo. Fast Lossless Frequent Itemset Mining in Data Streams using Crucial Patterns. *SDM*, 2016.
- Wenchao Yu, Ariyam Das, J. Wood, Wei Wang, Carlo Zaniolo, P. Luo. Max-Intensity: Detecting Competitive Advertiser Communities in Sponsored Search Market. *IEEE ICDM*, 2015.

CHAPTER 1

Introduction

Prolog's success with advanced applications demonstrated the ability of declarative languages to express powerful algorithms as "logic + control." Then, after observing that in relational database management systems, "control" and optimization are provided by the system implicitly, Datalog researchers sought the ability to express powerful applications using only declarative logic-based constructs. After initial successes, which e.g., led to the introduction of recursive queries in SQL, Datalog encountered two major obstacles as data analytics grew increasingly complex: (i) lack of expressive power at the language level, and (ii) lack of scalability and performance at the system level.

These problems became clear with the rise of more complex descriptive and predictive Big-Data analytics. For instance, the in-depth study of data mining algorithms [STA00] carried out in the late 90s by the IBM DB2 team concluded that the best way to carry out advanced predictive analytics, beyond simple classification models like Naive Bayesian classifiers, is to load the data from an external database into main memory and then write an efficient implementation in a procedural language to mine the data from the cache. However, recent advances in architectures supporting in-memory parallel and distributed computing have led to the renaissance of powerful declarative-language based systems like *LogicBlox* [ACG15], *BigDatalog* [SYI16], *SociaLite* [SPS13], *BigDatalog-MC* [YSZ17], *Myria* [WBH15] and *RASQL* [GWM19] that can scale efficiently on multi-core machines as well as on distributed clusters. In fact, some of these general-purpose systems like *BigDatalog* and *RASQL* have outperformed commercial graph engines like

GraphX for many classical graph analytic tasks in terms of performance and scalability. This has brought the focus back on to the first challenge (i) – how to express the wide spectrum of predictive and prescriptive analytics in declarative query languages. This problem has assumed great significance today with the revolution of machine learning driven data analytics, since "in-database analytics" can save data scientists considerable time and effort, which is otherwise repeatedly spent in extracting features from databases via multiple joins, aggregations and projections and then exporting the dataset for use in external learning tools to generate the desired analytics [ANN18]. Modern researchers have worked toward this "in-database analytics" solution by writing *user-defined functions* in procedural languages or using other low-level system interfaces, which the query engines can then import [FKR12]. However this approach raises three fundamental challenges:

- **Productivity and Developability:** Writing efficient implementations of advanced data analytic applications (or even modifying them) using low-level system APIs require data science knowledge as well as system engineering skills. This can strongly hinder the productivity of data scientists and thus the development of these advanced applications.
- **Portability:** User-defined functions written in one system-level API may not be directly portable to other systems where the architecture and underlying optimizations differ.
- **Optimization:** Here, the application developer is entrusted with the responsibility to write an optimal user-defined function, which is contrary to the work and vision of the database community in the 90s [IM96] that aspired for a high-level declarative language like SQL supported by implicit query optimization techniques.

Moving away from static databases, these aforementioned problems also existed and compounded for data stream systems. In fact, the rise of the Internet of Things (IoT) and the recent focus on a gamut of 'Smart City' initiatives world-wide have pushed for new advances in data stream systems to (1) support complex ML/AI-powered data analytics and evolving graph applications as continuous queries, and (2) deliver fast and scalable processing on large data streams. Unfortunately current continuous query languages (CQL) lack the features and constructs needed

to support the more advanced applications. For example recursive queries are now part of SQL, Datalog, and other query languages, but they are not supported by most CQLs, a fact that caused a significant loss of expressive power, which is further aggravated by the limitation that only non-blocking queries can be supported in data stream systems.

All these challenges can be largely addressed, if we have a powerful language-driven unified framework that can achieve scalability through parallelization independent of the underlying architecture, and where users can express even complex analytics using concise high-level declarative queries, which can be easily ported across multiple platforms. To that effect, in this thesis, we attack this problem at three different fronts and make the following contributions:

Theoretical and Semantic Level. Supporting aggregates within recursion in logic programs is an old and difficult problem primarily due to the non-monotonic nature of aggregates. We address this by introducing the notion of Pre-Mappability ($\mathscr{P}reM$) and show that, in many applications, aggregates can be used inside recursion to optimize the perfect-model semantics of aggregate-stratified programs. This allows us to combine the best of both worlds: we can preserve the declarative formal semantics of such logic programs and yet achieve a highly efficient operational semantics at the same time. Additionally, we show that with $\mathscr{P}reM$, a wide spectrum of classical algorithms of practical interest, ranging from graph analytics and dynamic programming based optimization problems to data mining and machine learning applications, can be succinctly written in declarative languages by using aggregates in recursion. We further illustrate how a data scientist or an application developer can very easily verify the semantic correctness of such declarative programs, which provide these complex ML/AI-powered data analytic solutions. Furthermore, we also demonstrate using different case studies, how the semantics of $\mathscr{P}reM$ allows the declarative specification of many complex continuous queries that can then be efficiently executed over data streams.

Performance and Ease-of-Use. We show that $\mathscr{P}reM$-optimized declarative programs offer better usability as compared to *locally stratified* programs (quasi-declarative) and are more easily conducive to scalable implementations on parallel and distributed platforms. In fact, $\mathscr{P}reM$ can

be easily incorporated into the data-parallel computation plans of different distributed systems, irrespective of their synchronization models. In addition, we show that non-linear recursive queries can be evaluated using a hybrid stale synchronous parallel (SSP) model on distributed environments. We experimentally show that recursive query evaluation with $\mathscr{P}reM$ under this relaxed synchronization model can offer significant performance gains.

Robust Online Optimization Techniques. Streaming computations are intrinsically complex in nature as they need to be executed in a non-blocking fashion (without seeing the end of the data) and in real time. Thus streaming algorithms require use of lightweight compact data structures, which incur low storage overhead and can be easily maintained over time as new data streams in. In addition, it is also desirable to exploit statistical approximation techniques that can offer early response with reasonably high confidence and are also amenable to parallelization with high resource utilization. In particular, we present two case studies to highlight this: Firstly, we use the classic problem of frequent itemset mining to empirically show how using a compact representation can reduce query response time by decreasing maintenance cost of the structure as well as the overall mining time. Secondly, we use the example of online decision tree models (Hoeffding Trees) to demonstrate how non-parametric bootstrap can be applied at scale with respect to data streams to maximize resource utilization on multi-core machines and considerably build online models faster.

The rest of this thesis is organized as follows. In Chapter 2, we formally define $\mathscr{P}reM$ and discuss its important properties. Thereafter, we show how $\mathscr{P}reM$ is easily amenable to parallelization and how it offers interesting opportunities for better scalability under relaxed synchronization settings on a distributed environment. Then, in Chapter 3, we analyze the usability, expressivity power and performance of $\mathscr{P}reM$-optimized queries with queries written in quasi-declarative programming methodologies. Next, we describe in Chapter 4 the host of BigData applications that can be expressed declaratively with $\mathscr{P}reM$. Chapters 5 and 6 present robust online optimization strategies with respect to two classic problems: frequent itemset mining and online decision tree construction. Finally, we conclude and discuss future research directions originating from this work in Chapter 7.

CHAPTER 2

Declarative Recursive Computation with $\mathscr{P}reM$

A large class of traditional graph and data mining algorithms can be concisely expressed in Datalog, and other Logic-based languages, once aggregates are allowed in recursion. However, the use of non-monotonic aggregates in recursion raises difficult semantic issues. We propose a property called *Pre-Mappability* ($\mathscr{P}reM$) that solves these semantic issues for most BigData algorithms. In fact, $\mathscr{P}reM$ assures that for a program with aggregates in recursion there is an equivalent aggregate-stratified program. In this chapter, we further show that, by bringing together the formal abstract semantics of stratified programs with the efficient operational one of unstratified programs, $\mathscr{P}reM$ can also facilitate and improve their parallel execution. We prove that $\mathscr{P}reM$-optimized lock-free and decomposable parallel semi-naive evaluations produce the same results as the single executor programs. Thus, $\mathscr{P}reM$ can be assimilated into the data-parallel computation plans of different distributed systems, irrespective of whether these follow bulk synchronous parallel (BSP) or asynchronous computing models. Moreover, $\mathscr{P}reM$ offers an interesting new opportunity for non-linear recursive queries to be evaluated using a hybrid stale synchronous parallel (SSP) model on distributed environments. We provide a formal correctness proof for the recursive query evaluation with $\mathscr{P}reM$ under this relaxed synchronization model and also show the experimental evidence of its benefits.

2.1 Pre-Mappability

In this section we formally define Pre-Mappability ($\mathscr{P}reM$) and present its properties [ZYD16], [ZYI18]. In the remainder of this thesis, we will use Datalog queries as examples, where each rule r has the following form with `H` representing the head atom, `B1`,...,`Bn` denoting the body atoms and the commas separating the goals in the body stand for logical `AND`. Also, any variable appearing *only once* in the rule is denoted by the "_" symbol.

$$r : \texttt{H} \leftarrow \texttt{B1}, \texttt{B2}, ..., \texttt{Bn}.$$

Now, consider the Datalog query in Example 2.1 that computes the shortest path between all pairs of vertices in a graph, given by the relation `arc(X, Y, D)`, where `D` is the distance between source node `X` and destination node `Y`. The `min`⟨`D`⟩ syntax in our example indicates *min* aggregate on the cost variable `D`, while `(X, Y)` refer to the group-by arguments. This head notation for aggregates directly follows from SQL-2 syntax, where cost argument for the aggregate consists of one variable and group-by arguments can have zero or more variables. Rules $r_{2.1.3}$ in the example shows that the aggregate *min* is computed at a stratum higher than the recursive rule ($r_{2.1.2}$).

Example 2.1. *All Pairs Shortest Path*

$$r_{2.1.1} : \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}).$$

$$r_{2.1.2} : \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{path}(\texttt{X},\texttt{Z},\texttt{Dxz}), \texttt{arc}(\texttt{Z},\texttt{Y},\texttt{Dzy}), \texttt{D} = \texttt{Dxz} + \texttt{Dzy}.$$

$$r_{2.1.3} : \texttt{shortestpath}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}).$$

Incidentally, $r_{2.1.3}$ can also be expressed with stratified negation as shown in rules $r_{2.1.4}$ and $r_{2.1.5}$. This guarantees that the program has a perfect-model semantics, although an iterated fixpoint computation of it can be very inefficient and even non-terminating in presence of cycles.

$$r_{2.1.4} : \texttt{shortestpath}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}), \neg\texttt{betterpath}(\texttt{X},\texttt{Y},\texttt{D}).$$

$$r_{2.1.5} : \texttt{betterpath}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}), \texttt{path}(\texttt{X},\texttt{Y},\texttt{Dxy}), \texttt{Dxy} < \texttt{D}.$$

$\mathscr{P}reM$ **Application.** The aforementioned inefficiency can be mitigated with $\mathscr{P}reM$, if the *min*

aggregate can be pushed inside the fixpoint computation, as shown in rules $r_{2.2.1}$ and $r_{2.2.2}$. The following program under $\mathscr{P}reM$ has a stable model semantics and [CDI18] showed that this transformation is indeed equivalence-preserving with an assured convergence to a minimal fixpoint within a finite number of iterations. In other words, without $\mathscr{P}reM$ the shortest path in our example (according to rule $r_{2.1.3}$) is given by the subset of the minimal model (computed from rules $r_{2.1.1}, r_{2.1.2}$) obtained after removing `path` atoms that did not satisfy the *min* cost constraint for a given source-destination pair. However, with $\mathscr{P}reM$, the transfer of *min* cost constraint inside recursion results in an optimized program, where the fixpoint computation is performed more efficiently, eventually achieving the same shortest path values (as those produced in the perfect model of the earlier program) by simply copying the atoms from `path` under the name `shortestpath` (rule $r_{2.2.3}$) after the least fixpoint computation terminates.

$$r_{2.2.1} : \mathtt{path}(\mathtt{X},\mathtt{Y},\mathtt{min}\langle\mathtt{D}\rangle) \leftarrow \mathtt{arc}(\mathtt{X},\mathtt{Y},\mathtt{D}).$$

$$r_{2.2.2} : \mathtt{path}(\mathtt{X},\mathtt{Y},\mathtt{min}\langle\mathtt{D}\rangle) \leftarrow \mathtt{path}(\mathtt{X},\mathtt{Z},\mathtt{Dxz}), \mathtt{arc}(\mathtt{Z},\mathtt{Y},\mathtt{Dzy}), \mathtt{D} = \mathtt{Dxz} + \mathtt{Dzy}.$$

$$r_{2.2.3} : \mathtt{shortestpath}(\mathtt{X},\mathtt{Y},\mathtt{D}) \leftarrow \mathtt{path}(\mathtt{X},\mathtt{Y},\mathtt{D}).$$

Formal Definition of $\mathscr{P}reM$. For a given Datalog program, let P be the rules defining a (set of mutually) recursive predicate(s) and T be the corresponding *Immediate Consequence Operator* (ICO) defined over P. Then, a constraint γ is said to be $\mathscr{P}reM$ to T (and to P) when, for every interpretation I of the program, we have $\gamma(T(I)) = \gamma(T(\gamma(I)))$.

In Example 2.1, the final rule $r_{2.1.3}$ imposes the constraint $\gamma = (\mathtt{X},\mathtt{Y},\mathtt{min}\langle\mathtt{D}\rangle)$ on $I = \mathtt{path}(\mathtt{X},\mathtt{Y},\mathtt{D})$ (representing all possible paths) to eventually yield the shortest path between all pairs of nodes. Thus, the aggregate-stratified program defined by rules $r_{2.1.1} - r_{2.1.3}$ is equivalent to $\gamma(T(I))$ in the definition of $\mathscr{P}reM$. On the other hand, with *min* aggregate pushed inside recursion, recursive rules $r_{2.2.1} - r_{2.2.2}$ represent $\gamma(T(\gamma(I)))$.

$\mathscr{P}reM$ **Properties.** We now discuss some important results about $\mathscr{P}reM$ from [ZYI17]. We refer interested readers to our paper [ZYI17] for the detailed proofs. Let T_γ denote the *constrained immediate consequence operator*, where constraint γ is applied after the ICO T, i.e., $T_\gamma(I) = \gamma(T(I))$.

The following results hold when γ is $\mathscr{P}reM$ to a positive program P with ICO T:

(i). If $I = T(I)$ is a fixpoint for T, then $I' = \gamma(I)$ is a fixpoint for $T_\gamma(I)$, i.e., $I' = T_\gamma(I')$.

(ii). For some integer n, if $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$, then $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is a minimal fixpoint for T_γ and $T_\gamma^{\uparrow n}(\emptyset) = \gamma(T^{\uparrow\omega}(\emptyset))$, where $T^{\uparrow\omega} = \bigcup_{n\geq 1} T^{\uparrow n}$

$\mathscr{P}reM$ **Provability.** We can verify if $\mathscr{P}reM$ holds for a recursive rule by explicitly validating $\gamma(T(I)) = \gamma(T(\gamma(I)))$, i.e., $T_\gamma(I) = T_\gamma(\gamma(I))$ at every iteration of the fixpoint computation. To simplify, this would indicate that we can verify if the *min* constraint can be pushed inside recursion in rule $r_{2.2.2}$ by inserting an additional goal *is_min* in the body of the rule as follows:

$$r'_{2.2.2} : \texttt{path(X,Y,min}\langle\texttt{D}\rangle\texttt{)} \leftarrow \texttt{path(X,Z,Dxz)}, \textbf{is_min}\texttt{((X,Z),Dxz)}, \texttt{arc(Z,Y,Dzy)}, \texttt{D} = \texttt{Dxz} + \texttt{Dzy}.$$

This additional goal in the body pre-applies the constraint γ on I, followed by the application of T_γ operator, i.e., it expresses $T_\gamma(\gamma(I))$. Note, the *is_min* constraint is satisfied by `Dxz`, if it is the minimum value seen yet in the fixpoint computation for the source-destination pair `(X, Z)`. It is also evident that any other distance value between `(X, Z)`, which violates the *is_min* constraint, will also not satisfy the *min* aggregate at the head of the rule, since the additional goal minimizes the sum `D` for each `Dzy`. Thus, this new goal in the body does not alter the ICO mapping defined by the original recursive rule, thereby proving γ is $\mathscr{P}reM$ in this example program. More broadly speaking, these additional goals can be formally defined as "half functional dependencies", borrowing the terminology from classical database theory of Functional and Multi-Valued Dependencies (FDs and MVDs). We next present the formal definition of *half FD* from [ZYI18], which will be used later for our proofs.

Definition 2.1. *(Half Functional Dependency). Let $R(\Omega)$ be a relation on a set of attributes Ω, $X \subset \Omega$ and $A \in \Omega - X$. Considering the domain of A to be totally ordered, a tuple $t \in R$ is said to satisfy the min-constraint is_min$((X),A)$ (denoted as $X \overset{min}{\rightharpoondown} A$), when R contains no tuple with the same X-value and a smaller A-value. Similarly, a tuple $t \in R$ satisfies a max-constraint is_max$((X),A)$ (denoted as $X \overset{max}{\rightharpoondown} A$) if R has no tuple with the same X-value and a larger A-value.*

For any *min* or *max* constraint to be $\mathscr{P}reM$ to a positive program P, the corresponding half FD should hold for the relational view of the relevant recursive predicate across every interpretation I of P, where a relational view for predicate q is defined as $R_q = \{(x_1,...,x_n)|q(x_1,...,x_n) \in I\}$ for a given I. [ZYI18] provides generic templates, based on Functional and Multi-valued Dependencies, for identifying constraints that satisfy $\mathscr{P}reM$.

$\mathscr{P}reM$ with Semi-Naive Evaluation. A naive fixpoint computation for a recursive query trivially generates new atoms from the entire set of atoms available at the end of the last fixpoint iteration. Semi-naive evaluation [SYI16] improves over this naive fixpoint computation with the aid of the following enhancements:

1. At every iteration, track *only* the new atoms produced.
2. Rules are re-written into their differential versions, so that only new atoms are produced and old atoms are never generated redundantly.
3. Ensure step (2) does not generate any duplicate atoms.

For programs where $\mathscr{P}reM$ can be applied, steps (1) and (2) remain identical. However, step (3) is extended so that (i) new atoms produced may not be retained, if they do not satisfy the constraint γ and (ii) existing atoms may get updated and thereafter tracked for the next iteration. For example, new atoms produced from rule $r_{2.2.2}$ are added to the working set and tracked only if a new source-destination (`X, Y`) path is discovered. On the other hand, if the new `path` atom, thus produced, has a smaller distance than the one in the working set, then the distance of the existing `path` atom is updated to satisfy the *min*-constraint. However, if new `path` atoms are generated, which have larger distances, then they are simply ignored. This understanding of $\mathscr{P}reM$ for semi-naive evaluation leads to a case for Stale Synchronous Parallel (SSP) model, where significant communication can be saved by condensing multiple updates into one. This is discussed in detail later in Section 2.5.

2.2 Motivation

The growing interest in Datalog-based declarative systems like *LogicBlox* [ACG15], *BigDatalog* [SYI16], *SociaLite* [SPS13], *BigDatalog-MC* [YSZ17] and *Myria* [WBH15] has brought together important advances on two fronts: (i) Firstly, Datalog, with support for aggregates in recursion [MSZ13], has sufficient power to express succinctly declarative applications ranging from complex graph queries to advanced data mining tasks, such as frequent pattern mining and decision tree induction [CDI18]. (ii) Secondly, modern architectures supporting in-memory parallel and distributed computing can deliver scalability and performance for this new generation of Datalog systems.

For example *BigDatalog* (bulk synchronous parallel processing on shared-nothing architecture), *BigDatalog-MC* (lock-free parallel processing on shared-memory multicore architecture), *Myria* (asynchronous processing on shared-nothing architecture) spearheaded the system-level scheduling, planning and optimization for different parallel computing models. This line of work was quite successful for Datalog, and also for recursive SQL queries that have borrowed this technology [GWM19]). Indeed, our recent general-purpose Datalog systems surpassed commercial graph systems like GraphX on many classical graph queries in terms of performance and scalability [SYI16].

Much of the theoretical groundwork contributing to the success of these parallel Datalog systems was laid out in the 90s. For example, in their foundation work [GST92] investigated parallel *coordination-free* (asynchronous) bottom-up evaluations of simple linear recursive programs (without any aggregates). In fact, many recent works have pushed this idea forward under the broader umbrella of *CALM conjecture* (Consistency And Logical Monotonicity) [ANV13] which establishes that monotonic Datalog (Datalog without negation or aggregates) programs can be computed in an *eventually consistent, coordination-free* manner [Ame14], [AKN15]. This line of work led to the asynchronous data-parallel (for *Myria*) and lock-free evaluation plans for many of the aforementioned systems (e.g. *BigDatalog-MC*). Simultaneously, another branch of research about 'parallel correctness' for simple non-recursive conjunctive queries [AGK17] focused on optimal

data distribution policies for re-partitioning the initial data under Massively Parallel Communication model (MPC). However, notably, this theoretical groundwork left out programs using aggregates in recursion, for which the existence of a formal semantics could not be guaranteed. But, this situation has changed recently with the advent of the notion of *Pre-Mappability* ($\mathscr{P}reM$) that has made possible the use of aggregates in recursion to express efficiently a large range of applications (as shown later in Chapter 4). As discussed before, $\mathscr{P}reM$ enables the use of non-monotonic aggregates and pre-mappable constraints inside recursion, while preserving the formal declarative semantics of aggregate-stratified programs and guaranteeing their equivalence. This is very encouraging, since unlike more complex non-monotonic semantics (as discussed later in Chapter 3), stratification is a syntactic condition that is easily checked by users (and compilers), who know that the presence of a formal declarative semantics guarantees the portability of their applications over multiple platforms. Naturally, we would like to examine the applicability of $\mathscr{P}reM$ under a parallel and distributed setting and analyze its potential gains using the rich models of parallelism previously proposed for Datalog and other logic systems.

In this chapter, therefore, we further examine how $\mathscr{P}reM$ interacts under a parallel setting, and address the question of whether it can be incorporated into the parallel evaluation plans on shared-memory and shared-nothing architectures. Furthermore, the current crop of Datalog systems supporting aggregates in recursion have only explored Bulk Synchronous Parallel (BSP) and asynchronous distributed computing models. However, the new emerging paradigm of Stale Synchronous Parallel (SSP) processing model [CCH14] has shown to speed up big data analytics and machine learning algorithm execution on distributed environments [LKZ14], [HCC13] with *bounded staleness*. SSP processing allows each worker in a distributed setting to see and use another worker's obsolete (stale) intermediate solution, which is out-of-date only by a limited (bounded) number of epochs. On the contrary, in a BSP model every worker coordinates at the end of each round of computation and sees each others' current intermediate results. This relaxation of the synchronization barrier in a SSP model can reduce idle waiting of the workers (time spent waiting to synchronize), particularly when one or more workers (stragglers) lag behind others in terms of computation. Thus, in this chapter, we also explore if declarative recursive computa-

tion can be executed under the loose consistency model of SSP processing and if it has the same convergence as that under a BSP processing framework. To our surprise, we find $\mathscr{P}$*reM* dovetails excellently with SSP model for a class of non-linear recursive queries with aggregates, which are not embarrassingly parallel and still require some coordination between the workers to reach eventual consistency [IT18]. Thus, the contributions can be summarized as follows:

- We show that $\mathscr{P}$*reM* is applicable to parallel bottom-up semi-naive evaluation plan, terminating at the same minimal fixpoint as the corresponding single executor based sequential execution.
- We further show how recursive query evaluation with $\mathscr{P}$*reM* can operate effectively under a SSP distributed model.
- Finally, we discuss the merits and demerits of a SSP model with empirical results on some recursive query examples, thus opening up an interesting direction for future research.

2.3 An Overview of Parallel Bottom-Up Evaluation

One of the early foundational works that established a standard technique to parallelize bottom-up evaluation of *linear recursive* queries was presented in [GST92]. The authors proposed a *substitution partitioned parallelization* scheme, where the set of possible ground substitutions, i.e., the base (extensional database) and derived relation (intensional database) atoms in the Datalog program are disjointedly partitioned, using a hash-based discriminating function, so that each partition of possible ground substitutions is mapped to exactly one of the parallel workers. The entire computation is then divided among all the workers, operating in parallel, where each worker only processes the partition of ground substitutions mapped to it during the bottom-up semi-naive evaluation. Since, each worker operates on a distinct non-overlapping partition of ground substitutions, no two workers perform the same or redundant computation, i.e., this scheme is *non-redundant*. Formally, if $v(r)$ is a non-repetitive sequence of variables appearing in the body of rule r and $\mathscr{W}$ denotes a finite set of parallel workers, then $h : v(r) \longrightarrow \mathscr{W}$ is a discriminating hash function that

divides the workload by assigning the ground substitution and corresponding processing to exactly one worker. The workers can send and receive information (ground instances from partially computed derived relations) to and from other workers to finish the assigned computation tasks. Ganguly et al. summarized the correctness of this parallelization scheme with the following result:

Correctness of Partitioned Parallelization Scheme. Let P be a recursive Datalog program to be executed over $\mathscr{W}$ workers. Under the partitioned parallelization scheme, let Q_i be the program to be executed at worker i and let $Q = \bigcup_{1 \leq i \leq \mathscr{W}} Q_i$. Then, for every interpretation, the least model of the recursive relation in Q is identical to the least model obtained from the sequential execution of P.

Note, the above parallelization strategy did not involve aggregates in recursion. But, nevertheless it was of significant consequence, since the scheme has been extended to derive *lock-free parallel* plans for shared-memory architectures as well as *sharded data parallel decomposable* plans for shared-nothing distributed architectures to parallelize bottom-up semi-naive evaluation of Datalog programs. We discuss them next with examples.

Shared-Memory Architecture. A trivial hash-based partitioning, as described above, can often lead to conflicts between different workers on a shared-memory architecture[1]. This can be prevented with the implementation of classical locks to resolve read-write conflicts. However, recently, [YSZ15] proposed a hash partitioning strategy based on *discriminating sets* that allows *lock-free* parallel evaluation of a broad class of generic queries including non-linear queries. We illustrate this with our running all pairs shortest path example.

Assume the relations `arc`, `path` and `shortestpath` from example 2.1 (rules $r_{2.1.1}$ – $r_{2.1.3}$) are partitioned by the first column[2] (i.e., the source vertex), using a hash function h that maps the source vertex to an integer between 1 to $\mathscr{W}$, latter denoting the number of workers. Now, a worker i can execute the following program in parallel:

[1] For example, two distinct workers may update a `path` atom for the same (`X`, `Y`) pair in rule $r_{2.1.2}$, if the hashing is done based on the ground instances of the sequence {`X`, `Z`, `Dxz`, `Z`, `Y`, `Dzy`} or even on the sequence {`X`, `Z`, `Y`}.

[2] The first attribute forms a *discriminating set* that is used for partitioning.

$r_{2.3.1}$: $\texttt{path(X,Y,D)} \leftarrow \texttt{arc(X,Y,D)}, \texttt{h(X)} = \texttt{i}.$

$r_{2.3.2}$: $\texttt{path(X,Y,D)} \leftarrow \texttt{path(X,Z,Dxz)}, \texttt{arc(Z,Y,Dzy)}, \texttt{D} = \texttt{Dxz} + \texttt{Dzy}, \texttt{h(X)} = \texttt{i}.$

$r_{2.3.3}$: $\texttt{shortestpath(X,Y,min}\langle\texttt{D}\rangle\texttt{)} \leftarrow \texttt{path(X,Y,D)}, \texttt{h(X)} = \texttt{i}.$

1. The ith worker executes rule $r_{2.3.1}$ by reading from the ith partition of arc.

2. Once all the workers finish step (1), the ith worker begins semi-naive evaluation with rule $r_{2.3.2}$, where it reads from the ith partition of `path`, joins with the corresponding atoms from the `arc` relation, which is shared across all the workers, and then writes new atoms into the same i^{th} partition of `path`.

3. Once all the workers finish step (2), the semi-naive evaluation proceeds to the next iteration and repeats step (2) till the least fixpoint is reached.

4. In the final step, the ith worker computes the `shortestpath` for the ith partition.

5. All the `shortestpath` data pooled across the workers produce the final query result.

It is easy to observe that the above parallel execution does not require any locks, since each worker is writing to exactly one partition and no two workers are writing to the same partition. We formally define the lock-free parallel bottom-up evaluation scheme next.

Definition 2.2. *(Lock-free Parallel Bottom-up Evaluation). Let P be a recursive Datalog program to be executed over $\mathscr{W}$ workers and let T be the corresponding ICO for the sequential execution of P. Under the lock-free parallel plan executed over $\mathscr{W}$ workers, let Q_i be the program to be executed at worker i, producing an interpretation I_i of the recursive predicate with the corresponding ICO T_i. Then, for every input of base relations, we have, $T_i^{\uparrow\omega}(\emptyset) \bigcap T_j^{\uparrow\omega}(\emptyset) = \emptyset$ for $1 \leq i, j \leq \mathscr{W}$, $i \neq j$. It also follows from the correctness of partitioned parallelization scheme that $\bigcup\limits_{1 \leq i \leq \mathscr{W}} T_i^{\uparrow\omega}(\emptyset) = T^{\uparrow\omega}(\emptyset)$.*

The underlying strategy of a lock-free parallel plan to use disjointed data partitions have also been adopted to execute data-parallel distributed bottom-up evaluations, as explained next.

Shared-Nothing Architecture. Distributed systems like *BigDatalog* [SYI16] also divide the entire dataset into disjointed data shards in an identical manner as the lock-free partitioning technique described above. Each data shard resides in the memory of a worker and this partitioning scheme reduces the data shuffling required across different workers [SYI16]. In the context of shared-nothing architecture, this sharding scheme and subsequent distributed bottom-up evaluation is termed as a *decomposable plan* [SYI16], [GWM19]. In the rest of this chapter, we will use the term 'lock-free parallel plan' in the context of shared-memory architecture and 'parallel decomposable plan' in the context of distributed environment for clarity.

Distributed systems like *BigDatalog* and *SociaLite* [SPS13] perform the fixpoint computation under BSP model with synchronized iterations. However, note that, if each node caches the `arc` relation, then each node can operate independently without any *co-ordination* or *synchronization* with other nodes (i.e., step 3 listed before in the lock-free evaluation plan becomes unnecessary). The *Myria* system follows this asynchronous computing model for the query evaluation. Interestingly, [GST92] showed that only a subclass of linear recursive queries[3] can be executed in a *co-ordination free* manner or asynchronously. Thus, for a large class of non-linear and even many linear recursive queries (e.g. same generation query [GST92]), BSP computing model has been the only viable option.

2.4 Parallel Evaluation with $\mathscr{P}reM$

In this section, we now examine if $\mathscr{P}reM$ can be easily integrated into the *lock-free parallel* and *parallel decomposable bottom-up evaluation* plans that have been widely adopted across shared-memory and shared-nothing architectures for a broad range of generic queries. We next provide some interesting theoretical results.

Lemma 2.3. *Let $R(\Omega)$ be a relation defined over a set of attributes Ω, where $X \subset \Omega$ and $A \in \Omega - X$. For a subset S of X ($S \subseteq X$), if R is divided into k disjoint subsets $R_1, R_2, ..., R_k$ using a hash function $h : S \to k$ such that R_i is defined as $R_i = \{e | e \in R \wedge h(e[S]) = i\}$, then a tuple $t \in R$ satisfying*

[3] The dataflow graph corresponding to the linear recursive query must have a cycle.

$X \overset{min}{\rightharpoondown} A$ *(or,* $X \overset{max}{\rightharpoondown} A$*) will also satisfy* $X \overset{min}{\rightharpoondown} A$ *(or,* $X \overset{max}{\rightharpoondown} A$ *respectively) over* R_i *and vice versa, where* $h(t[S]) = i$*.*

Proof. This follows directly from the fact that since $S \subseteq X$, for any two tuples $t_1, t_2 \in R$, if $t_1[X] = t_2[X]$, then $t_1[S] = t_2[S]$, i.e., any two tuples with the same X-value will be mapped into the same partition, decided by their common S-value. Since, all tuples with the same X-value belong to a single partition, any tuple $t \in R_i$ will satisfy $X \overset{min}{\rightharpoondown} A$ (or, $X \overset{max}{\rightharpoondown} A$) over both R and R_i. □

Theorem 2.4. *Let P be a recursive Datalog program, T be its corresponding ICO and let the constraint γ be $\mathscr{P}reM$ to T and P, resulting in the constrained ICO T_γ. Let P be executed over $\mathscr{W}$ workers under a lock-free parallel (or parallel decomposable) bottom-up evaluation plan, where Q_i is the program executed at worker i and T_i be the corresponding ICO defined over Q_i. If the group-by arguments used for the γ constraint also contain the discriminating set used for partitioning in the lock-free parallel (or parallel decomposable) plan, then:*

(i). *γ is also $\mathscr{P}reM$ to T_i and Q_i, for $1 \leq i \leq \mathscr{W}$.*

(ii). *For some integer n, if $T_\gamma^{\uparrow n}(\emptyset)$ is the minimal fixpoint for T_γ, then $T_\gamma^{\uparrow n}(\emptyset) = \bigcup_{1\leq i \leq \mathscr{W}} T_{i_\gamma}^{\uparrow n}(\emptyset)$, where T_{i_γ} denotes the constrained ICO with respect to T_i.*

Proof. The proof for (i) follows trivially from *Lemma* 2.3 and the $\mathscr{P}reM$ provability technique discussed earlier in Section 2.1.

Since, γ is $\mathscr{P}reM$ to T and P, $T_\gamma^{\uparrow n}(\emptyset) = \gamma(T^{\uparrow \omega}(\emptyset))$ according to the properties of $\mathscr{P}reM$. Similarly, since for $1 \leq i \leq \mathscr{W}$, γ_i is $\mathscr{P}reM$ to T_i and Q_i (from (i) of *Theorem* 2.4), $T_{i_\gamma}^{\uparrow n_i}(\emptyset) = \gamma(T_i^{\uparrow \omega}(\emptyset))$, for some integer n_i, where $T_{i_\gamma}^{\uparrow n_i}(\emptyset) = T_{i_\gamma}^{\uparrow (n_i+1)}(\emptyset)$ is the minimal fixpoint for T_{i_γ}. Thus, $\bigcup_{1\leq i \leq \mathscr{W}} T_{i_\gamma}^{\uparrow n_i}(\emptyset) = \bigcup_{1\leq i \leq \mathscr{W}} \gamma(T_i^{\uparrow \omega}(\emptyset))$. Now, γ constraints are also trivially $\mathscr{P}reM$ to union over disjoint sets [ZYI17], i.e., $\gamma(\bigcup_{1\leq i \leq \mathscr{W}} T_i^{\uparrow \omega}(\emptyset)) = \bigcup_{1\leq i \leq \mathscr{W}} \gamma(T_i^{\uparrow \omega}(\emptyset))$. Also recall from the definition of lock-free parallel (or parallel decomposable) plan that $\bigcup_{1\leq i \leq \mathscr{W}} T_i^{\uparrow \omega}(\emptyset) = T^{\uparrow \omega}(\emptyset)$. Combining these aforementioned equalities, we get,

$T_\gamma^{\uparrow n}(\emptyset) = \gamma(T^{\uparrow \omega}(\emptyset)) = \gamma(\bigcup_{1\leq i \leq \mathscr{W}} T_i^{\uparrow \omega}(\emptyset)) = \bigcup_{1\leq i \leq \mathscr{W}} \gamma(T_i^{\uparrow \omega}(\emptyset)) = \bigcup_{1\leq i \leq \mathscr{W}} T_{i_\gamma}^{\uparrow n_i}(\emptyset).$

Since, $T_{i_\gamma}^{\uparrow n_i}(\emptyset)$ is the minimal fixpoint with respect to T_{i_γ}, for $n > n_i$, $T_{i_\gamma}^{\uparrow n}(\emptyset) = T_{i_\gamma}^{\uparrow n_i}(\emptyset)$. Therefore, $T_\gamma^{\uparrow n}(\emptyset) = \bigcup_{1 \leq i \leq \mathscr{W}} T_{i_\gamma}^{\uparrow n}(\emptyset)$. □

Thus, following *Theorem* 2.4, we can push the *min* constraint within the parallel recursive plan expressed by rules $r_{2.3.1} - r_{2.3.3}$ and rewrite them for worker i as follows:

$r_{2.4.1}: \texttt{path}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}), \texttt{h}(\texttt{X}) = \texttt{i}.$

$r_{2.4.2}: \texttt{path}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{path}(\texttt{X},\texttt{Z},\texttt{Dxz}), \texttt{arc}(\texttt{Z},\texttt{Y},\texttt{Dzy}), \texttt{D} = \texttt{Dxz} + \texttt{Dzy}, \texttt{h}(\texttt{X}) = \texttt{i}.$

$r_{2.4.3}: \texttt{shortestpath}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}), \texttt{h}(\texttt{X}) = \texttt{i}.$

Thus, we observe that pre-mappable constraints can be also easily pushed inside parallel lock-free (or parallel decomposable) evaluation plans of recursive queries to yield the same minimal fixpoint, yet making them computationally more efficient and safe. Thus, $\mathscr{P}$*reM* can be easily incorporated into the parallel computation plans (equivalent to rules $r_{2.4.1} - r_{2.4.3}$) of different systems like *BigDatalog-MC*, *BigDatalog* and *Myria*, irrespective of whether they use (1) shared-memory or shared-nothing architecture, or (2) they follow BSP or asynchronous computing models.

2.5 A Case for Relaxed Synchronization

We now consider a non-linear query, which is equivalent to the linear all pairs shortest path program with the application of $\mathscr{P}$*reM* (rules $r_{2.2.1} - r_{2.2.3}$). Since this is a non-linear query (rules $r_{2.5.1} - r_{2.5.3}$), this program *cannot* be executed in a *coordination-free* manner or *asynchronously* following the technique described in [GST92].

$r_{2.5.1}: \texttt{path}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}).$

$r_{2.5.2}: \texttt{path}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{path}(\texttt{X},\texttt{Z},\texttt{Dxz}), \texttt{path}(\texttt{Z},\texttt{Y},\texttt{Dzy}), \texttt{D} = \texttt{Dxz} + \texttt{Dzy}.$

$r_{2.5.3}: \texttt{shortestpath}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}).$

However, as shown in [YSZ15], a simple query rewriting technique can produce an equivalent parallel decomposable evaluation plan for this non-linear query. Rules $r_{2.6.1} - r_{2.6.4}$ show the equivalent decomposable program, which can be executed by worker i on a distributed system following a bulk synchronous parallel model. In this following decomposable evaluation plan,

there is a mandatory synchronization step (rule $r_{2.6.3}$), where each worker i (operating on the ith partition) copies the new atoms or updates in `path` produced during the semi-naive evaluation from rule $r_{2.6.2}$ to $\mathtt{path}^{(1)}$ and the new $\mathtt{path}^{(1)}$ is then sent to other workers so that they can use it in the evaluation of rule $r_{2.6.2}$ in the next iteration.

$$r_{2.6.1}: \mathtt{path(X,Y,min\langle D\rangle)} \leftarrow \mathtt{arc(X,Y,D), h(X) = i.}$$

$$r_{2.6.2}: \mathtt{path(X,Y,min\langle D\rangle)} \leftarrow \mathtt{path(X,Z,Dxz), path^{(1)}(Z,Y,Dzy), D = Dxz + Dzy, h(X) = i.}$$

$$r_{2.6.3}: \mathtt{path^{(1)}(X,Y,min\langle D\rangle)} \leftarrow \mathtt{path(X,Y,D), h(X) = i.}$$

$$r_{2.6.4}: \mathtt{shortestpath(X,Y,D)} \leftarrow \mathtt{path(X,Y,D), h(X) = i.}$$

In a bulk synchronous distributed computing model, the communication between the workers in each iteration can be considerably more expensive than the local computation performed by each worker due to the bottleneck of network bandwidth. We now investigate if we can relax this synchronization constraint at every iteration.

Under a *stale synchronous parallel* (SSP) model, a worker i can use an obsolete or stale version of $\mathtt{path}^{(1)}$ that omits some recent updates, produced by other workers, for its local computation. In particular, a worker using $\mathtt{path}^{(1)}$ at iteration c will be able to use all the atoms and updates generated from iteration 0 to $c-s-1$, $s \geq 0$ is a user-specified threshold for controlling the staleness. In addition, the worker's stale $\mathtt{path}^{(1)}$ may have atoms or updates from iteration beyond $c-s-1$, i.e., from iteration $c-s$ to $c-1$ (although this is not guaranteed). The intuition behind this is that in a SSP model, a worker for its local computation should be able to see and use its own updates at every iteration, in addition to seeing and using as many updates as possible from other workers, with the constraint that any updates older than a given age are not missed. This is the *bounded staleness* constraint [CHK13]. This leads to two advantages:

1. Workers spend more time performing actual computation, rather than idle waiting for other workers to finish. This can be very helpful, when there are straggling workers present, which lag behind others in an iteration. In fact in distributed computing, stragglers present an acute problem since they can occur for several reasons like hardware differences [KTG11],

system failures [AKG10], skewed data distribution or even from software management issues and program interruptions caused from garbage collections or operating system noise, etc. [BIY06].

2. Secondly, workers can end up communicating less than under a BSP model. This is primarily because under $\mathscr{P}reM$, each worker can condense several updates computed from different local iterations into a single update before eventually sending it to other workers.

We illustrate the above advantages through an example. Figure 2.1 shows a toy graph which is distributed across two workers: (i) all edges incident on nodes 1-4 are available on *worker 0*, (ii) and the rest of the edges reside on *worker 1*. Now consider the shortest path between nodes 4 and 8, given by the path 4-3-2-1-5-6-7-8, which spans across 7 hops. The parallel program defined by rules $r_{2.6.1} - r_{2.6.4}$ with BSP processing would require at least three synchronized iterations to reach to the least fixpoint by semi-naive evaluation. Now consider *worker 1* to be a straggling node that lags behind *worker 0* during the computation because of hardware differences. Thus, *worker 0* spends significant time idle waiting for *worker 1* to complete, as shown in Figure 2.2a. But in this example, the shortest path between nodes 4 and 8 changes because of two aspects: (1) the shortest path between nodes 4 and 1 changes and (2) the shortest path between nodes 5 and 8 changes. Both of these computations can be done independently on the two workers and *worker 0* needs to know the eventual shortest path between nodes 5 and 8 calculated by *worker 1* and vice versa. It is important to note that this will only work if each worker can use the most recent *local* updates (newest atoms) generated by itself. In other words, *worker 0* should be able to see the changes of the shortest path between node 4 and node 1 in every iteration (which is generated locally) and use a stale (obsolete) knowledge about the shortest path between nodes 5 and 8 (as sent by *worker 1* earlier). This stale synchronization model is summarized in Figure 2.2b.

In this same example, note how the minimum cost for the path between node 1 and node 4 (computed by *worker 0*) changes in every iteration: (i) in the first iteration, the minimum cost was 10 given by the edge between node 1 and node 4, (ii) in the next iteration, the minimum cost drops to 7 given by the path 1-3-4 and (iii) in the third iteration the final minimum cost of 5 is given by

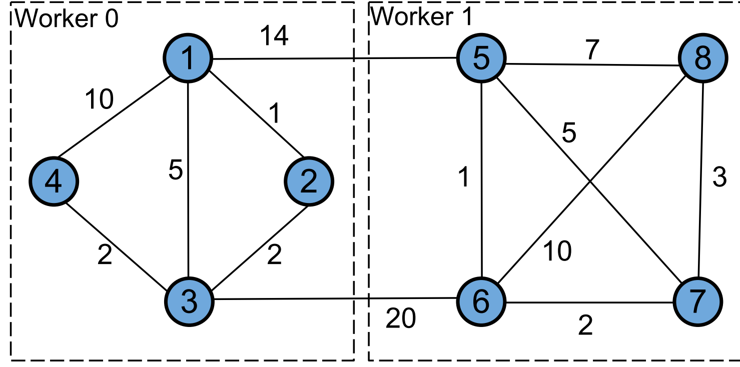


Figure 2.1: A toy graph distributed across two workers.

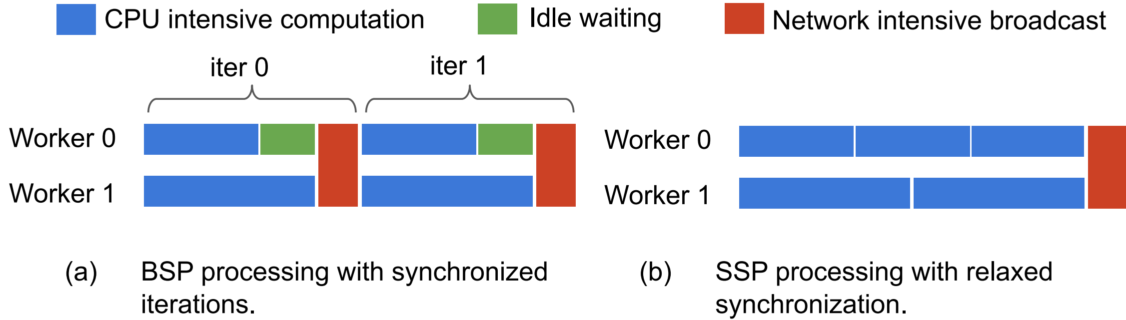


Figure 2.2: BSP vs. SSP model for evaluating *all pairs shortest path* query on two workers.

the sequence 1-2-3-4. In a BSP model, each of this update generated in every iteration needs to be communicated to all the remaining workers. However, in a SSP model due to the advantage of this staleness, these multiple updates from different local iterations can be condensed into one most recent update, which is then sent to other workers. In other words, SSP with $\mathscr{P}reM$ may skip sending some updates to remote workers, thus saving communication time.

Figure 2.3 formally presents the SSP processing based bottom-up evaluation plan for the non-linear all pairs shortest path example given by rules $r_{2.6.1} - r_{2.6.4}$. If the evaluation is executed over a distributed system of $\mathscr{W}$ workers, Figure 2.3 depicts the execution plan for a worker i. A coordinator marks the completion of the overall evaluation process by individually tracking the termination of each of the worker's task. For simplicity and clarity, we have used the naive fixpoint computation to describe the evaluation plan instead of using the optimized differential fixpoint algorithm. The γ used in the Figure 2.3 denotes the *min* constraint. Step (3) in this evaluation plan shows how worker i uses stale knowledge from other workers j (denoted by $\mathtt{path}_{\mathtt{j}}^{(\mathtt{r}')}$) during the recursive rule evaluation, shown by step (4). It is also important to note that in step (4), each worker

i is also using the most recent atoms generated by itself (denoted by $\mathtt{path}_\mathtt{i}^{(\mathtt{r})}$) for the evaluation. The condition in step (6) allows each local computation on worker i to reach local fixpoint or move further by at least $\mathscr{T}$ iterations. Thus, each worker i can condense multiple updates generated within these $\mathscr{T}$ iterations due to $\mathscr{P}reM$ into a single update. Finally, step (9) ensures that if any worker falls beyond the user-defined *staleness bound*, then other workers wait for it to catch up within the desired staleness level before starting their local computations again. We next present some theoretical and empirical results about the SSP model based bottom-up evaluation.

1: $\mathtt{path}_\mathtt{i}^{(0)}(\mathtt{X},\mathtt{Y},\mathtt{D}) := \{(\mathtt{X},\mathtt{Y},\mathtt{D}) | \mathtt{arc}(\mathtt{X},\mathtt{Y},\mathtt{D})\}$, $\mathtt{path}_\mathtt{j}^{(0)}(\mathtt{X},\mathtt{Y},\mathtt{D}) := \emptyset\ \forall i \neq j$, $\mathtt{r} = 0$, $\mathtt{s} = 0$
2: **repeat**
3: $\quad \mathtt{path}_\mathtt{j}^{(\mathtt{r}')}(\mathtt{X},\mathtt{Y},\mathtt{D})$:=Last received $\mathtt{path}_\mathtt{j}$ by worker $\mathtt{i}, \forall i \neq j$.
4: $\quad \mathtt{path}_\mathtt{i}^{(\mathtt{r}+1)}(\mathtt{X},\mathtt{Y},\mathtt{D}) := \gamma\Bigg(\Big(\bigcup_{i\neq j} \mathtt{path}_\mathtt{i}^{(\mathtt{r})}(\mathtt{X},\mathtt{Z},\mathtt{Dxz}) \bowtie \mathtt{path}_\mathtt{j}^{(\mathtt{r}')}(\mathtt{Z},\mathtt{Y},\mathtt{Dzy})\Big) \cup \Big(\mathtt{path}_\mathtt{i}^{(\mathtt{r})}(\mathtt{X},\mathtt{Z},\mathtt{Dxz}) \bowtie \mathtt{path}_\mathtt{i}^{(\mathtt{r})}(\mathtt{Z},\mathtt{Y},\mathtt{Dzy})\Big)\Bigg)$
5: $\quad \mathtt{r} := \mathtt{r}+1$, $\mathtt{s} := \mathtt{s}+1$
6: **until** $s < \mathscr{T}$ and $\mathtt{path}_\mathtt{i}^{(\mathtt{r})} \neq \mathtt{path}_\mathtt{i}^{(\mathtt{r}-1)}$
7: $\mathtt{s} := 0$
8: Send $\mathtt{path}_\mathtt{i}^{(\mathtt{r})}$ to other workers.
9: **if** for any worker $j, r - r' >$ staleness bound **then**
10: $\quad$ Wait for a new update from worker j before continuing
11: **if** $\mathtt{path}_\mathtt{i}^{(\mathtt{r})} \neq \mathtt{path}_\mathtt{i}^{(\mathtt{r}-1)}$ or a new update has been received from worker j **then**
12: $\quad$ repeat from Step (2)
13: **else**
14: $\quad$ Send a `finish` message to coordinator.
15: $\quad$ **if** any new update is received from worker j **then**
16: $\quad\quad$ Send a `resume` message to coordinator.
17: $\quad\quad$ Repeat from step (2).

Figure 2.3: SSP based bottom-up evaluation plan executed by worker i for computing all pairs shortest path.

2.6 Bottom-up Evaluation with SSP Processing

We now establish some theoretical guarantees for the recursive query evaluation with $\mathscr{P}reM$ constraints under a SSP model.

Definition 2.5. *(γ-Cover). Let P be a positive recursive Datalog program with T as its corresponding ICO. Let a constraint γ be defined over the recursive predicate on a set of k group-by arguments, denoted by $G_1, G_2, ..., G_k$ with the cost-argument denoted as C. Let γ be also $\mathscr{P}$reM to T and P. Let there be two sets S_1 and S_2, both of which contain tuples of the form $\{(g_1, g_2, ..., g_k, c) | g_i \in G_i \forall 1 \leq i \leq k, c \in \mathbb{R}\}$, where $\mathbb{R}$ represents the set of real numbers. Now, S_1 is defined as the γ-cover for S_2, if for every tuple $t \in S_2$, there exists only one tuple $t' \in S_1$ such that (i) $t'[G] = t[G]$ and (ii) $\gamma(t'[C], t[C]) = t'[C]$.*

It is important to note from the above definition that if S_1 is the γ-cover for S_2, then there can exist a tuple $t \in S_1$, such that $t[G] \neq t'[G]\ \forall t' \in S_2$ but the converse is never true.

Lemma 2.6. *Let P be a recursive Datalog program, T be its corresponding ICO and let the constraint γ be $\mathscr{P}$reM to T and P, resulting in the constrained ICO T_γ. Now, for any pair of positive integers m, n, where $m \geq n$, $T_\gamma^{\uparrow m}(\emptyset)$ is a γ-cover for $T_\gamma^{\uparrow n}(\emptyset)$.*

Proof. This directly follows from the fact that any atom in $T_\gamma^{\uparrow n}(\emptyset)$ with cost c can only exist in $T_\gamma^{\uparrow m}(\emptyset)$ with updated cost c', if $c = c'$ or $\gamma(c, c') = c'$. Note if $c = c'$, then $\gamma(c, c') = c'$ is trivially true. □

Lemma 2.7. *Let P be a recursive Datalog program with ICO T and let the constraint γ be $\mathscr{P}$reM to T and P. Let P also have a parallel decomposable evaluation plan that can be executed over $\mathscr{W}$ workers, where Q_i is the program executed at worker i and T_i is the corresponding ICO defined over Q_i. Let γ be also $\mathscr{P}$reM to T_i and Q_i, for $1 \leq i \leq \mathscr{W}$. After r rounds of synchronization (r rounds of synchronization in SSP model means every worker has sent at least r updates), if I_b and I_s denote the interpretation of the recursive predicate under BSP and SSP models respectively for any worker i, then I_s is a γ-cover for I_b.*

Proof. In a SSP based fixpoint computation, any worker i can produce an atom in three ways:

(1) From local computation not involving any of the updates sent by other workers.

(2) From a join with a new atom or an update sent by another worker j.

(3) From both cases (1) and (2) together.

Now, consider the base case, where before the first round of synchronization (i.e., at the 0th round) each worker performs only local computation, since it has not received/sent any update from/to any other worker. Since, in a SSP model, each local computation may involve multiple iterations (as shown in step (6) in Figure 2.3), I_s is trivially a γ-cover for I_b (from *Lemma* 2.6).

We next assume this hypothesis (*Lemma* 2.7) to be true for some $r \geq 0$. Under this assumption, we find that each worker i in SSP model for its fixpoint computation operates based on the information from its own I_s and from the ones sent by other workers after the rth round of synchronization. And since each of this I_s involved is a γ-cover for the corresponding I_b (when compared against the BSP model), the aforementioned cases (1)-(3) will also produce a γ-cover for the $(r+1)^{th}$ synchronization round.

Hence, by principle of mathematical induction, the lemma holds for all $r \geq 0$. □

Theorem 2.8. *Let P be a recursive Datalog program with ICO T and let the constraint γ be $\mathscr{P}$reM to T and P. Let P have a parallel decomposable evaluation plan that can be executed over $\mathscr{W}$ workers, where Q_i is the program executed at worker i and T_i is the corresponding ICO defined over Q_i. If γ is also $\mathscr{P}$reM to T_i and Q_i, for $1 \leq i \leq \mathscr{W}$, then:*

(i). The SSP processing yields the same minimal fixpoint of $\gamma(T^{\uparrow\omega}(\emptyset))$, as would have been obtained with BSP processing.

(ii). If any worker i under BSP processing requires r rounds of synchronization, then under SSP processing i would require $\leq r$ rounds to reach the minimal fixpoint, where r rounds of synchronization in SSP model means every worker has sent at least r updates.

Proof. Theorem 2.4 guarantees that the BSP evaluation of the datalog program with $\mathscr{P}$*reM* will yield the minimal fixpoint of $\gamma(T^{\uparrow\omega}(\emptyset))$. Note that in the SSP evaluation, for every tuple t produced by a worker i from the program Q_i, $t \in T^{\uparrow\omega}(\emptyset)$. In other words, if I represents the final

interpretation of the recursive predicate under SSP evaluation, then $I \subseteq T^{\uparrow\omega}(\emptyset)$ i.e. I is bounded. It also follows from *Lemma* 2.7, that I is a γ-cover for the final interpretation of the recursive predicate under BSP evaluation i.e. I is a γ-cover for $\gamma(T^{\uparrow\omega}(\emptyset))$. Since, $\gamma(T^{\uparrow\omega}(\emptyset))$ is the least fixpoint under the γ constraint, we also get $\gamma(T^{\uparrow\omega}(\emptyset)) \subseteq I$, as atoms in $\gamma(T^{\uparrow\omega}(\emptyset))$ must have identical cost in I.

Thus, we can write the following equation based on the above discussion,

$$\gamma(T^{\uparrow\omega}(\emptyset)) \subseteq I \subseteq T^{\uparrow\omega}(\emptyset) \tag{2.1}$$

Also recall, since γ is $\mathscr{P}reM$ to each T_i and Q_i, under the SSP evaluation, each worker i also applies γ in every iteration in its fixpoint computation (step (4) in Figure 2.3). Thus, we have,

$$I \subseteq \gamma(T^{\uparrow\omega}(\emptyset)) \tag{2.2}$$

Combining equations (2.1) and (2.2), we get $I = \gamma(T^{\uparrow\omega}(\emptyset)$. Thus, the SSP evaluation also yields the same minimal fixpoint as the BSP model.

Since, the interpretation of the recursive predicate in the least model obtained from BSP evaluation is identical to that in the least model obtained from SSP processing, it directly follows from *Lemma* 2.7, that the number of synchronization rounds required by worker i in SSP evaluation will be at most r, where r is the number of rounds i takes under BSP model. □

2.6.1 SSP Evaluation of Queries without $\mathscr{P}reM$ Constraint

We now consider the parallel decomposable plan of a transitive closure query, which does not contain any aggregates in recursion. We use the same non-linear recursive example from [YSZ17], given by rules $r_{2.7.1} - r_{2.7.3}$, which shows the program executed by worker i. Note, in this example every worker i eventually has to compute and send to other workers all `tc` atoms of the form `(X, Y)`, where `h(X)` $= i$. Without $\mathscr{P}reM$, a worker i does not update its existing `tc` atoms. In fact, during semi-naive evaluation of this query, at any time, only new unique atoms are appended to

tc. Thus, a SSP evaluation for the transitive closure query (without $\mathscr{P}reM$) does not save any communication cost as compared to a BSP model. However, as shown in our experimental results next, the SSP model can still mitigate the influence of stragglers.

$$r_{2.7.1}: \mathtt{tc(X,Y)} \leftarrow \mathtt{arc(X,Y)}, \mathtt{h(X)} = \mathtt{i}.$$

$$r_{2.7.2}: \mathtt{tc(X,Y)} \leftarrow \mathtt{tc(X,Z)}, \mathtt{tc}^{(1)}\mathtt{(Z,Y)}, \mathtt{h(X)} = \mathtt{i}.$$

$$r_{2.7.3}: \mathtt{tc}^{(1)}\mathtt{(X,Y)} \leftarrow \mathtt{tc(X,Y)}, \mathtt{h(X)} = \mathtt{i}.$$

2.6.2 Experimental Results

Setup. We conduct our experiments on a 12 node cluster, where each node, running on Ubuntu 14.04 LTS, has an Intel i7-4770 CPU (3.40GHz, 4 cores) with 32GB memory and a 1 TB 7200 RPM hard drive. The compute nodes are connected with 1Gbit network. Following the standard practices established in [SYI16], [YSZ17], we execute the distributed bottom-up semi-naive evaluation using an AND/OR tree based implementation in Java on each node. Each node executes one application thread per core. We evaluate both the non-linear all pairs shortest path and transitive closure queries on a subset of the real world *orkut* social network data[4].

Inducing Stragglers. In order to study the influence of straggling nodes in a declarative recursive computation, we induce stragglers in our implementation following the strategy described in [CCH14]. In particular, each of the nodes in our setup can be disrupted independently by a CPU-intensive background process that kicks in following a Poisson distribution and consumes at least half of the CPU resources.

Analysis. In this section, we empirically analyze the merits and demerits of a SSP model over a BSP model, by examining the following questions: (1) How does a SSP model compare to a BSP model when queries contain $\mathscr{P}reM$ constraints and aggregates in recursion? (2) How do these two processing paradigms compare when $\mathscr{P}reM$ cannot be applied? (3) And, how do the overall performances in the above scenarios change in presence and absence of stragglers? Table

[4] http://snap.stanford.edu/data/com-Orkut.html

2.1 captures the first case with the all pairs shortest path query (where $\mathscr{P}reM$ is applicable), while Table 2.2 presents the second case with the transitive closure query, which do not contain any aggregates or $\mathscr{P}reM$ constraints in recursion. For each of these two cases, as shown in the tables, we experimented with two different staleness values for a SSP model, both under the presence and absence of induced stragglers. Notably, a SSP model with bounded staleness (alternatively also called 'slack' and indicated by s in the tables) set as zero reduces to a BSP model. Tables 2.1 and 2.2 capture the average execution time for the query at hand under different configurations over five runs. This *run time* can be divided into two components— (1) *average computation time*, which is the average time spent by the workers performing semi-naive evaluation for the recursive computation, and (2) *average waiting time*, which is the average time spent by the workers waiting to receive a new update to resume computation. Tables 2.1 and 2.2 show the run time break down for the two aforementioned cases (with and without $\mathscr{P}reM$ respectively).

From Tables 2.1 and 2.2, it is evident that BSP processing requires the least compute time irrespective of straggling nodes. This is also intuitively true because the total recursive computation involved in a BSP based distributed semi-naive evaluation is similar to that of a single executor based sequential execution and as such a BSP model should require the least computational effort to reach the minimal fixpoint. On the other hand, a SSP model may perform many local computations optimistically with obsolete data using relaxed synchronization barriers, which can become redundant later on. As shown in the tables, average compute time indeed increases with higher slack indicating that a substantial amount of the work becomes unnecessary. However, as seen from both the tables, SSP plays a major rule in reducing the average wait time. This is trivially true, since in SSP processing, any worker can move ahead with local computations using stale knowledge, instead of waiting for global synchronization as required in BSP. However, note the reduction in average wait time under SSP model in Table 2.1 (with $\mathscr{P}reM$) is more significant than in Table 2.2 (without $\mathscr{P}reM$). This can be attributed to the fact that $\mathscr{P}reM$ with semi-naive evaluation (Section 2.1) under SSP model can batch multiple updates together before sending them, thereby saving communication cost. However, for the transitive closure query (without $\mathscr{P}reM$), the overall updates sent in BSP and SSP models are similar (since no aggregates are used, semi-

naive evaluation only produces new atoms, never updates existing ones). Thus, in the latter case (Table 2.2), the wait times between BSP and SSP models are comparable when there are no induced stragglers, whereas the wait time in SSP is marginally better than BSP when stragglers are present. Notably, inducing stragglers obviously increases the average wait time all throughout as compared to a no straggler situation. The compute time also increases marginally in presence of stragglers, primarily because the straggling nodes take longer time to finish its computations.

Thus, to summarize based on the run times in the two tables, we see that in absence of stragglers, the SSP model can reduce the run time of the shortest path query (with $\mathscr{P}reM$ constraint) by nearly 30%. However, the same is not true for the transitive closure query, which do not have any $\mathscr{P}reM$ constraint. Hence, a BSP model would suffice if there are no stragglers and the query does not contain any $\mathscr{P}reM$ constraint. However, in presence of stragglers or $\mathscr{P}reM$ constraints, SSP model turns out to be a better alternative than BSP model, as it can lead to a execution time reduction of as high as 40% for the shortest path query and nearly 7% for the transitive closure query. Finally, it is also worth noting from the results that too much of a slack can also increase the query latency. Thus, a moderate amount of slack should be used in practice.

Time consumption	No stragglers (time in sec)			With stragglers (time in sec)		
	BSP (s=0)	SSP (s=3)	SSP (s=6)	BSP (s=0)	SSP (s=3)	SSP (s=6)
Avg. compute time	**2224**	2443	3038	**2664**	2749	3435
Avg. wait time	1679	**302**	**408**	2786	**485**	**704**
Run time	3903	**2745**	**3446**	5450	**3234**	**4139**

Table 2.1: Comparing BSP vs. SSP model for all pairs shortest path query containing aggregates in recursion (with $\mathscr{P}reM$).

Time consumption	No stragglers (time in sec)			With stragglers (time in sec)		
	BSP (s=0)	SSP (s=3)	SSP (s=6)	BSP (s=0)	SSP (s=3)	SSP (s=6)
Avg. compute time	**682**	762	879	**754**	827	921
Avg. wait time	367	**345**	**334**	618	**456**	**412**
Run time	**1049**	1107	1213	1372	**1283**	1431

Table 2.2: Comparing BSP vs. SSP model for transitive closure query containing *no* aggregates in recursion (without $\mathscr{P}reM$).

2.7 Conclusion

𝒫reM facilitates and extends the use of aggregates in recursion, and this enables a wide spectrum of graph and data mining algorithms to be expressed efficiently in declarative languages. In this chapter, we explored various improvements to scalability via paralled execution with *𝒫reM*. In fact, *𝒫reM* can be easily integrated with most of the current generation Datalog engines like *BigDatalog*, *Myria*, *BigDatalog-MC*, *SociaLite*, *LogicBlox*, irrespective of their architecture differences and varying synchronization constraints. Moreover, we have shown that *𝒫reM* brings additional benefits to the parallel evaluation of recursive queries. For that, we established the necessary theoretical framework that allows bottom-up recursive computations to be carried out over stale synchronous parallel (SSP) model—in addition to the synchronous or completely asynchronous computing models studied in the past. These theoretical developments lead us to the conclusion, confirmed by our experiments, that the parallel execution of non-linear queries with *𝒫reM* constraints can be expedited with a SSP model. This model is also useful in the absence of *𝒫reM* constraints, where bounded staleness may not reduce communications, but it nevertheless mitigates the impact of stragglers. Experiments performed on a real-world dataset confirm the theoretical results, and are quite promising, paving the way toward future research in many interesting areas, where declarative recursive computation under SSP processing can be quite advantageous.

Finally, it is important to note that the methodologies developed here can also be applied to other declarative logic based systems beyond Datalog, like in SQL-based query engines [GWM19], which also use semi-naive evaluation for recursive computation. In addition, the SSP processing paradigm can also be adopted in many state-of-the-art graph-centric platforms such as Pregel [MAB10] and GraphLab [LBG12]. These modern graph engines use a vertex-centric computing model [YCL15], which enforces a strong consistency requirement among its model variables under the “Gather-Apply-Scatter” abstraction. Consequently, this makes the synchronization cost for these graph frameworks similar to that of standard BSP systems. Thus, for many distributed graph computation problems involving aggregators (like shortest path queries), SSP model, as demonstrated here, can be quite useful for these graph based platforms.

CHAPTER 3

Efficiently Computable Subclass of Stable Models

The efficient computation of stable models for large classes of programs that can express a wide range of applications represents a critical requirement for the logic paradigm in BigData applications. Due to their simplicity and amenability to efficient implementation many Datalog and database systems now support stratified programs. But, since this class of programs is very restrictive and cannot express most A.I. applications, many interesting approaches have been proposed to overcome such limitations, while preserving efficiency. In this chapter, we focus on and compare approaches that have proved effective on a wide range of applications.

A first approach uses *local stratification* restricted to a strictly templatized format that allows non-monotonic constructs inside recursion. For instance, $XY-$programs require queries to be written as a combination of $X-$rules and $Y-$rules that makes explicit the *local stratification* and the iterated fixpoint used to compute their unique stable models. Therefore, this approach does not fully conform with the declarative programming paradigm (of never specifying the control flow of the program) and therefore it misses some of the benefits of $\mathscr{P}reM$ that allows non-monotonic aggregates in recursion. As discussed previously, $\mathscr{P}reM$ delivers very efficient computations of their unique stable models for large number of applications. Therefore, in this chapter, we explore these two worlds of non-monotonic reasoning in terms of performance, usability and expressivity. On the one hand, we find how $\mathscr{P}reM$ offers strict adherence to declarative programming, better usability and amenability to efficient parallel implementation by a stale synchronous computing model on distributed environment. On the other hand, we illustrate nuances of $XY-$stratification

that allow a logic-based expression of complex procedural algorithms, which can be otherwise difficult to express even with $\mathscr{P}reM$.

3.1 Introduction

The powerful notion of stable model semantics [GL88] unifies several non-monotonic knowledge representation formalisms such as circumscription, autoepistemic logic, default theory, and also enable answer-set semantics [SP07], [FPL11] that delivers great power, but is NP-complete. While, in general, stable model computation for any arbitrary Datalog program can be of exponential complexity [PZ96], the former can be computed in polynomial time for stratified programs [CH80], [Prz88a]. In addition, since a program usually contains a handful number of predicates, stratification of a program can be easily verified from the corresponding *predicate dependency graph* [ZAO93] at compile-time, which in turn can also be used for optimizing the program execution. Overall, this led to the computation of unique stable model for stratified recursive programs by efficient procedures such as differential fixpoint computation (a.k.a. semi-naive evaluation). Incidentally, this success story also inspired the adoption of recursive queries in SQL. However, stratification also imposes a serious restriction that negation cannot be used inside recursion [Gel86], [Naq86] in order to preserve semantic correctness under the classic *Closed World Assumption* (CWA) [Rei87]. For example, consider the classic paradox where a villager is shaved by barber if he does not shave himself. In the query below, if we assume `¬shaves(barber, barber)`, then the rule would produce `shaves(barber, barber)` in direct contradiction. On the other hand, if we assume the converse, then `shaves(barber, barber)` would not be produced, thereby indicating `¬shaves(barber, barber)` to be true under CWA, which is again a direct contradiction. Since, other extrema operators like `max` or `min` are quintessentially based on negation, these non-monotonic constructs also could not be previously used inside recursion for stratified programs. This was a serious limitation for many practical applications ranging from graph analytics to data mining [Prz88b], [CDI18], [BBC12].

```
shaves(barber, X) ← villager(X), ¬shaves(X, X).

villager(barber).
```

To address the aforementioned problems, researchers proposed writing *locally stratified* programs[1] and computing their corresponding equivalent stable models instead. Unfortunately, verifying whether a program can be *locally stratified* is undecidable [BC94] and hence its semantic correctness cannot be examined, short of actually executing the program. Thus, additional restrictions were imposed [KRS95] to come up with *explicitly locally stratified* strict program templates, such as those of $XY-$stratification [ZAO93], where the existence of a stable model can be examined at the time of compilation similar to usual stratified programs and was used in several applications [BBC12, CAG03].

However, $XY-$stratification raises two fundamental challenges, which are adversarial to declarative style of programming: (i) declarative programmers often need to explicitly specify terminating criteria in the recursive rules in order to prevent infinite number of iterations and (ii) declarative programmers also have to express certain implementation details as logical rules (*copy/delete* rules) for an efficient execution. Both these issues add a 'procedural' flair to $XY-$stratified programs, akin to embedded SQL languages like PL/SQL. While this enables $XY-$stratified programs to express many complex classical procedural algorithms and various data mining and machine learning applications [BBC12,CAG03], it also burdens the declarative programmer with the effort to specify the details of *how to implement*, rather than just expressing logically *what to get*.

On the other hand, $\mathscr{P}reM$ ensures semantic correctness is preserved even when non-monotonic constraints are pushed inside recursion, provided certain conditions are met. In addition, as discussed before, $\mathscr{P}reM$ has proven to be amenable to highly efficient parallelism under bulk synchronous and stale synchronous distributed computing models. However, does $\mathscr{P}reM$ has the same expressiveness as $XY-$stratification? In this chapter, we thus explore these two different semantics of declarative programming and investigate (1) if complex *locally stratified* $XY-$programs can

[1]Here, stratification is done over all the ground atoms given by the Herbrand base, instead of over the program predicates.

be expressed with otherwise stratified recursive programs with non-monotonic $\mathscr{P}reM$ constraints inside recursion, and (2) if expressing $XY-$stratifiable programs with $\mathscr{P}reM$ can ensure (a) better usability from declarative point of view and (b) better performance? Thus, the contributions of this chapter can be summarized as follows:

- We show that many complex *locally stratified* $XY-$programs like temporal coalescing and Floyd-Warshal algorithm can be indeed expressed with $\mathscr{P}reM$, which produces the exact same stable model as the *locally stratified* program.
- Furthermore, we also elaborate how programs expressed with $\mathscr{P}reM$ are much more intuitive and more amenable to better usability since they completely follow the declarative paradigm, unlike the quasi-declarative $XY-$stratified programs.
- We further empirically show how expressing queries with $\mathscr{P}reM$ can open up opportunities for performance gains under a stale synchronous parallel (SSP) distributed model, which cannot be used in $XY-$stratified programs.
- Finally, we also demonstrate cases where $\mathscr{P}reM$ is not directly applicable. As such, for these scenarios, $XY-$stratification offers better expressiveness and still remains the best way to support such algorithms.

3.2 $XY-$Stratification

Consider the Datalog query, given by rules $r_{3.1.1} - r_{3.1.2}$, where `parent(X, Y)` is an extensional predicate representing `X` as a parent of `Y` and `all_anc(X, Y)` is an intensional predicate indicating `X` to be an ancestor of `Y`. This simple recursive query (rules $r_{3.1.1} - r_{3.1.2}$) computes all ancestors of a person using their successive parents information and can be easily computed using classical bottom-up semi-naive evaluation method [SYI16], which terminates once the fixpoint is reached. This stratified query written declaratively is concise, easy to understand and based on the simple intuition that if `X` is an ancestor of `Y` and `Y` is a parent of `Z`, then `X` is also an ancestor of `Z` (rule $r_{3.1.2}$).

$r_{3.1.1}$: $\texttt{all_anc}(\texttt{X},\texttt{Y}) \leftarrow \texttt{parent}(\texttt{X},\texttt{Y})$.

$r_{3.1.2}$: $\texttt{all_anc}(\texttt{X},\texttt{Z}) \leftarrow \texttt{all_anc}(\texttt{X},\texttt{Y}),\texttt{parent}(\texttt{Y},\texttt{Z})$.

Interestingly, the above query can also be expressed with $XY-$ stratification as shown with rules $r_{3.2.1} - r_{3.2.4}$. This query is *locally stratified* based on the first argument of `delta_anc` and `all_anc`, acting as a temporal argument. More precisely, the kth stratum comprises of atoms from `all_anc(`k`, X, Y)` and `delta_anc(`k`, X, Y)`, while the zeroth stratum comprises of non-recursive predicate `parent(X, Y)` and atoms of the form `all_anc(0, X, Y)` and `delta_anc(0, X, Y)` unified using `parent(X, Y)`. Furthermore, even within the same stratum (i.e., having the same temporal argument), atoms are partitioned into multiple substrata. For example, in the same k^{th} stratum `all_anc(`k`, X, Y)` occurs at a higher substratum than `delta_anc(`k`, X, Y)`. This inter-strata and intra-strata dependencies in this *local stratification* can be well understood by rewriting the query using an equivalent bi-state form, as shown using rules $r'_{3.2.1} - r'_{3.2.4}$, where we dropped the temporal argument from the recursive predicates and instead focused on any two successive strata, *new* (representing $(k+1)^{th}$ stratum) and *old* (representing k^{th} stratum). At the start of the computation at every stratum, the *new* predicates computed before are used as *old*. Strictly speaking, this rewriting was possible only because this *local stratification* followed $XY-$template, with $r_{3.2.4}$ being called a $X-$rule (having same temporal argument in every recursive predicate used) and $r_{3.2.2}$, $r_{3.2.3}$ being called $Y-$rules (computing atoms in $(\texttt{J+1})^{th}$ stratum from some atoms in `J` stratum) [ZAO93].

$r_{3.2.1}$: $\texttt{delta_anc}(0,\texttt{X},\texttt{Y}) \leftarrow \texttt{parent}(\texttt{X},\texttt{Y})$.

$r_{3.2.2}$: $\texttt{delta_anc}(\texttt{J}+1,\texttt{X},\texttt{Z}) \leftarrow \texttt{delta_anc}(\texttt{J},\texttt{X},\texttt{Y}),\texttt{parent}(\texttt{Y},\texttt{Z}),\neg\texttt{all_anc}(\texttt{J},\texttt{X},\texttt{Z})$.

$r_{3.2.3}$: $\texttt{all_anc}(\texttt{J}+1,\texttt{X},\texttt{Y}) \leftarrow \texttt{all_anc}(\texttt{J},\texttt{X},\texttt{Y}),\textbf{delta_anc}(\mathbf{J}+\mathbf{1},_,_)$.

$r_{3.2.4}$: $\texttt{all_anc}(\texttt{J},\texttt{X},\texttt{Y}) \leftarrow \texttt{delta_anc}(\texttt{J},\texttt{X},\texttt{Y})$.

$r'_{3.2.1}$: `new_delta_anc(X,Y)` ← `parent(X,Y).`

$r'_{3.2.2}$: `new_delta_anc(X,Z)` ← `old_delta_anc(X,Y),parent(Y,Z),¬old_all_anc(X,Z).`

$r'_{3.2.3}$: `new_all_anc(X,Y)` ← `old_all_anc(X,Y),new_delta_anc(U,V).`

$r'_{3.2.4}$: `new_all_anc(X,Y)` ← `new_delta_anc(X,Y).`

As evident, the *locally stratified* $XY-$query is more complex than the simple stratified query discussed before, primarily because of the following:

- The intuition behind the $XY-$stratified query is given by the logical expression in $r_{3.2.2}$, which represents the following: generate new ancestors, which have *not* been previously included in the list of all ancestors (expressed with `¬all_anc(J,X,Y)`) by joining *only* the new ancestors produced in the last stratum with `parent`. Interestingly, this optimization scheme expressed in the form of a rule in the $XY-$query is implicitly executed when the stratified query (rules $r_{3.1.1} - r_{3.1.2}$) is evaluated under the standard semi-naive evaluation.
- To make things even more cumbersome, a declarative programmer also needs to include additional implementation details via rules like $r_{3.2.3} - r_{3.2.4}$[2], which expresses the control flow of the program about how atoms must be appended from one stratum to another, instead of just expressing the logical intuition of the program declaratively, as is often the case with stratified programs.
- Finally, note rule $r_{3.2.3}$ needs to include the goal `delta_anc(J+1,_,_)` as a stopping criterion to terminate the program execution when no new ancestors are found. Without it, the program would indefinitely continue, computing for all positive integers. In other words, the programmer needs to explicitly ensure the termination of the program.

While the above complexities pose serious usability challenges for a declarative programmer, *local stratification* in the past was nevertheless necessary to use negation and other non-monotonic constraints within recursion, as shown in rule $r_{3.2.2}$. However, under certain conditions, we can avoid this complexities using $\mathscr{P}reM$, as discussed before.

[2]termed as *copy rules* [ZAO93]

3.3 Comparison of $XY-$Stratification with $\mathscr{P}reM$

We now present a nuanced comparative analysis between $XY-$stratification and $\mathscr{P}reM$ using popular query examples to emphasize the possible advantages of $\mathscr{P}reM$ over traditional *local* $XY-$*stratification*.

3.3.1 Temporal Coalescing

Temporal relations can comprise of several attributes and projecting out any of them can produce overlapping intervals. Consider the example shown in Figure 3.1, where temporal projection of an employee (denoted by `Eno`) from a database, leaving out other attributes, produces seven intervals (working spans), many of which are overlapping and need to be merged. The Datalog query given by rules $r_{3.3.1} - r_{3.3.10}$ performs this temporal coalescing recursively using *local* $XY-$*stratification* (specifically rules $r_{3.3.4} - r_{3.3.7}$). The stratified query given by rules $r_{3.3.1} - r_{3.3.3}$ initially computes the starting time points of all the intervals, which are not completely contained by another interval. For example, the start of interval `2002/05/01` — `2003/12/31` is not included in `lstart` because it is already covered by the interval `2001/01/01` — `2004/06/30`. Thereafter, the $XY-$rules ($r_{3.3.4} - r_{3.3.7}$) recursively coalesces the intervals in two basic computation steps: (i) the $Y-$rule $r_{3.3.5}$ selects two overlapping intervals (e.g. `2001/01/01` — `2004/06/30` and `2003/06/01` — `2007/05/31`), where the second interval succeeds the first and (ii) the $X-$rule $r_{3.3.6}$ merges the two overlapping intervals, selecting the larger of the two end points as the end point for the coalesced interval ($r_{3.3.9} - r_{3.3.10}$). Note, rule $r_{3.3.7}$ ensures that a coalesced interval from the previous stratum gets copied into the next stratum, only if it has not been selected in the overlapping step before and hence has not been merged. This is achieved by pushing the negation inside recursion via rule $r_{3.3.7}$, which is also termed as the *delete* rule [ZAO93], since the overlapping intervals from the previous stratum are deleted from the coalesced intervals in the next stratum. This data processing rule assures the efficiency of the recursive computation by gradually coalescing intervals and then removing the ones from consideration, which have been merged. The computation terminates when no more overlapping intervals can be found and hence the goal `overlap(J+1,_, _, _, _, _)` is not satisfied in $r_{3.3.7}$. Of course, this terminating

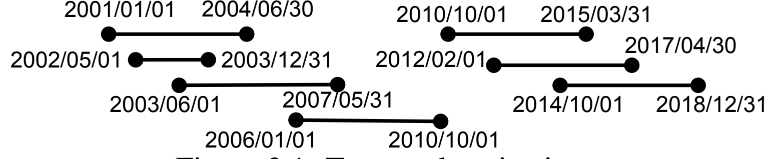


Figure 3.1: Temporal projections

criterion needs to be explicitly specified. The final merged intervals are obtained from the highest stratum of this *locally stratified* $XY-$ query as given by rule $r_{3.3.8}$.

$$r_{3.3.1}: \texttt{lstart(Eno,S)} \leftarrow \texttt{inter(Eno,S,E)}, \neg\texttt{covered(Eno,S,E)}.$$

$$r_{3.3.2}: \texttt{covered(Eno,S,E)} \leftarrow \texttt{inter(Eno,S,E)}, \texttt{inter(Eno,S1,E1)}, \texttt{S1} \leq \texttt{S}, \texttt{E1} > \texttt{E}.$$

$$r_{3.3.3}: \texttt{covered(Eno,S,E)} \leftarrow \texttt{inter(Eno,S,E)}, \texttt{inter(Eno,S1,E1)}, \texttt{S1} < \texttt{S}, \texttt{E1} \geq \texttt{E}.$$

$$r_{3.3.4}: \texttt{coal(0,Eno,S,E)} \leftarrow \texttt{lstart(Eno,S)}, \texttt{inter(Eno,S,E)}.$$

$$r_{3.3.5}: \texttt{ovrlap(J+1,Eno,S1,E1,S2,E2)} \leftarrow \texttt{coal(J,Eno,S1,E1)}, \texttt{coal(J,Eno,S2,E2)}, \texttt{S1} < \texttt{S2}, \texttt{S2} \leq \texttt{E1}.$$

$$r_{3.3.6}: \texttt{coal(J,Eno,S1,E)} \leftarrow \texttt{ovrlap(J,Eno,S1,E1,S2,E2)}, \texttt{larger(E1,E2,E)}.$$

$$r_{3.3.7}: \texttt{coal(J+1,Eno,S,E)} \leftarrow \texttt{coal(J,Eno,S,E)}, \texttt{ovrlap(J+1,_,_,_,_,_)}, \neg\texttt{ovrlap(J+1,Eno,S,E,_,_)}, \neg\texttt{ovrlap(J+1,Eno,_,_,S,E)}.$$

$$r_{3.3.8}: \texttt{final_intrvl(Eno,S,E)} \leftarrow \texttt{coal(J,Eno,S,E)}, \neg\texttt{coal(J+1,_,_,_)}.$$

$$r_{3.3.9}: \texttt{larger(E1,E2,E1)} \leftarrow \texttt{E1} \geq \texttt{E2}.$$

$$r_{3.3.10}: \texttt{larger(E1,E2,E2)} \leftarrow \texttt{E2} > \texttt{E1}.$$

On the other hand, this temporal coalescing can also be done recursively using $\mathscr{P}reM$ with the aid of extrema aggregates like `max` within recursion. Rules $r'_{3.3.4} - r'_{3.3.5}$ shows the corresponding example of temporal coalescing with $\mathscr{P}reM$. This simple recursion computed via semi-naive evaluation iteratively merges the overlapping the intervals by extending the interval end point with the `max` of the overlapping intervals. For example, intervals `2001/01/01` — `2004/06/30` and `2003/06/01` — `2007/05/31` are coalesced to produce `2001/01/01` — `2007/05/31`, whereas, intervals `2003/06/01` — `2007/05/31` and `2006/01/01` — `2010/10/01` are

merged to produce interval 2003/06/01 — 2010/10/01. These two newly produced intervals are merged in the next iteration to produce the interval 2001/01/01 — 2010/10/01. The computation stops when the fixpoint is reached with the final interval of 2001/01/01 — 2018/12/31.

$r'_{3.3.4}$: $\texttt{coal(Eno,S,E)} \leftarrow \texttt{lstart(Eno,S),inter(Eno,S,E)}$.

$r'_{3.3.5}$: $\texttt{coal(Eno,S,max}\langle\texttt{E}\rangle\texttt{)} \leftarrow \texttt{coal(Eno,S,E1),coal(Eno,S1,E),S} < \texttt{S1,S1} \leq \texttt{E1}$.

$r'_{3.3.6}$: $\texttt{final_intrvl(Eno,S,E)} \leftarrow \texttt{coal(Eno,S,E)},\neg\texttt{contain(Eno,S,E)}$.

$r'_{3.3.7}$: $\texttt{contain(Eno,S,E)} \leftarrow \texttt{coal(Eno,S,E),coal(Eno,S1,E1),S1} \leq \texttt{S,E1} > \texttt{E}$.

$r'_{3.3.8}$: $\texttt{contain(Eno,S,E)} \leftarrow \texttt{coal(Eno,S,E),coal(Eno,S1,E1),S1} < \texttt{S,E1} \geq \texttt{E}$.

The above queries highlight the following salient points:

• **Better *usability* with $\mathscr{P}reM$:** $\mathscr{P}reM$ allows more elegant, concise and intuitive representation of the logic, hiding away all the stopping criterion details and other control flow rules, which makes the $XY-$stratified query less comprehensible and user friendly from a declarative perspective.

• **Advantage of *delete* rules with XY:** The *delete* rules in $XY-$stratification can afford to remove or forget certain atoms, which will no longer be required in the subsequent computations. For example, once the overlapping intervals of 2003/06/01 — 2007/05/31 and 2006/01/01 — 2010/10/01 are merged at the $\texttt{J}^{th}$ stratum, they will no longer be retained at the $(\texttt{J+1})^{th}$ stratum (courtesy $r_{3.3.7}$). This is possible because the *locally stratified* $XY-$program can be implemented using a bi-state version (as discussed before) that only requires to maintain information about two stratum (discarding all the other lower strata) – the $(\texttt{J+1})^{th}$ (the new stratum where atoms are produced) and the $\texttt{J}^{th}$ (the old stratum whose atoms are used for the production in a $Y-$rule). However, this is not true for the $\mathscr{P}reM$ query. The coal predicate will contain the intervals 2003/06/01 — 2007/05/31 and 2006/01/01 — 2010/10/01, even after merging them, although once merged they would not be able to produce any new atoms from rule $r'_{3.3.5}$ under differential fixpoint evaluation. Thus, the final extraction rule $r'_{3.3.6}$ is required to obtain the eventual coalesced interval that is not contained in any of the intervals (determined using rules $r'_{3.3.7} - r'_{3.3.8}$).

• **Relaxed synchronization with $\mathscr{P}reM$:** As discussed earlier, relaxed synchronization with $\mathscr{P}reM$ can offer significant opportunities for optimization. Here also, assume that temporal coalescing is executed in parallel on two different compute nodes. One computer node, *worker 0* coalesces intervals beginning before `2010/10/01`, while the other node, *worker 1* merges intervals starting from or after `2010/10/01`. Interestingly, note in the first iteration *worker 0* can merge the sets of overlapping intervals (`2001/01/01` — `2004/06/30`, `2003/06/01` — `2007/05/31`) and (`2003/06/01` — `2007/05/31`, `2006/01/01` — `2010/10/01`), independent of *worker 1*, which can similarly coalesce the sets of overlapping intervals (`2010/10/01` — `2015/03/31`, `2012/02/01` — `2017/04/30`) and (`2012/02/01` — `2017/04/30`, `2014/10/01` — `2018/12/31`) without communicating with *worker 0*. In fact, even in the second iteration *worker 0* and *worker 1* can coalesce the merged overlapping intervals (`2001/01/01` — `2007/05/31`, `2003/06/01` — `2010/10/01`) and (`2010/10/01` — `2017/04/30`, `2012/02/01` — `2018/12/31`) respectively from the first iteration independent of each other. However, this acutely brings forth the challenge with the operational semantics of $XY-$stratification. The $XY-$rules ($r_{3.3.5} - r_{3.3.7}$) enforce that all computation in the $\mathtt{J}^{th}$ stratum must be completed before evaluating the $(\mathtt{J+1})^{th}$ stratum. This in turn, implies that *worker 0* and *worker 1* must wait for each other to finish their respective $\mathtt{J}^{th}$ iteration and *synchronize* with each other, even though sometimes their computations can be carried out independently. Thus, the $XY-$stratified programs can only be executed using *Bulk Synchronous Parallel* (BSP) computing model [SYI16], where each worker coordinates with others after every iteration. This is particularly a serious problem when there are straggling workers, i.e. those lag behind other workers in terms of computations which can be easily addressed with $\mathscr{P}reM$ under the *Stale Synchronous Parallel* (SSP) distributed computing model. $\mathscr{P}reM$ queries are primarily able to adopt SSP models for computing, because of the absence of a temporal argument `J`, as shown in rules $r'_{3.3.4} - r'_{3.3.5}$, which imposes a strict need for synchronization. While it can be argued that a better data partitioning strategy[3] can mitigate this problem with the $XY-$stratified temporal coalescing query, there are many other complex algorithms, like Floyd-Warshall algorithm, where this becomes unavoidable, as shown next.

[3] For example, data divided according to `Eno` instead of temporal intervals.

3.3.2 Floyd-Warshall Algorithm

We now express the classic Floyd-Warshall algorithm, which computes the shortest path between all pairs of vertices in a weighted graph. The Datalog query defined by rules $r_{3.4.1} - r_{3.4.8}$ shows how this classic algorithm can be expressed with $XY-$stratification. The intuition behind the algorithm is mainly expressed with rules $r_{3.4.4}$ and $r_{3.4.5}$ as follows: new paths between two nodes `X` and `Y` are only considered if (i) the cost of the path from `X` to `Y` via `Z` is lower than the earlier cost of the path from `X` to `Y` ($r_{3.4.4}$) or (ii) there did not exist a path between `X` and `Y` ($r_{3.4.5}$). Of all these new paths considered from the $\mathtt{J}^{th}$ stratum, we add the one with `min` cost to `delta` in the $(\mathtt{J+1})^{th}$ stratum. The new paths thus found or paths with updated costs are copied into the next stratum (rule $r_{3.4.7}$), while old paths found before which already have the `min` cost so far are retained as well (rule $r_{3.4.6}$).

Interestingly, the same logic can be more elegantly expressed with $\mathscr{P}reM$ constraint, as shown with rule $r'_{3.4.4}$. [DZ19] has shown that this all pairs shortest path query (rules $r'_{3.4.1} - r'_{3.4.4}$) with $\mathscr{P}reM$ constraint, when evaluated with differential fixpoint iteration is amenable to SSP computing models. In fact, in the distributed setting, the `allpaths` predicate is generally hash partitioned [AGK17] to different computing nodes and under a SSP model, a worker i at iteration c for computing its `allpaths` can use the `allpaths` atoms from other workers generated from iteration $c-s$ to $c-1$, where s is a *bounded staleness* constraint [CHK13]. Thus, we compare the performance of these two implementations: $XY-$stratified query ($r_{3.4.1} - r_{3.4.8}$) executed over a BSP environment, whereas query with $\mathscr{P}reM$ ($r'_{3.4.1} - r'_{3.4.4}$) executed over SSP environment. However, it is important to highlight a significant difference between these two queries other than the relaxed synchronization benefit. Rules $r_{3.4.4}$ and $r_{3.4.5}$ (unlike rule $r'_{3.4.4}$) does not operate similar to an optimized semi-naive evaluation, where only the new or updated `allpaths` atoms participate in the join. In other words, old `allpaths` atoms are joined again in rules $r_{3.4.4}$ and $r_{3.4.5}$ and re-examined, making this implementation highly inefficient in itself. Thus, the programmer has to re-write rule $r_{3.4.4}$ into rules $r_{3.4.4.1} - r_{3.4.4.3}$ (similarly for $r_{3.4.5}$) to ensure that only the new and updated atoms are used in the join for a better optimized implementation.

$$r_{3.4.1}: \texttt{allpaths}(0,\texttt{X},\texttt{X},0) \leftarrow \texttt{edge}(\texttt{X},_,_).$$

$$r_{3.4.2}: \texttt{allpaths}(0,\texttt{Y},\texttt{Y},0) \leftarrow \texttt{edge}(_,\texttt{Y},_).$$

$$r_{3.4.3}: \texttt{allpaths}(0,\texttt{X},\texttt{Y},\texttt{C}) \leftarrow \texttt{edge}(\texttt{X},\texttt{Y},\texttt{C}).$$

$$r_{3.4.4}: \texttt{delta}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{C}\rangle) \leftarrow \texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Z},\texttt{C1}),\texttt{allpaths}(\texttt{J},\texttt{Z},\texttt{Y},\texttt{C2}),$$
$$\texttt{C} = \texttt{C1}+\texttt{C2},\texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C3}),\texttt{C} < \texttt{C3}.$$

$$r_{3.4.5}: \texttt{delta}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{C}\rangle) \leftarrow \texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Z},\texttt{C1}),\texttt{allpaths}(\texttt{J},\texttt{Z},\texttt{Y},\texttt{C2}),$$
$$\texttt{C} = \texttt{C1}+\texttt{C2},\neg\texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Y},_).$$

$$r_{3.4.6}: \texttt{allpaths}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{C}) \leftarrow \texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}),\texttt{delta}(\texttt{J}+1,_,_,_),$$
$$\neg\texttt{delta}(\texttt{J}+1,\texttt{X},\texttt{Y},_).$$

$$r_{3.4.7}: \texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}) \leftarrow \texttt{delta}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}).$$

$$r_{3.4.8}: \texttt{minpaths}(\texttt{X},\texttt{Y},\texttt{C}) \leftarrow \texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}),\neg\texttt{allpaths}(\texttt{J}+1,_,_,_).$$

$$r'_{3.4.1}: \texttt{allpaths}(\texttt{X},\texttt{X},0) \leftarrow \texttt{edge}(\texttt{X},_,_).$$

$$r'_{3.4.2}: \texttt{allpaths}(\texttt{Y},\texttt{Y},0) \leftarrow \texttt{edge}(_,\texttt{Y},_).$$

$$r'_{3.4.3}: \texttt{allpaths}(\texttt{X},\texttt{Y},\texttt{C}) \leftarrow \texttt{edge}(\texttt{X},\texttt{Y},\texttt{C}).$$

$$r'_{3.4.4}: \texttt{allpaths}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{C}\rangle) \leftarrow \texttt{allpaths}(\texttt{X},\texttt{Z},\texttt{C1}),\texttt{allpaths}(\texttt{Z},\texttt{Y},\texttt{C2}),\texttt{C} = \texttt{C1}+\texttt{C2}.$$

$$r_{3.4.4.1}: \texttt{delta}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{C}\rangle) \leftarrow \texttt{delta}(\texttt{J},\texttt{X},\texttt{Z},\texttt{C1}),\texttt{allpaths}(\texttt{J},\texttt{Z},\texttt{Y},\texttt{C2}),$$
$$\neg\texttt{delta}(\texttt{J},\texttt{Z},\texttt{Y},\texttt{C2}),\texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C3}),$$
$$\texttt{C} = \texttt{C1}+\texttt{C2},\texttt{C} < \texttt{C3}.$$

$$r_{3.4.4.2}: \texttt{delta}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{C}\rangle) \leftarrow \texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Z},\texttt{C1}),\neg\texttt{delta}(\texttt{J},\texttt{X},\texttt{Z},\texttt{C1}),\texttt{delta}(\texttt{J},\texttt{Z},\texttt{Y},\texttt{C2}),$$
$$\texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C3}),\texttt{C} = \texttt{C1}+\texttt{C2},\texttt{C} < \texttt{C3}.$$

$$r_{3.4.4.3}: \texttt{delta}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{C}\rangle) \leftarrow \texttt{delta}(\texttt{J},\texttt{X},\texttt{Z},\texttt{C1}),\texttt{delta}(\texttt{J},\texttt{Z},\texttt{Y},\texttt{C2}),\texttt{allpaths}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C3}),$$
$$\texttt{C} = \texttt{C1}+\texttt{C2},\texttt{C} < \texttt{C3}.$$

- **Setup and Dataset:** We use a 12 node cluster, with each node running on Ubuntu 14.04 LTS and

having an Intel i7-4770 CPU, 32GB memory and a 1 TB 7200 RPM hard drive. Our experimental configurations are similar to the ones reported in previously in Chapter 2. As before, we conduct our experiments on a subset of the *orkut* social network data used in Chapter 2. We perform two sets of experiments comparing performances of Java-based implementations of $XY-$stratified query evaluation using BSP model with $\mathscr{P}reM$-enabled query evaluation using SSP model (i) in presence and (ii) absence of induced stragglers. The straggling nodes were induced as discussed before in Chapter 2.

• **Performance gains with $\mathscr{P}reM$:** Figure 3.2 reports the average query run time over five runs for the $XY-$stratified and $\mathscr{P}reM$-enabled query, using two different values of staleness (denoted by 's') for the SSP processing. As evident from Figure 3.2, the average wait time is considerable lower for all SSP evaluations. In fact, even though the average compute time for $XY-$stratified query in absence of stragglers is the smallest in Figure 3.2(a), its total run time is largely dominated by the average wait time, which makes it overall 35% slower than the best $\mathscr{P}reM$-enabled SSP evaluation (s=3). Furthermore, surprisingly even the compute time for $\mathscr{P}reM$-enabled SSP evaluation (s=3) is marginally lower than that of $XY-$stratified query in presence of stragglers (Figure 3.2(b)), thus implying that sometimes even explicit optimizations as done with rules $r_{3.4.4.1} - r_{3.4.4.3}$ are not sufficient to surpass in-built system level optimizations, as is the case with $\mathscr{P}reM$-enabled query computation via semi-naive evaluation. Similar trends have been reported in [GWM19], where performance of procedural programs could not compete with that of in-built optimizations supported by declarative systems.

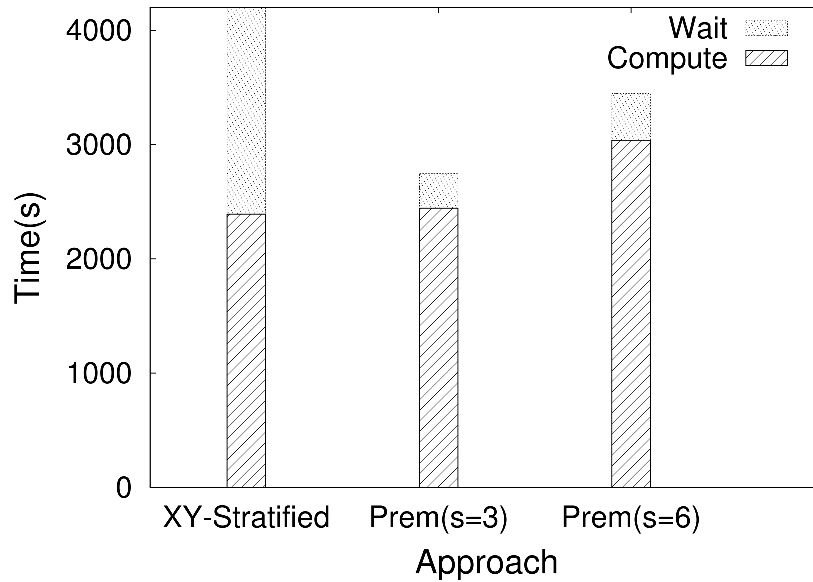
3.4 Stable Model Computation

Lemma 3.1. *Every* locally stratified *program, without any non-deterministic constructs*[4] *has an unique stable model.*

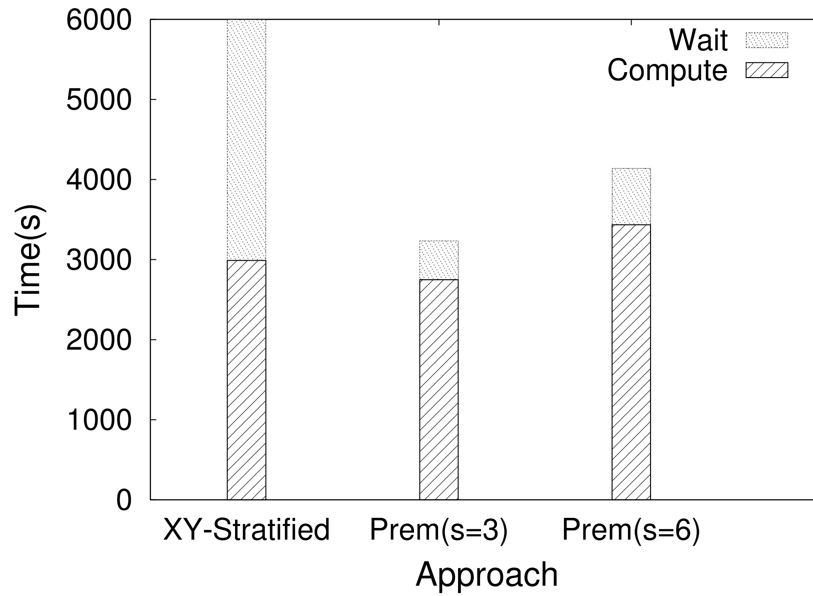
Proof. The proof directly follows from [ZCF97]. □

Lemma 3.2. *A recursive program with $\mathscr{P}$reM constraints inside recursion converges to a stable*

[4]For example, `choice` as used in [GSZ95]



(a) Without Stragglers



(b) With Stragglers

Figure 3.2: Comparing $XY-$stratified query evaluation with BSP model vs. $\mathscr{P}reM$ enabled query evaluation with SSP model for Floyd-Warshall algorithm.

model that is equivalent to the minimal perfect model of the corresponding aggregate stratified program that can be computed using iterated fixpoint computation.

Proof. The proof directly follows from [ZYI17] and implies that rules $r_{2.1.1} - r_{2.1.3}$ and $r_{2.2.1} - r_{2.2.3}$ in Chapter 2 would theoretically return the same minimal model, although the latter provides a more efficient operational semantics. □

Theorem 3.3. *Let P be a XY−stratified recursive program supporting non-monotonic constraints inside recursion without any non-deterministic constructs. Let P′ be the corresponding stratified recursive program for P with $\mathscr{P}$reM constraint γ within recursion. Then P and P′ both converge to the same stable model.*

Proof. Every $XY-$stratified program can be represented as a bi-state program (as shown before), which is thus trivially *locally stratified*. Thus P has an unique stable model (by *lemma* 3.1), say M. Also, by *lemma* 3.2, P' has a stable model, say M'. By the definition of stable models, M and M' are minimal models. Therefore, if $M \neq M'$, then P should have more than one stable model, which contradicts *lemma* 3.1. Thus, $M = M'$. □

3.5 When $\mathscr{P}$*reM* is Inapplicable

We now present an example where direct application of $\mathscr{P}$*reM* fails, even though it can be expressed with recursion using $XY-$stratification. Consider a case where a person wants to get an item in the least possible time. The item can either be assembled from its subparts or it can be bought directly from a supplier. Different suppliers take different days to supply a part. The time to assemble a part from its subparts is the `max` of the days required to receive all its subparts. We assume the actual assembly time is zero. In Figure 3.3, subparts `A` and `B` can be delivered by three suppliers. The optimal case is `A` gets delivered in 10 days and `B` gets delivered in 6 days. We can then either assemble `C` from `A` and `B` in 10 days (time waiting to receive `A`) or we can get `C` from a supplier directly in 12 days. Thus, the `min` time to get part `C` is 10 days. Similarly, the least time to get part `D` is 20 days. Thus, it is better to buy `E` directly from a supplier and get it in 15 days,

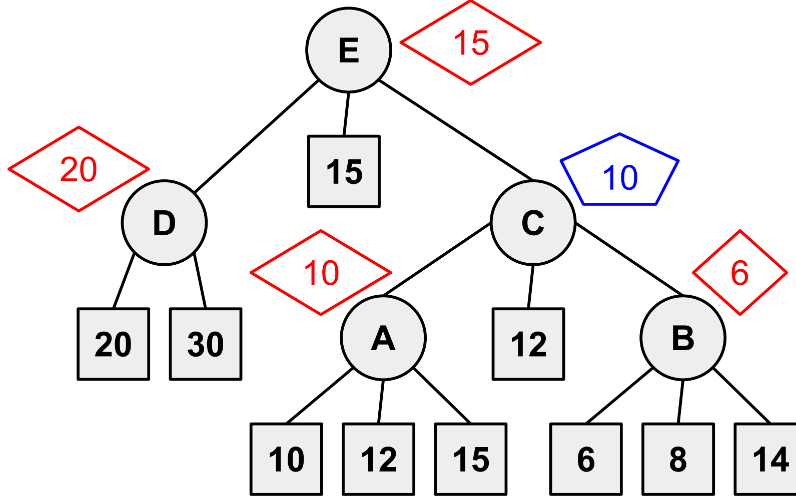


Figure 3.3: Optimal time to get part E.

instead of assembling it from C and D, which would take 20 days. However, say, D can be delivered in 8 days, then the optimal strategy would have been to buy D, A and B and then assemble E.

The example program, given by rules $r_{3.5.1} - r_{3.5.6}$, shows how to evaluate the number of days required to get an item using $XY-$stratification. Note this query uses two constraints: (i) a max constraint (say, γ_1) in $r_{3.5.2}$ that decides the assembly time on the basis of which subpart requires the highest number of days for delivery, and (ii) a min constraint (say, γ_2) in $r_{3.5.4}$ (defined by smaller in $r_{3.5.5} - r_{3.5.6}$) that decides whether the part should be bought directly or assembled depending on which one is smaller. Also note, to simplify the query, we have used an explicit goal of J$< \mathscr{T}$ (where, $\mathscr{T}$ denotes the upper bound of total number of items at hand) in $r_{3.5.3} - r_{3.5.4}$, which can be a sub-optimal stopping criterion, thereby leading to redundant computations. Now, we examine if these two constraints γ_1 and γ_2 are $\mathscr{P}reM$ or not. Let T_1 be the Immediate Consequence Operator (ICO) that determines the assembly time (rule $r_{3.5.2}$) and T_2 be the ICO that decides whether to assemble or buy (i.e., rules $r_{3.5.3} - r_{3.5.4}$). Now, by definition of $\mathscr{P}reM$, γ_2 is $\mathscr{P}reM$ to T_1, when for every interpretation I, $\gamma_1(T_1(I)) = \gamma_1(T_1(\gamma_2(I)))$ and similarly, γ_1 is $\mathscr{P}reM$ to T_2, if $\gamma_2(T_2(I)) = \gamma_2(T_2(\gamma_1(I)))$ holds. However, note in our example in Figure 3.3, if the max constraint γ_1 is not applied first, then with respect to E, only the min constraint γ_2 will return 10 days (the min of the assembly time of the subparts of E which is incorrect) i.e. $\gamma_2(T_2(I)) \neq \gamma_2(T_2(\gamma_1(I)))$. This is also intuitively true, since the min of max across different sets

is not same as just taking the `min` across all sets. Thus, the semantics and correctness of $\mathscr{P}reM$ does not hold directly here and *local* $XY-$*stratification* offers the best solution to express such queries. The classic Minimax algorithm from game theory is another example where $\mathscr{P}reM$ is similarly inapplicable.

$r_{3.5.1}$: $\texttt{days}(0,\texttt{P},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{supplier}(\texttt{P},\texttt{D}).$

$r_{3.5.2}$: $\texttt{assemble}(\texttt{J}+1,\texttt{P},\texttt{max}\langle\texttt{D}\rangle) \leftarrow \texttt{days}(\texttt{J},\texttt{S},\texttt{D}),\texttt{sub}(\texttt{S},\texttt{P}).$

$r_{3.5.3}$: $\texttt{days}(\texttt{J}+1,\texttt{P},\texttt{D}) \leftarrow \texttt{days}(\texttt{J},\texttt{P},\texttt{D}),\neg\texttt{assemble}(\texttt{J}+1,\texttt{P},_),\texttt{J} < \mathscr{T}.$

$r_{3.5.4}$: $\texttt{days}(\texttt{J}+1,\texttt{P},\texttt{D}) \leftarrow \texttt{days}(\texttt{J},\texttt{P},\texttt{D1}),\texttt{assemble}(\texttt{J}+1,\texttt{P},\texttt{D2}),\texttt{smaller}(\texttt{D1},\texttt{D2},\texttt{D}),\texttt{J} < \mathscr{T}.$

$r_{3.5.5}$: $\texttt{smaller}(\texttt{D1},\texttt{D2},\texttt{D1}) \leftarrow \texttt{D1} \leq \texttt{D2}.$

$r_{3.5.6}$: $\texttt{smaller}(\texttt{D1},\texttt{D2},\texttt{D2}) \leftarrow \texttt{D2} < \texttt{D1}.$

3.6 Conclusion

Local $XY-$*stratification* and recursive queries with $\mathscr{P}reM$ constraints — both these declarative semantics are quite general, robust and allow efficient computation of stable models in polynomial time, thereby making them very attractive for a large number of modern Bigdata applications. However, they represent two different worlds of non-monotonic reasoning. In this chapter, we showed stratified recursive queries with $\mathscr{P}reM$ constraints offer better usability, intuitive understanding and comprehensibility than $XY-$stratification from the vantage point of declarative programming. Also, in terms of performance, $\mathscr{P}reM$ offers the benefits of relaxed synchronization with SSP distributed computing model, which under proper configurations can deliver great performance gains, particularly in presence of stragglers. This is all the more significant because now even programmers with a strong procedural coding background can attempt to write $XY-$stratified code first, which has a "procedural" flair and then possibly reduce it later to a pre-mappable query for better performance. On the other hand, we also examined cases where $\mathscr{P}reM$ does not hold. As such, for those cases, $XY-$stratification still remains the foremost choice for its superior expressivity.

CHAPTER 4

Expressing BigData Applications with $\mathscr{P}reM$

In this chapter, we show that several interesting applications can be expressed declaratively with the help of $\mathscr{P}reM$ using aggregates in recursion. Our examples are also used to show that $\mathscr{P}reM$ can be checked using simple techniques and templatized verification strategies. Thus a wide range of advanced BigData applications can now be expressed declaratively in logic-based languages, including Datalog, Prolog, and even SQL, which can then be executed efficiently at scale, since $\mathscr{P}reM$ optimizes the perfect-model semantics of these aggregate-stratified programs as well offers better amenability to parallelization as discussed in Chapter 2. Furthermore, we also formalize how the semantics of $\mathscr{P}reM$ can be combined with the continuous query semantics with respect to data streams. We elucidate that $\mathscr{P}reM$, not only allows the declarative specification of many complex continuous queries, but can also efficiently optimize their execution for real-time query response.

4.1 Evolution of Declarative Semantics

The growth of BigData applications added new vigor to the vision of Datalog researchers who sought to combine the expressive power demonstrated by recursive Prolog programs with the performance and scalability of relational database management systems (DBMSs). Earlier, research led to the delivery of a first commercial Datalog system [ACG15] and also had a significant impact on other languages and systems. In particular, DBMS vendors introduced support for recursive queries into their systems and into the SQL-2003 standards by adopting Datalog's (a) stratified

semantics for negation and aggregates, (b) optimization techniques that include (i) seminaive fixpoint computation, (ii) constant pushing for left/right-linear rules, and (iii) magic sets for linear rules. However, many algorithms and queries of practical interest cannot be expressed efficiently, or cannot be expressed at all, using stratified programs. This has motivated much research work seeking to go beyond stratification, often through the introduction of more powerful semantics, including semantics based on locally stratified programs, well-founded models and stable models, as discussed in Chapter 3.

On the other hand, concise expression and efficient support for a wide range of polynomial time algorithms, while keeping a stratification-based formal semantics, was made possible by the recent introduction of $\mathscr{P}reM$. In fact, in this chapter, we will use different case studies to show that simple aggregates in declarative recursive computation can express concisely and declaratively a host of advanced BigData applications ranging from graph analytics and dynamic programming (DP) based optimization problems to data mining and machine learning (ML) algorithms. However, in order to realize the significant potential offered by $\mathscr{P}reM$, the concept must be well-understood and its validity must be easy to verify for the applications of interest. Therefore, we next present different types of $\mathscr{P}reM$ that occur for different constraints and rules, which can be easily verified for semantic correctness by a data scientist or an application developer. These simple $\mathscr{P}reM$ verification strategies are very helpful in the context of the complex queries, discussed later in this chapter, to understand their correctness.

4.2 Different Manifestations of $\mathscr{P}reM$

We revisit the all pairs shortest path query from Chapter 2. The aggregate-stratified program (rules $r_{4.1.1} - r_{4.1.3}$) is shown below followed by the $\mathscr{P}reM$-optimized program (rules $r_{4.2.1} - r_{4.2.3}$):

$$r_{4.1.1} : \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}).$$

$$r_{4.1.2} : \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{path}(\texttt{X},\texttt{Z},\texttt{Dxz}), \texttt{arc}(\texttt{Z},\texttt{Y},\texttt{Dzy}), \texttt{D} = \texttt{Dxz} + \texttt{Dzy}.$$

$$r_{4.1.3} : \texttt{shortestpath}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}).$$

$$r_{4.2.1}: \texttt{path}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}).$$

$$r_{4.2.2}: \texttt{path}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{path}(\texttt{X},\texttt{Z},\texttt{Dxz}),\texttt{arc}(\texttt{Z},\texttt{Y},\texttt{Dzy}),\texttt{D} = \texttt{Dxz} + \texttt{Dzy}.$$

$$r_{4.2.3}: \texttt{shortestpath}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{path}(\texttt{X},\texttt{Y},\texttt{D}).$$

Using the symbols and notations from Chapter 2, which have the same meaning as described before, it is easy to that one can conveniently verify if $\mathscr{P}reM$ holds during the execution of a program by simply comparing $\gamma(T(I))$ and $\gamma(T(\gamma(I)))$ at each step of the recursive evaluation for a given set of facts I. However, strictly speaking, more formal tools [ZYI18] are required to prove that $\mathscr{P}reM$ holds for any possible execution of a given program.

We next introduce two special cases of $\mathscr{P}reM$ along with their formal definitions. These narrow definitions of specific instances of $\mathscr{P}reM$ are much easier to observe and verify.

- **Intrinsic** $\mathscr{P}reM$ (or $i\mathscr{P}reM$): The $\mathscr{P}reM$ of a constraint γ upon $T(I)$ will be called intrinsic when $T(I) = T(\gamma(I))$. To understand this, assume we replace $\texttt{D} = \texttt{Dxz} + \texttt{Dzy}$ with, say, $\texttt{D} = 3.14 * \texttt{Dzy}$ in rule $r_{4.2.2}$. Then obviously, the value of Dxz does not, in any way, affect the result computed in the head of the rule $r_{4.2.2}$. Thus, we could even select the min of these Dxz values, thereby eventually having $T(I) = T(\gamma(I))$ i.e. γ is $i\mathscr{P}reM$ in this case.

- **Radical**$\mathscr{P}reM$ (or $r\mathscr{P}reM$): The $\mathscr{P}reM$ of a constraint γ upon $T(I)$ will be called radical when $\gamma(T(I)) = T(\gamma(I))$. Consider the constraint $\texttt{X} = \texttt{a}$ in rule $r_{4.2.3}$ that specifies that we are only interested in the shortest paths that originate from node a. This constraint can be pushed all the way to the non-recursive base rule $r_{4.2.1}$, leaving the recursive rule unchanged, yielding the exact same results. This exemplifies $r\mathscr{P}reM$, which has been extensively researched in the Datalog literature, since it provides a very efficient optimization for equality constraints.

We next discuss the different category of BigData applications that can be expressed with $\mathscr{P}reM$ and how their correctness can be verified using the above special instances. In the rest of this chapter, we will separately mark out the use of $i\mathscr{P}reM$ and $r\mathscr{P}reM$, if applicable.

4.3 Dynamic Programming based Optimization Problem

Consider the classic *coin change* problem: given a value V and an infinite supply of each of $C_1, C_2, ..., C_n$ valued coins, what is the minimum number of coins needed to get change for V amount? Traditionally, declarative programming languages attempt to solve this through a stratified program: the lower stratum recursively enumerates over all the possible ways to make up the value V, while the min aggregate is applied at the next stratum to select the desired answer. Obviously, such simple stratified recursive solutions are computationally extremely inefficient. In procedural languages, these problems are solved efficiently with dynamic programming (DP) based optimization. Such DP based solutions utilize the "optimal substructure property" of the problem i.e., the optimal solution of the given problem can be evaluated from the optimal solutions of its sub-problems, which are, in turn, progressively calculated and stored in memory (memoization). For example, consider an extensional predicate coins having the atoms coins(2), coins(3) and coins(6), which represent coins with values 2 cents, 3 cents and 6 cents respectively. Now, we need at least 2 coins to make up the value $V = 9$ cents (3 cents + 6 cents). Note, we can also make up 6 cents using 3 coins of 2 cents each. However, the optimal solution to make up 9 cents should also in turn use the best alternative available to make up 6 cents, which is to use 1 coin of 6 cent itself. Based on this discussion, the example program below, described by rules $r_{4.3.1} - r_{4.3.2}$, shows how this solution can be succinctly expressed in Datalog with aggregate in recursion. This program can be executed in a top-down fashion and the optimal number of coins required to make up the change is determined by passing the value of V (9 in our example) to the recursive predicate num (as shown by the query goal).

$r_{4.3.1}: \texttt{num}(\texttt{C},1) \leftarrow \texttt{coins}(\texttt{C}).$

$r_{4.3.2}: \texttt{num}(\texttt{V},\texttt{min}\langle\texttt{N}\rangle) \leftarrow \texttt{coins}(\texttt{C}), \texttt{C} < \texttt{V}, \texttt{X} = \texttt{V} - \texttt{C}, \texttt{num}(\texttt{X},\texttt{Y}), \texttt{N} = \texttt{Y} + 1.$

$? - \texttt{num}(9,\texttt{N}).$

The successive bindings for the predicate num are calculated from the coin value C under consideration (as V - C) and are passed in a top-down manner (top-down information passing) till the exit rule $r_{4.3.1}$ is reached. The min aggregate inside recursion ensures that for every top-down

recursive call (sub-problem) only the optimal solution is retained. With this said materialization of the intensional predicate `num` (analogous to memoization), this program execution is almost akin to a DP based solution except one difference — pure DP based implementations are usually executed in a bottom-up manner. In the same vein, it is worth mentioning that many interesting DP algorithms (e.g., computing minimum number of operations required for a chain matrix multiplication) can also be effectively computed with queries, containing aggregates in recursion, using bottom-up semi-naive evaluation identical to the DP implementations. We next focus our attention on validating $\mathscr{P}reM$ for the above program. Note the definition of $\mathscr{P}reM$, $i\mathscr{P}reM$ or $r\mathscr{P}reM$ does not refer to any evaluation strategy for processing the recursive query i.e. the definitions are agnostic of top-down, bottom-up or magic sets based recursive query evaluation strategies. Interestingly, the use of "optimal substructure property" in DP algorithms itself guarantees the validity of $\mathscr{P}reM$. This can be illustrated as follows with respect to the `min` constraint: consider inserting an additional constraint $\bar{\gamma} = \mathtt{num}(\mathtt{X},\mathtt{min}\langle\mathtt{Y}\rangle)$ on $I = \mathtt{num}(\mathtt{V},\mathtt{N})$ in the recursive rule $r_{4.3.2}$. Naturally, any `Y`, which does not satisfy $\bar{\gamma}$, will produce a `N` that violates the `min` aggregate in the head of rule $r_{4.3.2}$ and hence will be discarded. Since, the imposition of $\bar{\gamma}$ in the rule body does not change the result when γ in the head (of rule $r_{4.3.2}$) is applied, the `min` constraint can be pushed inside recursion i.e., $\gamma(T(I)) = \gamma(T(\gamma(I)))$, thus validating $\mathscr{P}reM$.

4.4 K-Nearest Neighbors Classifier

K-nearest neighbors is a popular non-parametric instance-based lazy classifier, which stores all instances of the training data. Classification of a test point is computed based on a simple majority vote among K nearest[1] training instances of the test point, where the latter is assigned into the class that majority of the K neighbors belong to.

In the Datalog program, defined by rules $r_{4.4.1} - r_{4.4.7}$, the predicate `te(Id,X,Y)` denotes a relational instance of two-dimensional test points represented by their `Id` and coordinates `(X,Y)`. Likewise, the predicate `tr(Id,X,Y,Label)` denotes the relational instance of training points represented by their `Id`, coordinates `(X,Y)` and corresponding class `Label`. In this example,

[1] Based on metrics like Euclidean distance.

rule $r_{4.4.1}$ calculates the Euclidean distance between the test and all the training points, while the recursive rule $r_{4.4.3}$ with aggregate determines the nearest K neighbors for each of the test point. Symbolically, the predicate `nearestK(IdA,D,IdB,J)` represents the training instance `IdB` is the `J`-th nearest neighbor of the test point `IdA` located at a distance of `D` apart. Finally, rules $r_{4.4.4} - r_{4.4.5}$ aggregates the votes for different classes and performs the classification by majority voting. `cMax` in rule $r_{4.4.5}$ is a special construct that extracts the corresponding class `Label` that received the maximum votes for a given test point. Rule $r_{4.4.5}$ can be alternatively expressed without `cMax`, as shown in rules $r'_{4.4.5}, r''_{4.4.5}$. In terms of simple relational algebra, the constructs `cMin` or `cMax` can be thought of denoting the projection of specific columns (attributes like $\mathtt{Id_2}$ in $r_{4.4.3}$ and `Label` in $r_{4.4.5}$) from a tuple, which satisfies the `min` or `max` aggregate constraint respectively. However, these special constructs are mere syntactic sugar as illustrated before with equivalent rules $r'_{4.4.5}, r''_{4.4.5}$, which do not use any of these constructs.

$r_{4.4.1}: \mathtt{dist(Id_1,Id_2,D)} \leftarrow \mathtt{te(Id_1,X_1,Y_1),tr(Id_2,X_2,Y_2,Label),D = (X_1 - X_2)^2 + (Y_1 - Y_2)^2}.$

$r_{4.4.2}: \mathtt{nearestK(Id,-1,-1,nil)} \leftarrow \mathtt{te(Id,X,Y)}.$

$r_{4.4.3}: \mathtt{nearestK(Id_1,min\langle D\rangle,cMin\langle Id_2\rangle,J_1)} \leftarrow \mathtt{dist(Id_1,Id_2,D),nearestK(Id_1,S,Id_3,J),}$
$\mathtt{larger(S,Id_3,D,Id_2),J_1 = J+1,J_1 \leq K}.$

$r_{4.4.4}: \mathtt{votes(Id_1,Label,count\langle Id_2\rangle)} \leftarrow \mathtt{nearestK(Id_1,D,Id_2,J),tr(Id_2,X,Y,Label)}.$

$r_{4.4.5}: \mathtt{classify(Id_1,max\langle V\rangle,cMax\langle Label\rangle)} \leftarrow \mathtt{votes(Id_1,Label,V)}.$

$r_{4.4.6}: \mathtt{larger(S,Id_3,D,Id_2)} \leftarrow \mathtt{D > S}.$

$r_{4.4.7}: \mathtt{larger(S,Id_3,D,Id_2)} \leftarrow \mathtt{D = S,Id_2 > Id_3}.$

$r'_{4.4.5}: \mathtt{classify(Id_1,V,Label)} \leftarrow \mathtt{votes(Id_1,Label,V),\neg higher(Id_1,V)}.$

$r''_{4.4.5}: \mathtt{higher(Id_1,V)} \leftarrow \mathtt{votes(Id_1,Label,V),votes(Id_1,Label',W),W > V}.$

We now verify that the `min` aggregate in the recursive rule $r_{4.4.2} - r_{4.4.3}$ satisfies $\mathscr{P}reM$ and ensures semantic correctness. Note the exit rule $r_{4.4.2}$ always trivially satisfies the $\mathscr{P}reM$ definition, since the interpretation, I of the recursive predicate is initially an empty set. Thus, we focus our attention only on the recursive rule $r_{4.4.3}$. We now prove that $r_{4.4.3}$ satisfies $i\mathscr{P}reM$: consider

inserting an additional constraint $(\mathtt{Id_1}, \mathtt{J}, \mathtt{min}\langle\mathtt{S}\rangle)$ in the body of the rule $r_{4.4.3}$ that defines the `min` constraint on the recursive predicate `nearestK` in the body (creating an interpretation $\gamma(I)$ in the rule body). If this `min` constraint in the body ensures that for a given $\mathtt{Id_1}$ and J, S is the minimum distance of the J-th nearest neighbor, then for the corresponding valid $\mathtt{J_1}(\leq \mathtt{K})$, $r_{4.4.3}$ without the `min` aggregate in the head will produce all potential $\mathtt{J_1}$-th neighbors whose distances are higher than S (i.e., distance of J-th neighbor), thereby being identical to $T(I)$. Thus, we have, $T(I) = T(\gamma(I))$ validating $r_{4.4.3}$ satisfies *i𝒫reM*, since the recursive rule remains invariant to the inclusion of the additional constraint $(\mathtt{Id_1}, \mathtt{J}, \mathtt{min}\langle\mathtt{S}\rangle)$ in the rule body.

Similar to K-nearest neighbor classifier, several other data mining algorithms like K-spanning tree based graph clustering, vertex and edge based clustering, tree approximation of Bayesian networks, etc. — all depend on the discovery of a sub-sequence of elements in sorted order and can likewise be expressed with *𝒫reM* using aggregates in recursion. It is also worth observing that while our declarative K-nearest algorithm requires more lines of code than the other cases presented in this chapter, it can still be expressed with only seven lines of logical rules as compared to standard learning tools like *Scikit-learn* that implements this in 150+ lines of procedural or object-oriented code.

4.5 Iterative-Convergent Machine Learning Models

Iterative-convergent machine learning (ML) models like SVM, perceptron, linear regression, logistic regression models, etc. are often trained with batch gradient descent and can be written declaratively as Datalog programs with XY-stratification, as shown in [BBC12]. Rules $r_{4.5.1} - r_{4.5.3}$ show a simple XY-stratified program template to train a typical iterative-convergent machine learning model. J denotes the temporal argument, while `training_data` (in $r_{4.5.2}$) is an extensional predicate representing the training set and `model(J, M)` is an intensional predicate defining the model M learned at iteration J. The model is initialized using the predicate `init_model` and the X-rule $r_{4.5.2}$ computes the corresponding error E and gradient G at every iteration based on the current model and the training data using the predicate `compute` (defined according to the learning algorithm under consideration). The final Y-rule $r_{4.5.3}$ assigns

the new model for the next iteration based on the current model and the associated gradient using the `update` predicate (also defined according to the learning algorithm at hand). Since many iterative-convergent ML models are formulated as convex optimization problems, the error gradually reduces over iterations and the model converges when the error reduces below a threshold δ.

$$r_{4.5.1}: \mathtt{model(0,M)} \leftarrow \mathtt{init_model(M)}.$$

$$r_{4.5.2}: \mathtt{stats(J,E,G)} \leftarrow \mathtt{model(J,M), training_data(Id,R), compute(M,R,E,G)}.$$

$$r_{4.5.3}: \mathtt{model(J+1,M')} \leftarrow \mathtt{stats(J,E,G), model(J,M), update(M,G,M'), E} > \delta.$$

Interestingly, an equivalent version of the above program can be expressed with aggregates and pre-mappable constraints in recursion, as shown with rules $r'_{4.5.1} - r'_{4.5.4}$. The stopping criterion $\gamma: \mathtt{E} > \delta$ pushed inside the recursion in rule $r'_{4.5.3}$ satisfies $r\mathscr{P}reM$, since $T(\gamma(I))$ and $\gamma(T(I))$ would both generate the same atoms in `find`, where the error `E` is above the threshold δ (assuming convex optimization function). Also note, the `max` aggregate defined over the recursive predicate `find` trivially satisfies $i\mathscr{P}reM$.

$$r'_{4.5.1}: \mathtt{model(0,M)} \leftarrow \mathtt{init_model(M)}.$$

$$r'_{4.5.2}: \mathtt{stats(J,E,G)} \leftarrow \mathtt{model(J,M), training_data(Id,R), compute(M,R,E,G)}.$$

$$r'_{4.5.3}: \mathtt{find(max\langle J\rangle, cMax\langle M\rangle, cMax\langle E\rangle, cMax\langle G\rangle)} \leftarrow \mathtt{model(J,M), stats(J,E,G), E} > \delta.$$

$$r'_{4.5.4}: \mathtt{model(J_1,M')} \leftarrow \mathtt{find(J,M,E,G), update(M,G,M'), J_1 = J+1}.$$

4.6 $\mathscr{P}reM$ for Continuous Queries on Data Streams

The formal semantics of continuous query language (CQL) [ABW06] for data stream management systems (DSMS) was proposed in the last decade following the footsteps of structured query language (SQL) in a bid to streamline and speedup application programming for data streams, much like SQL standardized and simplified high-level database application programming in the 1990s. This eventually led to the development of a host of query engines for data streams like C-SPARQL [Gno10], EP-SPARQL [AFR11], Streaming SPARQL [BGJ08], etc. These systems have been quite successful inasmuch as they provided support for logical/physical window operators and also delivered continuous real-time *aggregated* query results with non-blocking implementa-

tion [BBD02] of aggregate (`SUM`, `COUNT`) and extrema operators (`MAX`, `MIN`), as opposed to using their blocking implementations [BBD02] that are commonly prevalent in the traditional relational database management systems. However, the dawn of IoT era brought the focus on advanced stream reasoning applications, which cannot be easily supported by CQL based systems. For example, consider the query of finding the minimum cost to travel from a place A to another place B considering different modes of transport, where the expenses for some of the transports vary over time (e.g. flight prices, on-demand cab fares, etc.). This example of a shortest path query on evolving graph sequences (EGS) [RLK11] illustrates the need for languages supporting recursive queries with aggregates. Unfortunately, most of the current generation of CQL systems do not support aggregates in recursion, or even simple recursion for that matter. This precludes them from expressing most graph queries. While some complex event processing engines [Hir12, MZZ12] are able to detect patterns and composite events over incoming streaming sequences via regular expression matching, these systems still fall woefully short of expressing many machine learning and data mining algorithms that can be expressed using aggregates in recursion [Yan17]. Without such generic support, these systems are only able to perform some basic specific inductive stream reasoning tasks using user-defined functions [BBC10]. In fact, this lack of expressive power is a major setback in building advanced stream reasoning applications quickly, as application programmers now have to largely rely on writing an increasing number of their own user defined functions in a procedural language [MD14].

Recently, many researchers discussed theoretical temporal extensions to Datalog [RKG18, Zan12, BDE15, MD14], which could serve as high-level declarative APIs for advanced stream reasoning applications. However, unfortunately, these Datalog extensions lacked robustness [ZYI17] and were still not adequate to deliver low-latency analytics on high velocity data streams [SGW16]. For example, monotonic aggregates are conceived as continuous integer functions and hence monotonic aggregate for `SUM` can only operate on integer arguments [MSZ13]; in addition the operational semantics of monotonic aggregates were still computationally too expensive and not conducive for real time responses. Therefore, in this section we will examine if we can efficiently combine the semantics of $\mathscr{P}reM$ with that of the semantics of continuous query, in particular the

Dataog semantics laid out by *Streamlog* [Zan12] for data streams.

4.6.1 Streamlog: Formal Semantics

We first briefly review the formal semantics of *Streamlog* here. The formal semantics of logic programming bounds negation of a predicate to be evaluated only when all the tuples for the predicate are known. In order to avoid this blocking implementation, *Progressive Closing World Assumption* (PCWA) was proposed in [Zan12] for data streams, where negation is applied considering tuples only seen till now. For example, the following query maintains the highest weight observed so far on a conveyer belt. The first arguments of the predicates denote the temporal argument/timestamp.

$$\texttt{maxWeight}(\texttt{T},\texttt{W}) \leftarrow \texttt{sensor}(\texttt{T},\texttt{W}), \sim \texttt{larger}(\texttt{T},\texttt{W}).$$

$$\texttt{larger}(\texttt{T},\texttt{W}) \leftarrow \texttt{maxWeight}(\texttt{T}_1,\texttt{W}_1), \texttt{T}_1 < \texttt{T}, \texttt{W}_1 > \texttt{W}.$$

If the readings from `sensor` stream arrive with increasing timestamps, then under PCWA, the above query can be stratified with respect to negation and can be hence evaluated in a non-blocking fashion. Such a query is called a *sequential program* [Zan12]. We next present several programs that use the operational semantics of $\mathscr{P}$*reM* for efficacy and robustness.

4.6.2 Streaming Applications with $\mathscr{P}$*reM*

We now present three streaming applications that uses $\mathscr{P}$*reM*. The semantics used here is similar to *Streamlog* [Zan12] under PCWA, except that all streaming records are assigned a *tuple identifier* as their first attribute, which consists of two parts: (i) a timestamp, `ts`, that is either explicitly specified or internally generated by the system, and (ii) an `id`, which is only used for specifying physical window lengths. The `id` (similar to `ROWID` in SQL) is always initialized internally by the system in successive order according to the arriving timestamp. The timestamp (`ts`) value in the head of a Datalog rule can be initialized internally using a built-in predicate that returns the system time periodically. It can also be initialized using the latest timestamp values of the tuples in the body of the rule. We follow similar syntax as before, except that the temporal arguments (i.e. the first attributes) are never considered as *group-by* arguments during aggregation; instead they are either initialized with the system time, or with the latest timestamp among all the tuples to be

aggregated upon. The declarative queries for each of the case studies are commented for clarity.

• **Computing Number of Reachable Paths.** In this example, we consider the temporal trust network maintained by the Bitcoin Alpha platform [KSS16], where each node represents a user and a directed edge from *u* to *v* denotes that *u* can trust *v*. Bitcoin platforms often need to monitor how many paths are reachable from a given node. This monitoring is essential to select paths, which can sufficiently obfuscate end-to-end transactions. We present a simple recursive query (`reachability`) enabled by $\mathscr{P}$*reM* that achieves this goal over an evolving trust network:

%Schema declaration.

$\mathtt{stream}(\{\mathtt{edge}(\mathtt{T}:\mathtt{TupleIdentifier},\mathtt{Source}:\mathtt{string},\mathtt{Target}:\mathtt{string})\})$.

%The base case.

$\mathtt{path}(\mathtt{T},\mathtt{S},\mathtt{D}) \leftarrow \mathtt{edge}(\mathtt{T_1},\mathtt{S},\mathtt{D}),\mathtt{T.ts} = \mathtt{T_1.ts}$.

%The recursive rule. Initializes/updates the tuple with latest timestamp.

%'larger' grounds T.ts to the larger of the two values: $T_1.ts$ or $T_2.ts$.

$\mathtt{path}(\mathtt{T},\mathtt{S},\mathtt{D}) \leftarrow \mathtt{path}(\mathtt{T_1},\mathtt{S},\mathtt{S_1}),\mathtt{path}(\mathtt{T_2},\mathtt{S_1},\mathtt{D}),\mathtt{larger}(\mathtt{T.ts},\mathtt{T_1.ts},\mathtt{T_2.ts})$.

%The final continuous query returns results as the system time updates.

$\mathtt{reachability}(\mathtt{T},\mathtt{S},\mathtt{count}\langle\mathtt{D}\rangle) \leftarrow \mathtt{path}(\mathtt{T_1},\mathtt{S},\mathtt{D}),\mathtt{system}(\mathtt{T.ts}),\mathtt{T.ts} \geq \mathtt{T_1.ts}$.

• **Computing Optimal Paths.** In this case study, we consider the fluctuating cab prices across different routes in Chicago [Chi]. Additionally, we also consider the city bus fares, which remain static over time. Bus rides are relatively cheaper but buses do not operate across all routes. Thus, modeling this data as a temporal graph, we can execute a continuous recursive query on it to monitor the minimum price to travel from one place to another. Such queries exemplify the use of extrema in recursion. Furthermore, this query also shows how streams can be merged with static databases, if necessary. This class of queries are particularly important in social networks to monitor metrics like (i) how the diameter of a network changes over time, or (ii) how the distance between two friends vary [LKF05]. It is also important to reiterate that such query computations in traditional Datalog systems may not terminate in presence of cycles. However, such situations

can be elegantly handled with $\mathscr{P}reM$ [ZYI17].

In this query, as the prices for a route decrease in real time, the classic semi-naive evaluation method [SYI16] automatically updates the other routes in a recursive manner, till the fixpoint is reached. In normal DSMS, such applications are often executed with user-defined functions which would require recomputation of all the affected paths whenever a route prices increases or decreases. However, with $\mathscr{P}reM$ recomputation is only necessary when route price rises. Thus, $\mathscr{P}reM$ also offers better query optimization opportunities.

```
%Input stream schema.
stream({cab(T : TupleIdentifier, Pickup : string, Drop : string, Fare : float)}).
%Input database schema.
database({bus(Stop1 : string, Stop2 : string, Fare : float)}).
%Optimal route computation.
route(T, S, D, min⟨P⟩) ← cab(T1, S, D, P), T.ts = T1.ts.
route(T, S, D, min⟨P⟩) ← bus(S, D, P), system(T.ts).
route(T, S, D, min⟨P⟩) ← route(T1, S, S1, P1), route(T2, S1, D, P2),
                          P = P1 + P2, larger(T.ts, T1.ts, T2.ts).
```

• **Computing Publication Impact.** We consider the high-energy physics citation network [LKF05] next. The impact of a publication within the research community can be gauged by computing its *pagerank* as the network grows over time. We present a query, which monitors the pagerank of a publication, as computed only from its recent citations (published within the last 365 days), while ignoring all its older citations. This query demonstrates how aggregates like `SUM` can be effectively computed on fractions inside recursion with $\mathscr{P}$*reM*; something which monotonic aggregates were unable to perform due to its operational semantics [MSZ13]. This query further demonstrates how logical window operators can be applied using our semantics. Physical windows can be similarly constructed using `id` instead of `ts`. For sake of clarity, we present a simplified pagerank algorithm without considering the damping factor.

%Input stream schema.

$\texttt{stream}(\{\texttt{pub}(\texttt{T}:\texttt{TupleIdentifier},\texttt{PaperId}:\texttt{string},\texttt{ReferredPaper}:\texttt{string})\}).$

%Computing total references for each paper.

$\texttt{ref}(\texttt{T},\texttt{P},\texttt{count}\langle\texttt{R}\rangle) \leftarrow \texttt{pub}(\texttt{T}_1,\texttt{P},\texttt{R}),\texttt{T.ts} = \texttt{T}_1\texttt{.ts}.$

%Simplified pagerank computation.

%'round' limits the number of decimal places for a floating point number.

%'in_days' returns time difference between two timestamps in days.

$\texttt{prank}(\texttt{T},\texttt{P},\texttt{V}) \leftarrow \texttt{ref}(\texttt{T}_1,\texttt{P},\texttt{R}),\texttt{V} = 1.0,\texttt{T.ts} = \texttt{T}_1\texttt{.ts}.$

$$\begin{aligned}\texttt{prank}(\texttt{T},\texttt{P},\texttt{sum}\langle\texttt{V}\rangle) \leftarrow\ & \texttt{prank}(\texttt{T}_1,\texttt{C},\texttt{W}),\texttt{pub}(\texttt{T}_2,\texttt{C},\texttt{P}),\texttt{ref}(\texttt{T}_3,\texttt{C},\texttt{R}),\\ & \texttt{V} = \texttt{round}(\frac{\texttt{W}}{\texttt{R}},2),\texttt{system}(\texttt{T.ts}),\\ & \texttt{in_days}(\texttt{T.ts} - \texttt{T2.ts}) \leq 365.\end{aligned}$$

4.7 Conclusion

Today BigData applications are often developed and operated in silos, which only support a particular family of tasks — e.g. only descriptive analytics or only graph analytics or only some ML models and so on. This lack of a unifying model makes development extremely ad hoc, and hard to port efficiently over multiple platforms as discussed in Chapter 1 . For instance, on many

graph applications native Scala with Apache Spark cannot match the performance of systems like RaSQL [GWM19], which can plan the best data partitioning/swapping strategy for the whole query and optimize the semi-naive evaluation accordingly. However, with $\mathscr{P}reM$, a simple extension to declarative programming model, which allows use of aggregates and easily verifiable pre-mappable constraints in recursion, can enable developers to write concise declarative programs (in Datalog, Prolog or SQL) and express a plethora of applications ranging from graph analytics to data mining and machine learning algorithms. This will also increase the productivity of developers and data scientists, since they can work only on the logical aspect of the program without being concerned about the underlying physical optimizations.

In this chapter, we also discussed how $\mathscr{P}reM$ can be combined with the formal semantics of *Streamlog* to express complex streaming applications. This is quite promising, paving the way toward future research in many interesting areas, where declarative recursive computation under SSP processing can be quite advantageous. For example, declarative advanced stream reasoning systems [DGZ18], supporting aggregates in recursion, can adopt distributed SSP model to query evolving graph data, especially when one portion of the network changes more rapidly as compared to others. SSP models under such scenario offer the flexibility to batch multiple network updates together, thereby reducing the communication costs effectively.

CHAPTER 5

Fast Frequent Itemset Mining from Data Streams

In this chapter, we study how compact data structures and representations can be very useful in several low latency data mining and analytical tasks. This assumes even more significance with respect to data streams, since online maintenance of such data structures can play a critical role in the overall latency of the tasks. We demonstrate this using the problem of mining exact frequent itemsets from data streams. Since the number of frequent patterns is often quite large, concise representations that save resources by avoiding redundancy are critical for an efficient lossless extraction of frequent patterns. Therefore, in this chapter, we introduce the novel concept of crucial patterns, and formally prove them to be an effective subset of closed frequent itemsets that assures lossless extraction. Furthermore, we also present a novel Crucial Pattern Mining (CPM) algorithm for data streams that includes robust optimization strategies for online maintenance of such compact data representations.

5.1 Introduction

Extracting frequent itemsets[1] over a continuous stream of transactions is an important task in many online data mining applications [CH08]. While many applications prefer approximate lossy extraction techniques [MM02], certain critical applications for credit card fraud detection, stock market prediction and anomaly detections rely on exact frequent itemset mining methods [MTZ08] that do not generate any false positives or false negatives. In addition, many of these data mining

[1] We will use the terms "itemsets" and "patterns" interchangably in this chapter.

applications also require the exact support of these itemsets for advanced machine learning tasks such as association rule deduction [AS94], classification [CRB06], etc. Often, however, there can be too many frequent patterns containing redundant information. In fact, the number of frequent itemsets is usually quite huge, particularly for low support thresholds or dense datasets that consist of strongly correlated transactions. In such cases, it may become cumbersome for analysts to gain meaningful insights from these patterns [TPB00]. As a result, several condensed representations [CRB06], [GLM14] have been proposed that concisely represent the complete information contained in all the frequent itemsets. Researchers also state that these concise representations are more useful, since analysts can easily deduce non-redundant association rules and other actionable knowledge from them [TPB00]. Moreover, these representations typically require less storage than all the frequent patterns together, thereby making them more suitable for data streams [CWY04]–[LC09]. Lossless representations allow one to recover all the frequent itemsets with their exact supports without accessing the original data. Furthermore, a good lossless representation should also allow fast and complete generation of important information from it (e.g. deducing relevant association rules or frequent itemsets with their exact supports) [CRB06]. This is particularly important for data stream applications that require real-time response. As of today, the most popular lossless condensed representation is the closed frequent itemsets [GLM14], [TPB00]. Unlike other representations, the latter is a small subset of all the frequent patterns and allows speedy extraction of relevant information from itself. In this chapter, we investigate an important research question: *is the set of closed frequent patterns an optimal subset of the frequent itemsets for lossless extraction. Or, does there exist a set of patterns with smaller cardinality from which one can quickly construct all the frequent itemsets with exact supports?*

5.2 Related Work

Over the years, a large number of algorithms have been proposed for lossless extraction that ranges from mining all frequent patterns to generating different lossless condensed representations. However, traditional data mining approaches proposed in the context of static databases like Apriori [AS94], FP-growth [HPY00], Charm [ZH02], Ndi [CG02], etc. require multiple scans of the

entire dataset. On the contrary, streaming algorithms need to process the incoming data in a single scan (one-pass constraint [LK06]) to cope with the high data arrival rate and provide real-time response. Furthermore, they also need to quickly detect and respond to concept-drifts [MTZ08]. Thus, several stream mining algorithms have been proposed following different window models [LJA14] to deal with these additional constraints, as discussed next.

Most of the recent lossless mining algorithms [MTZ08], [LKL05]– [TAJ09], are largely inspired from the traditional FP-growth method [HPY00]. The FP-growth algorithm recursively mines frequent itemsets from the FP-tree, which is created from two scans of the database; the first scan determines the order between the frequent items based on their supports and the second scan sorts the frequent items within a transaction and inserts them into the FP-tree (prefix tree). However, owing to the one-pass constraint for data streams, many of the reported methods like CanTree [LKL05] and DSTree [LK06] insert transactions into the prefix tree following a lexicographic or pre-determined canonical order, thereby omitting the need for the first scan. Consequently, the resulting prefix tree (with higher average depth and branching factor) does not have the same compact structure as an optimal FP-tree. This, in turn, drastically increases the memory usage and time required to mine frequent patterns from the prefix tree by FP-growth, as shown in our experimental results. The SWIM algorithm [MTZ08] attempts to reduce this mining time by implementing fast counting techniques (verifiers), but their eventual performance is also bottlenecked by the large size of the lexicographic prefix trees. The most recent work CPS-tree [TAJ09] tries to mitigate this by maintaining an optimal prefix tree (based on the descending order of support). This significantly reduces the mining time, but adds considerable overhead towards maintaining the tree, since it needs to continuously rearrange the prefix tree optimally with the arrival of incoming data (particularly for concept-drifting streams). Thus, in this chapter, we also examine *how to maintain a close-to-optimal prefix tree that can be mined quickly, but also has a low maintenance cost.*

The approaches mentioned above [MTZ08], [LKL05]– [TAJ09] use the conventional recursive FP-growth technique [HPY00] to extract frequent itemsets from conditional pattern trees. This leads to unnecessary FP-growth calls over subsequent windows, performing repeated construction

and mining of similar conditional pattern trees, which eventually yield the same frequent patterns most of the times (although their corresponding supports may vary). Other related algorithms, like Moment [CWY04], CFI-stream [JG06] and NDFIoDS [LC09], attempt to avoid this by maintaining an information lattice (or index [CKN08]), comprising of a subset of the frequent patterns (condensed representations) along with some other gateway (border) elements. As the new data arrives, these algorithms update the support of the existing patterns in the information lattice accordingly. However, this update operation is quite expensive, since it often has to re-mine the old data along with the new one, whenever it appears that a new pattern (absent in the lattice) may become frequent (although the potential candidate may turn out to be infrequent in the end). Thus, we also study in this chapter *how to reduce the redundant FP-growth calls and restrict their execution (i) to only when we are certain of identifying new frequent patterns from existing branches, or (ii) to mine new branches inserted into the prefix tree*.

5.3 Contributions

In the rest of this chapter, we address the issues highlighted in Sections 5.1 and 5.2 and make the following contributions:

- We prove that computing the optimal lossless condensed representation for a set of frequent patterns in general is an NP-hard problem.
- Thereafter, we introduce the notion of "crucial patterns" and show that, under reasonable constraints, the set of crucial patterns does become an optimal subset of the closed frequent itemsets for lossless extraction.
- Furthermore, we propose a novel Crucial Pattern Mining (CPM) algorithm for data streams. In this approach, we construct and maintain a close-to-optimal prefix tree over subsequent window slides.
- We also present a robust delta-maintenance strategy that intelligently mines the prefix tree, avoiding unnecessary computation.

5.4 Preliminaries

Let $I = \{i_1, ..., i_n\}$ be a set of items and S be a continuous stream of incoming transactions t_i, where every $t_i \subseteq I$. We will analyze S using the popular sliding window model [LJA14], where only a fixed length of recently generated data is considered, as shown in Figure 5.1. We now formally define the following terms:

- *Support: Support* of any pattern $A \subseteq I$ is defined as the number of transactions in window W that contain A.
- *Frequent Pattern:* A pattern is defined to be *frequent*, if its support is not less than a minimum support threshold ε.
- *Set Enumeration Tree:* Given an order $\prec$ between the items in I, the complete set of all the frequent patterns with their supports can be represented by a prefix tree, known as the *set enumeration tree*. Figure 5.2(a) shows an example, where lexicographic ordering is used to represent all the frequent patterns mined from the first sliding window (Figure 5.1) with $\varepsilon = 3$. Alternatively, since the set of frequent patterns form a partially ordered set (poset) with respect to the subset relation, the former can also be represented by a *Hasse* diagram [ES13], as shown in Figure 5.2(b). We will use these frequent patterns as a running example in the subsequent definitions.
- *Lossless Frequent Itemset Mining:* Given a set of transactions and ε, the *lossless frequent itemset mining* problem should return a set of frequent patterns with their supports and some additional information (if needed) from which the exact set enumeration tree can be constructed uniquely. However, if the set enumeration tree cannot be built precisely every time for any ε, then the mining approach is termed as *lossy extraction*.
- *Candidate free construction:* Given the output of a lossless mining algorithm, if the corresponding set enumeration tree can be built from it, without generating any infrequent candidates, then the latter process is called *candidate free construction*.

- *Candidate pruning based construction:* Contrary to the previous definition, a *candidate pruning based construction* generates infrequent candidates and later discards them to build the set enumeration tree. This process is usually extremely time consuming.
- *Closed Pattern:* A pattern P is defined to be *closed*, if it has no superset Q ($P \subset Q$) with the same support (e.g. x, bx and xy). It is easily verifiable that the exact set enumeration tree can be built from the closed frequent itemsets (and their supports) by generating all of their possible subsets (lossless extraction). This is an example of candidate free construction.
- *Maximal Itemset:* A frequent itemset P is called *maximal*, if it has no superset Q ($P \subset Q$) which is also frequent (e.g. bx and xy). One can always get all the frequent patterns from the maximal itemsets, but not their exact supports (lossy extraction). E.g., support of x cannot be inferred from the supports of bx and xy.
- *Non-Derivable Itemset:* Upper and lower bounds on the support of an itemset can be determined from all its subsets using the inclusion-exclusion principle [CG02]. Based on this, itemsets are called *derivable*, if the maximum lower bound and minimum upper bound on their supports are equal; otherwise they are termed as *non-derivable* (e.g. b,x and y). It is possible to build the set enumeration tree from frequent non-derivable itemsets using candidate pruning based construction. E.g. by will be generated and pruned.

Other less popular representations, like *frequent generators* and *disjunction free sets* [CRB06], require substantial amount of additional information (like border elements) which increases their overall size (as later discussed in our experimental results).

5.5 Crucial Patterns

In this section, we formally introduce the notion of crucial patterns and theoretically prove its important properties. For this purpose, we first define the term *branch id* in the context of FP-trees. A branch id is an unique identifier which is assigned to a FP-tree node, *(i)* if its support exceeds the total support of its children, or *(ii)* if it does not have any child. If a node is not

TID	Items
t_1	a, b, x, y
t_2	c, x, y
t_3	a, b, c, x, z
t_4	b, x, y, z
t_5	c, x, y, z
t_6	x, y, z

Time

Window #1

Window #2

Figure 5.1: A sliding window example with *window size=4* and *slide size=2*.

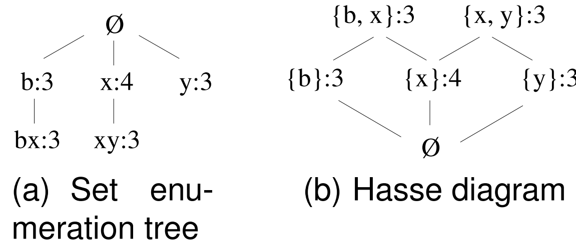


Figure 5.2: All frequent patterns represented with their respective supports.

assigned a branch id, it collectively uses the branch ids of all its children. Figure 5.3(a) shows an example. These branch ids can be treated as virtual transaction ids (several times fewer than actual number of transactions) while creating the conditional pattern bases during FP-growth. Figure 5.3(b) depicts the corresponding conditional FP-tree after retaining the original branch ids. As the figure shows, the support at each node in the conditional FP-tree is a linear sum of the supports contained in the actual FP-tree nodes with the associated branch ids. In other words, the support for any frequent itemset generated from the conditional pattern base (projected database) of an element i in the FP-tree is simply a linear combination of the supports of the actual FP-tree nodes that contain the element i. Figure 5.3(c) presents an example. We will call the set of branch ids whose linear sum forms the support of a frequent pattern as its *valid branch combination*. For example, the set $\{6,9\}$ is the valid branch combination of the frequent pattern *ed*.

Now, let $\mathscr{F}$ denote the set of all the frequent patterns such that every $f \in \mathscr{F}$ has a valid branch combination b_f with storage cost c_f. Given the set of all the branch ids of the frequent patterns (V), we are interested in finding a subset $E \subseteq \mathscr{F}$ with the minimum total storage cost that can be used for lossless extraction (*optimal lossless extraction*). The corresponding decision version L of

this optimization problem can be stated as follows: given the input pair $(\mathscr{F},V)$ and an integer k, the decision problem should answer whether there exists a subset $E \subseteq \mathscr{F}$ for lossless extraction with $\sum_{e \in E} c_e \leq k$.

Claim 5.1. *L is in **NP**.*

Proof. The following verifier for L runs in time polynomial in the length of the inputs:

Verifier $\mathbb{V}$ $(<\mathscr{F},V,k>,<E>)$:

1. Construct $U = \{u | u \in X \wedge X \in \mathscr{F}\}$
2. Construct $T = \{e | e \in X \wedge X \in E\}$
3. $\forall e \in U$, Construct $V_e = \{i | e \in X \wedge X \in \mathscr{F} \wedge i \in b_X\}$
4. $\forall e \in T$, Construct $W_e = \{i | e \in X \wedge X \in E \wedge i \in b_X\}$
5. If the following are all true then accept else reject:

   i $U = T$ (all frequent items included)

   ii $\forall e \in U, V_e = W_e$ (all <branch id, item> included)

   iii $\sum_{e \in E} c_e \leq k$ (total cost $\leq k$)

□

Claim 5.2. Weighted Set Cover $\leq_p L$.

Proof. The weighted set cover problem takes as input $<U,\mathscr{A},k>$, where $\mathscr{A}$ is a set whose member $i \subseteq U$ with cost c_i and $k \in \mathbb{N}$, to answer the question if U has an $\mathscr{A}$-cover of cost k.

Let us define a function f that takes as input $<U,\mathscr{A},k>$ and output $<\mathscr{F},V,k>$. We perform this by computing mutually exclusive sets D_i from $\mathscr{A}$ in polynomial time, such that $D_i \cap D_j = \emptyset$, $\forall i \neq j$ and $\bigcup_i D_i = U$, where $i \in \mathbb{N}$.

Now, $\mathscr{F}$ can be computed as follows:

$\forall w \in \mathscr{A} \implies w \in \mathscr{F}$ with cost c_w and valid branch combination $\{i\}$ where $w \subseteq D_i$.

Suppose U has a weighted $\mathscr{A}$-cover $\mathscr{B}$ of cost $\leq k$. Then it can be trivially shown, $\mathscr{B}$ can be used for lossless extraction of $\mathscr{F}$, since all frequent items and their corresponding valid branches are covered.

Conversely, if $\mathscr{B}$ can be used for lossless extraction of $\mathscr{F}$, then $\bigcup_{w \in \mathscr{B}} w = U$ and total cost of $\mathscr{B} \leq k$. Thus $\mathscr{B}$ is also a weighted $\mathscr{A}$-cover of cost $\leq k$. □

Theorem 5.3. *The decision version L of the lossless extraction problem is NP-complete.*

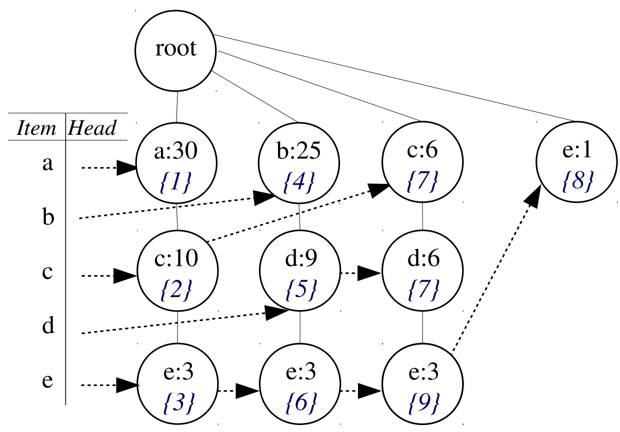
Proof. It directly follows from *Claim* 5.1 and 5.2 that *(i) L* is in *NP* and *(ii)* the *Weighted Set Cover NP*-complete problem can be reduced to L in polynomial time. □

Since the decision version is *NP*-complete, it follows that the optimal lossless extraction problem is *NP*-hard. Moreover, given an optimal subset E, one may need to use the expensive candidate pruning based construction (discussed in Section 5.4) to build the full set enumeration tree, which involves joining of patterns in E on their branch ids (e.g. *eca* derived from *ec* and *ea*) and examining several infrequent candidates. However, if we restrict only to candidate free construction (defined in Section 5.4) for building the set enumeration tree, then it can be shown that, an optimal subset exists under this new constraint. We now formally present the definition of crucial patterns.

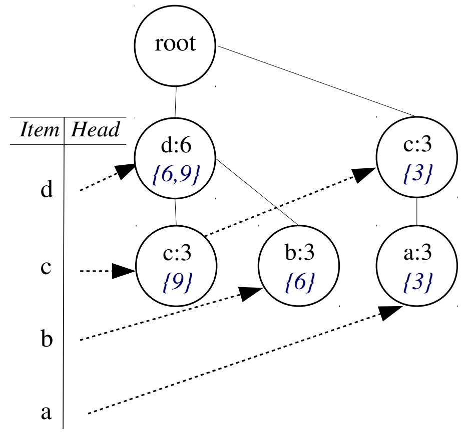
A frequent itemset P is called a *crucial pattern* if its valid branch combination has at least one branch id, which is not present in the valid branch combination of any of its frequent superset Q ($P \subset Q$). The unique branch ids corresponding to P will be called its *demarcating branches*. E.g., *e*, *edc*, *edb*, *eca* in Figure 5.3(c) are the only 4 crucial patterns with demarcating branches $\{8\}, \{9\}, \{6\}$ and $\{3\}$ respectively. Except *e*, all other crucial patterns are also maximal patterns, whereas, adding *ed* to the set of crucial patterns, we obtain the complete set of closed frequent itemsets. Now we prove some important properties of crucial patterns.

5.6 Properties of Crucial Patterns

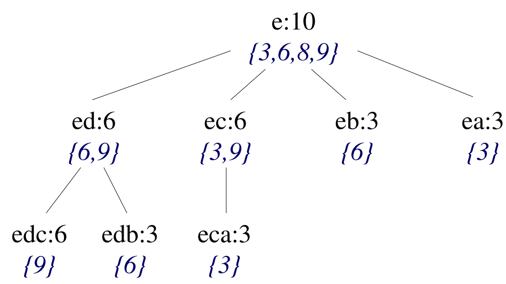
Lemma 5.4. *Let P, Q be frequent patterns with valid branch combinations B_P and B_Q respectively such that $P \subset Q$. Then, (i) $B_P \supseteq B_Q$ and (ii) if, support of P = support of Q, we have, $B_P = B_Q$.*



(a) A FP-tree with branch ids



(b) Conditional FP-tree built for e with $\varepsilon = 3$



(c) Final set enumeration tree derived from the projected database of e

Figure 5.3: Mining frequent itemsets with associated branch ids from FP-tree.

Proof. (i) Since, $P \subset Q$, all the branches (prefixes in the FP-tree) containing Q must also contain P i.e. $B_P \supseteq B_Q$. (ii) Furthermore, if support of P = support of Q, then B_P cannot contain any branch id which is absent in B_Q, otherwise support of P would be greater. Therefore, $B_P = B_Q$. □

Lemma 5.5. *Let P be a crucial pattern with valid branch combination B_P. If $Q \supset P$ is a frequent pattern with valid branch combination B_Q, then, $B_P \supset B_Q$.*

Proof. Since, every crucial pattern is also a frequent pattern, it follows directly from *Lemma* 5.4(i) that $B_P \supseteq B_Q$. However, by the definition of crucial pattern, P must contain at least one branch id which is not present in Q. Therefore, $B_P \neq B_Q$. Thus, $B_P \supset B_Q$. □

Lemma 5.6. *Every maximal pattern is also a crucial pattern.*

Proof. By definition, a maximal pattern does not have any superset which is also frequent. Hence, the lemma follows trivially from the definition of crucial pattern. □

Lemma 5.7. *Every crucial pattern is also a closed frequent itemset.*

Proof. Let P be a crucial pattern. We assume that P is not a closed itemset. Since, P is not closed, there must exist a pattern $Q \supset P$, such that support of Q = support of P. Since P is frequent, Q is also frequent. Now, according to *Lemma* 5.4(ii), the valid branch combination of P and Q must be same. But this violates *Lemma* 5.5, since P is a crucial pattern. Therefore, P must be a closed itemset. Thus, every crucial pattern is frequent (by definition) and closed. □

Theorem 5.8. *If M, E, C are the sets of maximal, crucial and closed frequent patterns respectively, then $M \subseteq E \subseteq C$.*

Proof. Follows directly from *Lemma* 5.6 and 5.7. □

Theorem 5.9. *If X is the set of crucial patterns, then there does not exist any $Y \subset X$ that can build the set enumeration tree, using candidate free generation.*

Proof. Let us assume that such a set $Y \subset X$ exists. Since, Y is a proper subset, there must exist at least one crucial pattern $P \in X - Y$. By our assumption, P and its support can be calculated from

the crucial patterns in Y using candidate free generation. However, exact support of P cannot be calculated, if any of its branch id is missing from the valid branch combinations of its supersets in Y. And, by definition, P must contain atleast one unique demarcating branch id. Therefore, this contradicts our assumption. Thus, the set of crucial patterns is an optimal subset of the closed frequent patterns, which can be used for lossless extraction, based on candidate free generation. □

5.7 Set Enumeration Tree Construction

Let the set $V_i = \{1, .., k\}$ denotes the valid branch combination of the ith frequent item. Also, let there be a column vector $\mathbf{C_i} = [s_1 s_2 ... s_k]^T$, where s_j is the support of branch id j $(1 \leq j \leq k)$. The valid branch combination of any crucial pattern derived from the projected database of i can be represented by a $1 \times k$ Boolean row vector (bitmap), where a '1' at position j indicates the presence of branch id j in the valid branch combination. Since, the crucial patterns are a superset of the maximal patterns (*Theorem* 5.8), we can trivially find all the frequent patterns from them by generating all of their subsets (candidate free construction). Now interestingly, the bitmap representing the valid branch combination of any non-crucial frequent pattern f is simply the logical *OR* of those bitmaps, whose corresponding crucial patterns generate the subset f. With this information, we can prepare an $n \times k$ matrix $\mathbf{R}$ for any n relevant frequent patterns, whose exact supports we are interested to know. The rows of $\mathbf{R}$ contain the bitmaps of the relevant frequent patterns and the matrix multiplication $\mathbf{RC_i}$ results in their exact supports. The set enumeration tree corresponding to the projected database of i can be thus trivially constructed, if the supports of all the frequent patterns are calculated. Since, in FP-trees, all projected databases are created by taking only the prefix paths, frequent patterns computed from these projected databases are mutually exclusive and completely exhaustive [HPY00]. Thus, the forest of all the set enumeration trees, derived from the conditional pattern bases of the frequent items, represent the overall set enumeration tree.

5.8 Crucial Pattern Mining Algorithm

We now present our crucial pattern mining (CPM) algorithm, which comprises of three main components as described below:

5.8.1 Initialization

At the end of the first window, we construct an optimal prefix tree by trivially building a lexicographic tree in one pass and then restructuring it at the end (similar to the CPS-tree). We also maintain a list of the items with their running support. At every node, the support for each of the individual window panes [LK06], [TAJ09] is stored in an array. When a slide expires, this array needs to be shifted accordingly (Figure 5.5). This is performed lazily following the strategy discussed in [LK06]. Once the tree is constructed, we assign branch ids only to the nodes containing frequent items as discussed in Section 5.5. All branch ids are reassigned when complete restructuring is performed on the tree (Section 5.8.2.2). The crucial patterns are computed next following algorithm 5.1. The term T_{val} in the algorithm is used for a pattern to denote the highest support among the infrequent items contained in its projected database. T_{val} is used later in Section 5.8.3.1.

5.8.2 Delta Maintenance of Prefix Tree.

5.8.2.1 Close-to-optimal Prefix Tree.

The prefix tree constructed before has a distinct node boundary that separates the frequent items from the infrequent ones. It is easy to see that the performance of the mining algorithm depends on the compactness of the subtree consisting of the frequent items (with branch ids). The following method tries to retain this compactness as much as possible without incurring high maintenance cost. In this approach, given a new transaction, we will use two ordering schemes to sort it – $\prec_{est}$ (established order) for "once frequent" items and $\prec_{desc}$ (descending order of current support) for "still infrequent" items. $\prec_{est}$ indicates the ordering between the frequent items when they were first inserted into the tree above the said node boundary. Once these items were inserted above the boundary, their relative ordering has been established and should not change. On the contrary,

Algorithm 5.1 Crucial Pattern Computation

Input: $T \leftarrow$ Prefix-tree with branch ids, $e \leftarrow$ frequent item, $V_e \leftarrow$ valid branch combination of e in T, $B \leftarrow$ set of all *uncovered* branches, $\alpha \leftarrow null$

Output: Append triplets $< P, V_P, \delta_P >$ to list L, where P is a crucial pattern, computed from projected database of e, with corresponding valid branch combination V_P and $T_{val} = \delta_P$

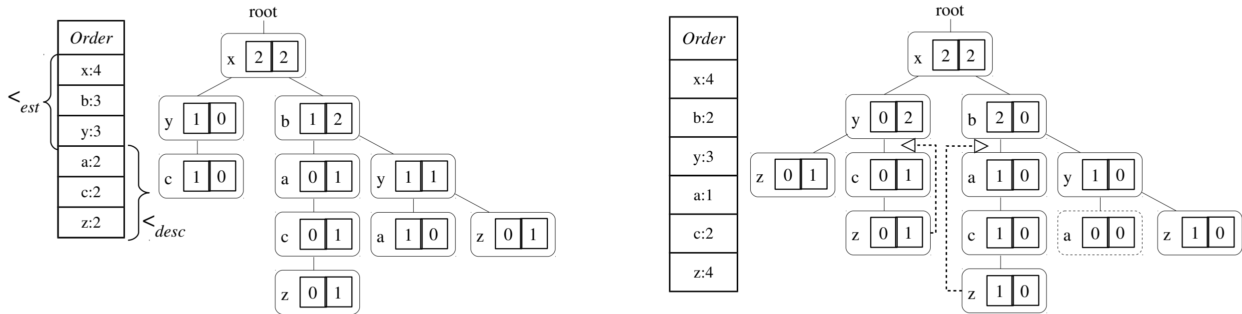
1: **procedure** COMP–CRUCIAL–PATTERN(T, e, V_e, α)
2:     **for all** frequent items f in $T|e$ **do**// cond. on e
3:         $compCrucialPattern(T|e, f, V_f, \alpha \cup f)$
4:     **end for**
5:     $isCrucial \leftarrow$ **false**
6:     **if** $V_e \cap B \neq \emptyset$ **then**
7:         $isCrucial \leftarrow$ **true**
8:     **else if** $V_e - \cup_{x \in L \wedge x \supset (\alpha \cup e)} V_x \neq \emptyset$ **then**
9:         $isCrucial \leftarrow$ **true**// by definition
10:     **end if**
11:     **if** $isCrucial =$ **true then**
12:         $\delta \leftarrow$ max. support of infrequent item $\in T|e$
13:         $L \leftarrow L \cup < \alpha \cup e, V_e, \delta >$
14:         $B \leftarrow B - V_e$
15:     **end if**
16: **end procedure**

Figure 5.4: Algorithm for computing crucial patterns.

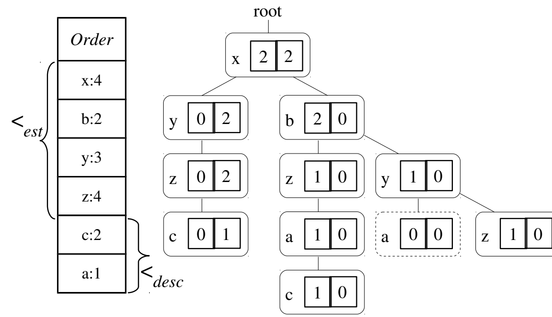
$\prec_{desc}$ can change as the window slides. But only the infrequent items are ordered according to the latter. Now as the window slides, it is important to consider two cases – *(a)* frequent item becoming infrequent and *(b)* infrequent item becoming frequent. For *(a)*, the nodes containing the item are in the upper portion of the prefix tree and no action is taken with regards to it. This is because all crucial patterns have been computed earlier by projecting conditional pattern bases, based on $\prec_{est}$ and hence changing it can impact the results. Therefore the relative ordering of the "once frequent" items ($\prec_{est}$) and their established position in the tree is not disturbed. However, for *(b)*, the infrequent item, that became frequent, was never conditionalized before. Hence, we push this element up through the boundary and append it to $\prec_{est}$. This element from now onwards will be sorted following $\prec_{est}$. This "*bubble up*" movement may result in a trivial split up or merging of branches. Figure 5.5 illustrates the entire process for the sliding window example in §2. The branch ids are omitted in this figure for clarity. Note, the subtree comprising of the items, defined by $\prec_{est}$, may not be optimal but is still significantly better than the corresponding lexicographically ordered tree resulting in smaller mining cost. Moreover, this adjustment is considerably less expensive than restructuring the entire tree (including the infrequent items) at every window slide, as performed in the CPS-tree.

5.8.2.2 Periodic Complete Restructuring.

The semi-optimal prefix tree, discussed above, ensures that the conditional pattern bases (obtained during mining) remain small in size, thus leading to reduced mining time. However, over many slides, the semi-optimal tree will accumulate considerable number of "*garbage*" nodes (n_1) i.e. nodes with zero support and many infrequent elements (n_2) in the upper portion. Although garbage nodes will predominantly occur in the lower portion of the tree (Figure 5.5(c)), it nevertheless can adversely impact the performance as discussed in [TAJ09] and also shown in our experiments. Therefore, we use a threshold θ, such that, if $(n_1 + n_2)/N > \theta$ (N being the number of nodes in the upper portion), then we restructure the complete tree using path adjusting method [KS04], assign new branch ids and generate the crucial patterns once again.



(a) Tree at end of slide 1 (b) New frequent item will bubble up



(c) Tree at end of slide 2

Figure 5.5: Maintaining a close-to-optimal prefix tree.

5.8.3 Update of Crucial Patterns.

5.8.3.1 Incoming Slide.

As new transactions arrive and are inserted into the tree, the following cases can occur: *(a)* infrequent items can become frequent, *(b)* new branches for a frequent item can appear and *(c)* existing branches of a frequent item can get updated (*support* incremented). For *(a)*, after the subsequent bubble up, we assign branch ids to the nodes containing the new frequent item and compute new crucial patterns from its projected database only. As for *(b)*, we add the new branches to B and execute algorithm 5.1 for the associated items only. Lastly, for *(c)*, we check if the support in the branches increased by $\varepsilon - T_{val}$ or more for any crucial pattern (T_{val} constraint), where ε is the minimum support threshold. This is because we can derive new crucial patterns simply by re-mining only those valid branch combinations whose corresponding patterns have violated the T_{val} constraint. Overall, these strategies mitigate the issues discussed in Section 5.2.

5.8.3.2 Expiring Slide.

As slides are expired, the corresponding support of the crucial patterns are reduced and they can become obsolete. It is possible to still build the set enumeration tree from these patterns, even when some of them are infrequent. But, we remove these obsolete patterns to reduce memory overhead and keep the search space small. Therefore, in this case, we add the demarcating branches of the obsolete pattern P to B and then apply the algorithm 5.1 on the valid branch combinations of P's subsets to include the frequent itemset(s) that was previously dependent on P (because of these demarcating branches) as a crucial pattern.

5.9 Experiments

In this section, we report the experiments conducted to evaluate the performance of our crucial pattern mining algorithm.

5.9.1 Setup

We designed a common framework using the Complex Event Processing engine *Esper* and implemented all the lossless extraction algorithms in Java[2]. All the experiments were run using Esper-5.1.0[3] and Java 1.7 on a machine, with one Intel i5-4210U CPU and 8GB memory, running Ubuntu 14.04 LTS. Each of the experiments was conducted 10 times for physical sliding windows and the averaged results have been reported here.

5.9.2 Datasets

We ran our experiments on both sparse (*T10I4D100K*, *Retail*) and dense (*Chess*) datasets[4], which have been used in many of the previous related works. *T10I4D100K* is a synthetic dataset (produced by the IBM market-basket generator) while *Chess* and *Retail* are real-world datasets. We

[2]Some of the codes were reused from
`http://www.philippe-fournier-viger.com/spmf` and
`http://adrem.ua.ac.be/~goethals/software.`

[3]`http://www.espertech.com/esper.`

[4]Datasets and their descriptions are available at
`http://fimi.ua.ac.be/data.`

highlight the results from the *T10I4D100K* dataset, since results from other datasets were consistent as well.

5.9.3 Results

We first run experiments to compare the computation time of CPM with different lossless extraction methods discussed in §1.1. We mainly compare the results of CPM with DSTree [LK06], CPS-tree [TAJ09] and SWIM [MTZ08], as they are the recent state-of-the-art methods which have outperformed previous techniques like CanTree [LKL05] and Moment [CWY04] by several orders of magnitude. Furthermore, since the performances of Moment and NDFIoDS [LC09] were comparable, the latter was not compared with CPM. Lastly, due to unavailability of CFI-stream [JG06] source code, we could not compare it to CPM. However, the results reported in [JG06], [MTZ08] imply CFI-stream to be slower than SWIM. Figures 5.6 - 5.8 present the corresponding experimental results obtained from the T10I4D100K dataset for window size 10K, slide size 2.5K and $\theta = 0.25$.

Figure 5.6 plots the average mining time per slide for different minimum support thresholds. As shown in the figure, CPM is orders of magnitude faster than other state-of-the-art methods in terms of mining time. In fact, even for higher supports, CPM is more than 3 times faster than its nearest competitor SWIM. This is mainly due to the fact that CPM avoids repeated mining of the same crucial patterns between the slides. On the other hand, CPS-tree, though optimal in size, has to repeatedly mine the same patterns, while SWIM has to run its verifiers against the window panes several times to update the support of the frequent patterns. In case of DSTree, mining time is significantly higher because of its non-optimal structure and the accumulation of garbage nodes over time. Interestingly, unlike its competitors, the mining time of CPM is not strictly decreasing with increasing support thresholds (also observed in [CWY04]). This is primarily because, in CPM, the number of mining calls in a slide is directly dependent on how many new crucial patterns are emerging. If the overlap among the crucial patterns between two consecutive slides is less (e.g. at support=0.1%), the number of mining calls will be more, thus leading to a higher mining time. The distribution of the data eventually decides this overlap. But, even in the worst case,

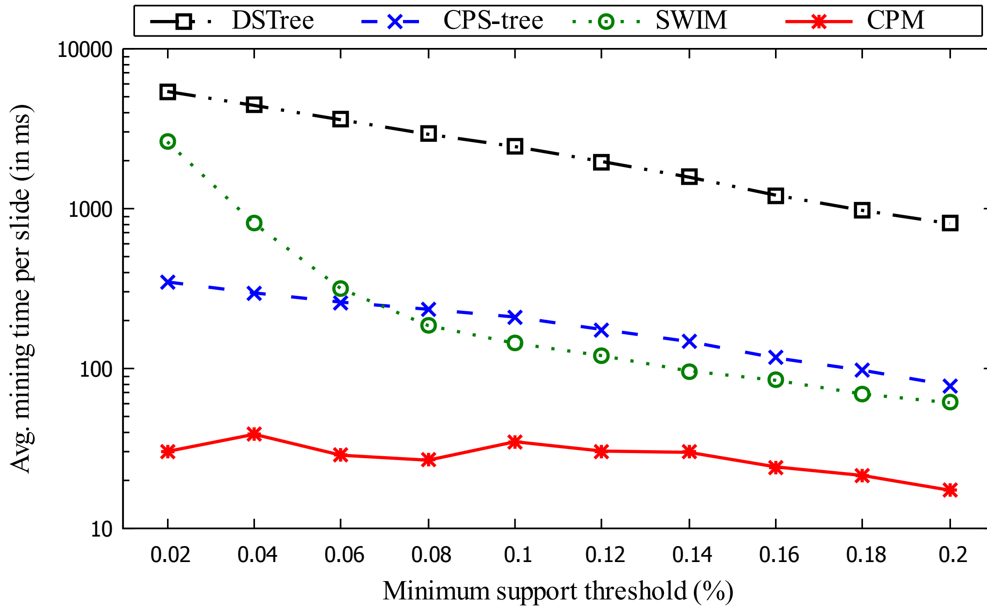


Figure 5.6: Mining time comparison for lossless extraction methods.

the performance of CPM will be better than its competitors, because it enumerates through the smallest subset of the frequent patterns (by *Theorem* 5.9).

Figure 5.7 presents the average delta maintenance time per slide for each of the above mentioned algorithms across different minimum supports. The best delta maintenance time is offered by SWIM which is only marginally less than CPM. This is because SWIM simply creates a lexicographic prefix tree from the incoming transactions and expires the oldest window pane; whereas CPM, apart from inserting transactions into the prefix tree, also moves up new frequent items and occasionally realigns the tree (although rarely) adding to the delta maintenance. It must be pointed out that, when new crucial patterns emerge or old ones become obsolete at a higher rate (e.g. at support=0.1%), the maintenance time is slightly higher due to increased number of adjustments for new frequent items. On the other hand, at higher supports, when the rate of emergence of new frequent items is less (i.e. mainly dealing with the heavy hitters), maintenance time of CPM is almost same as SWIM. On the contrary, CPS-tree is realigned at every slide to form an optimal prefix tree, resulting in an almost constant maintenance cost, which is higher than both SWIM and CPM. Although transactions are simply inserted into the DSTree, their delta maintenance cost is surprisingly very high. We investigated this further and found that this is again mainly due to the

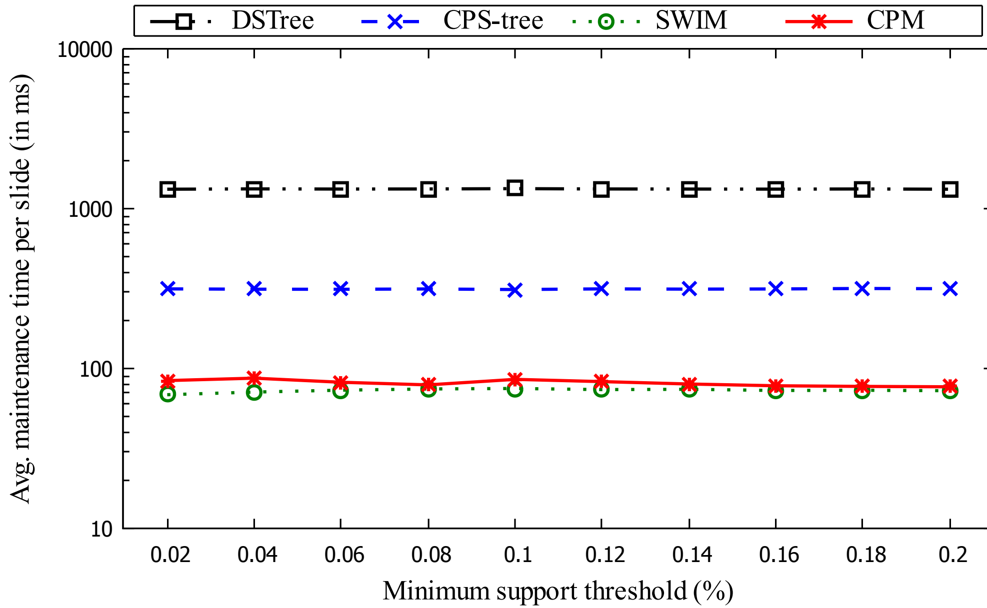


Figure 5.7: Delta maintenance time comparison for lossless extraction methods.

accumulation of large amount of garbage nodes over time, which makes search for a child at a node much more time consuming and thus, even inserting or updating a transaction becomes cumbersome and expensive. SWIM and CPS-tree does not suffer from the problem of garbage nodes. Although CPM can potentially suffer from this problem, but our periodic realignment resolves it without much overhead.

Figure 5.8 reports the average running time (sum of delta maintenance and mining time) per slide of the different algorithms for the same support thresholds. It is evident from the figure that CPM is several times faster than CPS-tree and orders of magnitude faster than DSTree for the entire support range. For low support thresholds ($< 0.4\%$), CPM is 10 times or more faster than SWIM. This is also true for dense datasets like *chess* even for moderate support thresholds. However, as the support increases, the performance of SWIM and CPM becomes comparable, as indicated in the figure.

Next, we compare the compression ratio among different lossless representations for varying supports on the T10I4D100K dataset. The compression ratio is calculated as the size of the condensed representation (including any additional information required for lossless extraction) to the size of all the frequent patterns with their supports. The results are shown in Figure 5.9. As in-

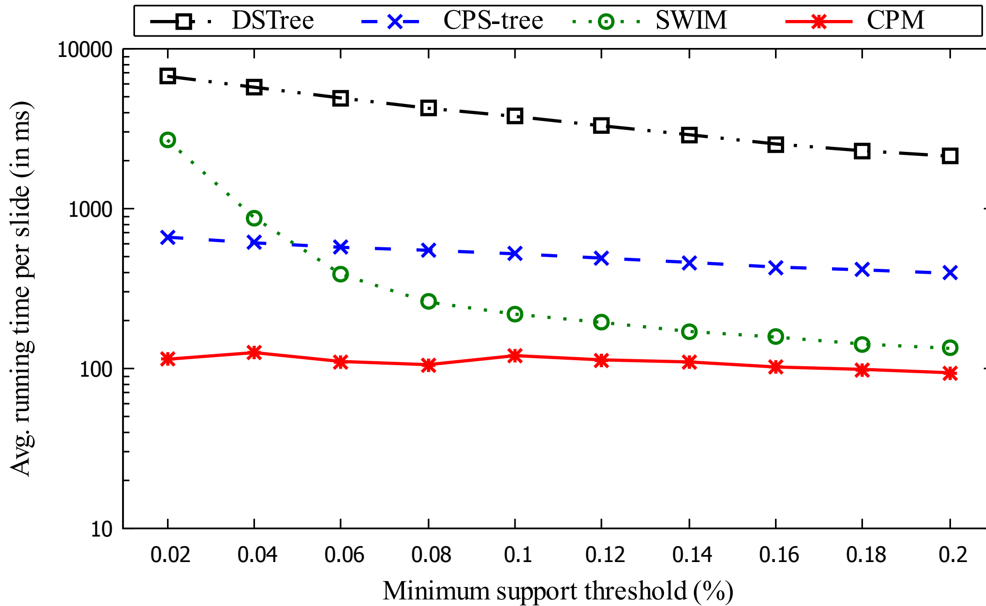


Figure 5.8: Running time comparison for lossless extraction methods.

dicated in the figure, crucial patterns offer the best lossless compaction when compared against closed [TPB00], non-derivable [CG02], disjunction-free [Kry01] itemsets and frequent generators (also known as 0-free sets [CRB06]). The last two lossless representations were coupled with positive border elements instead of negative border ones in order to obtain better size estimates [LLW07]. Also interestingly, crucial patterns have the slowest convergence to the compression ratio 1 with increasing support values.

5.10 Conclusion

In this chapter, we introduced the novel concept of "crucial patterns" for lossless frequent itemset mining. We formally proved that these patterns are a subset of the closed frequent itemsets from which all the frequent patterns can be derived with their exact supports. We also proposed the crucial pattern mining (CPM) heuristic for data streams that maintains a close-to-optimal prefix tree using two ordering schemes – *(i)* one for "once frequent" items and *(ii)* another for "still infrequent" items. Then, we also discussed how to persist the set of crucial patterns between subsequent window slides to avoid unnecessary mining calls. Our experiments indicate that CPM yields the best running time when compared against other state-of-the-art approaches. Finally, we

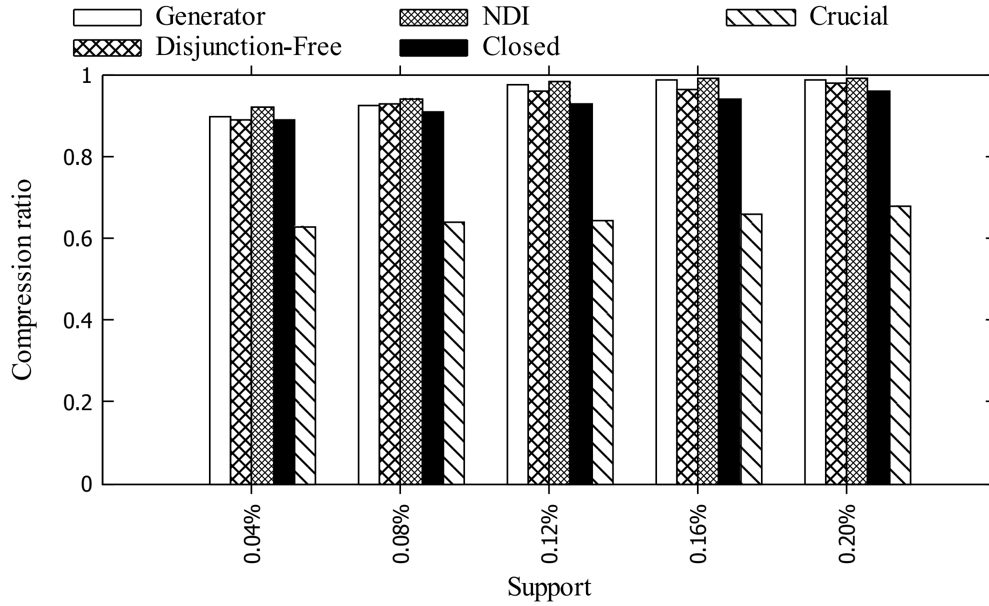


Figure 5.9: Compression ratio comparison for lossless representations.

also noted that the crucial patterns offer the best concise lossless representation of the frequent itemsets, especially for dense correlated datasets and low minimum support thresholds. In fact, in such cases, crucial patterns may be more intuitive and insightful to analysts than other lossless condensed representations.

CHAPTER 6

Scalable Construction of Online Decision Trees

In this chapter, we discuss a scalable method to efficiently employ statistical resampling techniques for building online machine learning models faster on streaming data. Since naive implementation of resampling techniques like non-parametric bootstrap does not scale on data streams due to large memory and computational overheads, we propose a robust memory-efficient bootstrap simulation heuristic (Mem-ES) that successfully expedites the learning process. We demonstrate this using online decision tree models as a case-study, since the latter are extensively used in many industrial machine learning applications for real-time classification tasks.

6.1 Introduction

Decision trees have been widely adopted by machine learning practitioners across different domains for their efficiency [BZF17], scalability [ABL16] and comprehensibility [Fre14]. It has been used in myriad applications ranging from Higgs boson classification [CH14] to predicting protein-protein interactions [KS08] in computational biology. In the streaming scenario, incremental decision tree induction algorithms have emerged as the predominant choice for a broad array of industrial classification tasks like real-time telecommunications network management and planning [BZF17], stock market prediction [KPP02], vehicle monitoring [KBL04], health indicator tracking [HTS17] and biosensor measurements [Agg06].

The Very Fast Decision Tree [DH00] (VFDT), a.k.a Hoeffding tree, is the most popular incremental decision tree induction algorithm. Figure 6.1 illustrates how a VFDT is built incrementally

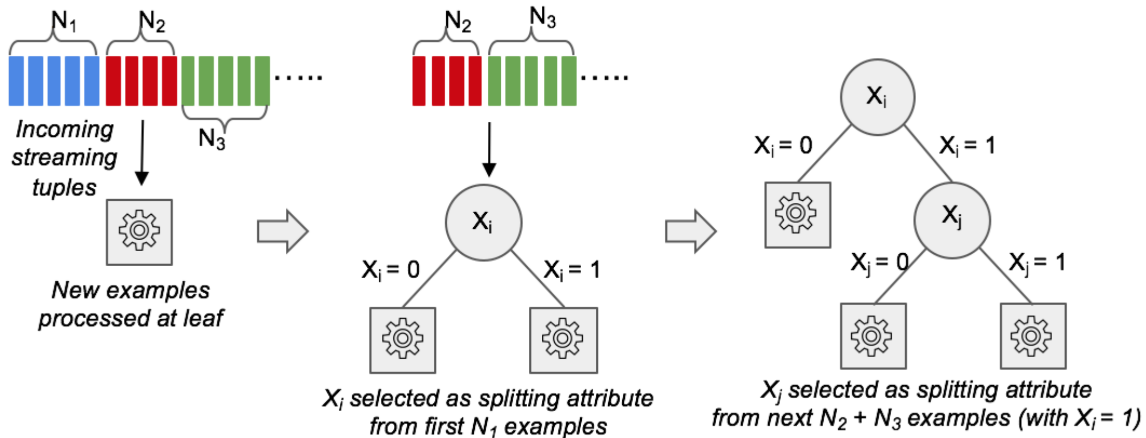
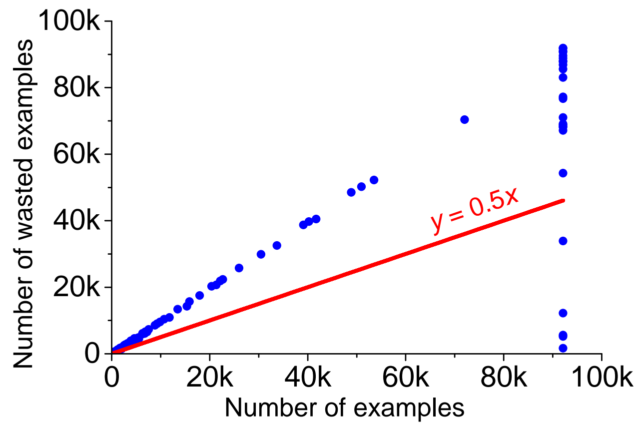


Figure 6.1: Online decision tree construction

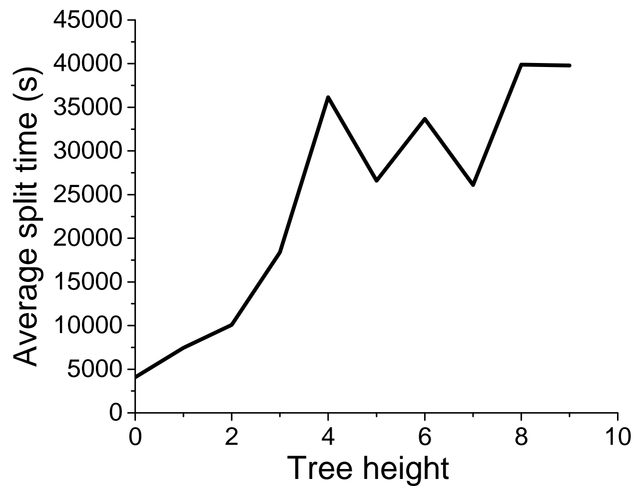
in a top-down manner as more examples are streamed in. It is typically constructed by observing "enough" training examples from an unbounded data stream and then deciding the splitting criterion from these observations with reasonably high confidence. The main idea is that for any small value δ, the splitting attribute chosen from a finite subsample would be the same as the one selected by traditional learners [Qui93, BFO84] with at least a probability of $1-\delta$. The minimal sample size is ensured via the classic *Hoeffding bound inequality*, which determines the number of points that need to be observed before deciding the split. Theoretically, VFDT is guaranteed to be asymptotically identical to a decision tree built by a conventional learner [BFO84, Qui93]. However, the Hoeffding bound is a very conservative measure, since it is independent of the underlying data distribution. This means that a larger number of observations are needed at every node to make a split with the same confidence level $1-\delta$, as compared to distribution-dependent bounds.

We demonstrate this by an example. We built a VFDT with 99% confidence (i.e. $\delta = 10^{-2}$) on 5 million streaming tuples produced from the standard MOA RandomTreeGenerator [MWS18] using the same parameters[1] mentioned in [DH00]. Figure 6.2(a) plots the total number of examples observed at a leaf before splitting vs. the number of *'redundant'* examples accumulated at the same leaf node. The latter represents the observations trailing a split, which have been collected at the leaf just to reach the eventual Hoeffding bound, but they do not change the selection of the best

[1] 100 binary attributes and two classes



(a) Redundant examples



(b) Time to split increases with depth

Figure 6.2: Observations over decision tree node split

splitting attribute in any way. That is, the same best split could have been obtained without these 'redundant' examples, although the Hoeffding bound would not have been met. As shown in the figure, even for a moderate confidence level[2], majority of the leaves have 50% to 90% wasted examples (above the red line in Figure 6.2(a)).

In fact, this problem becomes more acute as the decision tree grows. Under the same experimental settings as before, Figure 6.2(b) shows how the average time taken by a leaf to split increases across the decision tree levels. This is primarily because the probability of an incoming tuple being assigned to a leaf reduces as the number of branches increases. This can be particularly concerning with today's modern shared memory and distributed memory architectures, where decision tree construction can be massively parallelized on multicore machines [JA03a] and on multi-node clusters [ABL16]. However, the conservative nature of Hoeffding bound under these same parallel settings can force more resources (workers or threads) to idle-wait for longer durations. Naturally, these problems can be mitigated to a large extent, if an online induction tree model can learn *faster with fewer training examples* using tighter data distribution dependent bounds. Since the online induction tree model eventually grows and improves via leaf node splitting, this calls for a better and faster approach to split a leaf correctly. This is particularly needed when dealing with sensitive data streams (e.g. stock market or health sensor data), where inaccurate predictions can incur considerable damages.

In this chapter, we propose a memory-efficient bootstrap simulation strategy (Mem-ES), which consumes fewer examples in deciding when to split a leaf. While simple adoption of resampling techniques like non-parametric bootstrap is not conducive to stream processing due to large memory and computational overheads as shown later in Section 6.3.2, Mem-ES only uses constant memory space per leaf to ensure accelerated learning and superior performance. Our proposed approach Mem-ES presented in Section 6.3.3 is able to estimate the distribution dependent bounds for node splitting and can operate robustly on any dataset. Mem-ES is the first of its kind resource-efficient resampling strategy proposed in the context of incremental decision tree learners that empirically learns the distribution dependent bounds.

[2]More stringent confidence levels are used in [DH00, MWS18].

6.2 Related Work

6.2.1 Incremental Decision Tree Learning

Traditional decision tree [Qui93] model is a cornerstone of classification tasks in machine learning. However, it is not well suited for real-time learning on data streams, where low latency bounded memory models are required. The most famous online decision tree induction algorithm, VFDT [DH00] mitigates this by learning from massive data streams incrementally in a *single* pass using *constant memory per leaf*. CVFDT [HSD01] extends VFDT to incorporate gradual changes in the underlying data distribution for concept-drifting data streams. [Fan04] and [WFY03] combined VFDT with other ensemble methods to improve the performance. Some previous studies also aimed at improving the node split latency of VFDT, which is the major bottleneck for the overall learning stage. [JA03b] deduced smaller theoretical bounds for the sample size required to make a split decision correctly. These bounds are independent of the input data distribution and have been derived from the mathematical properties of information gain [Qui93] and Gini index [BFO84]. However, unlike VFDT, these bounds are not agnostic to the split measures used. In addition, they also do not exploit the underlying input data distribution to come up with tighter bounds to hasten the split.

Recent studies like Extremely Fast Decision Tree [MWS18] (EFDT) improves the splitting process by allowing revision on the split decisions and achieves state-of-the-art performance on many datasets. Other recently proposed methods like One-Sided Minimum OSM [LWH18] utilizes local node statistics to optimize the frequency of evaluation of split decisions . These recent improvements still rely on the Hoeffding bound inequality for the actual split and are thus independent of our proposed method. Hence, our proposed algorithm Mem-ES can be plugged into these frameworks to further improve their performance, as shown in our experimental results. Many data stream mining systems also provide parallel and distributed implementations of online decision trees [JA03a], such as MOA [BHK10] and STREAMDM-C++ [BZF17], where our optimizations discussed in this chapter can also be integrated.

6.2.2 Resampling Methods

Statistical resampling techniques provide conceptually simple and powerful tools to (1) measure an estimate and (2) assess the quality of the corresponding estimation from an empirical distribution. The classical Bootstrap method [BF81,GZ90] was the first to quantify the uncertainty in an estimator using confidence intervals via repeated Monte Carlo trials, where in each trial the estimator was computed over a resample drawn from the entire observed data. These estimates were more accurate, robust and consistent than those obtained using asymptotic approximations [Hal13] on a wide domain of problems. Variants of the bootstrap algorithm like subsampling [PRW99] and m out of n bootstrap [BGZ12] are computationally much less demanding than the classic bootstrap algorithm, since the estimators are repeatedly computed on significantly smaller resamples. Unlike the traditional bootstrap method, these variants are less generic in nature, and often require rescaling and analytic asymptotic approximation of the output. This is addressed by the most recent Bag of Little Bootstraps (BLB) [KTS12] method, which combines the weighted results of bootstrapping multiple small subsets of a larger dataset in a robust manner. Our proposed algorithm Mem-ES is a variant of BLB that performs the Monte Carlo approximation over incoming streaming tuples under bounded memory constraints using a computationally efficient strategy.

6.3 Methodology

6.3.1 Preliminary

First we provide a detailed overview of the general online decision tree induction algorithm. Suppose $\mathscr{S}$ denotes a sequence of examples s, where $s = \langle \mathbf{x}, y \rangle$. Let $\mathbf{X}$ denote the set of attributes an example s has. The goal is to predict the class label y given the attribute values $\mathbf{x}$ of an incoming example s. VFDT, in particular, adopts the *Hoeffding bound* to decide when a node in the decision tree can be split with a confidence level of $1 - \delta$. Formally, the Hoeffding bound inequality states that for n independent observations of a real-valued random variable r with range R and observed

mean $\bar{r}$, the true mean of $r \geq \bar{r} - \varepsilon$ with probability $1-\delta$, where

$$\varepsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}} \tag{1}$$

Thus intuitively, if $G(X_i)$ denotes the information gain[3] [Qui93] for an attribute $\mathbf{X}_i$ computed from n training examples, and $G(\mathbf{X}_a), G(\mathbf{X}_b)$ indicate the highest and second highest information gain among all the attributes, then $G(\mathbf{X}_a) - G(\mathbf{X}_b) > \varepsilon$ implies $\mathbf{X}_a$ can be judged as the best splitting attribute with probability $1-\delta$ without considering additional points for this particular split. In other words, by Hoeffding bound, we have $G(\mathbf{X}_a) - G(\mathbf{X}_b) > 0$ with probability $1-\delta$ over entire data.

Algorithm 6.1 demonstrates the process of constructing an online decision tree. The decision tree is first initialized with a single node (line: 2). Then for each incoming example $s \in \mathscr{S}$, it assigns the example into a leaf node l using the existing decision tree (line: 5). The leaves do not store the actual examples. Instead they maintain statistics like the *number of examples* seen for each attribute $\mathbf{X}_i \in \mathbf{X}$, with value j and class label $y = k$, which are denoted as n_{ijk} and are used for computing $G(\mathbf{X}_i)$. These statistics are updated along with new incoming examples (line: 6). A leaf node in VFDT is split (line: 8) when its best attribute satisfies the Hoeffding inequality (as shown in the function `Attempt to Split`). The split attempts on different leaves can be executed in parallel and asynchronously [JA03a]. However, more importantly, as shown in equation 1, the Hoeffding bound does not take into account the underlying distribution of the examples seen and hence most leaf nodes consume considerably more examples than necessary in order to make a successful split with reasonable confidence.

6.3.2 Non-parametric Bootstrap Driven Split

According to the bootstrap principle, given any unknown distribution F and a sample S drawn i.i.d from F, the quality of an estimation θ of some unknown population value, associated with F, can be assessed by drawing *with replacement* sufficient resamples from S of size $|S|$. The confidence

[3] Other heuristic measures such as Gini index [BFO84] can also be used.

Algorithm 6.1: Online Decision Tree Induction ($\mathcal{S}$, $\mathbf{X}$, δ)

Input: $\mathcal{S}$: A sequence of examples; $\mathbf{X}$: The set of attributes; δ: One minus the desired probability

Output: $\mathcal{T}$: Online decision tree learned from $\mathcal{S}$

```
1  begin
2      Initialize T with a single root node;
3      Initialize the statistics for tree growth;
4      foreach s ∈ S do
5          Sort s into leaf node l using T;
6          Update the statistics at l for tree growth;
7          if Examples at l are not from the same class then
8              async Attempt to Split(l, X, δ);
9      return T;
10 end
```

Function `Attempt to Split` (l, $\boldsymbol{X}$, δ) in VFDT

```
1  begin
2      X_a, X_b ← The two attributes with highest G(X_i) values, where X_i ∈ X;
3      Compute ε using Equation 6.1;
4      G(X_∅) ← Gain corresponding to no split;
5      if G(X_a) − G(X_b) > ε and G(X_a) ≠ G(X_∅) then
6          Replace l with an internal node;
7      foreach branch of split on X_a do
8          Add a new child l_m to l and set X_m ← X − X_a;
9  end
```

Figure 6.3: Algorithm for constructing online decision tree

interval of θ computed via a form of Monte Carlo approximation from the resampling (or empirical) distribution F^* holds well in practice, since by the law of large numbers the relative variation among F and F^* are similar [GZ90]. Interestingly, bootstrap does not necessarily improve upon

the actual value of the estimation θ. Instead it provides a good assessment of the quality of the estimate via standard error or confidence intervals.

We next summarize a bootstrap driven split attempt method: For a given leaf node l, let $\mathscr{A}$ denote all examples observed in l. $\mathbf{X}_a, \mathbf{X}_b$ are the best and second best splitting attributes based on $G(\mathbf{X}_a)$ and $G(\mathbf{X}_b)$ respectively as computed from $\mathscr{A}$. Let $\theta = G(\mathbf{X}_a) - G(\mathbf{X}_b)$. Now, we can perform the classical bootstrap for T Monte Carlo iterations, where in each iteration i we draw a sample S_i with replacement from $\mathscr{A}$ and compute from S_i, $\theta_i^* = G(\mathbf{X}_a) - G(\mathbf{X}_b)$ and $\Delta_i^* = \theta_i^* - \theta$. Given a series of sorted Δ_i^*, for $1 \leq i \leq T$, we can select Δ_L^* and Δ_U^* as the lower and upper percentiles for calculating the bootstrap confidence interval corresponding to $1-\delta$ confidence. For example, for a confidence level of 95%, Δ_L^* and Δ_U^* would be the 2.5th and 97.5th percentile respectively. Thus, the bootstrap confidence interval for the estimate θ is given by $[\theta - \Delta_U^*, \theta - \Delta_L^*]$. Now, considering only the lower bound of the confidence interval, if equation 2 is satisfied, then $\theta = G(\mathbf{X}_a) - G(\mathbf{X}_b) > 0$ holds with probability $1-\delta$. Therefore, we can split the node on $\mathbf{X}_a$.

$$\theta > \Delta_U^* \tag{2}$$

It is important to note that for calculating Δ_U^*, we only need to store a few top Δ_i^* values. For example, for calculating the 97.5th percentile of 100 Δ_i^* values, we only need to store and maintain the four top Δ_i^* values. Note this bootstrap driven method decides to split based on the empirical distribution. However, this accelerated split comes at the cost of larger memory and computational overheads, which are discussed next.

6.3.2.1 Space and Time Complexity Analysis

Unlike the VFDT, a bootstrap driven split attempt would need to store the examples in the leaf in order to be able to resample from it. Considering nominal data with c classes and d attributes, where each attribute can have at most v values, VFDT can maintain the n_{ijk} counts at each leaf in $O(dvc)$ memory. Thus for l leaves, total space complexity for VFDT is given by $O(ldvc)$. On the contrary, in order to store examples at the leaves, non-parametric bootstrap would require a space of $O(ldn)$, where n is the highest number of points accumulated at any leaf. Since $n >> vc$, the

bootstrap driven split method has a space complexity of $O(ldn)$.

Information gain computation takes $O(c)$ time and each of the d attributes needs to calculate at most v information gains. Thus the time to find the best and second best split attribute at any node in VFDT is $O(dvc)$. However, in bootstrap, T iterations are conducted, where in each iteration resamples are drawn in $O(n)$ time and Δ_U^* is computed in $O(vc)$ time. Again, since $n >> vc$, the overall time complexity for bootstrap based split attempt at any leaf is $O(Tn + dvc)$, or simply $O(Tn)$, assuming the split attempts at each node are conducted in parallel and asynchronously (see Algorithm 6.1). Thus, in terms of both space and time complexity, the dependency on n is the major bottleneck, which makes the bootstrap based approach practically hard to scale.

6.3.3 Memory-Efficient Bootstrap Simulation (Mem-ES)

To mitigate the above problems, we propose the Mem-ES method based on the principle of Bag of Little Bootstraps (BLB) [KTS12]. BLB selects few small samples (possibly disjoint) and then *artificially* generates large bootstrap samples from them, which are consequently used to compute the quality of the estimators. Under BLB, we simulate selecting large bootstrap samples of size n from a considerably smaller sample of size w ($w << n$) by drawing n trials from a multinomial distribution with parameters $n, \mathbf{1}_w/w$, where $\mathbf{1}_w/w$ denotes the 1-by-w vector of multinomial probabilities, each initialized with value $\frac{1}{w}$. The final estimator quality is assessed by averaging across all the results. However, this BLB template was proposed in the context of bounded static data. Mem-ES extends BLB and adapts it for unbounded data streams. For example, Algorithm 6.2 uses Mem-ES to specifically check for a potential split with high statistical confidence in a memory-efficient manner by only relying on the most recent batch of w points at any leaf to simulate the bootstrap process. Also note that in Algorithm 6.2, unlike the non-parametric bootstrap, the size of $\mathscr{A}$ is bounded to an user-defined value w (lines: 3 and 18). We can easily integrate Mem-ES into VFDT or other variants by simply replacing line 8 in Algorithm 6.1. We next discuss some key aspects of Mem-ES.

6.3.3.1 Discussion

Each leaf node in Mem-ES maintains at most w last seen examples. Thus, the overall memory complexity for Mem-ES is $O(ldvc + ldw)$. Typically, w values (as used in our experiments) are comparable to vc.

Similarly, we can draw a sample of size n from the multinomial distribution of w distinct objects in $O(w)$ time, thereby the overall time complexity for a split attempt at any leaf is reduced to $O(Tw + Tvc + dvc)$. Typically Tw is comparable to dvc. However, online decision tree construction is mostly I/O or network bound [DH00], since the streaming rate is primarily throttled by the I/O rate or network bandwidth. As such, the in-memory computation time of a split attempt, specially under parallel settings, is largely overshadowed by the time required to read the corresponding tuples. Lastly, it is worth mentioning that in practice, instead of two attributes (line: 7 in Algorithm 6.2), we maintain top 4 or 5 promising attributes since X_a, X_b may change as more examples arrive.

6.4 Evaluation

6.4.1 Experimental Setup

6.4.1.1 Baseline Methods

We evaluate the effectiveness of Mem-ES against two category of baselines:

• First, we integrate Mem-ES into VFDT (denoted as *VFDT+ME*) and compare its performance against standard VFDT and [JA03b] (denoted by '*VFDT+IG*'). Simple VFDT uses Hoeffding bound, whereas *VFDT+IG* uses sample size estimates deduced theoretically from the property of information gain.

• Second, we incorporate Mem-ES into a VFDT variant like EFDT and benchmark its performance. *EFDT+ME* denotes integration of Mem-ES into EFDT.

6.4.1.2 Datasets

We evaluate the performance of Mem-ES against the above baselines on two large real world classification datasets used in [MWS18] and one synthetic dataset.

• Gas Sensors dataset(Gas) from UCI repository consists of 900K+ records. The data have 15 continuous attributes and 3 classes in total.

• Human Activity Recognition (WISDM) dataset [KWM11] consists of 1M+ records. The data have 5 continuous attributes and 6 classes in total.

• In addition, we also created a synthetic dataset(SYN) spanning across 10M+ records with 100 binary attributes and 2 classes (generated from the standard MOA RandomTreeGenerator) to test Mem-ES for scalability.

VFDT and its variants like EFDT theoretically converge towards the decision tree built by a traditional learner when the incoming streaming examples are i.i.d. Hence, the data sets are shuffled for the experiments as prescribed in [MWS18]. Here we report the metrics averaged over 5 such shuffles.

6.4.1.3 Environment

Our experiments are conducted on a standard commodity machine with 4 cores and 32GB memory running Ubuntu 14.04 LTS. We used Java implementation of Mem-ES and other baselines with 8 threads. The main program was tasked with reading the incoming data tuple and assigning it to a leaf in the decision tree. Other workers concurrently and asynchronously attempt splits at different leaves[4]. All the experiments were performed using a confidence level of 98%, $T = 150$, $w = 40$. We used the standard value of $n_{min} = 200$ (parameter which controls how frequently a split attempt is made at a leaf) [DH00].

6.4.2 Results and Discussion

In this section, we will try to answer the following questions: (1) Does Mem-ES help to learn a decision tree *correctly* with *fewer* examples? (2) Is memory a major bottleneck for Mem-ES or can it scale at an affordable cost?

We examine the first question by plotting the number of instances seen vs. the error rate for all three data sets. Figure 6.5 shows the corresponding three plots. Figure 6.5(a) shows that *VFDT+ME* learns better from considerably fewer examples, as compared to VFDT and *VFDT+IG*, yielding around 8 percentage points lower error rate than VFDT and *VFDT+IG* in the first 200K examples. In addition, Figure 6.5(a) further shows that even *EFDT+ME* learns faster with fewer number of examples than EFDT, although the gap reduces after 150K examples. Interestingly, simple *VFDT+ME* has outperformed the optimized EFDT for the first 400K examples on the Gas data. Figure 6.5(b) on WISDM data further reiterates that Mem-ES learns and converges faster with fewer examples than other baseline methods. In fact, Figure 6.5(b) exemplifies the utility of Mem-ES as a 'plug-in', since it can be incorporated into VFDT to outperform VFDT and *VFDT+IG* and can similarly be integrated into EFDT to improve its performance. For some datasets, VFDT variants can outperform EFDT, as shown in Figure 6.5(c), where *VFDT+IG* produces lower error rate than simple EFDT. Nevertheless, Mem-ES still yields a marginal improvement (2 percentage points lower error in first 400K examples) over *VFDT+IG*.

It is worth highlighting here that all online models eventually converge as examples stream in [MWS18]. But online models that learn and converge faster with fewer examples are naturally more preferable. This is more important for sensitive data streams, where inaccurate classifications can incur significant penalties. Thus, the nomenclature of 'fast' in systems like VFDT or EFDT refers to how many fewer examples are required in the learning stage. In other words, at any time instant during the learning stage, the accuracy of the online learner is determined mainly from its ability to learn from fewer examples. And in this context of learning from fewer examples, Figure 6.5 presents Mem-ES as a very effective and robust strategy that works well across different data

[4] Leaves are assigned to threads in a round-robin fashion so that the load is evenly distributed.

sets, since *EFDT+ME* (Figures 6.5(a), 6.5(b)) or *VFDT+ME* (Figure 6.5(c)) produces the best results.

Next, we investigate the memory cost of Mem-ES by examining how the number of nodes and the memory consumption varies as the examples stream in. Figures 6.6 and 6.7 show this interplay with double Y-axis graphs, where the line graph indicates the total memory consumed and the vertical bar represents the total number of nodes created in the corresponding decision tree. Figure 6.6 presents the comparison between VFDT, *VFDT+IG* and *VFDT+ME*. As shown in Figure 6.6, the three methods incrementally builds the tree, but *VFDT+ME* constructs at the fastest rate i.e. it builds a more deeper decision tree model as compared to other baselines after processing the same number of examples. Also recall that Mem-ES needs to store at most w examples at each leaf. As a result, *VFDT+ME* ends up requiring more memory than VFDT or *VFDT+IG*, since it grows better trees with more nodes at an expedited rate as well as retains some data points at the leaves. But nevertheless, even for the synthetic data of over 10M records (Figure 6.6(c)), *VFDT+ME* requires only around 100MB memory, which is an order of magnitude less than the default allocated heap size in standard Java virtual machines as used in our implementation. Furthermore, *VFDT+ME* requires less than 10MB and 3MB memory for Gas and WISDM datasets respectively. Figure 6.7 shows the corresponding plots for EFDT and *EFDT+ME*. EFDT re-examines the splits of internal nodes in the decision tree and aggressively prunes the sub-trees if a split better than the original one is found. Consequently, as shown in Figure 6.7, the number of nodes and memory consumption of EFDT and *EFDT+ME* can decrease as well. This makes Mem-ES more suitable for integration into strategies like EFDT where memory can be aggressively freed. *EFDT+ME* consumes around 700KB, 300KB and 45MB memory for Gas, WISDM and SYN datasets respectively.

6.5 Conclusion

In this chapter, we presented Mem-ES that efficiently performs resampling techniques to accelerate the node splits for online decision tree learning. The success of Mem-ES is particularly exciting, since the idea of applying resampling techniques like non-parametric bootstrap on data streams had always been considered very difficult in the past due to its cumbersome nature and

high time and space complexity. But, this first of its kind realization and experimental validation of approximate bootstrapping can invite further research investigations in other stream mining algorithms used for frequent pattern mining [DZ16], episode mining [ASW19, ALW18], complex pattern detection and ranking [GWZ16], where bootstrapping can be useful.

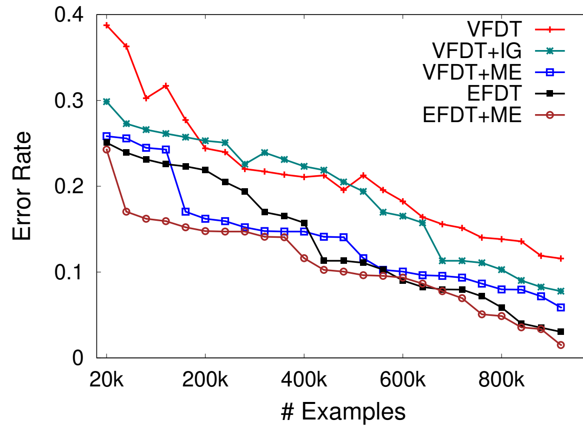
Algorithm 6.2: Attempt to split with Mem-ES

Input: l: Leaf to be split; $\mathbf{X}$: The set of attributes; δ: One minus the desired probability; s: current example; $\mathcal{A}$: Queue of fixed size w; n: User-specified parameter; T: Number of Monte Carlo iterations; r, sum: Variables initialized with 0

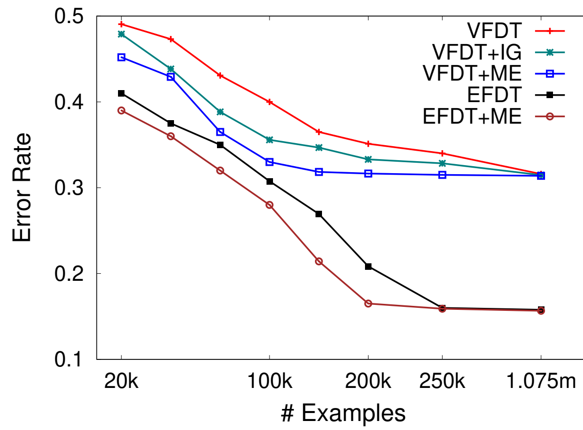
```
1  begin
2      Enqueue s to A;
3      if |A| is not full then
4          return;
5      Q ← Min-Heap of fixed size for Δ*_U computation;
6      r ← r + 1; // count of disjoint sets
7      X_a, X_b ← The two attributes with highest G(X_i) values computed from n_ijk
       counts at l;
8      θ ← G(X_a) − G(X_b);
9      for t ∈ [1, T] do
10         Draw sample S_t from Multinomial(n, 1_w/w);
11         θ*_t ← G(X_a) − G(X_b) from S_t;
12         Δ*_t ← θ*_t − θ;
13         if Δ*_t > min(Q) or Q is not full then
14             Add Δ*_t to Q;
15     Δ*_U ← min(Q);
16     sum ← sum + Δ*_U;
17     mean(Δ*_U) ← sum / r;
18     Empty A and Q;
19     if θ > mean(Δ*_U) then
20         Replace l with an internal node;
21         foreach branch of split on X_a do
22             Add a new child l_m to l and set X_m ← X − X_a;
23 end
```

Line 5: $Q \leftarrow$ Min-Heap of fixed size for Δ^*_U computation;
Line 7: $\mathbf{X}_a, \mathbf{X}_b \leftarrow$ The two attributes with highest $G(\mathbf{X}_i)$ values computed from n_{ijk} counts at l;
Line 8: $\theta \leftarrow G(\mathbf{X}_a) - G(\mathbf{X}_b)$;
Line 10: Draw sample S_t from Multinomial$(n, \mathbf{1}_w/w)$;
Line 11: $\theta^*_t \leftarrow G(\mathbf{X}_a) - G(\mathbf{X}_b)$ from S_t;
Line 12: $\Delta^*_t \leftarrow \theta^*_t - \theta$;
Line 13: **if** $\Delta^*_t > min(Q)$ *or* Q *is* not *full* **then**
Line 17: $\overline{\Delta^*_U} \leftarrow \frac{sum}{r}$;
Line 19: **if** $\theta > \overline{\Delta^*_U}$ **then**
Line 22: Add a new child l_m to l and set $\mathbf{X}_m \leftarrow \mathbf{X} - \mathbf{X}_a$;

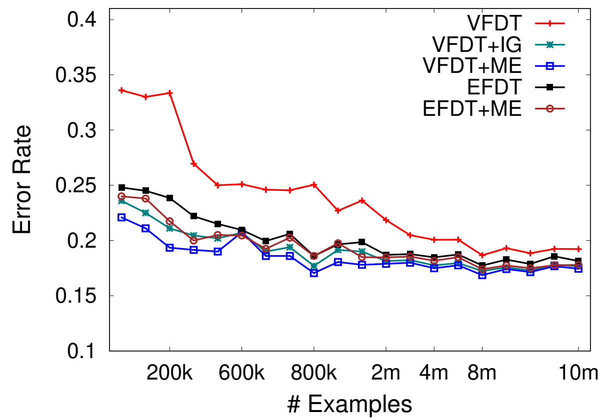
Figure 6.4: Algorithm for Mem-ES



(a) Gas

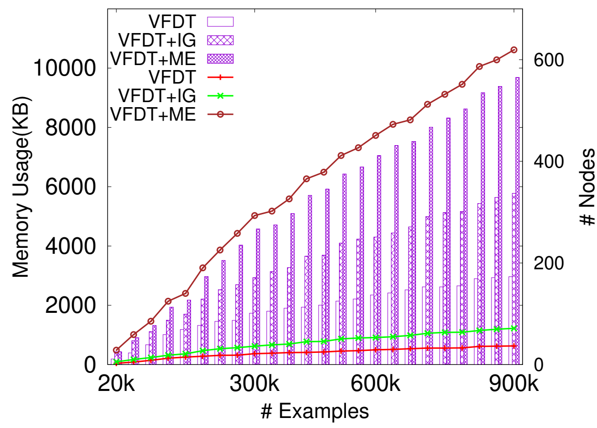


(b) WISDM

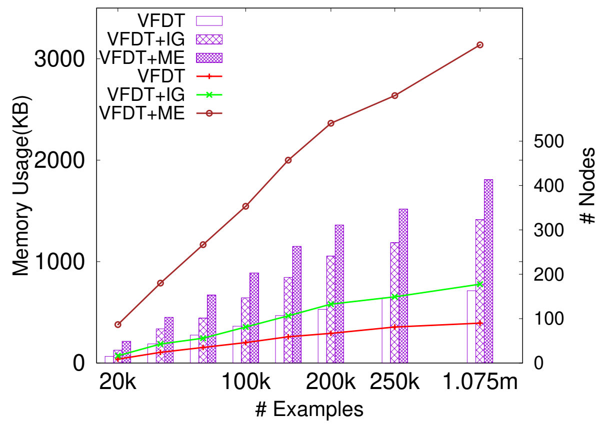


(c) SYN

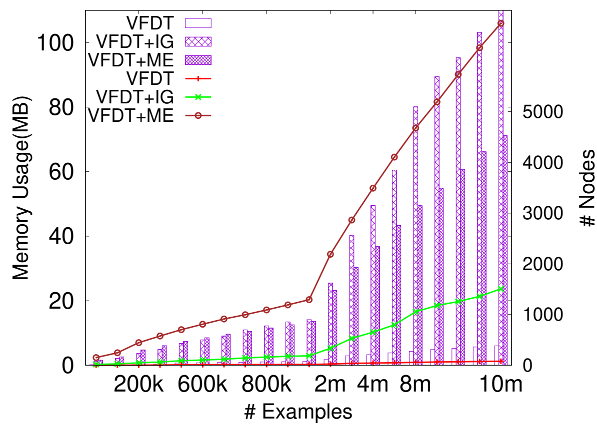
Figure 6.5: Comparison with state-of-the-art methods: Error Rate



(a) Gas

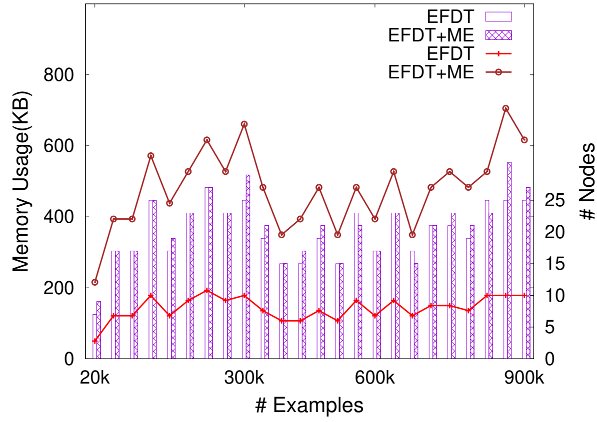


(b) WISDM

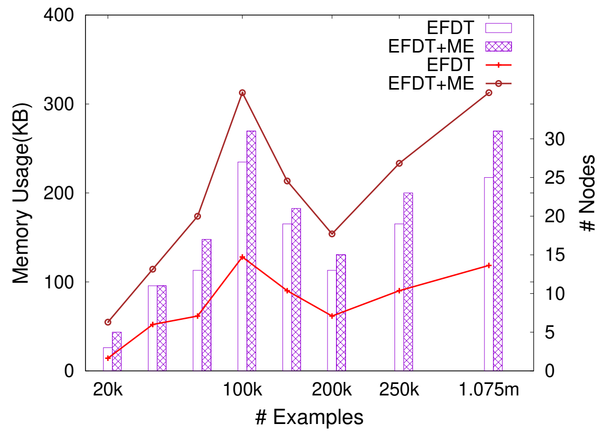


(c) SYN

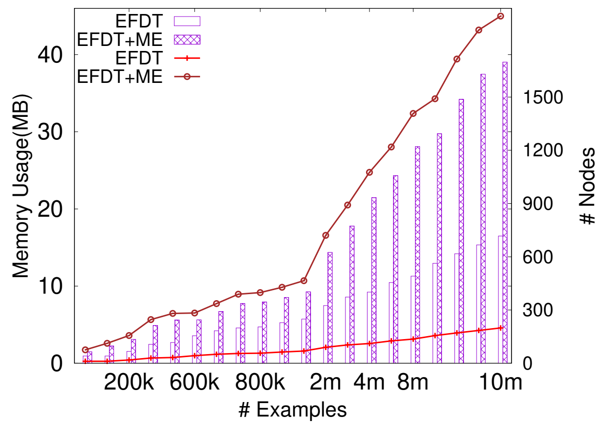
Figure 6.6: Tree growth and memory consumption: Mem-ES vs. VFDT and *VFDT+IG*



(a) Gas



(b) WISDM



(c) SYN

Figure 6.7: Tree growth and memory consumption: Mem-ES vs. EFDT

CHAPTER 7

Conclusion and Future Work

By embracing the Horn-clause logic of Prolog but not its operational constructs such as the cut, Datalog researchers, 30 years ago, embarked in a significant expedition toward declarative languages in which logic alone rather than "Logic+Control" [Kow79] can be used to specify algorithms. Significant progress toward this ambitious goal was made in the 90s with techniques such as semi-naive fixpoint and magic sets that support recursive Datalog programs by bottom-up computation and implementation techniques from relational database systems. However, declarative semantics for algorithms that require aggregates in recursion largely remained an unsolved problem for this first generation of deductive database systems. Moreover, Datalog scalability via parallelization was only discussed in papers, until recently when the availability of new parallel platforms and an explosion of interest in BigData renewed interest in Datalog and its parallel implementations on multicore and distributed systems, as discussed in chapter 2.

In this thesis, we have made major progress toward a unified environment for developing various types of big data applications in declarative languages (Datalog, SQL, ...), enabling end-to-end query optimization for complex algorithms. This was made possible with $\mathcal{P}$*reM*, which allows aggregates in recursion with rigorous semantics guarantee. We demonstrated that $\mathcal{P}$*reM* offers *greater expressivity, usability, portability, better performance and scalability through parallelization* for several graph queries. In fact, there are unique advantages that $\mathcal{P}$*reM* offers on comparison with other BigData application development frameworks, like, (1) a Stale Synchronous Parallel computing model and $\mathcal{P}$*reM*-optimized queries dovetail and combine to further expedite many

graph and potential ML applications, and (2) $\mathscr{P}reM$ also offers optimization scope for continuous queries on data streams.

An immediate future work is to develop ML applications and queries which can benefit from relaxed synchronization. Another interesting future direction is to develop a rich library of data mining and graph applications, where $\mathscr{P}reM$ can be applied with formal proof of validity, and also build language extensions and system support for user-defined aggregates that can aid developability for programmers. Another very exciting direction for future research is to examine support for non-deterministic constructs within recursion. More efforts are needed to understand if limitations of $\mathscr{P}reM$, discussed in Chapter 3 can be solved with alternative problem formulations using these non-deterministic constructs.

Furthermore, we believe that the use of aggregates in recursive rules made possible by $\mathscr{P}reM$ can lead to beneficial extensions in several application areas, e.g., data mining algorithms, and in related logic-based systems, including, e.g., those that use tabled logic programming [SW12] and Answer Sets [EGL16]. Therefore we see many interesting new topics deserving further investigation, suggesting that logic and databases remains a vibrant research area although many years have passed since it was first introduced [MSZ14].

References

[ABL16] F. Abuzaid, J. K. Bradley, F. T. Liang, A. Feng, L. Yang, M. Zaharia, and A. Talwalkar. "Yggdrasil: An Optimized System for Training Deep Decision Trees at Scale." In *NIPS*, pp. 3817–3825. 2016.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL Continuous Query Language: Semantic Foundations and Query Execution." *The VLDB Journal*, **15**(2):121–142, June 2006.

[ACG15] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. "Design and Implementation of the LogicBlox System." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1371–1382, 2015.

[AFR11] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. "EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning." In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pp. 635–644, New York, NY, USA, 2011. ACM.

[Agg06] Charu C. Aggarwal. *Data Streams: Models and Algorithms (Advances in Database Systems)*. Springer-Verlag, 2006.

[AGK17] Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. "Parallel-Correctness and Transferability for Conjunctive Queries." *J. ACM*, **64**(5):36:1–36:38, 2017.

[AKG10] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. "Reining in the Outliers in Map-reduce Clusters Using Mantri." In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pp. 265–278, 2010.

[AKN15] Tom J. Ameloot, Bas Ketsman, Frank Neven, and Daniel Zinn. "Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture." *ACM Trans. Database Syst.*, **40**(4):21:1–21:45, 2015.

[ALW18] X. Ao, P. Luo, J. Wang, F. Zhuang, and Q. He. "Mining Precise-Positioning Episode Rules from Event Sequences." *IEEE TKDE*, 2018.

[Ame14] Tom J. Ameloot. "Declarative Networking: Recent Theoretical Work on Coordination, Correctness, and Declarative Semantics." *SIGMOD Rec.*, **43**(2):5–16, 2014.

[ANN18] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. "In-Database Learning with Sparse Tensors." In *SIGMOD/PODS'18*, 2018.

[ANV13] Tom J. Ameloot, Frank Neven, and Jan Van Den Bussche. "Relational Transducers for Declarative Networking." *J. ACM*, **60**(2):15:1–15:38, 2013.

[AS94] R. Agrawal and R. Srikant. "Fast algorithms for mining association rules in large databases." In *VLDB*, 1994.

[ASW19] X. Ao, H. Shi, J. Wang, L. Zuo, H. Li, and Q. He. "Large-scale Frequent Episode Mining from Complex Event Sequences with Hierarchies." *ACM TIST*, 2019.

[BBC10] Davide Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, Yi Huang, Volker Tresp, Achim Rettinger, and Hendrik Wermser. "Deductive and Inductive Stream Reasoning for Semantic Social Media Analytics." *IEEE Intelligent Systems*, **25**(6):32–41, November 2010.

[BBC12] Vinayak Borkar, Yingyi Bu, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. "Declarative Systems for Large-Scale Machine Learning." In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2012.

[BBD02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. "Models and Issues in Data Stream Systems." In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pp. 1–16, New York, NY, USA, 2002. ACM.

[BC94] H. A. Blair and P. Cholak. "The Complexity of the Class of Locally Stratified Prolog Programs." *Fundamenta Informaticae*, **21**(4):333–344, 1994.

[BDE15] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. "LARS: A Logic-Based Framework for Analyzing Reasoning over Streams." In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pp. 1431–1438, 2015.

[BF81] Peter J. Bickel and David A. Freedman. "Some Asymptotic Theory for the Bootstrap." *The Annals of Statistics*, **9**(6):1196–1217, 11 1981.

[BFO84] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.

[BGJ08] Andre Bolles, Marco Grawunder, and Jonas Jacobi. "Streaming SPARQL Extending SPARQL to Process Data Streams." In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pp. 448–462, Berlin, Heidelberg, 2008. Springer-Verlag.

[BGZ12] Peter J Bickel, Friedrich Götze, and Willem R van Zwet. "Resampling fewer than n observations: gains, losses, and remedies for losses." In *Selected works of Willem van Zwet*, pp. 267–297. Springer, 2012.

[BHK10] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. "MOA: Massive Online Analysis." *Journal of Machine Learning Research*, **11**:1601–1604, 2010.

[BIY06] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. "The influence of operating systems on the performance of collective operations at extreme scale." In *2006 IEEE International Conference on Cluster Computing*, pp. 1–12, 2006.

[BZF17] A. Bifet, J. Zhang, W. Fan, C. He, J. Zhang, J. Qian, G. Holmes, and B. Pfahringer. "Extremely Fast Decision Tree Mining for Evolving Data Streams." In *SIGKDD*, pp. 1733–1742, 2017.

[CAG03] Flavio Cruz, Michael P. Ashley-Rollman, Seth C. Goldstein, Ricardo Rocha, and Frank Pfenning. "Bottom-Up Logic Programming for Multicores." 2003.

[CCH14] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. "Exploiting Bounded Staleness to Speed Up Big Data Analytics." In *USENIX ATC*, pp. 37–48, 2014.

[CDI18] Tyson Condie, Ariyam Das, Matteo Interlandi, Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. "Scaling-up reasoning and advanced analytics on BigData." *TPLP*, **18**(5-6):806–845, 2018.

[CG02] T. Calders and B. Goethals. "Mining all non-derivable frequent itemsets." In *PKDD*, pp. 74–85, 2002.

[CH80] Ashok K. Chandra and David Harel. "Structure and Complexity of Relational Queries." In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, SFCS '80, pp. 333–347, 1980.

[CH08] G. Cormode and M. Hadjieleftheriou. "Finding frequent items in data streams." In *VLDB*, pp. 1530–1541, Newport Beach, CA, USA, 2008.

[CH14] Tianqi Chen and Tong He. "Higgs Boson Discovery with Boosted Trees." In *HEPML@NIPS*, pp. 69–80, 2014.

[Chi] "Chicago Taxi Data Released." `https://cloud.google.com/bigquery/public-data/chicago-taxi`. Accessed: 2018-05-22.

[CHK13] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. "Solving the Straggler Problem with Bounded Staleness." In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pp. 22–22, 2013.

[CKN08] J. Cheng, Y. Ke, and W. Ng. "Maintaining frequent closed itemsets over a sliding window." *Journal of Intelligent Information Systems*, **31**(3):191–215, 2008.

[CRB06] Toon Calders, Christophe Rigotti, and Jean-François Boulicaut. "A Survey on Condensed Representations for Frequent Sets." In *Constraint-Based Mining and Inductive Databases*, pp. 64–80, 2006.

[CWY04] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. "Moment: Maintaining closed frequent itemsets over a stream sliding windows." In *ICDM*, pp. 59–66, 2004.

[DGZ18] Ariyam Das, Sahil M. Gandhi, and Carlo Zaniolo. "ASTRO: A Datalog System for Advanced Stream Reasoning." In *CIKM'18*, pp. 1863–1866, 2018.

[DH00] Pedro M. Domingos and Geoff Hulten. "Mining high-speed data streams." In *SIGKDD*, pp. 71–80, 2000.

[DZ16] Ariyam Das and Carlo Zaniolo. "Fast Lossless Frequent Itemset Mining in Data Streams using Crucial Patterns." In *SDM*, pp. 576–584, 2016.

[DZ19] Ariyam Das and Carlo Zaniolo. "A Case for Stale Synchronous Distributed Model for Declarative Recursive Computation." *TPLP*, 2019.

[EGL16] Esra Erdem, Michael Gelfond, and Nicola Leone. "Applications of Answer Set Programming." *AI Magazine*, **37**(3):53–68, 2016.

[ES13] D. Eppstein and J. A Simons. "Confluent Hasse Diagrams." *Journal of Graph Algorithms and Applications*, **17**(7):689–710, 2013.

[Fan04] Wei Fan. "Systematic Data Selection to Mine Concept-drifting Data Streams." In *SIGKDD*, 2004.

[FKR12] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. "Towards a Unified Architecture for in-RDBMS Analytics." In *SIGMOD'12*, pp. 325–336, 2012.

[FPL11] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. "Semantics and Complexity of Recursive Aggregates in Answer Set Programming." *Artif. Intell.*, pp. 278–298, 2011.

[Fre14] Alex A. Freitas. "Comprehensible Classification Models: A Position Paper." *SIGKDD Explor. Newsl.*, **15**(1):1–10, 2014.

[Gel86] Allen Van Gelder. "Negation as Failure Using Tight Derivations for General Logic Programs." In *Proceedings of the 1986 Symposium on Logic Programming, Salt Lake City, Utah, USA, September 22-25, 1986*, pp. 127–138, 1986.

[GL88] Michael Gelfond and Vladimir Lifschitz. “The Stable Model Semantics for Logic Programming.” In *Proceedings of International Logic Programming Conference and Symposium*, pp. 1070–1080, 1988.

[GLM14] Arnaud Giacometti, Dominique H. Li, Patrick Marcel, and Arnaud Soulet. “20 Years of Pattern Mining: A Bibliometric Survey.” *SIGKDD Explor. Newsl.*, **15**(1):41–50, 2014.

[Gno10] Alessandro Gnoli. *C - SPARQL: A Continuous Query Language for Resource Description Framework Data Streams*. LAP Lambert Academic Publishing, Germany, 2010.

[GST92] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. “Parallel Bottom-up Processing of Datalog Queries.” *J. Log. Program.*, **14**(1-2):101–126, 1992.

[GSZ95] Sergio Greco, Domenico Saccà, and Carlo Zaniolo. “DATALOG Queries with Stratified Negation and Choice: from P to D^P.” In *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, pp. 82–96, 1995.

[GWM19] Jiaqi Gu, Yugo Watanabe, William Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. “RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark.” In *SIGMOD'19*, 2019.

[GWZ16] Jiaqi Gu, Jin Wang, and Carlo Zaniolo. “Ranking support for matched patterns over complex event streams: The CEPR system.” In *ICDE*, 2016.

[GZ90] Evarist Gine and Joel Zinn. “Bootstrapping General Empirical Measures.” *The Annals of Probability*, **18**(2):851–869, 04 1990.

[Hal13] Peter Hall. *The bootstrap and Edgeworth expansion*. Springer Science & Business Media, 2013.

[HCC13] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. “More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server.” In *NIPS*, pp. 1223–1231, 2013.

[Hir12] Martin Hirzel. “Partition and Compose: Parallel Complex Event Processing.” In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pp. 191–200, New York, NY, USA, 2012. ACM.

[HPY00] J. Han, J. Pei, and Y. Yin. “Mining frequent patterns without candidate generation.” In *SIGMOD*, 2000.

[HSD01] Geoff Hulten, Laurie Spencer, and Pedro M. Domingos. “Mining time-changing data streams.” In *SIGKDD*, pp. 97–106, 2001.

[HTS17] Mostafa Haghi, Kerstin Thurow, and Regina Stoll. "Wearable devices in medical internet of things: scientific research and commercially available devices." *Healthcare informatics research*, **23**(1):4–15, 2017.

[IM96] Tomasz Imielinski and Heikki Mannila. "A Database Perspective on Knowledge Discovery." *Commun. ACM*, **39**(11):58–64, November 1996.

[IT18] Matteo Interlandi and Letizia Tanca. "A datalog-based computational model for coordination-free, data-parallel systems." *Theory and Practice of Logic Programming*, **18**(5-6):874–927, 2018.

[JA03a] Ruoming Jin and Gagan Agrawal. "Communication and Memory Efficient Parallel Decision Tree Construction." In *SDM*, pp. 119–129, 2003.

[JA03b] Ruoming Jin and Gagan Agrawal. "Efficient Decision Tree Construction on Streaming Data." In *SIGKDD*, pp. 571–576, 2003.

[JG06] N. Jiang and L. Gruenwald. "CFI-Stream: Mining closed frequent itemsets in data streams." In *SIGKDD*, 2006.

[KBL04] H. Kargupta, R. Bhargava, K. Liu, M. Powers, P. Blair, S. Bushra, J. Dull, K. Sarkar, M. Klein, M. Vasa, and D. Handy. "VEDAS: A Mobile and Distributed Data Stream Mining System for Real-Time Vehicle Monitoring." In *SDM*, 2004.

[Kow79] Robert A. Kowalski. "Algorithm = Logic + Control." *Commun. ACM*, **22**(7):424–436, 1979.

[KPP02] H. Kargupta, B.-H. Park, S. Pittie, L. Liu, D. Kushraj, and K. Sarkar. "MobiMine: Monitoring the Stock Market from a PDA." *SIGKDD Explor. Newsl.*, **3**(2):37–46, 2002.

[KRS95] David B. Kemp, Kotagiri Ramamohanarao, and Peter J. Stuckey. "ELS Programs and the Efficient Evaluation of Non-Stratified Programs by Transformation to ELS." In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, DOOD '95, pp. 91–108, 1995.

[Kry01] M. Kryszkiewicz. "Concise representation of frequent patterns based on disjunction-free generators." In *ICDM*, pp. 305–312, 2001.

[KS04] J.-L. Koh and S.-F. Shieh. "An efficient approach for maintaining association rules based on adjusting FP-tree structures." In *DASFAA*, pp. 417–424, 2004.

[KS08] Carl Kingsford and Steven L. Salzberg. "What are decision trees?" *Nature Biotechnology*, **26**(1011), 2008.

[KSS16] Srijan Kumar, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos. "Edge weight prediction in weighted signed networks." In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pp. 221–230. IEEE, 2016.

[KTG11] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. "Disks Are Like Snowflakes: No Two Are Alike." In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pp. 14–14, 2011.

[KTS12] Ariel Kleiner, Ameet Talwalkar, Purnamrita Sarkar, and Michael I. Jordan. "The Big Data Bootstrap." In *ICML*, 2012.

[KWM11] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. "Activity Recognition Using Cell Phone Accelerometers." *SIGKDD Explor. Newsl.*, **12**(2):74–82, 2011.

[LBG12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud." *Proc. VLDB Endow.*, **5**(8):716–727, 2012.

[LC09] H. Lia and H. Chen. "Mining non-derivable frequent itemsets over data stream." *International Journal of Data & Knowledge Engineering*, pp. 481–498, 2009.

[LJA14] V. E. Lee, R. Jin, and G. Agrawal. "Frequent pattern mining in data streams." *Frequent Pattern Mining*, pp. 199–224, 2014.

[LK06] C. K. S. Leung and Q. I. Khan. "DSTree: a tree structure for the mining of frequent sets from data streams." In *ICDM*, pp. 928–932, 2006.

[LKF05] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations." In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pp. 177–187, New York, NY, USA, 2005. ACM.

[LKL05] C. K.-S. Leung, Q. I Khan, Z. Li, and T. Hoque. "CanTree: A tree structure for efficient incremental mining of frequent patterns." In *ICDM*, 2005.

[LKZ14] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A. Gibson, and Eric P. Xing. "On Model Parallelization and Scheduling Strategies for Distributed Machine Learning." In *NIPS*, pp. 2834–2842, 2014.

[LLW07] G. Liu, J. Li, and L. Wong. "A new concise representation of frequent itemsets using generators and a positive border." *Knowledge and Information Systems*, **15**(1):55–86, 2007.

[LWH18] Viktor Losing, Heiko Wersing, and Barbara Hammer. "Enhancing Very Fast Decision Trees with Local Split-Time Predictions." In *ICDM*, pp. 287–296, 2018.

[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'10*, pp. 135–146, 2010.

[MD14] Jayanta Mondal and Amol Deshpande. "Stream querying and reasoning on social data." In *Encyclopedia of Social Network Analysis and Mining*, pp. 2063–2075. Springer, 2014.

[MM02] G. S. Manku and R. Motwaniu. "Approximate frequency counts over data streams." In *VLDB*, pp. 346–357, 2002.

[MSZ13] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. "Extending the Power of Datalog Recursion." *The VLDB Journal*, **22**(4):471–493, August 2013.

[MSZ14] Jack Minker, Dietmar Seipel, and Carlo Zaniolo. "Logic and Databases: A History of Deductive Databases." In *Computational Logic*, pp. 571–627. 2014.

[MTZ08] B. Mozafari, H. Thakkar, and C. Zaniolo. "Verifying and mining frequent patterns from large windows over data streams." In *ICDE*, pp. 179–188, 2008.

[MWS18] Chaitanya Manapragada, Geoffrey I. Webb, and Mahsa Salehi. "Extremely Fast Decision Tree." In *SIGKDD*, pp. 1953–1962, 2018.

[MZZ12] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. "High-performance Complex Event Processing over XML Streams." In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 253–264, New York, NY, USA, 2012. ACM.

[Naq86] Shamim A. Naqvi. "A Logic for Negation in Database Systems." In *Workshop on Database Theory, University of Texas at Austin, TX, USA, August 13-15, 1986*, 1986.

[PRW99] Dimitris Politis, Joseph P Romano, and Michael Wolf. "Weak convergence of dependent empirical measures with application to subsampling in function spaces." *Journal of statistical planning and inference*, **79**(2):179–190, 1999.

[Prz88a] Teodor C. Przymusinski. "On the Declarative Semantics of Deductive Databases and Logic Programs." In *Foundations of Deductive Databases and Logic Programming*, pp. 193–216. 1988.

[Prz88b] Teodor C. Przymusinski. "On the Relationship Between Logic Programming and Nonmonotonic Reasoning." In *Proceedings of the 7th National Conference on Artificial Intelligence*, pp. 444–448, 1988.

[PZ96] Luigi Palopoli and Carlo Zaniolo. "Polynomial-time computable stable models." *Annals of Mathematics and Artificial Intelligence*, **17**(2):261–290, 1996.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[Rei87] R. Reiter. "Readings in Nonmonotonic Reasoning." chapter On Closed World Data Bases, pp. 300–310. 1987.

[RKG18] Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, Boris Motik, and Ian Horrocks. "Stream Reasoning in Temporal Datalog." In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 2018.

[RLK11] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. "On querying historical evolving graph sequences." *Proceedings of the VLDB Endowment*, **4**(11):726–737, 2011.

[SGW16] Xiang Su, Ekaterina Gilman, Peter Wetz, Jukka Riekki, Yifei Zuo, and Teemu Leppänen. "Stream Reasoning for the Internet of Things: Challenges and Gap Analysis." In *Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics*, WIMS '16, pp. 1:1–1:10, New York, NY, USA, 2016. ACM.

[SP07] Tran Cao Son and Enrico Pontelli. "A Constructive semantic characterization of aggregates in answer set programming." *Theory and Practice of Logic Programming*, 2007.

[SPS13] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. "Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis." *Proc. VLDB Endow.*, **6**(14):1906–1917, 2013.

[STA00] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. "Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications." *Data Mining and Knowledge Discovery*, **4**(2), 2000.

[SW12] Terrance Swift and David Scott Warren. "XSB: Extending Prolog with Tabled Logic Programming." *TPLP*, **12**(1-2):157–187, 2012.

[SYI16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. "Big Data Analytics with Datalog Queries on Spark." In *SIGMOD*, pp. 1135–1149, New York, NY, USA, 2016. ACM.

[TAJ09] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong, and Y.-K. Lee. "Sliding window-based frequent pattern mining over data streams." *Information Sciences*, **179**(22):3843–3865, 2009.

[TPB00] R. Taouil, N. Pasquier, Y. Bastide, and L. Lakhal. "Mining bases for association rules using closed sets." In *ICDE*, 2000.

[WBH15] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. "Asynchronous and Fault-tolerant Recursive Datalog Evaluation in Shared-nothing Engines." *Proc. VLDB Endow.*, **8**(12):1542–1553, 2015.

[WFY03] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. "Mining concept-drifting data streams using ensemble classifiers." In *SIGKDD*, pp. 226–235, 2003.

[Yan17] Mohan Yang. *Declarative Languages and Scalable Systems for Graph Analytics and Knowledge Discovery*. PhD thesis, UCLA, 2017.

[YCL15] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. "Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation." In *WWW*, pp. 1307–1317, 2015.

[YSZ15] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. "Parallel Bottom-Up Evaluation of Logic Programs: DeALS on Shared-Memory Multicore Machines." In *Technical Communications of ICLP*, 2015.

[YSZ17] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. "Scaling up the performance of more powerful Datalog systems on multicore machines." *VLDB J.*, **26**(2):229–248, 2017.

[Zan12] Carlo Zaniolo. "Logical Foundations of Continuous Query Languages for Data Streams." In *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, pp. 177–189, 2012.

[ZAO93] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. "Negation and Aggregates in Recursive Rules: the LDL++ Approach." In *Deductive and Object-Oriented Databases, Third International Conference, DOOD'93*, pp. 204–221, 1993.

[ZCF97] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard Thomas Snodgrass, V. S. Subrahmanian, and Roberto Zicari. *Advanced Database Systems*. 1997.

[ZH02] M. J. Zaki and C.-J. Hsiao. "CHARM: An efficient algorithm for closed itemset mining." In *SDM*, 2002.

[ZYD16] Carlo Zaniolo, Mohan Yang, Ariyam Das, and Matteo Interlandi. "The Magic of Pushing Extrema into Recursion: Simple, Powerful Datalog Programs." In *AMW*, 2016.

[ZYI17] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. "Fixpoint semantics and optimization of recursive Datalog programs with aggregates." *TPLP*, **17**(5-6):1048–1065, 2017.

[ZYI18] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. "Declarative BigData Algorithms via Aggregates and Relational Database Dependencies." In *AMW*, 2018.