COMPILATION TECHNIQUES, ALGORITHMS, AND DATA STRUCTURES FOR EFFICIENT AND EXPRESSIVE DATA PROCESSING SYSTEMS

by

Supun Madusha Bandara Abeysinghe Tennakoon Mudiyanselage

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

December 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Tiark Rompf, Chair
Department of Computer Science

Dr. Walid Aref
Department of Computer Science

Dr. Milind Kulkarni
Elmore Family School of Electrical and Computer Engineering

Dr. Yongle Zhang
Department of Computer Science

Approved by:
Dr. Kihong Park

To my beloved wife, Udeshika

ACKNOWLEDGMENTS

I extend my heartfelt appreciation to my advisor, Prof. Tiark Rompf, for his invaluable guidance and mentorship throughout my doctoral journey. Tiark introduced me to the fascinating world of compilers research and provided unwavering support as I navigated the intricacies of my research work. Tiark stands out as one of the most brilliant individuals I have had the privilege to collaborate with.

I would like to express my thanks to Prof. Walid Aref, Prof. Milind Kulkarni, and Prof. Yongle Zhang for their roles on my committee and for the feedback and comments they provided. I truly appreciate their time, dedication, and contributions, which were instrumental in helping me successfully complete my Ph.D.

I would like to extend my deep gratitude to Anxhelo Xhebraj and Qiyang He, who played pivotal roles as close collaborators at various stages of my Ph.D. journey. Their unwavering support and expertise were instrumental in successfully completing several challenging projects. Furthermore, I am indebted to my lab mates and my circle of friends who provided continuous assistance and encouragement. Their willingness to address my queries, offer guidance during moments of difficulty, and assist with the review of early drafts, etc. was instrumental to my progress. As I reflect on this chapter of my life, I will fondly remember the camaraderie and stimulating conversations shared with all of you at LWSN 3133. These memories will be cherished, and I will undoubtedly miss the intellectual and personal connections we formed during this time.

I am immensely grateful to my group of Sri Lankan friends in West Lafayette, whose friendship and companionship outside of my Ph.D. studies added a vital element of fun and joy to my life. The countless weekend hangouts, trips, and a mix of fun and intellectual conversations, among other activities, provided a welcome respite during my long period away from my family.

I also wish to express my appreciation to the Sri Lankan taxpayers, whose contributions funded my education up to my bachelor's degree. This has paved the way for my academic journey, and I am truly thankful for the opportunities this has provided.

Additionally, I owe a debt of gratitude to Dr. Malaka Walpola, Dr. Charith Chitraranjan, and Dr. Uthayasanker Thayasivam for their unwavering support and guidance throughout the application process for graduate school. Their assistance was invaluable in making my academic pursuits a reality.

Last but certainly not least, I extend my heartfelt thanks to my wife, parents, and friends and family. Their unwavering support and belief in me have been the cornerstone of my success. Without their love and encouragement, none of my achievements would have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The proliferation of digital data, driven by factors like social media, e-commerce, etc., has created an increasing demand for highly processed data at higher levels of fidelity, which puts increasing demands on modern data processing systems. In the past, data processing systems faced bottlenecks due to limited main memory availability. However, as main memory becomes more abundant, their optimization focus has shifted from disk I/O to optimized computation through techniques like compilation. This dissertation addresses several critical limitations within such compilation-based data processing systems.

In modern data analytics pipelines, combination of workloads from various paradigms, such as traditional DBMS and Machine Learning, is common. These pipelines are typically managed by specialized systems designed for specific workload types. While these specialized systems optimize their individual performance, substantial performance loss occurs when they are combined to handle mixed workloads. This loss is mainly due to overheads at system boundaries, including data copying and format conversions, as well as the general inability to perform cross-system optimizations.

This dissertation tackles this problem in two angles. First, it proposes an efficient post-hoc integration of individual systems using generative programming via the construction of common intermediate layers. This approach preserves the best-of-breed performance of individual workloads while achieving state-of-the-art performance for combined workloads. Second, we introduce a high-level query language capable of expressing various workload types, acting as a general substrate to implement combined workloads. This allows the generation of optimized code for end-to-end workloads through the construction of an intermediate representation (IR).

The dissertation then shifts focus to data processing systems used for incremental view maintenance (IVM). While existing IVM systems achieve high performance through compilation and novel algorithms, they have limitations in handling specific query classes. Notably, they are incapable of handling queries involving correlated nested aggregate subqueries. To address this, our work proposes a novel indexing scheme based on a new data structure and a corresponding set of algorithms that fully incrementalize such queries. This approach result

in substantial asymptotic speedups and order-of-magnitude performance improvements for workloads of practical importance.

Finally, the dissertation explores efficient and expressive fixed-point computations, with a focus on Datalog–a language widely used for declarative program analysis. Although existing Datalog engines rely on compilation and specialized code generation to achieve performance, they lack the flexibility to support extensions required for complex program analysis. Our work introduces a new Datalog engine built using generative programming techniques that offers both flexibility and state-of-the-art performance through specialized code generation.

1. INTRODUCTION

1.1 Problem statement

The widespread proliferation of the internet, driven by an array of factors such as social media, news, human communication, e-commerce, and the growing prevalence of digital content, has led to an unprecedented accumulation of data in various formats. This surge has resulted in a tension between (A) increasingly huge volumes of data and (B) an increasing demand for highly-processed data at increasing levels of fidelity (cleaned, linked, analyzed and in various ways, fed through neural networks, etc.), which puts increasing demand on data processing systems. For instance, consider an e-commerce website where the ability to extract valuable insights and predictions regarding customer behavior, often through the application of machine learning (ML) techniques, becomes of paramount importance as the volume of accumulated customer transaction data continues to expand.

Historically, due to the limitations of main memory availability, only a fraction of the data could be loaded into main memory for processing at any given time. This limitation made data communication between persistent storage and memory a critical bottleneck in most cases. Consequently, traditional optimization efforts predominantly revolved around improving the efficiency of this data transfer.

However, in recent years, as main memory resources have become more abundant, modern systems are no longer constrained by disk I/O, thereby shifting the optimization focus towards improving efficiency by reducing the actual number of instructions executed. This entails a significant pivot towards optimizing computations themselves.

To this end, most state-of-the-art modern data processing systems employ a form of compilation to generate specialized code tailored to specific use cases. This approach effectively eliminates the overhead associated with interpretation. For instance, modern relational query engines take SQL queries and compile them into specialized native code. This code contains the specialized logic for query operators and incorporates intermediate data structures that are fully specialized (*e.g.*, hash tables specific for a given schema).

The use of compilation techniques extends beyond batch-processing systems or SQL-based relational data processing. For example, DBToaster [1] executes SQL queries on data streams in the form of incremental view maintenance (IVM). It compiles these queries into specialized update triggers, which are invoked as underlying relations change with the arrival of new data.

Similarly, Souffle [2], widely employed for declarative program analysis, compiles programs written in a different declarative language, Datalog. It adopts a similar approach of generating specialized code. Additionally, ML compilers generate native code that incorporates potentially fused, specialized kernels. These kernels are constructed from optimized computation graphs derived from high-level deferred APIs of ML frameworks and are configured for the most optimal performance.

While these systems have demonstrated significantly improved performance compared to their interpreted counterparts, there remain certain limitations and areas for further improvement as discussed below.

- **Efficient and Expressive Multi-Paradigm Workloads** Specialized systems, such as relational query compilers and ML compilers, have demonstrated exceptional performance within their respective individual workloads. However, their performance degrades when these systems are integrated to handle real-world data analytics pipelines. This degradation primarily stems from the overhead at system boundaries when data is transferred from one system to another, along with the absence of comprehensive cross-system, global optimizations. As a result, the challenge of efficiently managing multi-paradigm workloads using compilation-based systems remains an open problem.

- **Efficient and Expressive Incremental Updates** Systems like DBToaster have achieved state-of-the-art performance in incremental view maintenance for a broad range of queries by employing higher-order delta queries and compilation techniques. However, these approaches encounter limitations when dealing with specific classes of queries. Specifically, in cases involving correlated nested aggregate subqueries, they resort to naively re-evaluating the results from scratch, rather than performing incremental updates. This fallback strategy is suboptimal, emphasizing the need for efficient solutions in scenarios where full incrementalization is infeasible purely via higher order deltas.

- **Efficient and Expressive Iterative Fixpoints** While pure Datalog serves as a foundation for declarative program analysis, its expressiveness is occasionally insufficient for handling complex use cases. Consequently, prior works have introduced various extensions, such as aggregates [2], user-defined lattices [3], and SMT constraints [4]. However, in many cases, the implementations of these extensions sacrifice efficiency in favor of achieving greater expressivity. This trade-off can be attributed, in part, to the limited flexibility of existing compiled Datalog engines. Consequently, there exists a significant challenge in architecting extensible Datalog engines capable of accommodating a wide array of extensions while maintaining optimal performance.

In this dissertation, we delve deeper into the aforementioned limitations and propose novel solutions for each. We address the challenge of managing multi-paradigm workloads, such as combined relational and ML processing, by doing an efficient post-hoc integration of existing systems that are built using generative programming. This integration is facilitated through the creation of common intermediate layers, which ultimately compile multi-paradigm workloads into a unified executable. Generative programming-based systems have previously demonstrated considerable success in designing systems that achieve performance levels comparable to highly-engineered counterparts with significantly less engineering effort. In our work, we illustrate how the integration of such systems at a common intermediate layer effectively handles mixed workloads that are encountered in practical scenarios, resulting in state-of-the-art end-to-end performance.

To address the challenge of performing full incrementalization of complex queries containing correlated nested aggregate sub queries, we propose a novel indexing scheme based on a tree-based data structure. This approach is accompanied by a set of corresponding algorithms that enable the full incrementalization of such queries. Our proposed method is asymptotically faster than the current state-of-the-art and achieves order-of-magnitude performance improvements in workloads of practical importance.

In our efforts to construct flexible and performant Datalog engines, we adopt an approach rooted in generative programming. We draw inspiration from existing works on query compilation and develop a Datalog compiler with engine logic implemented in a high-level language that resembles a textbook interpreter. To enhance flexibility in the frontend, we design an embedding of Datalog in Scala, allowing us to leverage Scala's infrastructure to add support for various extensions with minimal effort.

Lastly, we tackle the challenge of managing efficient and expressive multi-paradigm workloads from a different angle. Instead of building common IRs across existing systems, we introduce a high-level query language capable of expressing various types of workloads. This query language serves as a common substrate for different kinds of workloads and enables various optimizations and code generation through the construction of an IR.

The rest of this dissertation provides an elaborate discussion of the identified problems and the proposed solutions.

1.2 Overview

The subsequent chapters of the dissertation delve into a comprehensive exploration of each of the identified problems and present innovative solutions. Below is an overview of the content in each chapter:

Efficient and Expressive Multi-Paradigm Workloads via Common IRs In Chapter 2, we introduce a mechanism for seamlessly integrating specialized systems to manage multi-paradigm workloads. Our approach involves constructing common intermediate layers to facilitate the integration of a state-of-the-art query compiler and a machine learning system. We demonstrate that this integrated approach achieves state-of-the-art end-to-end performance for combined DB and ML workloads, while preserving the individual best-of-breed performance of each system.

Efficient and Expressive Incremental Updates Chapter 3 presents a novel indexing scheme based on an innovative tree-based index structure. We focus on building indexes for partial aggregate values and propose efficient algorithms for shifting ranges of these partial aggregates. This approach enables efficient incrementalization of queries that contain correlated nested aggregate subqueries, addressing limitations observed in previous incrementalization techniques. The proposed method demonstrates significant performance improvements for practical workloads.

Efficient and Expressive Iterative Fixpoints In Chapter 4, we introduce a new Datalog engine developed using generative programming. Our Datalog engine achieves state-of-the-art performance across a variety of workloads. Unlike existing compiled Datalog engines, our system offers flexibility in the frontend through a Scala embedding and in the backend by structuring the engine logic to resemble a textbook interpreter. Simultaneously, it maintains the capability to achieve high performance through specialized code generation.

A New Query Language for Multi-Paradigm Workloads Chapter 5 introduces a novel high-level query language capable of expressing a wide range of workload types, including tensor computations and traditional DBMS operations like joins and aggregates. We also design a novel intermediate representation that facilitates various optimizations and the eventual generation of highly performant code.

The next several subsections provides a summary of each subsequent chapters.

1.2.1 Flern: Efficient Composition of Data Management and Machine Learning Systems via Common Intermediate Layers

Modern data analytics workloads combine relational data processing with machine learning (ML). Most DBMS handle these workloads by offloading these ML operations to external specialized ML systems. While both DBMS and ML systems go to great lengths to optimize performance for their specific workloads, significant performance is lost when used in combination, due to data movement across system boundaries, conversions between incompatible internal data formats, and the general inability to perform optimizations across systems.

A key idea to remove these bottlenecks is to integrate existing data manipulation systems with ML systems by building a common intermediate layer (IR). Although this idea has been explored before (Weld, Delite), previous such attempts require significant re-engineering of prior systems and still fail to achieve best-of-breed performance for individual tasks (*e.g.*, SQL, Deep Learning). Specifically, they rely on re-implementing existing systems using a generic set of operators and fail to match best-of-breed individual performance due to the inability to recover high-level optimizations from this generic IR through compiler analysis.

In Chapter 2, we present Flern, the first intermediate-layer integration between DB and ML systems that are best-of-breed individually (competitive with the best compiled query engines such as HyPer on a full relational benchmark TPC-H and competitive with TensorFlow and PyTorch in state-of-the-art deep learning models *e.g.*, DeepSpeech, SqueezeNet) and also represents a new state-of-the-art for integration. A key realization is to architect intermediate layers based on generative programming capabilities, which preserves high-level contextual information for cross-optimizations and enables the construction of a variety of complex structures and cross-system optimizations with minimal effort.

1.2.2 Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI)

Incrementalization of queries is imperative in cases where data arrives as streams and output is latency-critical and/or desired before the full data has been received. Incremental execution computes the output at a given time by reusing the previously computed outputs or maintained views rather than re-evaluating the query from scratch. There are various approaches to perform this incrementalization ranging from query-specific algorithms and data structures (*e.g.*, DYN, AJU) to general systems (*e.g.*, DBToaster, Materialize).

DBToaster is a state-of-the-art system that comes with an appealing theoretical background based on the idea of applying Incremental View Maintenance (IVM) recursively, maintaining a hierarchy of materialized views via delta queries. However, one key limitation of this approach is its inability to efficiently incrementalize correlated nested-aggregate queries due to an inefficient delta rule for such queries. Moreover, none of the other specialized approaches have shown efficient ways to optimize such queries either. Nonetheless, these types of queries can be found in many real-world application domains (*e.g.*, finance), for which efficient incrementalization remains a crucial open problem.

In Chapter 3, we propose an approach to incrementalize such queries based on a novel tree-based index structure called Relative Partial Aggregate Indexes (RPAI). Our approach is asymptotically faster than other systems and shows up to 1100× speedups in workloads of practical importance.

1.2.3 Flan: An Expressive and Efficient Datalog Compiler for Program Analysis

Datalog has gained prominence in program analysis due to its expressiveness and ease of use. Its generic fixpoint resolution algorithm over relational domains simplifies the expression of many complex analyses. The performance and scalability issues of early Datalog approaches have been addressed by tools such as Soufflé through specialized code generation. Still, while pure Datalog is expressive enough to support a wide range of analyses, there is a growing need for extensions to accommodate increasingly complex analyses. This has led to the development of various extensions, such as Flix, Datafun, and Formulog, which enhance Datalog with features like arbitrary lattices and SMT constraints.

Most of these extensions recognize the need for full interoperability between Datalog and a full-fledged programming language, a functionality that high-performance systems like Soufflé lack. Specifically, in most cases, they construct languages from scratch with first-class Datalog support, allowing greater flexibility. However, this flexibility often comes at the cost of performance due to the conflicting requirements of prioritizing modularity and abstraction over efficiency. Consequently, achieving both flexibility and compilation to highly-performant specialized code poses a significant challenge.

As presented in Chapter 4, in our work, we reconcile the competing demands of expressiveness and performance with Flan, a Datalog compiler fully embedded in Scala that leverages multi-stage programming to generate specialized code for enhanced performance. Our approach combines the flexibility of Flix with Soufflé's performance, offering seamless integration with the host language that enables the addition of powerful extensions while generating specialized code for the entire computation. Flan's simple operator interface allows the addition of an extensive set of features, including arbitrary aggregates, user-defined functions, and lattices, with multiple execution strategies such as binary and multi-way joins, supported by different indexing structures like specialized trees and hash tables, with minimal effort. We evaluate our system on a variety of benchmarks and compare it to established Datalog engines. Our results demonstrate competitive performance and achieves speedups in the range of 1.4× to 12.5× compared to state-of-the-art systems for workloads of practical importance.

1.2.4 Rhyme: A Data-Centric Expressive Query Language for Nested Data Structures

In Chapter 5, we present Rhyme, an expressive language designed for high-level data manipulation, with a primary focus on querying and transforming nested structures such as JSON and tensors, while yielding nested structures as output. Rhyme draws inspiration from a diverse range of declarative languages, including Datalog, JQ, JSONiq, Einstein summation (Einsum), GraphQL, and more recent functional logic programming languages like Verse. It has a syntax that closely resembles existing object notation, is compositional, and has the ability to perform query optimization and code generation through the construction of an intermediate representation (IR). Our IR comprises loop-free and branch-free code with program structure implicitly captured via dependencies. To demonstrate Rhyme's versatility, we implement Rhyme in JavaScript (as an embedded DSL) and illustrate its application across various domains, showcasing its ability to express common data manipulation queries, tensor expressions (à la Einsum), and more.

1.3 Hypothesis

Our thesis hypothesis posits that the utilization of generative programming-based compilation techniques, novel algorithms, and innovative query language designs results in significantly enhanced performance across a diverse range of data processing systems.

1.4 Contributions

The key contributions of our work are as follows:

1. **Efficient and Expressive Multi-Paradigm Workloads via Common IRs** In Chapter 2, we present Flern, the first intermediate-layer integration between DB and ML systems that are best-of-breed individually, and achieves state-of-the-art end-to-end performance.

   (a) We analyze the limitations of existing approaches to build common intermediate layers and discuss why generative programming is a better choice to integrate systems efficiently with minimal re-engineering cost (Section 2.2)

   (b) We demonstrate how generative programming enables efficient post-hoc integration of prior systems by combining two state-of-the-art systems in DB and ML domains. Specifically, we show how our generative approach can be used to eliminate overheads at systems boundaries and implement cross optimizations while preserving the best-of-breed performance of individual systems (Section 2.3)

   (c) We present two sets of benchmarks where we first evaluate the performance impact of each optimization, and secondly comparing the performance to state-of-the-art baselines to show that the our approach (1) either outperforms or achieves competitive results when used in isolated tasks (*i.e.*, data manipulation and ML independently) and (2) achieves state-of-the-art performance (showing up to order of magnitude speedups) when used in combined tasks (Section 2.4).

2. **Efficient and Expressive Incremental Updates** In Chapter 3, we present RPAI, a novel tree based index structure with the associated algorithms for efficient incrementalization of correlated nested-aggregate queries.

   (a) We present a case study with two examples of nested aggregate queries and analyze how existing approaches handle those queries. Then, we motivate our approach of using aggregate indexes (PAI Maps and RPAI Trees) and demonstrate how such structures can improve the incrementalization (Section 3.3).

   (b) We design an efficient tree-based data structure for RPAI and present algorithms for the key operators with an analysis of their time complexity (Section 3.4).

   (c) We present a novel general algorithm for incrementalizing correlated nested aggregate queries, followed by further optimizations using PAI Maps and RPAI Trees. Then, we discuss the limitations and overheads associated with our approach. (Section 3.5).

    (d) We evaluate the performance of our algorithm against DBToaster, a state-of-the-art system that supports incremental execution of SQL queries, and show that our approach performs significantly better in real-world datasets in line with the expected performance behaviors due to asymptotic speedups (Section 3.6).

3. **Efficient and Expressive Iterative Fixpoints** In Chapter 4 we present Flan, a novel Datalog engine that is capable of handling various Datalog extensions while having the ability to achieve performance via compilation.

    (a) We elucidate the rationale behind our choice of employing generative programming, specifically, LMS, as the basis for constructing Flan (Section 4.2).

    (b) We review essential background and demonstrate the construction of a simple Datalog interpreter from scratch in Scala (Section 4.3).

    (c) We illustrate how to effortlessly transform the Datalog interpreter into a compiler that generates fast, specialized code by utilizing LMS (Section 4.4.1). We demonstrate the integration of various extensions such as constraints, UDFs, negations, and aggregations through streamlined abstractions (Section 4.4.2). We highlight Flan's backend flexibility by adding support for multiple join evaluation strategies and index structures (Sections 4.4.3 and 4.4.4).

    (d) We showcase the Datalog compiler can be seamlessly embedded into a full programming language, capitalizing on existing features of the language (e.g., type system, abstractions) for enabling composable, polymorphic, higher-order Datalog programs. Moreover, we demonstrate how this simplifies the process of enriching Datalog with features such as user-defined lattices (Section 4.5).

    (e) We compare our engine with state-of-the-art Datalog engines such as Soufflé, Ascent, Crepe, and Flix across a diverse range of benchmarks. Flan consistently delivers competitive or superior performance in each benchmark (Section 4.6).

4. **A New Query Language for Multi-Paradigm Workloads** In Chapter 5, we present Rhyme, a new query language capable of expressing a variety of data processing

workload while having the ability to performance optimizations and code generation via the construction of an IR.

(a) We introduce the syntax of Rhyme, showcasing the ability to express common data manipulation operators such as selections, group-bys, joins, user-defined functions (UDFs), and others (Section 5.2).

(b) We highlight the versatility of Rhyme across various use cases, including the expression of visual elements in web applications (e.g., tables, charts using SVG), declarative tensor computations (akin to Einsum), and alternative 'pipe' APIs via metaprogramming (Section 5.3).

(c) We elucidate the process of lowering queries into an IR that features loop-free and branch-free code, with dependencies implicitly representing the program structure. Then, we illustrate how this IR facilitates code generation by constructing the optimal program structure from dependencies (Section 5.4).

(d) We evaluate the performance of Rhyme on several JSON analytics workloads to demonstrate the effectiveness of our code generation approach (Section 5.5).

1.4.1 Publications

The content of the dissertation is based on the following papers:

- *Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI).*
  Supun Abeysinghe, Qiyang He, Tiark Rompf
  Appeared in Proceedings of the 2022 International Conference on Management of Data (SIGMOD 2022).

- *Architecting Intermediate Layers for Efficient In-Database Machine Learning*
  Supun Abeysinghe, Fei Wang, Gregory Essertel, Tiark Rompf
  Presented at the 7th Annual Symposium on Machine Programming (MAPS 2023).

- *Flan: An Expressive and Efficient Datalog Compiler for Program Analysis*
  Supun Abeysinghe, Anxhelo Xhebraj, Tiark Rompf
  Will appear in Proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024).
- *Rhyme: A Data-Centric Expressive Query Language for Nested Data Structures*
  Supun Abeysinghe, Anxhelo Xhebraj, Tiark Rompf
  Will appear in Proceedings of 26th International Symposium on Practical Aspects of Declarative Languages (PADL 2024).

2. ARCHITECTING INTERMEDIATE LAYERS FOR EFFICIENT IN-DATABASE MACHINE LEARNING

Portions of this chapter have appeared in *Architecting intermediate layers for efficient composition of data management and machine learning systems, arXiv preprint arXiv:2311.02781, 2023* [5].

2.1 Introduction

Today's data analytics workloads often combine traditional relational processing with modern machine learning (ML) operations. These two workload types are significantly different from each other and have different domain-specific properties that lead to the construction of independent specialized systems (i.e., DBMS for relational processing and ML frameworks for ML operations). These systems go to great lengths to optimize their workloads by taking domain-specific properties into account (e.g., query compilation, binding to low-level specialized kernels).

However, when we combine these systems to build complex data analytics pipelines, the end-to-end performance becomes suboptimal. This can be due to multiple reasons including the general inability to perform global optimizations (i.e., across individual system boundaries) and data movement and conversion overheads present at the boundaries (e.g., due to incompatible data formats). For example, a simple relational query that uses an ML model as a function, constructed inside Postgres (using PyTorch for ML) can be more than 50x slower than a manual hand-optimized program written for the same task.

In this work, we address the following problem. How to incorporate full ML capabilities to existing DBMS while (1) eliminating expensive overheads at system boundaries when external ML systems are used, (2) performing global optimizations on the entire computation rather than treating ML computations as black boxes, (3) without re-engineering systems from scratch.

One approach is to add support for ML computations inside DBMS by extending SQL engines with new operators (e.g., user-defined functions, iterative computations, etc.) [6–11].

However, these approaches have limited support for the types of models they can run and do not support modern ML models (e.g., Transformers [12]). Moreover, the performance is not on par with specialized ML systems (e.g., PyTorch, TensorFlow, etc.) mainly due to the inability to incorporate domain-specific optimizations.

Another approach, specifically to alleviate the data copying overheads at system boundaries is introducing common low-level data layouts that can be used across multiple systems. Potential candidates include NumPy arrays and Apache Arrow [13]. However, such common data formats typically do not *directly* support efficient data manipulation as required internally by some systems. Moreover, these common data formats have to be sufficiently generic, therefore, yielding suboptimal performance compared to workload-specific, optimized data layouts, and are thus not well suited as internal data formats, so in practice, format conversion to and from interchange formats remains at system boundaries. Besides, even an approach based on shared internal formats would not enable cross-optimization of computation, such as eliminating entire materialized data structures via operator fusion across systems.

Another, more intrusive but also more powerful, approach is to build a common intermediate layer across different systems. The intermediate layer must be sufficiently generic to support all operations of disjoint systems. That is, it should support all SQL/DataFrame operations (e.g., different types of joins, aggregation queries, etc.) and all deep learning operators (e.g., automatic differentiation (AD), sophisticated recursive ML models, etc.).

This could be achieved by making a single IR that contains the union of SQL engines and ML systems (e.g., TensorFlow [14], PyTorch [15]). However, in practice, such an approach is extremely challenging to realize due to the need to implement a plethora of special cases, and essentially rewriting multiple systems as a unified whole from ground up or subsume one into the other which requires an incredible engineering effort to match existing capabilities (i.e., need to implement a best-of-breed RDBMS **and** a best-of-breed tensor framework **and** achieve end-to-end optimal performance).

Therefore, existing work that follows the idea of building common intermediate layers [16–18] imposes a single, fixed intermediate layer as a *one-size fits all* consisting of a handful of generic operators (e.g., *map*, *reduce*, etc.). While there is considerable flexibility for optimization, this generic IR lacks vital contextual information which makes it difficult to recover low-level performance through compiler analysis. Moreover, all the high-level operators (e.g., Joins, Tensor operations, etc.) of existing systems (or their front-ends) should be re-implemented using these generic IR constructs which requires a significant engineering effort and can be challenging for complex operators (for instance, in Section 2.2, we investigate a Hash Join implementation of such a system and show how it differs from a classic textbook implementation).

In this work, we explore the use of generative programming to architect such common intermediate layers to combine pre-existing systems. Generative programming has demonstrated great effectiveness in architecting large-scale systems for individual workloads with relatively smaller engineering effort compared to highly engineered counterparts, while achieving competitive performance [19, 20]. This generative approach allows programmers to develop their systems using a high-level language with user-friendly features (e.g., high-level type system, high-level data structures, abstractions such as classes, interfaces, generics, etc.) while achieving native performance by translating this high-level code to low-level native code (thus, *generative*) which does not contain any of the high-level abstractions, hence, *abstraction without regret* [21].

Our approach based on generative programming differs from previous intermediate layer integrations as we have different layers of abstractions that are responsible for specific optimizations (e.g., tensor transformations, specialized data structures, etc.) as opposed to a fixed generic IR (Figure 2.1). A key realization of our work is, this ability to preserve domain-specific abstractions and optimizations of individual systems and the ability to add more cross optimizations using these abstractions is vital for building systems that are competitive with specialized systems for individual workloads and also outperforming previous integrations for combined workloads. Moreover, properties of generative programming such as the ability to build high-level data structures, having programmatic control, etc., allows us to eliminate overheads at system boundaries and implement key optimizations with minimal effort.

We demonstrate our approach enables efficient post-hoc integration of specialized pre-existing systems with smaller engineering effort. Specifically, we build Flern, a query compiler that supports full ML capabilities, by doing an intermediate layer integration between a state-of-the-art query compiler (Flare [19, 20]) and an ML framework (Lantern [22, 23]) that are developed based on the same generative programming technique called Lightweight Modular Staging (LMS) [21]. Flern is the first system that is best-of-breed individually (competitive with state-of-the-art in-memory SQL engines like HyPer [24] on full relational benchmarks [20] e.g., TPC-H [25] and competitive with TensorFlow and PyTorch in sophisticated deep learning models [22] [23] e.g., DeepSpeech [26], SqueezeNet [27], Transformer [12]) and also the new state-of-the-art for integration. Flern can be used as both a query compiler for queries that combine relational processing with ML (i.e., in-Database ML) or to accelerate end-to-end data science pipelines that consists of a data manipulation phase followed by ML operations (e.g., Spark + PyTorch workload). For the later case, Flern supports widely used high-level front ends (e.g., Pandas and PyTorch) using existing tools [28], making it possible to accelerate existing implementations with minimal code changes.

2.1.1 Contributions

The main intellectual contribution of this work is to analyze the limitations of existing approaches to extend DBMS with ML capabilities (e.g., incorporating ML operators into the relational model, introducing common data formats) and to present an approach to tackle this problem of combining systems in general: architect systems based on generative programming, so that they can be adapted more effectively and at a lower engineering cost. We demonstrate that architecting intermediate layers based on generative programming that preserves contextual information (as opposed to generic IR layers) is the key to build state-of-the-art systems that handles combined DB and ML workloads.

Our specific contributions are summarized as follows:

- We analyze the limitations of existing approaches to build common intermediate layers and discuss why generative programming is a better choice to integrate systems efficiently with minimal re-engineering cost (Section 2.2)

- We demonstrate how generative programming enables efficient post-hoc integration of prior systems by combining two state-of-the-art systems in DB and ML domains.

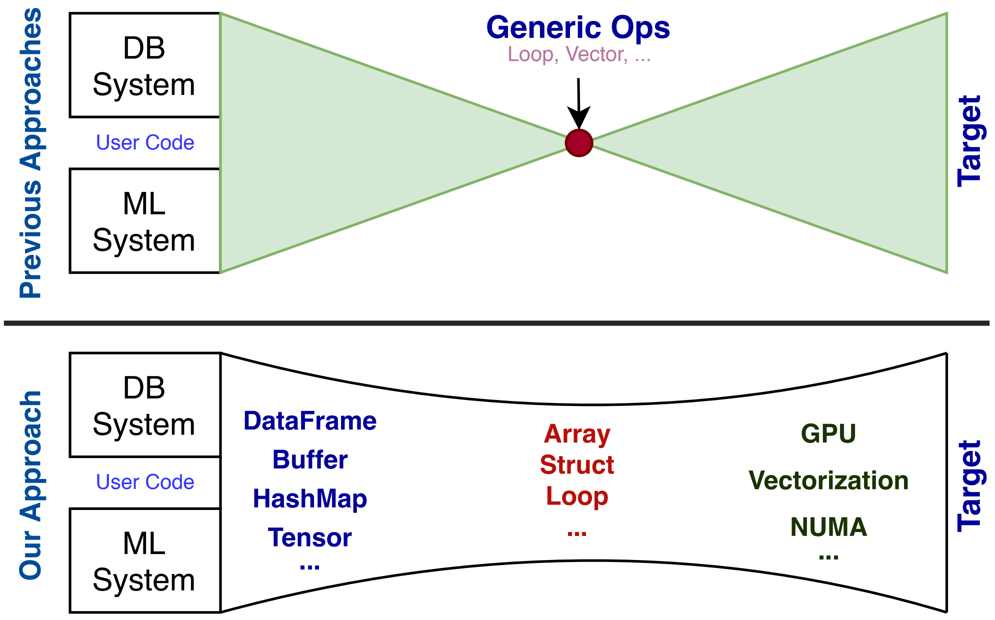


Figure 2.1. Prior approaches map the user program directly into a single common generic IR. This IR lacks vital information to perform high-level global optimizations, hence, becoming a bottleneck. In contrast, Flern consists of multiple abstraction levels each responsible for optimizations at their corresponding level with rich contextual information.

Specifically, we show how our generative approach can be used to eliminate overheads at systems boundaries and implement cross optimizations while preserving the best-of-breed performance of individual systems (Section 2.3)

- We present two sets of benchmarks where we first evaluate the performance impact of each optimization, and secondly comparing the performance to state-of-the-art baselines to show that the our approach (1) either outperforms or achieves competitive results when used in isolated tasks (i.e., data manipulation and ML independently) and (2) achieves state-of-the-art performance (showing up to order of magnitude speedups) when used in combined tasks (Section 2.4).

In Section 2.5, we present an analysis of related work in this domain. Finally, in Section 2.6, we draw conclusions and discuss potential future research directions.

2.2 Motivation

2.2.1 Why Intermediate Layer Integration?

Specialized ML systems employ different techniques to achieve good performance for ML workloads. These include implementing operations using high-level abstractions (e.g., Tensor) that make it relatively easy to add domain-specific optimizations (e.g., Tensor transformations). Therefore, to match the performance of these specialized ML systems when doing in-Database ML, the ability to incorporate these optimizations is crucial. This can be challenging in cases where ML operations are added to DBMS using different abstractions and mechanisms (e.g., extended SQL execution engines, user-defined functions, etc.) due to the difficulty of translating existing optimizations strategies to the new setting.

Another approach is to integrate DBMS with an existing ML system that already contains all the domain-specific optimizations, eliminating the need for significant re-engineering costs. For example, PL/Python plugin allows functions to be written in Python inside PostgreSQL. Therefore, any ML framework that has interfaces in Python (e.g., PyTorch, TensorFlow) can be used inside the DB to implement ML operations. With this, the DBMS gets full capabilities of fully-fledged ML systems to be used for combined relational and ML workloads. However, one of the major drawbacks of this approach is the fact that both systems treat the other system as a black-box, incurring data copying and format conversions at system boundaries and preventing any cross-system optimizations.

Specifically, in the case of Postgres, first, the data needs to be moved from the DB to the Python environment. Then, this data is converted into tensors, the data format supported by PyTorch. Once the ML computation is completed, the resultant tensors should be converted back to PL/Py objects that are then copied back to Postgre where they finally get converted into the original record format. For example, when running a three-layer neural network model on NYC-Taxi dataset[1] on this stack, the actual execution time of the ML computations is around 6 seconds whereas the total time comes to 638 seconds, showing the magnitude of these overheads (analyzed in Section 2.4.4).

[1]↑https://www.kaggle.com/c/nyc-taxi-trip-duration

To investigate whether we can eliminate or significantly reduce these overheads, we look into query compilers [19, 20, 24]. Query compilers accelerate the query execution time by generating low-level code that is specialized to a given query. Generally, these systems construct an intermediate representation (IR) for the input query, perform various optimizations on the IR and generate code from this optimized IR. Then, this generated code is compiled (by a general-purpose compiler) and executed to obtain the result. A key idea to eliminate the aforementioned cross-system data movement overheads is to integrate systems at this IR level. Specifically, lowering both DBMS (i.e., query compiler) and ML computations into a single common IR such that global optimizations can be performed at the IR level, and data movement and conversion overheads can be minimized by carefully analyzing the IR.

2.2.2 Why Generative Programming?

Delite [18] and Weld [16, 29] are two examples of prior systems that follow the approach of building common IRs across multiple systems where they design the IR constructs to support multiple front-end use cases. Both these approaches share the common characteristic of having a fixed, generic intermediate layer which consists of a few generic operators that the front-end should use to implement their functionality. This approach comes with great flexibility. That is, any optimizations we add to this generic IR layer can be reused for any of the front ends. However, in practice, this flexibility comes with a significant cost. First, due to the generic nature of this IR, most of the contextual information from the high-level abstractions (e.g., `Tensor`, `DataFrame`, etc.) is lost at the common IR level. For example, if a 2D Tensor is transposed twice consecutively, it is relatively easy to figure out this results in a no-op if we have access to contextual information about what the data is (i.e., a 2D Tensor) rather than capturing this by performing compiler analysis on the looping structures of the generic IR. Second, it can be challenging to implement some of the complex operators using these minimal IR constructs due to the likelihood of implementations varying significantly from the textbook implementations.

For instance, Weld defines a minimal IR that captures the structure of common data-parallel algorithms, and a runtime API that lets disjoint libraries construct Weld IR fragments. The host libraries should completely re-implement their operators to emit the IR fragments required for the operator logic. These emitted IR fragments are accumulated to a single global IR by the backend. Then, the combined IR will go through multiple layers of optimization and will generate optimized machine code (via LLVM [30]).

Figure 2.2 shows the implementation of the `Join` operator in Weld[2] [16]. The implementation differs significantly from textbook pseudo-code and contains a stringified code template for the `Join` operator and the holes will be filled based on the function parameters. Since this kind of template expansion is limited by the fact that the code is manipulated as strings, it is impossible or challenging to utilize features of high-level programming languages such as type checking on these code templates, and generally harder to implement and maintain. This typically results in systems that only support a certain set of cases (e.g. only one type of joins, or limited ML ops, etc.) due to the required engineering effort and other complexities. Therefore, even though there is potential for improvement in end-to-end performance, these approaches either do not support or fail to achieve best-of-breed performance for specific workloads in full benchmarks (e.g. full TPC-H).

In contrast, generative programming is fine-grained, supports lots of programmatic control, and can rely on rich contextual information. For instance, Lightweight Modular Staging (LMS) [21] is a generative programming framework in Scala based on multi-stage programming (staging, for short) [31] and runtime code generation. Multi-stage programming is a paradigm that allows developers to write generic programs using higher degrees of abstractions without incurring a runtime penalty [32]. The high-level idea behind staging is to delay computations of certain operations to a later stage, generating code for such operations with the information known in the current stage. Code snippet in Figure 2.4 shows LMS in action for a small example. Observe that the implementation is similar to programming in a regular language (i.e., does not use big chunks of stringified code templates).

LMS is type-driven and uses `Rep` types to indicate the values that are going to be computed in the next stage, and hence, should be computed by the generated code (`Rep` **rep**resents next-stage expressions). All the operations including language constructs (e.g., control flow) on normal non-`Rep` expressions (e.g., `Int`, `String`, etc.) are evaluated at stag-

[2]↑https://github.com/weld-project/weld

```python
def join(expr1, expr2, d1_keys, d2_keys, keys_type, d1_vals, df1_vals_ty
    , d2_vals, df2_vals_ty):
  weld_obj = WeldObject(encoder_, decoder_)

  df1_var = weld_obj.update(expr1)
  if isinstance(expr1, WeldObject):
    df1_var = expr1.obj_id
    weld_obj.dependencies[df1_var] = expr1
  df2_var = weld_obj.update(expr2)
  if isinstance(expr2, WeldObject):
    df2_var = expr2.obj_id
    weld_obj.dependencies[df2_var] = expr2
  #Some String manipulations constructing holes of the template elided
  weld_template = """
  let df2_join_table = result(
    for(
      %(df2)s,
      groupmerger[%(kty)s, %(df2ty)s],
       b, i, e  merge(b, {%(df2key)s, %(df2vals)s})
    )
  );
  result(for(
    %(df1)s,
    appender,
     b, i, e
      for(
        lookup(df2_join_table, %(df1key)s),
        b,
         b2, i2, e2  merge(b, {%(df1key)s, %(df1vals)s, %(df2vals2)s})
      )
  ))"""

  weld_obj.weld_code = weld_template % {"df1":df1_var, "df1key":
   d1_key_struct, "df1vals": d1_val_fields, "df2":df2_var, "kty":
   keys_type, "df2ty":df2_vals_ty, "df2key":d2_key_struct, "df2vals":
   d2_val_struct, "df2vals2":d2_val_fields2}

  return weld_obj
```

Figure 2.2. Join Implementation of Weld (for accelerating Pandas). The operators emit code for IR construction as blocks of strings. This kind of manipulation of code in stringified form is generally error-prone, relatively harder to maintain and implement.

```
case class HashJoinOp(left: Op, right:Op)(/* elided */) extends Op {
  // HashMap specialized for corresponding Schemas
  val map = LinkedHashMap(keySchema, left.schema)
  def exec(callback: Record => None) = {
    // Store records from left operator tree in the HashMap
    left.exec { tuple =>
      map.update(leftHash(tuple), tuple)
    }

    // Retrive records from right subtree and join
    right.exec { tuple =>
      for (lTuple <- map(rightHash(rTuple)) if joinCond(lTuple, rTuple))
        callback(lTuple ++ rTuple)
    }
  }
}
```

Figure 2.3. Hash-Join Implementation of Flare which is a query compiler based on generative programming. The implementation contains the operator logic implemented using a high-level programming language (Scala) as opposed to emitting blocks of stringified code and has access to all the features of the host language (e.g., type system, abstractions, etc.).

ing time whereas all `Rep`-typed expressions generate code. Figure 2.5 shows the generated C code.

This example also shows a glimpse of one of many optimizations done in LMS. In particular, because of common subexpression elimination, power values computed midway (`power(b, 3)`) are reused. Another key observation is the original source code (in Scala) is almost the same as a normal Scala code except for the type annotation which indicates the variables that need to be staged. Generally, this pattern is true for most cases, that is, a developer just needs to change the types of a normal Scala program to convert it into a staged program.

Under the hood, LMS maintains an extensible graph-like IR to capture the constructs and operations of the staged program. It comes with several optimizations at the IR level (e.g., loop fusion, common sub-expression elimination, loop unrolling, function inlining, etc.)

```
def power(b: Rep[Int], n: Int):Rep[Int] =
  if (n == 0)
    1
  else if (n % 2 == 0)
    power(b, n/2) * power(b, n/2)
  else
    b * power(b, n-1)

def main(args: Rep[Array[String]]) =
  println(power(args(0).toInt, 7))
```

Figure 2.4. Implementing `power` function using LMS where the normaly typed `n` is known at staging time and is evaluated while `Rep`-typed `b` generates code. A key observation here is that the code is similar to implementing the same function in regular programming.

```
int main(int argc, char** argv) = {
  int x0 = atoi(argv[0]); // args(0).toInt
  int x1 = x0 * (x0 * x0); // x1 = x0 ** 3
  printf("%d\n", x0 * (x1 * x1));
  // x0 * ((x0 ** 3)* (x0 ** 3)) = x0 ** 7
}
```

Figure 2.5. Code generated by LMS for the `power` function in Figure 2.4. Observe that the program is specialized to the specified `n` value.

and on top of that, library developers can write their domain-specific optimizations easily due to the extensible nature of the IR [33]. In this context, we can think of LMS as enabling us to use all of Scala as a macro language in building the intermediate layer. This makes it efficient and easy to implement sophisticated data structures and algorithms required for advanced data manipulations and supporting full deep learning capabilities.

Generally, generative programming can be viewed as template expansion in the limit, where operators are primitive operations (e.g., arrays, pointers, etc.) instead of abstract operators like `OuterJoin`, `TableScan` (in Spark [34]), and the templating engine is a full Turing-complete and expressive language. Prior work has demonstrated the effectiveness of generative programming as a systems-building technique that dramatically simplifies the construction of high-performance systems that have a high degree of internal variability and need specialization to achieve performance. Consider the case of databases and Flare's [20] query engine. Systems like Oracle or PostgreSQL consist of 10M LOC, and e.g. PostgreSQL has 7 BTree implementations and 10 page abstractions, each specialized differently for performance [35]. But this level of specialization is still not sufficient for top-of-the-line performance. HyPer [24] was the first system to use LLVM [30] for custom query code generation with 10x speedups, but with the drawback that each query operator needs to manage basic blocks, virtual registers, etc. DBLAB [36] is another system that used a complicated stack of 5 different intermediate languages to achieve specialized index data structures, for additional performance gains.

Generative programming based Flare/LB2 [19, 20] achieves all that and more in 3000 lines of high-level Scala code that looks like a textbook implementation of relational algebra. In other words, generative programming vastly increases the power of each LOC. For example, Figure 2.3 shows the implementation of `HashJoin` operator in LB2. The implementation is equivalent to writing the hash join algorithm for a simple query interpreter in a high-level programming language (Scala). The developer has full access to high-level data structures (e.g. `LinkedHashMap`) specialized for the given schemas, abstractions (e.g. `Record`), and the Scala type system as opposed to writing *brittle* stringified code templates.

In the case of composing such systems by constructing common IRs, these properties of generative programming are useful for adapting system boundaries with minimal overhead and implementing key global optimizations that span across system boundaries. We can implement such optimizations using different levels of abstractions (see Figure 2.1) while using a high-level programming language with full access to contextual information about the high-level operators (i.e, cross optimizations can be written using high-level abstractions like `Tensor`, `DataFrame`, etc.) as opposed to having to implement all global optimizations on a generic IR. In Section 2.3, we show how these properties of generative programming can be leveraged to combine a generative programming based query compiler with a ML system with minimal overheads at system boundaries and how we can implement cross system operations and optimizations to achieve state-of-the-art performance in combined DB and ML workloads.

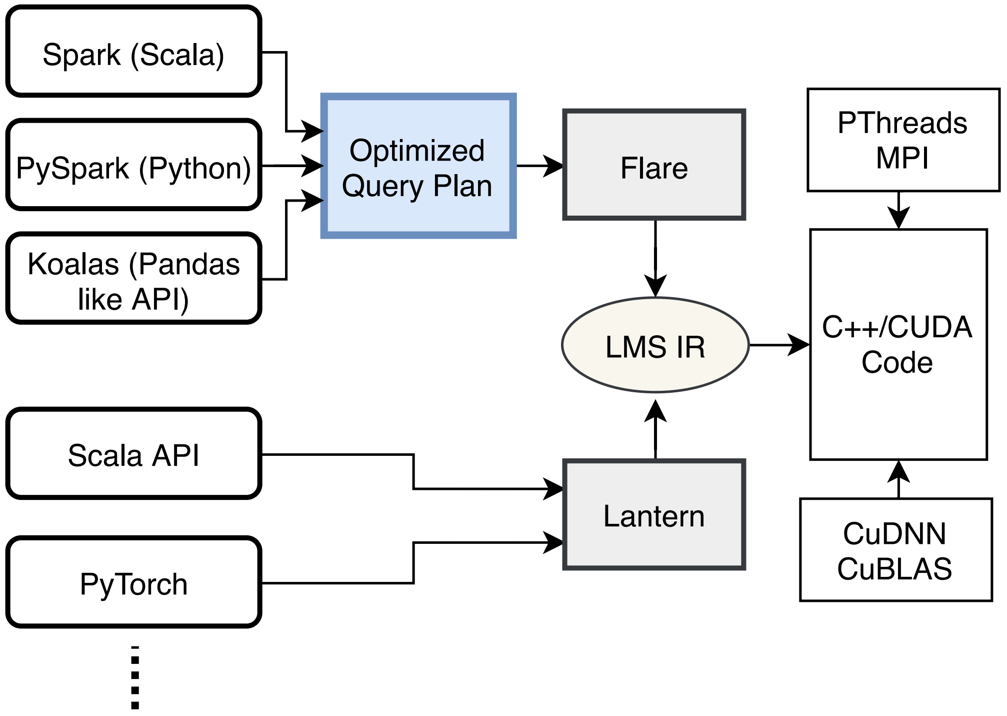


Figure 2.6. The overall architecture of Flern. The end users can use their familiar Spark, Pandas, PyTorch, etc. APIs.

2.3 Flern Overview

In this section, we demonstrate how to leverage the benefits of generative programming to combine two pre-existing, independently developed, systems for the domains of DB (relational) and ML (tensor) workloads. Specifically, we build Flern, by combining Flare [20], a query compiler for SQL, and Lantern [22, 23, 37], an ML framework, which are developed based on the same generative programming approach, Lightweight Modular Staging (LMS) [21] (Figure 2.6). These two systems support full functionality in their respective domains (e.g., full SQL, support for any DL model) and achieve competitive performance with best-of-breed individual systems. Figure 2.7 shows a code snippet of Flern where it is used to accelerate a Spark query that uses an ML classifier.

Flare is built on top of LB2 [19], a high-level query compiler developed using generative programming techniques. LB2 implements relational operators in a way that is similar to a simple query interpreter and converts it to a query compiler using Futamura projections and partial evaluation [38]. Flare is used as an accelerator for Apache Spark [39, 40] specifically targeting scale-up execution. Flare operates on the optimized query plans generated by Spark's query optimizer (Catalyst [34]) and performs query compilation and runtime native code generation. Flare shows that Spark's focus on individual query blocks (i.e. at the granularity of operator pipelines) holds back performance due to added overheads at the boundaries of queries and instead performs whole query compilation [20]. This code generation strategy of Flare achieves orders of magnitude speedups over Spark and other RDBMS and achieves competitive performance with state-of-the-art in-memory query compilers like Hyper [24] in full relational benchmarks (e.g., TPC-H).

Lantern [22, 23, 37] is a differentiable programming framework that handles automatic differentiation via delimited continuations [41], and code generation via LMS [21]. Delimited continuations allow Lantern to support ML models with in-graph control-flows such as conditionals, loops, and functions. LMS reifies the computation graph (after automatic differentiation) for code generation that utilizes various BLAS and neural network kernel libraries for multiple hardware platforms (such as CPU and GPU). Lantern performs competitively with existing deep learning frameworks (e.g., PyTorch and TensorFlow) in state-of-the-art deep learning models.

Flern can be mainly used in two use cases. First, it can be used inside a DB as a query compiler for queries that combine relational processing with ML (or tensor) computations (i.e., in-Database ML). For example, a query containing one or more pre-trained ML classifiers as functions in a SQL query. Second, Flern can be used to accelerate end-to-end data science pipelines that are built using data manipulation (e.g., Pandas [42], Spark) and ML frameworks (e.g., PyTorch, TensorFlow). Here, the ML system trains models by repeatedly retrieving batches of data by querying the data manipulation system (or from a materialized query output). In both cases, integration between DB and ML systems is needed and data needs to be transferred from one system to the other.

Remember, our primary design goal is to extend DBMS with ML capabilities while incurring minimal re-configuration to pre-existing systems that ensure the ability to retrain their individual performance and improving combined performance. Flare and Lantern are two state-of-the-art systems in their respective workloads. Therefore, one can simply run these two systems independently and transfer data between the two systems using a commonly used data format like CSV (or a common intermediate exchange format like Arrow [13]). This would not require any substantial modifications to the two systems and can be achieved by adding a CSV exporter and an importer in the respective system boundaries. This would retain the performance of individual systems, however, incurs a significant overhead due to the need for exporting the data in a chosen format, loading the data, parsing the loaded data, etc (evaluated in Section 2.4.2). A simple improvement over this naive implementation would be to use a binary format that can reduce the overhead associated with formatting the data and parsing the loaded data, and communicating over memory (e.g., using pipes). Yet, overheads remain due to communication over different runtime environments. Moreover, in cases where we build complex relational queries mixed with tensor computations (e.g., ML Classifiers as UDFs), this data movement may have to happen multiple times back and forth and both systems should be modified carefully to handle such interactions with external systems.

2.3.1 Adapting the Boundaries of Flare and Lantern

A crucial characteristic of these two systems is that they are based on the same generative programming approach (LMS). As we discussed in Section 2.2, a key benefit of this generative approach is that these high-performance systems are implemented using Scala, which is a feature-rich high-level programming language. Therefore, we can simply combine by importing them as Scala libraries which would give us access to functionalities of both systems. However, rather than combining them using the traditional way of passing data across functions of respective libraries which causes data movement and conversion overheads, we can bring the two systems to a common layer and build a single IR graph at the LMS level. Specifically, we create a common layer that supports the union of operators from both systems while ensuring to preserve all the transformation and optimizations of individual workloads.

```
// initialize spark context, data schemas, etc.
withSpark("flern-test") { spark =>
  val dataDf =
  spark.createDataFrame(/* data path, schema, etc.*/)

  dataDf.createOrReplaceTempView("data")

  // define ML model
  case class Model() {
    /* define the model (PyTorch style) */
  }

  val model = Model(/*load with pretrained weights*/)

  def classifier(a: Value): FloatValue = {
    val output = model.inference(a.toTensor())
    FloatValue(output)
  }

  spark.udf.register("classifier", classifier)

  val outputDf = spark.sql("select p, sum(classifier(xs)) from r")
  outputDf.show()

}
```

Figure 2.7. Code snippet shows a case where Flern is used to accelerate a Spark query that contains a user-defined ML UDF.

This is enabled by the fact that both systems are written in a modular fashion. Specifically, functionalities are encapsulated in Scala traits as modules and these relatively smaller modules are aggregated using class compositions with mixins [43] to form the larger system. For example, Flare defines `CompileProject`, `CompileJoin`, `CompileAggregate`, etc. traits to handle the operator logic for Project, Join and Aggregate operations. Then these smaller building blocks are combined into `FlareOps`. The design of Lantern follows a similar approach. Therefore, the functionalities of the two systems can be combined by following a similar mechanism. Then, we configure the LMS backend such that a single IR graph is constructed for operations of both workloads.

However, the two systems still use different high-level abstractions and are not aware of the abstractions of the other system (e.g., Flare is not aware of `Tensor`, Lantern is not aware of `Record`, `Buffer`, etc.). Hence, we need to do conversions between these high-level abstractions at the system boundaries. Yet, unlike other systems, a key characteristic of generative programming based code generation is that these high-level abstractions are dissolved into native data structures in the generated code (e.g., `Tensor` and `Buffer` become native C arrays in the generated code). Therefore, even though we do conversions between these abstractions in the high-level code, these conversions can be done in the form of mere variable assignments in the generated code, resulting in zero performance penalty at runtime.

Implementing these conversions is convenient due to the extensible nature of LMS [21, 33, 44]. For example, to convert a `Buffer` to a `Tensor`, we can define a `toTensor` method that creates a new type of node (e.g., `to-tensor`) in the IR. Then, we need to implement the code generation logic for this new node which would generate a simple variable assignment instruction in the generated code. Alternatively, since both `Buffer` and `Tensor` are built on top of the same lower-level abstraction (`Rep[Array[T]]`), the conversion can be done at that level. For example, Figure 2.11 (Line 11) shows how the records are simply converted to a tensor (which is also represented as an array) by doing a simple variable assignment.

Though this approach sounds relatively simple, this simple implementation eliminates data movement and data conversion overheads at system boundaries which is a significant bottleneck in existing tools and a challenging problem as identified in Section 2.1. And most importantly, we preserve the individual optimizations of respective systems, thus, individual best-of-breed performance is maintained. Moreover, since we construct a single IR for the entire program at the LMS level, all the optimizations at the IR level (e.g., Dead Code Elimination (DCE), code motion, loop fusion, etc.) are performed globally. Plus, further **global** optimizations are performed by the downstream general-purpose compilers (GCC or LLVM) when compiling the generated code.

This not just eliminates the overheads at system boundaries but also opens up room for more cross-system optimizations that would further enhance the performance. For example, we can avoid the materialization of large amounts of intermediate data produced by the data processing systems (that gets fed to ML models) by fusing the data processing loops with ML loops. In the following sections, we discuss several such key cross optimizations that improved the end-to-end performance of Flern. A key characteristic of all these implementations is that we can use high-level abstractions such as `Tensor`, `Buffer`, etc. to implement cross-system operations and optimizations rather than relying on compiler analysis on a fixed, generic IR layer (see Figure 2.1).

2.3.2 Optimizing GPU Data Movement

Typically, larger ML workloads are executed in GPUs due to their ability to perform tensor computations efficiently compared to CPUs. Running ML computations in GPU requires the host (i.e., CPU) to transfer the corresponding data (i.e., inputs and weights of the models) to the GPU. In this section, we introduce two optimizations to improve the efficiency of this data movement and show how we incorporated them into Flern.

Any memory allocations done by the host are pageable by default. However, GPUs cannot directly access this pageable memory allocated by the host. Therefore, whenever data needs to be moved from such a pageable host array, the GPU driver first allocates a temporary page-locked (or "pinned") host array. Then, data is copied to this newly allocated pinned array before copying to the GPU memory. In Flern, since we are aware of which data buffers need to be transferred to the GPU, we can eliminate this additional data copying by directly allocating the corresponding data buffers (in data manipulation) in pinned memory (e.g., using `cudaHostAlloc`).

We add this functionality to Flern by adding a new operator that can create a `Buffer` with underlying data (i.e., the C array in the generated code) directly allocated in pinned memory. Then, for any `Buffer` that we invoke `toTensor()` followed by a `toGPU()`, we allocate the original buffer in pinned memory.

Allocation of host arrays in pinned memory makes it possible to move data asynchronously to the GPU. Generally, when two independent systems (or naively integrated systems) are used for data manipulation and machine learning, GPU sits idle until the data processing

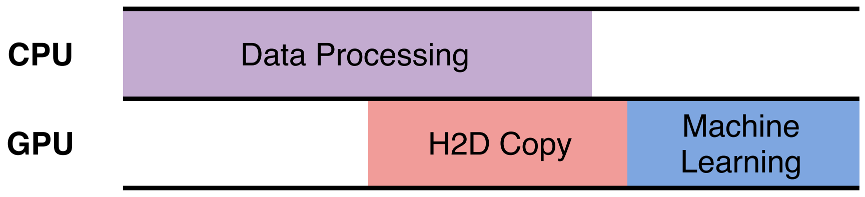


Figure 2.8. Overlapping host (CPU) to device (GPU) data copying with data processing such that batches of necessary records are moved to the GPU as they are produced

phase completes. A chief advantage of generating a single piece of code for the end-to-end task is, we can fuse these host to device data copying operations with the data manipulation operators. In that case, data can be moved to the GPU while the output (from data manipulation) is being produced. Here, we can use asynchronous memory copying mechanisms (e.g., `cudaMemCpyAsync`) to overlap data copying with data processing (see Figure 2.8).

2.3.3 Running ML UDFs Efficiently

Generally, data processing systems do not have support for ML operators natively. Therefore, whenever queries involving such ML computations need to be executed, ML library functionalities are integrated as calls to external systems (e.g., PyTorch or TensorFlow classifier in Spark). However, these external functions are opaque to the data management system, making it difficult or impossible to optimize and add significant overheads in the form of data serialization and invocation overhead (e.g., need to perform computations outside the DBMS runtime) when crossing system boundaries.

With Flern, we have access to a set of fully-fledged ML operations inside the DBMS. Therefore, Flern can run queries involving ML operations as *internal* functions. There are multiple benefits of running ML computations internally instead of relying on external systems. For example, these internal ML UDFs are transparent to optimizations, making it easier to mix the UDF computations with the rest of the computations and perform optimizations when generating code (e.g., inline the UDF invocation, code motion, etc.). Moreover, this opens UDFs for further global optimizations from the downstream compilers

```
def classifierUDF(a: Value) = {
  val inputTensor = a.toTensor()
  // run the pre-trained model
  // (can do any arbitrary tensor computation here
   )
  val output = model(inputTensor)
  output.toValue
}
```

Figure 2.9. An example end-user defined UDF that can be called from SQL. Since we have a tight integration with the ML framework, we can perform any arbitraty ML computations inside the UDF.

(e.g., GCC optimizations). Furthermore, since all execution happens in the same runtime, the overhead of invoking the UDF becomes negligible.

Consider the following simple query with a UDF.

```
SELECT x, classifierUDF(y) FROM r;
```

Figure 2.9 shows how an end-user would define such a UDF in Flern (in Scala). The end-user can write their UDFs as normal Scala functions that use functionalities from both Flare (DB) and Lantern (ML). For example, in the below code snippet, the user defines a UDF that runs a pre-trained model on an input value. The function signature should match the number of arguments we pass to the UDF in the SQL query. Then, whenever the user registers the UDF, it will be added to a map called `udfMap`.

We add this functionality to Flern by defining a new `ScalaUDF` operator that would generate code to invoke UDFs as normal function calls (shown in Figure 2.10). Then, relevant Flare methods are overridden to use the newly introduced UDF operator in cases where UDFs are invoked. This method extracts the corresponding function for the UDF and invokes the function with relevant parameters. The code generation for the function invocation is handled by LMS under the hood. Figure 2.11 shows how the generated code would look like.

With this approach, one of the key impedance for performance is that the query execution model used in Flare (i.e., a modified version of data-centric model [19]) operates on one record at a time. This is not ideal for running relatively large UDFs since the performance

```
override def compilerExpr(/* elided */) = {
  case ScalaUDF(function, dataType, children, ...) =>
    // extract the function from udfMap
    val extractedFunc = udfMap(function)

    // extract the UDF arguments
    // (need to recursively call compilerExpr because the arguments can
   be subqueries)
    val values = children map { compileExpr(_)(rec:_*) }
    // call the function
    extractedFunc(values:_*)
  case _ =>
    super.compilerExpr(/* elided */)
}
```

Figure 2.10. Adding a new `ScalaUDF` operator that retrieves the corresponding user-registered UDF and call it with the correct arguments.

can be dominated by the overheads of kernel launches. Therefore, we have introduced a `VectorizedUDF` operator where the kernels are launched for batches of data instead of single instances, amortizing the kernel launch overhead. The implementation follows a similar approach as above, but rather than invoking the function on single `Record`s of data, batches of data are processed.

Using CPUs for running ML models as UDFs works well for smaller models. However, most modern deep learning models (e.g., a transformer-based sentiment classifier for text data) are too slow to run on just CPUs. Therefore, it is imperative to have support for running these UDFs in GPUs. We can add GPU support by simply running the `VectorizedUDF` kernels using the Lantern GPU backend. However, this hinders the overall performance because the CPU data manipulation gets blocked until we retrieve the output for the current batch. Moreover, each data processing thread would perform GPU kernel launches and data transfers to/from the device independently and concurrently with other (CPU) threads, making both data transferring and execution inefficient. To mitigate this problem, we introduced a new thread pool (having one thread per GPU) that is separate from the data processing

```
/* select p, classifier(xs) from r; */
int main() {
  struct r_record* data = /* load data */
  long record_count = /* data count */

  float *w1 = /* classifier weigths*/
  float *b1 = /* classifier bias*/
  float *w2 = /* classifier weigths*/

  for(long i = 0; i < record_count; i++) {
    float *tensor = data[i]->xs; // conversion
    // ML Computation
    float *y1 = gemm_kernel(tensor, w1, ...);
    for(int j = 0; j < 4; j++) {
      y1[j] += b1[j];
    }
    float *y2 = gemm_kernel(y1, w2, ...);
    if (*y2 > 0.5) {
      printf("%d true", data[i]->p);
    } else {
      printf("%d false", data[i]->p);
    }
  }
}
```

Figure 2.11. Generated code for a simple query that uses an ML classifier (variables renamed, some parts elided to improve clarity). Here, `xs` is an array of data and the classifier computes the output by doing two matrix multiplications. This code looks exactly like a manually written version of the query and performs the combined DB + ML workload in a single unit.

threads (see Figure 2.12). This will retrieve records as they are being produced and then batch all the smaller transfers and kernel invocations together so that overhead is amortized.

2.3.4 Supporting other Popular Front Ends

For the case where we use Flern to accelerate existing end-to-end data science pipelines, it is imperative to support widely used user-friendly front ends [45] (e.g., Pandas, Spark, PyTorch, TensorFlow, etc.). There are existing tools that perform language virtualization

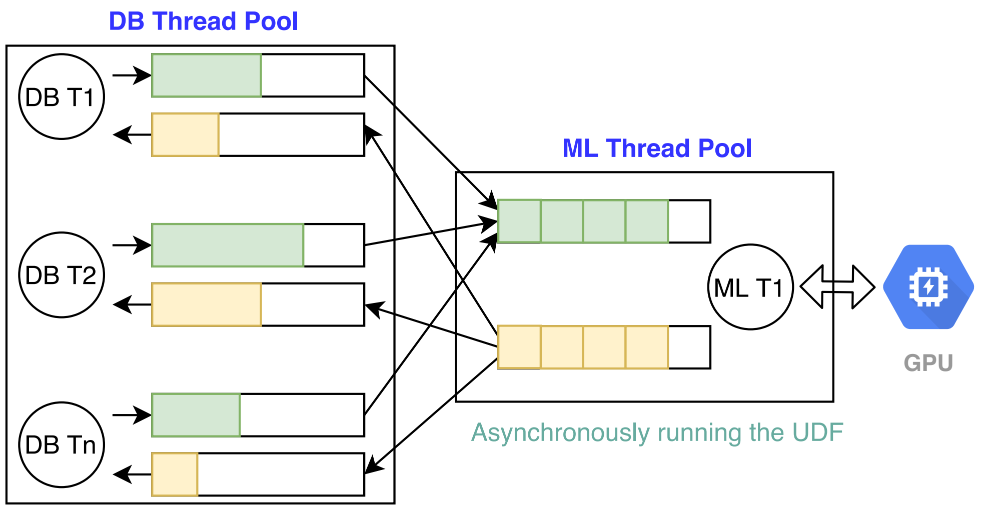


Figure 2.12. Adding a separate thread pool to run ML UDFs on GPU. These threads retrieve records as they are produced, accumulate them and run on the GPU while data processing is ongoing.

and source code transformations for Python. For instance, Snek-LMS [28] implements a multi-stage programming framework similar to LMS for Python. This can be used to convert Python code into an intermediate representation that then can be used to generate code in a different language. Snek-LMS has shown that PyTorch code can be translated into an S-Expression intermediate representation that gets translated into Lantern with minimal modifications (e.g., adding annotations) to the original code.

We can extend the same idea for Flare and build a translator that generates Flare code from a Python source. However, a better alternative approach is to use the existing front ends of Spark. Spark currently supports multiple front-end APIs such as PySpark and Koalas (a Pandas-like interface) which under the hood relies on the Catalyst optimizer to build optimized query plans and core Spark backend (Scala) for query execution. Since Flare only relies on optimized query plans built by Spark (see Figure 2.6), we can extract the query plans generated by these high-level front ends and use Flare runtime for execution. With Lantern and Flare having the support for popular Python ML and data manipulation APIs respectively, Flern can accelerate end-to-end data science pipelines written using high-level user-friendly APIs with minimal modifications to the original code.

2.4 Experimental Evaluation

We evaluate the performance of Flern in two phases. First, we analyze the impact of the main global optimizations we implemented in Section 2.3 such as minimizing data movement overheads, efficiently running UDFs, and overlapping ML computations (e.g., GPU) with data manipulations. Second, we compare the performance with existing systems. One of the key differentiators of Flern compared to existing work is its ability to have competitive performance with best-of-breed systems of individual workloads. Therefore, we first evaluate the individual performance of Flern using a set of representative data manipulation and machine learning benchmarks. In this analysis, we compare the performance with the state-of-the-art systems for the specific workloads.

Subsequently, we analyze the performance of the combined tasks. This includes relational queries that combine ML computations and end-to-end data science pipelines where data manipulation is followed by ML. Here, we show that trivial integrations between workload-specific libraries (e.g., Spark for data manipulation, PyTorch for deep learning) introduces additional overheads and precludes potential global optimizations, making their end-to-end performance suboptimal. Our main goal of the experiments section is to demonstrate Flern's ability to perform both types of workloads comprehensively and the ability to achieve state-of-the-art performance in all cases.

2.4.1 Benchmark Environment

For CPU only workloads, we used a NUMA machine with 4 sockets, 24 Intel(R) Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total). For GPU enabled workloads, we conducted experiments on a NUMA machine with 2 sockets, 12 Intel(R) Xeon(R) Gold 6126 cores per socket, and 100GB RAM per socket (200 GB total), a GPU cluster with four NVIDIA GeForce GTX 1080 Ti 11 GB GPUs. Both servers ran Ubuntu 18.04.4 LTS as the OS.

We use the following versions of libraries/frameworks in the experiments (and the respective versions). Python 3.7.6, Pandas 1.0.3, PyTorch 1.5.0, TensorFlow 2.2.0, NumPy 1.18.1, Weld 0.4.0, Dask 2.15.0, CuDF 0.14, Spark 2.4.5, Postgres 12.8, and GCC 7.5.0.

Each reported result is an average of five runs after removing the lowest and highest value. Unless otherwise specified, all the tools are running in multi-threaded mode with the default number of threads which in most cases equal to the number of logical cores in the system.

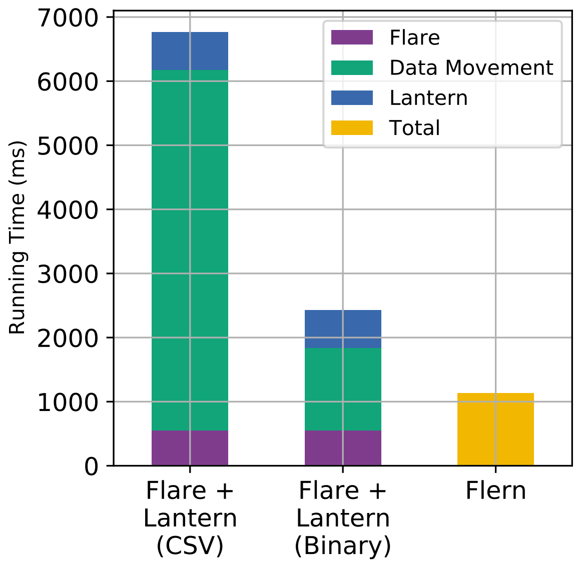
2.4.2 Effects of Key Optimizations

In this section we evaluate the performance impact of key optimization steps we implemented. We used the NYC-Taxi dataset from Kaggle[3] where ML models are trained and used to predict the duration of taxi trips given some information (e.g., pickup and drop off locations, time of the day, etc.) about the trip. The ML model we used for this task is a three layer fully connected neural network with ReLu activations [46] between each layer.

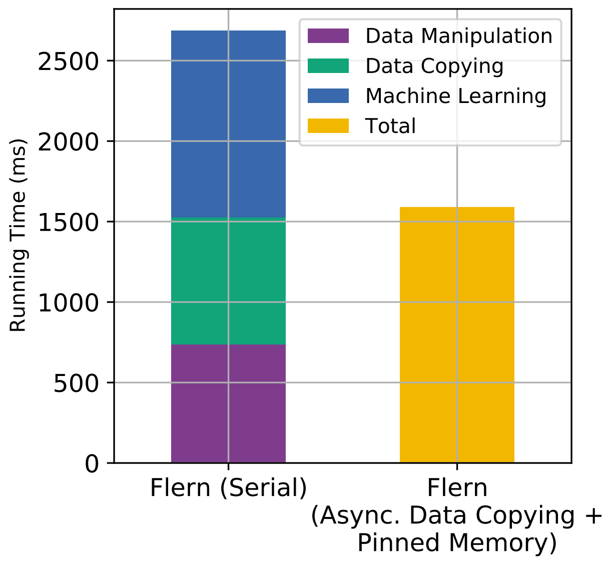
First, we evaluate the impact of minimizing the data movement overhead at system boundaries. Figure 2.13a shows the performance for three cases. One common approach to combine systems is to dump the output from one system and read it from the other system. Such a naive integration between Flare and Lantern using a standard data format (in this case CSV) incurs a significant data movement overhead accounting for more than 80% of the time. This is because, such formats are not optimized for data transferring, and hence, both the writer and reader need to spend time formatting and parsing data respectively. An improvement over this approach would be to use a binary format that is accepted by both systems, speeding up the data movement time notably (around 4x). In Flern, since high-level abstractions (e.g., Tensors, FlatBuffer) dissolves into native data structures (e.g., C Arrays) in the generated code, we can bring the two systems into the same memory address space and use shared data buffers to minimize any unnecessary data copying. This essentially eliminates the data movement overhead between the systems, achieving more than a 6x speedup over the naive integration.

In the next experiment, we evaluate the impact of improving the data movement from CPU memory to GPU memory. Figure 2.13b shows the results for this experiment. The first bar corresponds to the serial case where the data manipulation portion of the workload is completed first and then the produced data is copied to the GPU afterward. In this par-

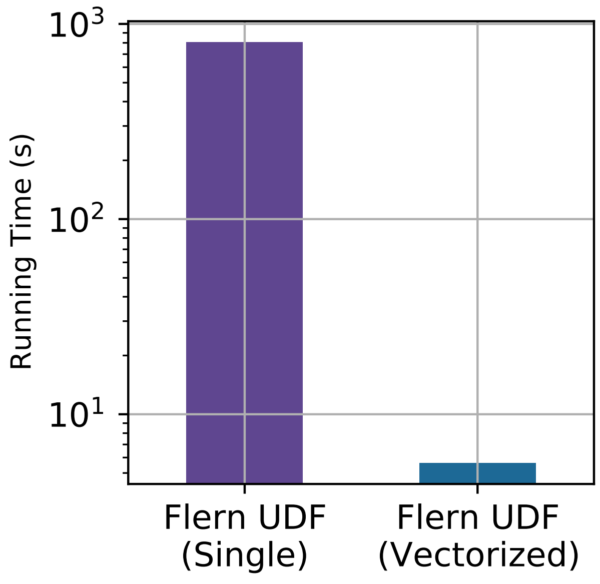
[3]↑https://www.kaggle.com/c/nyc-taxi-trip-duration



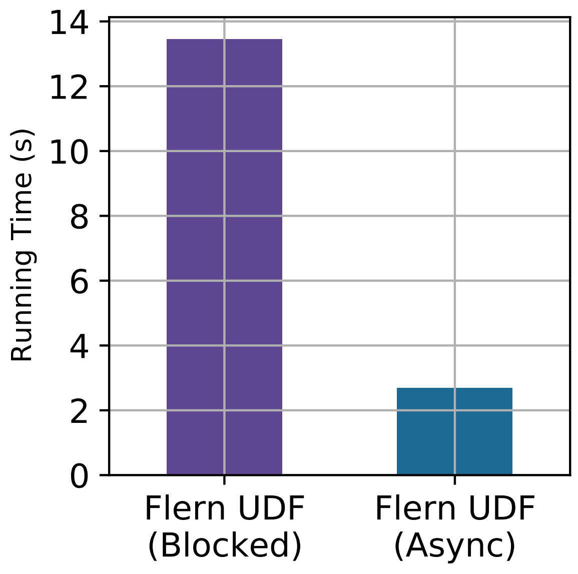
(a)



(b)



(c)



(d)

Figure 2.13. Performance impact of global optimizations; (a) using shared data buffers to minimize data duplication (CPU), (b) using pinned memory and performing asynchronous data movement to GPU (c) vectorizing the UDFs (CPU) (d) running the UDFs asynchronous (in GPU)

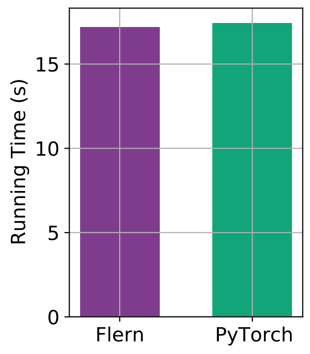
ticular experiment, this data movement takes almost 30% of the end-to-end running time. In the optimized implementation, data is moved (in batches) as the records are being produced rather than waiting for the entire data manipulation phase to complete. This means, data movement is overlapped with the data manipulation computations (see Figure 2.8). Moreover, additional data copying is eliminated by allocating relevant buffers in page-locked (i.e., pinned) memory (as discussed in Section 2.3.2). This overlapped memory transfers and efficient copying reduces the end-to-end running time by 40%.

The next set of experiments evaluate the performance of running ML models (or any tensor computations) as internal UDFs in Flern. Figure 2.13c shows the performance gains of running UDFs in a vectorized manner as opposed to running them in an instance by instance manner. The execution time for the vectorized version is more than 140x faster than the serial version. This can be attributed to the fact that the kernel (in this case, cblas kernels) launch overheads are amortized over the records in the batch in the vectorized case whereas in the serial case, each record would incur that cost.

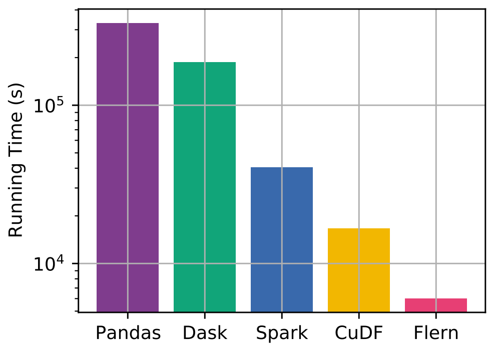
Figure 2.13d shows the execution time for running the same experiment on GPUs. We omit the results for the single instance version because the performance is significantly worse than the vectorized versions. The first bar in Figure 2.13d corresponds to the case where the UDF call is blocking and invoked by each data processing thread independently. That is, each thread invokes data transfers (between CPU and GPU) and all the kernel invocations separately. This leads to a relatively larger number of smaller data transfers and kernel invocations (due to limited GPU memory) adding up launching overheads. To mitigate this, Flern adds a separate worker thread pool (see Figure 2.12) where all the data movement and kernel invocations are handled collectively. In this setting, these separate ML threads can move data and invoke kernels for large batches of data, minimizing the total kernel launch overheads. This optimization shows a speedup of 5x over the previous case.

2.4.3 Evaluating Individual Workloads

In this section, we evaluate the performance of Flern in individual workloads. As we discussed in Section 2.3, Flern preserves all the optimizations of the individual systems. The goal of the experiments in this section is to ensure that the best-of-breed performance of individual workloads is maintained in the absence of interactions with other systems. In contrast,



(a)



(b)

Figure 2.14. Performance comparison for individual workloads (a) ML (b) DB. (a) Per epoch execution time for Transformer model [12] on WMT'14 Machine Translation dataset [47] (b) Running Time (s) for executing multiple (nested) join queries, followed by a *group by* and an *aggregation* (vertical axis is in log-scale).

prior approaches that build common IRs across libraries fail to perform competitively for individual workloads.

Data Manipulation

We evaluate the performance of data manipulation using the Favorita Dataset from Kaggle [4]. The dataset consists of six tables representing information about sales of a retail store. We perform a natural join across the six tables, followed by a *group by* and an *aggregation* query. This benchmark covers a representative data manipulation workload since we perform joins across large relations and performing aggregated queries on grouped relations.

Figure 2.14b shows the results for this benchmark. Pandas is the go-to framework for many data scientists [45] due to its popularity and simple API. However, when used in medium-scale workloads (>1GB), the performance of Pandas becomes relatively poor mainly due to lack of multi-threaded execution, taking the longest time to complete the data manipulation query. Dask accelerates Pandas by operating on chunks of Pandas DataFrames

[4]↑https://www.kaggle.com/c/favorita-grocery-sales-forecasting

and executing them in parallel [48]. Though we see a performance gain over Pandas, the execution time is still orders of magnitude higher than the state-of-the-art systems. We note that Dask supports cluster execution and might see better speedups over Pandas in such settings. Generally, these two approaches are limited by the fact that they are doing most of their computations on the Python runtime.

Spark is the de facto standard for big data processing. However, Spark is known to have sub-optimal performance when used in medium-sized workloads [20] yielding lower performance compared to CuDF and Flern. CuDF performs the entire computation on GPUs with minimal CPU intervention. We note that we used Dask-CuDF to operate on chunks of the dataset as operating on the full dataset is not possible due to limited GPU memory. Dask-CuDF utilizes Dask's task-scheduling to split the task into several smaller sub-tasks that operate on portions of the data. We can observe that CuDF performs better than Spark, however, has higher running time compared to Flern. This performance gap can be caused by the data spilling (from GPU to CPU) occur due to limited GPU memory and general differences of internal algorithms used in different platforms (e.g., Hash Join vs Sort-merge Join). Comparing the performance of Spark and Flern, we can see that Flern's query compilation approach accelerates Spark workloads significantly. Flern achieves around 6x speedup (compared to Spark) for the larger workloads. The performance gain can be attributed to the limitations of Spark's query compilation strategy (e.g., granularity of code generation) as discussed in Section 2.3.

Machine Learning

In the previous section, we saw Flern significantly outperforms most of the widely used libraries and systems in the data science community for data processing. In this section, we evaluate the performance of Flern when running state-of-the-art deep learning models. Specifically, we trained a transformer based machine translation model [12] for WMT'14 Multimodal Translation dataset [47]. The model uses 6 encoder and decoder layers, embedding size of 256, and similar configurations to the original Transformer-base model [12] and parameters are updated using Adagrad optimizer.

Figure 2.14a shows the per iteration running time for Flern compared against PyTorch. The performance is almost equivalent due to the similarity of the internal CUDA kernels used in both systems. A key result of this experiment is to show that Flern can run these state-of-the-art deep learning models with competitive performance which prior work that build common intermediate layer integrations [16, 49] has not demonstrated the ability to support. Specifically, in their approaches, it is challenging to capture composite computations from a generic IR (e.g., Multi-head Attention) to map to off-the-shelf kernels (e.g., CuDNN `multiHeadAttn`).

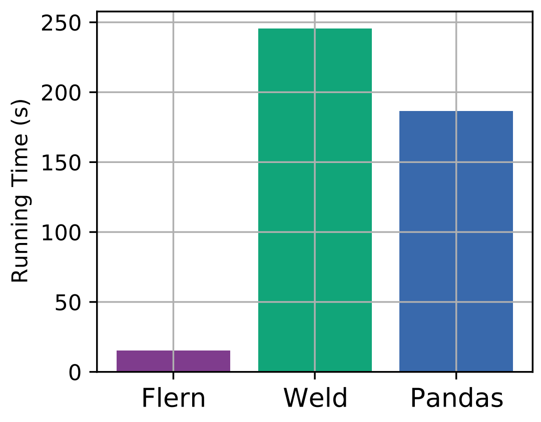
2.4.4 Evaluating Combined Workloads

To assess the performance of Flern in combined workloads, we present three sets of experiments. First, we run an enhanced UDF where a tensor computation is used inside the UDF. Then, a second set of experiments where a complete ML model is used as a UDF and run in CPU and GPU. Finally, an end-to-end pipeline where an ML model is trained using the data queried via a relational query.

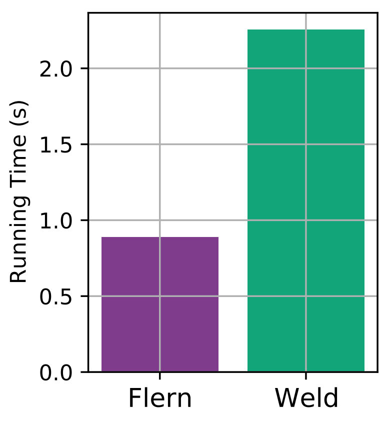
Enhanced UDFs

This section evaluates the performance of Flern compared to Weld which is a similar system at a superficial level. We chose a benchmark from one of the openly available Weld implementations where a scalar value (crime index) is computed by performing a dot product of a selected set of columns (crime-related data) with a predefined constant vector (openly available Weld implementation does not support the workloads in Section 2.4.4 and 5). This benchmark demonstrates a use case where multiple libraries are used jointly. Specifically, in the case of Weld, the data manipulation portion is handled by Grizzly (a Weld-based Pandas accelerator), and the dot product is handled by Weld-NumPy. Similarly, Flern uses Lantern to handle the dot product.

Figure 2.15a shows the end-to-end execution time including the time for data loading, code generation, and compilation (for Weld and Flern). Most notably, the end-to-end time for Weld is higher than Pandas. This is caused mainly by the fact that Weld relies on Pandas



(a) End-to-end time



(b) Execution time of the generated code

Figure 2.15. Execution time (s) for performing a dot product as UDF

to load the data and the time spent on marshaling (encoding) and demarshaling (decoding) of data between the host language and the Weld runtime. Figure 2.15b shows the actual time taken for running the generated code in Weld and Flern excluding data loading, and any other communication overheads. The generated Weld code has gone through multiple passes of optimization including loop fusion, vectorization, loop unrolling, etc. [29] and LLVM level optimizations. Similarly, generated Flern code has gone through optimizations at multiple levels (domain specific optimizations, code specialization, low-level compiler optimizations, etc.). The performance of the resultant Flern code has a speedup of 2.5x compared to Weld. This implies that, though both systems build a common IR across systems and generate low-level code, the approach taken for architecting such IRs has implications on the final performance.

Running ML Classifiers as UDFs

In this section, we evaluate the performance of running ML classifiers as UDFs in relational queries. We compare the performance of Flern with Spark and Postgres with PyTorch used as the ML framework for both cases. In the case of Spark, it uses Apache Arrow [13] which is a common in-memory columnar data format to transfer data between JVM and

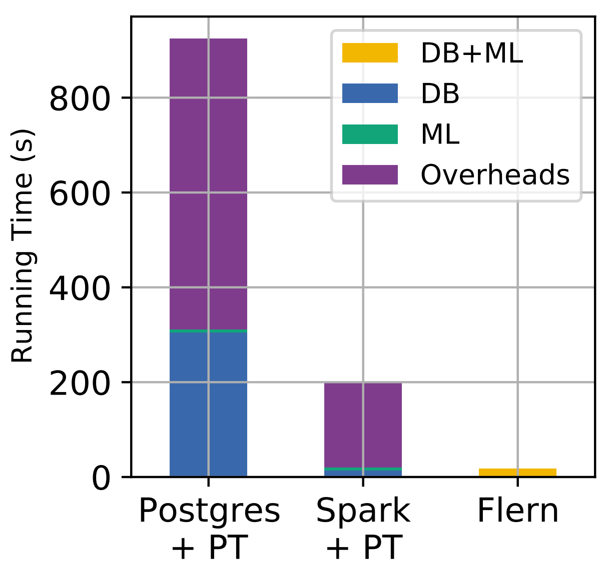
```sql
SELECT
  stars,
  SUM(CASE WHEN sentiment < 0.5 THEN 1 ELSE 0 END) AS negative,
  SUM(CASE WHEN sentiment >= 0.5 THEN 1 ELSE 0 END) AS positive
FROM
  (SELECT stars, classifier(review) AS sentiment
   FROM review NATURAL JOIN business)
GROUP BY stars
```

Figure 2.16. An SQL query that uses a pretrained Transformer model to predict the sentiment of business reviews and summarize according to stars.

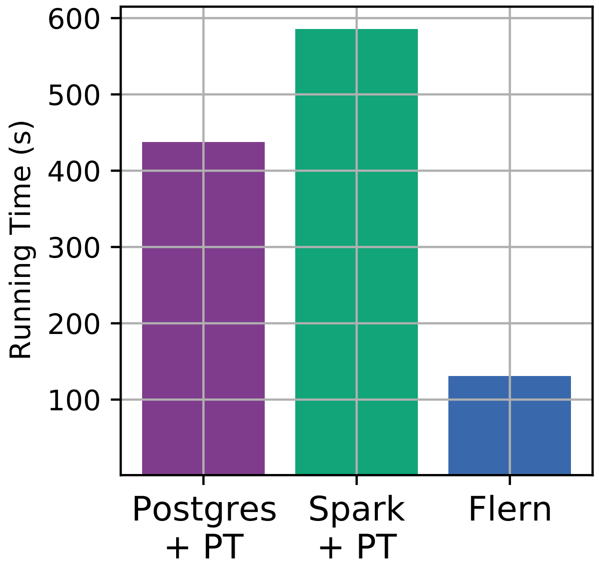
Python environments. Arrow reduces the cost of data serialization and movement, reducing the overhead of communication across the system boundary. This approach is significantly faster than the usual way of running the UDF as an external function on a per-record basis and serializing/deserializing at system boundaries (we do not report the numbers for this case). For Postres, we used PL/Python plugin and implemented the ML operations using PyTorch. In this experiment, we use the same dataset and model as Section 2.4.2 for the CPU case and run a Transformer based sentiment classifier on Yelp Reviews Dataset [5] (extracted 1 million reviews with their corresponding business information) for the GPU case (query shown in Figure 2.16). Specifically, for the sentiment analysis case, we compute the aggregated number of positive and negative reviews based on the star rating of restaurants (query shown below). The sentiment classifier consists of word and positional embeddings [12], 6 encoder layers, followed by a three-layer fully connected layer with ReLu [46] activations.

Figure 2.17a shows execution time for the CPU case where Flern demonstrates speedups over 50x against Postgres and around 12x over Spark. We did several follow-up micro-benchmarks to figure out the reason behind this large speedup. For the Postgres+PT and Spark+PT case, we ran the corresponding workloads on individual systems (i.e., Postgres, Spark and PyTorch separately) to observe the magnitude of the overheads added due to the integration (Figure 2.17a highlights the portion of time spent for actual computation). These overheads mainly come in the form of data copying and format conversion overheads that

[5]↑https://www.yelp.com/dataset



(a)



(b)

Figure 2.17. Performance comparison for (a) Running a 3-layer Neural Network (Regression) UDF on CPU (b) Running a Transformer-based Sentiment Classifier UDF on GPU

occur at system boundaries. Specifically, for Spark, this includes the cost of converting Spark DataFrames to Arrow, moving data from one execution environment to another, converting Arrow data to Pandas DataFrames, etc. Similar data copying occurs in the Postgres case. Such overheads accounts for more than 90% of the total execution time in both cases. Since Flern generates a single piece of low-level code with the UDFs inlined and fused with the data processing loops, none of these overheads exists. Moreover, these UDFs are transparent to the compiler and go through further global optimizations.

Figure 2.17b shows the execution time for the GPU case. Interestingly, Postgres implementation runs faster than Spark. This can be attributed to the fact that there is no coordination between the Spark workers when performing computations on the GPU. That is, each worker sends computations to the GPU independently which leads to a case where each worker launches kernels for smaller batches of data (because of limited amount of GPU memory) simultaneously. This leads to more kernel launches, more (smaller) data move-ments between host and device, hence, more launch overheads which becomes dominant. As we also saw in our micro-benchmarks (in Section 2.4.2), this lack of coordination can significantly hinder performance. In the case of Postgres, a single PyTorch worker is spawned by the DB that handles all the computations in the GPU, minimizing such kernel launch overheads. As discussed in Section 2.3.2 (Figure 2.12), Flern has a separate thread pool that batches ML computations from data processing workers which minimizes the kernel launch overheads. This, and the absence of system crossing overheads, makes Flern significantly faster than the other two systems.

End-to-end Training

This section evaluates the performance for training an ML model on data extracted by a relational query. We chose the Favorita dataset from Kaggle (same as Section 2.4.3) where the task is to train a model to accurately predict the number of sales for a large grocery chain. The same task has been used as an evaluation benchmark in prior related studies as well [10, 50]. The dataset consists of multiple relations (tables) that include related information such as daily oil price, information about items (e.g., whether the item was on promotion), history of transactions, etc. Similar to [50], we first perform a natural join between all the relations and select all the numerical columns. Then we train a neural network, composed of three layers with ReLU [46] activations in the first two layers. We compute the Mean Squared Error (MSE) loss, compute the gradients with respect to loss, and use Stochastic Gradient Descent (SGD) to optimize the model parameters. We run this benchmark on a set of most widely used libraries and systems in the data science community.

Figure 2.18 summarizes the result for this benchmark. The performance of data manipulation follows a similar pattern to that of Section 2.4.3 and can be attributed to the same limitations discussed. CuDF failed to complete the experiment due to suffering from insufficient GPU memory. Since Flare performs runtime native code generation and executes the generated code, there is no *direct* way of integrating Flare with a library like PyTorch. Hence, data movement becomes a bottleneck in that scenario. Flern, which builds a compiled path for the end-to-end data science pipeline execution, performs best compared to all

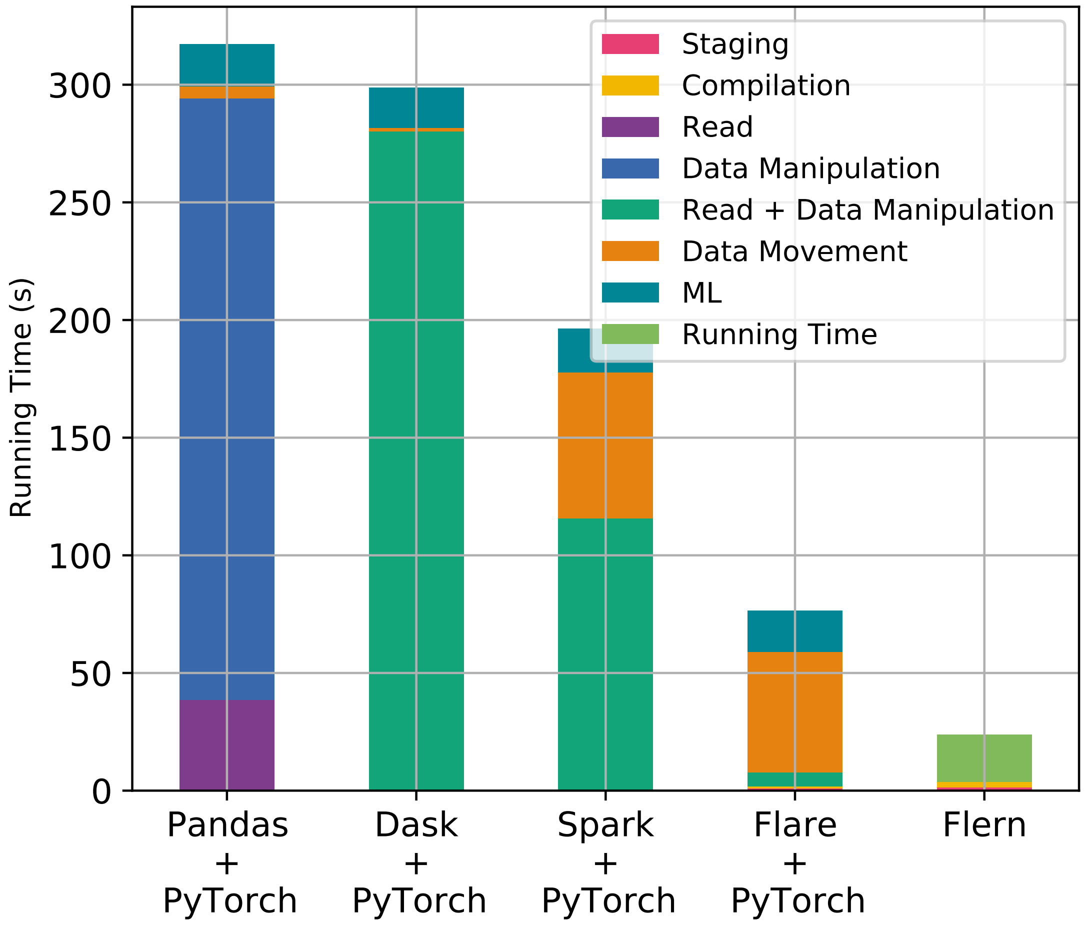


Figure 2.18. Running time (ms) for an end-to-end data science workload consisting of querying a relational source and training a machine learning model. `Flern` achieves a speed up of more than 13x and 3.2x over `Pandas + PyTorch` and `Flare + PyTorch` respectively while supporting user friendly APIs similar to `Pandas + PyTorch`

the other approaches. Flern achieves a speedup of over 13x compared to the initial Pandas version and achieves around 3.2x speedup over a naive integration between Flare and PyTorch. This can be attributed to the zero-cost data sharing between the two systems and the global optimizations enabled from having a single IR and a single generated code (evaluated in Section 2.4.2).

2.5 Related Work

Weld [16, 29] is a common runtime specifically targeting data-intensive applications with a main focus on physical data movement optimizations. Weld provides a common runtime for multiple libraries (e.g., Pandas, NumPy) by exposing an API to build the IR (Weld API). Delite [18, 49] is a compiler framework for implementing embedded domain-specific languages (DSLs). The Distributed Multiloop Language (DMLL) [51] performs cross optimizations on data processing and machine learning operations. Although both systems achieve good performance in combined workloads, their performance in individual workloads is below state-of-the-art or they do not provide enough functionality to run full benchmark suites like TPC-H [25]. Moreover, they require building individual systems from scratch or need significant modifications to existing systems. Split Annotations [52] identifies these limitations and presents an approach that can combine existing systems without modification. Specifically, they treat library functions as black boxes and adds a cross-function data pipelining layer to minimize data movement overheads. However, this approach sacrifices the benefits of building common IRs and performing code generation (e.g., cross-system operator fusion). On the other hand, our approach requires no re-implementation of existing operators and can introduce cross-system optimizations, attaining the best of both worlds.

There are several works that incorporate ML computations into DB systems to avoid expensive data movement [6–8, 11, 53]. MADLib [11] and Bismarck [8] define ML computations as UDFs inside the DBMS. Factorized ML approaches perform ML computations on multi-table data by pushing ML operations down through to the normalized relations, eliminating the need to materialize large join results [9, 10, 54–59]. However, these approaches are algorithm specific and does not support the full spectrum of modern ML models (e.g., deep learning).

Specialized data manipulation frameworks such as Pandas [42], Pandas accelerators (e.g., Modin [60], Dask [48]), Spark [34, 40], HyPer [24], Flare [20], etc. have to rely on external systems to handle ML workloads. TensorFlow [14] and PyTorch [15] are by far the most widely used deep learning frameworks. TensorFlow has its data loading API (`tf.data`) which builds efficient data preprocessing pipelines. However, that lacks full SQL/DataFrame like (e.g., Joins, Aggregate queries) operations. Spark MLlib [61] and FlinkML are machine learning libraries specifically designed for Spark and Flink [62] respectively. They utilize the data manipulation capabilities of Spark and Flink, however, they do not provide support for deep learning. Therefore, the need for combining specialized systems for data processing and machine learning remains essential.

2.6 Conclusions

In this chapter, we presented Flern, the first intermediate layer that achieves both best-of-breed performances on individual tasks and best-of-breed performance on combined workloads. Flern tackles a pressing problem in the DBMS community: incorporating full ML capabilities into DBMS efficiently. We demonstrated that our approach based on generative programming to construct common intermediate layers enables efficient post-hoc integration between existing systems without a significant re-engineering cost. We believe this chapter provides a principled approach to attacking this problem in general: architect systems based on generative programming so that they can be adapted more effectively and at a lower engineering cost.

3. EFFICIENT INCREMENTIALIZATION OF CORRELATED NESTED AGGREGATE QUERIES USING RELATIVE PARTIAL AGGREGATE INDEXES (RPAI)

Portions of this chapter have appeared in *Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI), in Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12-17, 2022, Philadelphia, PA, USA* [63].

3.1 Introduction

Many real-world systems perform analytical queries on continuously arriving streams of data that result in dynamic datasets with high update rates. These types of workloads are prevalent in many modern big data application domains including finance, social media, etc., and generally focus on extracting insights, trends, and anomalies from data streams [64]. For example, in algorithmic trading, such queries are performed on fast arriving streams of data to compute key metrics that drive trading decisions [64].

This is generally done in the form of incremental processing where, given a query Q, a database db, the task is to efficiently compute $Q(db + \Delta db)$ under updates Δdb using $Q(db)$ (i.e., the previous result) and any additionally maintained auxiliary data structures. Incremental processing is not just useful for processing streams of data, but also in other use cases such as, producing low-latency approximate results for batch queries by operating on mini-batches of data [65], and eager processing of queries (i.e., before all the data is ready) to produce faster results with limited resources [66], intermittent query processing [67], etc.

Traditional relational databases (DBMS) perform this in the form of Incremental View Maintenance (IVM) when maintaining materialized views. Specifically, users can materialize query outputs (e.g., for faster retrieval) and the DBMS engine maintains these views under updates to the underlying data sources by incrementally updating the view accordingly (via *delta queries*). This is generally faster compared to evaluating the query from scratch on the updated database. Although these traditional IVM techniques handle simple queries well, they fall back to re-computation for complex queries with nested and correlated subqueries [1].

DBToaster [1, 64] is a state-of-the-art incremental query evaluation system that takes this notion of delta queries and applies it recursively (i.e., delta of delta queries and so on) so that the views are maintained by not just a single materialized structure, but a set of auxiliary views corresponding to each level of the recursive delta query. This notion of higher-order delta queries works well in practice and has shown significant performance gains over commercial DBMS and Stream Processing systems [1].

However, this recursive approach is only effective for a class of queries that is guaranteed to have delta queries simpler than the higher-level query (further discussed in Section 3.3.1). Importantly, queries with nested aggregates do not satisfy this key property. Specifically, the delta of such queries is simply running the query twice and computing $Q(db + \Delta db) - Q(db)$ which is worse than simply re-evaluating the query [68]. Therefore, systems like DBToaster fall back to re-computing for such queries. In Section 3.3.2, we show that DBToaster takes $O(n^2)$ time to maintain such a query under updates which is the same asymptotic time as re-computation whereas we demonstrate the same can be done in $O(\log n)$ time by using an additional index to maintain partial aggregates.

Follow-up work on DBToaster [68] presented a technique called *domain extraction* for specifically optimizing this re-evaluation strategy by narrowing down the iteration space. However, this only works for cases where the nested aggregates are correlated to the outer query on equality predicates. Even for the queries that use this technique, we show in Section 3.3.1 that this approach yields sub-optimal asymptotic performance (e.g., takes $O(n)$ where $O(1)$ is possible). Another line of related work focuses on building specialized data structures and algorithms to efficiently incrementalize certain classes of queries (e.g., acyclic conjunctive queries (CQ) with equalities [69] or inequalities [70], acyclic foreign key joins [71] etc.). However, the presented specialized data structures either do not support or do not efficiently handle nested aggregate queries.

In this work, we focus on improving the incrementalization efficiency of aggregate queries containing nested aggregate subqueries in their predicates. First, we introduce a general algorithm that is based on the idea of identifying and updating only the aggregate values affected by a tuple insertion (or deletion). This is done by analyzing the correlated columns of inner queries and the corresponding uncorrelated columns used in the inner query predicates. Second, we focus on a specific subset of queries that appear commonly in practice (especially in finance-related use cases) and propose a way to further optimize them. Specifically, we build additional index structures that are indexed by aggregate values such that a range of aggregate values can be shifted efficiently, providing the ability to compute the final result directly from these indexes. We observe that none of the existing data structures support the required operations in a reasonable time, and hence, design a novel tree-based data structure called relative partial aggregate index (RPAI) that stores aggregate values (i.e., keys) in a parent-relative manner to enable range key shifting in logarithmic time. Our analysis shows that the use of these aggregate indexes result in significant asymptotic speedups (e.g., $O(n^2)$ to $O(\log n)$) and up to 1100× speedups in workloads of practical importance.

3.2 Contributions

- We present a case study with two examples of nested aggregate queries and analyze how existing approaches handle those queries. Then, we motivate our approach of using aggregate indexes (PAI Maps and RPAI Trees) and demonstrate how such structures can improve the incrementalization (Section 3.3).

- We design an efficient tree-based data structure for RPAI and present algorithms for the key operators with an analysis of their time complexity (Section 3.4).

- We present a novel general algorithm for incrementalizing correlated nested aggregate queries, followed by further optimizations using PAI Maps and RPAI Trees. Then, we discuss the limitations and overheads associated with our approach. (Section 3.5).

- We evaluate the performance of our algorithm against DBToaster, a state-of-the-art system that supports incremental execution of SQL queries, and show that our approach performs significantly better in real-world datasets in line with the expected performance behaviors due to asymptotic speedups (Section 3.6).

3.3 Incrementalization of Nested Aggregates

In this section, we consider two example nested aggregate queries of different forms and explore how existing systems and approaches handle these queries. Then, we identify the key limitations of these approaches, which motives the ideas presented in this work. Specifically, we demonstrate how our approach can improve the efficiency of incrementalization of these queries.

3.3.1 Nested Aggregate with Equality

Consider the query in Figure 3.1 that contains two nested-aggregate subqueries predicated on equality conditions. This computes an aggregate of values that is responsible for a given fraction ($\frac{1}{2}$ in this case) of all the records. Queries of this structure are commonly found in many use cases [64, 72].

```
Q = SELECT Sum(r.A * r.B) FROM R r
WHERE                           ┌─lhs_sum
  0.5 * (SELECT Sum(r1.B) FROM R r1) =    ┌─rhs_sum
      (SELECT Sum(r2.B) FROM R r2 WHERE r2.A = r.A)
```

Figure 3.1. Query containg nested-aggregate subqueries predicated on only equality conditions.

The query operates on a single relation $R(A, B)$ and consists of two nested aggregate queries. The inner query on the left-hand side is not correlated with the outer query whereas the right-hand sub-query is correlated implying that the corresponding aggregate value varies for different records from the outer query.

Figure 3.2, 3.3 and 3.4 shows the code corresponding to different execution strategies for the query in Figure 3.1.

```
R = []
def on_new_R(t: R):
  R.append(t)
  # re-evaluate from scratch
  res = 0.0
  for r in R:
    # evaluate lhs nested aggregate   (uncorrelated)
    lhs_sum = 0.0
    for r1 in R:
      lhs_sum += r1.B
    lhs_sum *= 0.5
    # evaluate rhs nested aggregate   (correlatead)
    rhs_sum = 0.0
    for r2 in R:
      if r2.A = r.A:
        rhs_sum += r2.B
    # evaluate the predicate and update aggregate
    if (lhs_sum == rhs_sum):
      res += r.A * r.B
```

Figure 3.2. Naive Re-evaluation strategy for query in Figure 3.1. Takes $O(|R|^2)$ time (Section 3.3.1)

Naive Re-evaluation Strategy

Figure 3.2 shows a naive re-evaluation computation of the given query in Python. The code is self-explanatory and follows the same structure as the query (i.e., nested loops for the subqueries). Notice that this computation happens every time $R(A, B)$ gets updated (i.e., tuple insertions, or deletions). That is, when updates to R is ΔR, $Q(R + \Delta R)$ is evaluated from scratch irrespective of the fact that $Q(R)$ was previously computed. Due to the presence of nested loops, overall asymptotic time complexity of producing the final result upon updates to R comes down to $O(|R|^2)$ where $|R|$ is the number of tuples in R.

```
# materialized views
map1 = {}  # A -> sum(A * B)
map2 = 0.0 #   -> sum(B)
map3 = {}  # A -> sum(B)

def on_new_R(t: R):
  res = 0.0
  # update the maps
  map1[t.A] += t.A * t.B * t.X
  map2 += t.B * t.X
  map3[t.A] += t.B * t.X

  # compute lhs_sum
  lhs_sum = 0.5 * map2
  # find each rhs_sum
  for a in map1:
    rhs_sum = map3(a)
    # evaluate the predicate and update aggregate
    if lhs_sum == rhs_sum:
      res += map1(a)
  return res
```

Figure 3.3. Code geberated by DBToaster for the query in Figure 3.1. Some portion of the query is incrementalized over the naive approach in Figure 3.2. Takes $O(|R|)$ time (Section 3.3.1)

DBToaster Incremental Query Execution

As discussed in Section 3.1, DBToaster falls back to recomputation due to the lack of an efficient delta query for queries containing aggregate subqueries in the predicates. Figure 3.3 shows the code generated by DBToaster for the query in Example 3.1. We converted the generated C++ code to Python to improve readability and combined the tuple addition and deletion triggers into a single (equivalent) trigger to make the code more concise ($t.X = 1$ for insertions and $t.X = -1$ for deletions).

The code shows that even though it relies on re-evaluation for connecting the nested aggregate with the outer query, other parts of the query are efficiently incrementalized (compared to naively re-evaluating). For instance, the `lhs_sum` computation which required

```
# materialized views
map1 = {}  # A -> sum(A * B)
map2 = 0.0 #   -> sum(B)
map3 = {}  # A -> sum(B)

def on_new_R(t: R):
res = 0.0
# update the maps
map1[t.A] += t.A * t.B * t.X
map2 += t.B * t.X
map3[t.A] += t.B * t.X

# compute lhs_sum
lhs_sum = 0.5 * map2
# find each rhs_sum
for a in map1:
  rhs_sum = map3(a)
  # evaluate the predicate and update aggregate
  if lhs_sum == rhs_sum:
    res += map1(a)
return res
```

Figure 3.4. Code generated by our approach based on aggregate indexes for the query in Figure 3.1. This query is fully incremental and only takes $O(1)$ time (assuming (1) hash maps) (Section 3.3.1)

iterating through all the records (line 8-11 in Figure 3.2), is now computed in constant time by incrementally maintaining the sum of B values using `map2`. Since `rhs_sum` depends on the $r.A$ values from the outer query, `map3` maintains the relevant $sum(B)$ values for different $r.A$ values. Here, rather than re-evaluating all the `rhs_sum` values (lines 13-16 in Figure 3.2), only the sum affected by the new tuple is updated. `map1` maintains the $sum(A * B)$ for each A value and is used to compute the final aggregate sum. Specifically, once `lhs_sum` and `rhs_sum` are computed, the set of A values that satisfy the condition are found and the corresponding $sum(A * B)$ values are summed up.

Lines 9-11 in Figure 1b correspond to updating the maps based on the incoming tuple and lines 16-20 show the computation of the final result. The overall time complexity of maintaining the final result under updates to R is $O(|R|)$.

Using Aggregate Indexes

A key observation at this point is that even though `lhs_sum` changes with incoming tuples, the aggregate value remains a fixed value for all the outer tuples as `lhs_sum` is not correlated to the outer query. Therefore, if we have an additional index that maps different `rhs_sum` values to the corresponding final aggregate sums (in this case `SUM(A * B)`), we can query that map using the current (updated) `lhs_sum` to compute the updated final result in constant time. However, populating such an aggregate index naively by computing `rhs_sum` for each tuple (using `map3`) would still leave the overall time complexity at $O(|R|)$.

Now the key question is, rather than populating this aggregate index by iterating and computing `rhs_sum` for all the tuples in R, is it possible to maintain it in less than $O(|R|)$ time. Observe that whenever a new tuple t arrives, three things change. First, the `lhs_sum` gets incremented by $t.B * t.X$ which can be updated in constant time. Second, the new record has a corresponding `rhs_sum` and that needs to be added to the `aggrMap`. This can also be done in constant time by computing the `rhs_sum` using `map3` as we saw before and updating the corresponding entry in `aggrMap`.

Third, the addition of the new tuple to $r2$ (in the right subquery) will trigger a change of `rhs_sum` values of all tuples having the same A value as $t.A$ (because the condition of the nested query predicate is $r.A = r2.A$). Therefore, all the tuples with the same A value will have the same `rhs_sum` because the set of tuples responsible for their respective `rhs_sum` are only the ones having the same A. That is, with the addition of t, the `rhs_sum` corresponding to $t.A$ gets incremented by $t.B * t.X$ and we need to update the `aggrMap` to reflect this change. We can do this by moving the corresponding `sum(A*B)` from the old `rhs_sum` key to `rhs_sum` $+ t.B * t.X$.

However, we cannot simply move the value for `rhs_sum` to `rhs_sum` $+ t.B * t.X$ because there can be the same `rhs_sum` for different A values (i.e., different groups of tuples can have the same aggregate value). We can handle this by using `map3` to move the respective portion of the value from `aggrMap[old_rhs_sum]` (i.e., only $sum(A * B)$ of records having $t.A$ in A) to `aggrMap[new_rhs_sum]`.

We call indexes of type `aggrMap`, Partial Aggregate Indexes (PAI) since they store aggregates values as keys and map to aggregates. Figure 3.4 shows the implementation based on the PAI Map based approach. The entire update routine only contains updates to hash maps and does not require any form of iteration. Therefore, the query in Example 3.1 can be incrementally maintained in $O(1)$ time using our approach which is more efficient than re-evaluation which takes $O(|db|^2)$ time, and DBToaster which takes $O(|db|)$.

3.3.2 Nested Aggregate with Inequalities

In the previous example, we demonstrated how a query that has an $O(|R|^2)$ re-evaluating cost and an $O(|R|)$ incremental maintenance cost (in DBToaster) can be incrementalized in $O(1)$ time by using PAI Maps. The query consisted of just equality predicates, making it possible to use hash maps to maintain the aggregate indexes under updates in constant time. In this section, we consider a real-world query (Figure 3.5) of a similar structure having inequality predicates.

```
SELECT
  Sum(b.price * b.volume)
FROM
  bids b
WHERE
  0.75 * (SELECT Sum(b1.volume) FROM bids b1)
  <
  (SELECT Sum(b2.volume)
          FROM bids b2 WHERE b2.price <= b.price)
```

Figure 3.5. The query computes the volume-weighted average price (VWAP) over bids that is in the final quartile (i.e, more than 75%) of total stock volume [64].

Example 3.5 is a query from the finance benchmark which is used in multiple other related works [1, 64] to evaluate the performance of incrementalization. The query operates on traces of records from order books that contain information about bids and asks for shares (or any other asset) in a financial market. These trades occur at rapid rates making it extremely important to have fast refresh rates as the queries compute key metrics that drive efficient algorithmic trading [64]. Moreover, the transactions in these financial markets often contain updates or retractions of older transactions, requiring the incremental query engines to maintain these queries incrementally under both insertions and deletions.

The *bids* relation consists of five attributes *timestamp, id, broker_id, volume, price*, plus an additional attribute (*bids.X*) to distinguish between record deletion (-1) and insertion (+1). Similar to Example 3.1, this query contains two nested aggregate queries with one of them being correlated to the outer query. The main difference is the use of inequality predicates in the query.

Naive Re-evaluation Strategy

Figure 3.6 shows the code for a naive re-evaluation approach to compute the output of this query. Here, the *bids* is updated as records arrive and the result is computed from scratch by looping through the records while computing `lhs_sum` and `rhs_sum` using nested loops. The overall asymptotic time complexity is $O(|bids|^2)$ where $|bids|$ is the cardinality of the *bids* relation at the given point.

DBToaster Incremental Query Execution

Figure 3.7 shows the code generated by DBToaster for the Example 3.5 query. DBToaster creates a set of maps representing intermediate materialized views to incrementally maintain the query. The two nested subqueries are fully incrementalized using two maps `map2` (for `lhs_sum`) and `map3` (for `rhs_sum`). `map1` maintains the sum of `price * volume` per each `price`. This is useful when computing the final result as we need to accumulate `price * volume` based on a set of `price` that satisfies the given conditions.

Similar to the previous example, DBToaster fails to incrementalize the computation of the final result that requires finding records from the outer query that satisfy the query conditions. Therefore, it falls back to computing the final result by iterating through records

```
 1 bids = []
 2 def on_new_bids(t: record):
 3   # update the base table
 4   bids.addRecord(t)
 5   # re-evaluate from scratch
 6   res = 0.0
 7   for b in bids:
 8     # evaluate the lhs nested aggregate (uncorrelated)
 9     lhs_sum = 0.0
10     for b1 in bids:
11       lhs_sum +=
12               b1.volume * b1.X
13     lhs_sum *= 0.75
14 
15     # evaluate the rhs nested aggregate (correlated)
16     rhs_sum = 0.0
17     for b2 in bids:
18       if b2.price <= b.price:
19         rhs_sum +=
20               b2.volume * b2.X
21 
22     # evaluate the predicate and update the aggregate
23     if (lhs_sum < rhs_sum):
24       res +=
25         b.price * b.volume * b.X
```

Figure 3.6. Naive Re-evaluation, Takes $O(|bids|^2)$ time (Section 3.3.2)

which is essentially similar to the naive re-evaluation strategy in Figure 3.6 (Lines 16-20). Hence, the asymptotic time complexity remains at the same level as naive re-evaluation, at $O(|bids|^2)$.

Using Aggregate Indexes to Fully Incrementalize

To further incrementalize this query, we can follow a similar intuition to the previous example where we introduced the use of PAI Maps to index the final aggregate sum using `rhs_sum` as a key. Assuming such an index can be efficiently maintained, finding the final aggregate can be done in linear time by iterating through `rhs_sum` keys that are greater

```
map1 = {}
  # price -> sum(price*volume)
map2 = 0.0
  # sum(volume)
map3 = {}
  # price -> sum(volume)

def on_new_bids(t: record):
  # update the maps
  map1[a.price] += t.X * t.price
                        * t.volume
  map2 += t.X * t.volume
  map3[a.price] += t.X
                        * t.volume
  #  evaluating correlated nested aggregate
    for each outer distinct price
  res = 0.0
  for b_price in map1:
    rhs_sum = 0.0
    for b2_price in map1:
      if b2_price <= b_price:
        rhs_sum += map3[b2_price]
    # evaluating condition
    if 0.75 * map2 < rhs_sum:
      res += map1[t.price]
  return res
```

Figure 3.7. Code generated by DBToaster. Takes $O(|bids|^2)$ time (partially incremental; Section 3.3.2)

than `lhs_sum` and accumulating the corresponding `sum(price * volume)` values. For that purpose, we can define a `getSum(key)` method for hash maps that finds the sum of values having keys less than or equal to a given key by simply iterating over the keys. The `compute()` method in Figure 3.8 (lines 23-27) shows how to use `getSum` to compute the final result.

Now the question is: can we efficiently maintain the `aggrIndex` under updates? We cannot simply follow the same approach as before to shift the aggregates as the predicate inside

```
aggrIndex = {}
  # <rhs_sum> --> sum(price * volume)
map2 = 0.0
  # sum(volume)
map3 = {}
  # price --> sum(volume)

def on_new_bids(t: record):
  # rhs_sum for new record (before update)
  rhs_sum = getSum(map3, t.price)
  volume = map3[t.price]
  # update the aggregate index
  shiftKeys(aggrIndex, rhs_sum-volume,
                          t.X * t.volume)
  # update the maps
  map3[t.price] += t.volume*t.X
  map2 += t.volume*t.X
  aggrIndex[rhs_sum + t.X * t.volume]
                 += t.X * t.price * t.volume
  #compute the output
  return compute()

def compute():
  lhs_sum = map2 * 0.75
  res = getSum(aggrIndex, inf)
                 - getSum(aggrIndex, lhs_sum)
  return res
```

Figure 3.8. Our approach based on aggregate indexes. Takes $O(|bids|)$ & $O(\log|bids|)$ (fully incremental; Section 3.3.2)

the correlated nested subquery (i.e., `b2.price <= b.price`) is not an equality. Specifically, insertion (or deletion) of a new record t does not only increment (or decrement) the `rhs_sum` of outer records having the same price as `t.price` but also the ones with `price > t.price`. This update cannot be done by simply iterating over the price values since the `aggrIndex` does not store the corresponding price values. For this, we can potentially use an additional index that maps `price` to the corresponding `rhs_sum` value (similar to `map1` in Figure 3.7).

Alternatively, rather than relying on an additional index, we can exploit a key characteristic of `rhs_sum` to do this mapping. Specifically, the `rhs_sum` will grow monotonically in the order of increasing price values since `rhs_sum` for a given `t.price` is the sum of all `volume` value of records having a smaller or equal price compared to `t.price` (based on the correlated predicate). Therefore, when a new record arrives, we can first compute the corresponding `rhs_sum` for that particular record (without considering the new record) and then we know that every `rhs_sum` greater than or equal to that value must be incremented by the new `t.volume`. For this, we can define a method `shiftkeys(key, offset)` that will shift all the keys greater than the given key by the given `offset`.

Figure 3.8 shows the final code for this approach. We first compute the `rhs_sum` for the new record using the `getSum` method (Line 10). Then, we shift all the keys (i.e., aggregate values) that are greater than or equal to `rhs_sum` using `shiftKeys` (Line 13). Since we defined `shiftKeys` to shift the keys that are *strictly* greater than a given key k, we pass the immediate lesser key (i.e., `rhs_sum - volume`) to shift all qualifying keys including `rhs_sum` (Lines 11-13).

Overall, the final code consists of a set of constant time hash map lookups, and linear time `getSum` and `shiftKeys` method calls. Hence, the total time complexity is $O(|bids|)$ which is asymptotically faster than DBToaster and re-evaluation that took $O(|bids|^2)$. In the next section, we design a new tree-based index structure called Relative Partial Aggregate Indexes (RPAI) that can perform `getSum` and `shiftKeys` operators in logarithmic running time, bringing the overall time complexity down to $O(\log|bids|)$.

3.4 Relative Partial Aggregate Indexes (RPAI)

In Section 3.3.2, we saw how the use of PAI Maps that are indexed on aggregate values can improve the incrementalization efficiency of nested aggregate queries. Specifically, for queries with inequality correlated predicates, we identified `getSum` and `shiftKeys` as two main operations and showed that PAI Maps can support those operations in linear time. In this section, we design a tree-based data structure called the Relative Partial Aggregate Index (RPAI) that can support both of these operations in logarithmic time. Our data structure needs to be a map (i.e., maintains key-value pairs with unique keys) and should support `getSum` and `shiftKeys` in addition to the regular map operations (i.e., `get` and `put`).

3.4.1 Optimizing `getSum`

Although hash maps support the regular map operations in constant time, `getSum` and `shiftKeys` require iterating through all the keys in the map. Alternatively, tree-based data structures are known to be efficient for operations similar to `getSum`. For instance, Segment Trees [73] and Binary Indexed Trees (Fenwick Trees) [74] can perform range sum queries in logarithmic time which require finding the sum of all elements between two given indices.

We follow a similar intuition and augment a typical TreeMap data structure [75] to maintain the required information in the nodes of the tree. Specifically, we create a Binary Search Tree (BST) indexed by the keys of the map and in each node, store the sum of values of its subtree in addition to storing respective value. Now, we can perform `getSum` in logarithmic time (assuming the tree is balanced) by recursively traversing the tree leveraging the BST property and calling `getSum` recursively on the corresponding subtrees.

Algorithm 1 shows the pseudo-code for the implementation (note that we omitted null checks and recursion base cases to make the code more concise).

Algorithm 1 Get Sum

Result: Sum of all the values having key $\leq$ `k`

```
def getSum(node, k):
  # Rule 1
  if (k < node.key):
    return getSum(node.left, k)
  # Rule 2
  else if (k == node.key):
    return node.value + getSum(node.left, k)
  # Rule 3
  else:
    return node.left.sum + node.value
              + getSum(node.right, k)
```

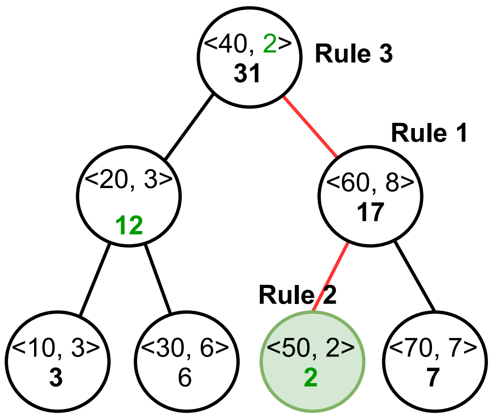


Figure 3.9. An example run of the `getSum(50)`. The red lines show the traversal path, and the values in green color contributes to the final answer which is 12+2+2. Each node shows <key, value> and the subtree sum.

Figure 3.9 shows an example run of the algorithm where it finds the sum of all values having keys less than or equal to 50 (i.e., `getSum(50)`). Having access to subtree sums avoids exhaustive iteration of nodes of subtrees that are known to have keys less than or equal to the given key. For example, in Figure 3.9, the algorithm avoids traversing the left subtree of the root node by directly extracting the corresponding subtree sum. This enables the algorithm to perform the complete `getSum` operation in logarithmic time.

3.4.2 Optimizing `shiftKeys`

`shiftKeys(key, offset)` shifts all the keys > `key` by the given `offset` (can be negative). Performing this operation on the tree-based data structure is not as trivial as PAI Maps where we iterate through the keys and update the qualifying keys. The complexity comes from the need to ensure that the BST property is intact whenever keys are updated. For cases where `offset` > 0, we can simply update all qualifying keys since `shiftkeys` shifts all the keys greater than `key` without the need for restructuring. That is, the resultant tree (after updating) is guaranteed to satisfy the BST property because (1) for the keys that get updated, they will be shifted by the same value, and (2) for the remaining keys, the set of keys that got incremented were originally greater than them. For cases where `offset`

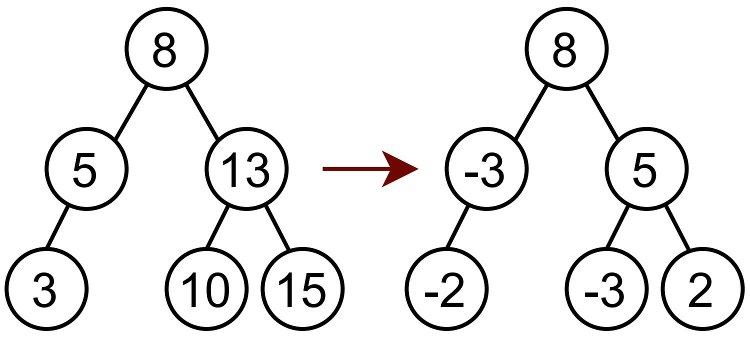


Figure 3.10. Storing the keys of the nodes relative to the parent rather than storing the raw key values. The actual key can be derived by summing up the keys from the parent to the particular node. The figure shows the parent-relative representation (right) for a given normal tree (left)

< 0, special care is needed as the tree structure may need to be changed after the update (discussed in Section 3.4.2). Nonetheless, the overall asymptotic time complexity remains at $O(n)$ since it still needs to update all the nodes with satisfying keys.

Parent-Relative Keys

To improve the time complexity beyond $O(n)$, we must design an approach that does not require visiting each qualifying node to perform the update. For that, rather than simply storing the respective key in the node, we can augment the tree structure so that each node stores the key relative to its parent. Specifically, in this setting, the actual key of a node will be the sum of keys along the path from root to the current node (see Figure 3.10 for an example). With such a structure in place, changing the key of a node is equivalent to updating the keys of the entire subtree rooted by that node (e.g., incrementing root key by 1 is equivalent to incrementing all keys by 1). Note that this changes the semantics of the raw keys stored in the node, therefore, we need to update the operations (i.e. `get`, `put`, and `getSum`) to take this change into account. Specifically, this is done by replacing the `key` in any recursive call with `(key - node.key)` (similar to lines 3 and 10 in Algorithm 3).

For example, all the keys passed to the recursive calls in Algorithm 1 should be fixed to handle the parent-relative keys. Specifically, instead of passing k, we should pass $(k -$

Algorithm 2 Get Sum (Updated to handle parent-relative trees)

Result: Sum of all the values having key $\leq$ `k`

```
def getSum(node, k):
  if (k < node.key):
    return getSum(node.left, k - node.key)
  else if (k == node.key):
    return node.value + getSum(node.left, k - node.key)
  else:
    return node.left.sum + node.value
             + getSum(node.right, k - node.key)
```

node.key) in Line 4,7, and 11. Other methods also follow a similar change to their corresponding BST implementation.

`shiftKeys` for Positive Offsets

Now that we have a way to update the value of multiple nodes in constant time (ignoring traversal time), we need to design an algorithm that performs `shiftKeys` better than $O(n)$ time. Since negative values can trigger changes in the tree structure which can be nontrivial to handle, we will first look at the case where the given offset value is positive. Algorithm 3 shows the implementation for the `shiftKeys` operator for that case (ignore `minKey` and `maxKey` related operations, we will get back to them later). The algorithm relies on the basic BST property. That is, if a node has a key greater than or equal to the given key, then that node and all the keys on the right subtree need to be incremented with the given offset (lines 2-5). However, unlike a normal tree where we would traverse the entire subtree to update the keys individually, we can simply update the key of the subtree root (line 4). This update also indirectly shifts all the keys of the left subtree which has already shifted the qualifying keys (because of the function call in line 3). Therefore, it is important to make sure that this is corrected by updating the key of the left node (line 5).

Figure 3.11 shows an example run of the Algorithm 3. Specifically, it runs `getSum(k=9, d=10)` on the input tree where all keys strictly greater than 9 is shifted by 10. This example

Algorithm 3 Shift Keys Operation (for `d` > 0)

Result: All the keys $> k$ increased by `d`

```
def shiftKeys(node, k, d):
  if (k < node.key):
    node.left = shiftKeys(node.left, k - node.key, d)
    node.key += d
    node.left.key -= d

    node.minKey -= d
    node.maxKey -= d
  else:
    node.right = shiftKeys(node.right, k-node.key, d)
  node.minKey = node.left.minKey + node.key
  node.maxKey = node.right.maxKey + node.key
  return node
```

demonstrates the benefit of storing keys relative to their parent. For instance, notice that all keys on the right subtree of the node with `key=13` are shifted without visiting any of the nodes individually. The overall time complexity of this approach is $O(\log n)$ which is an improvement over the complexity of the PAI Map data structure, $O(n)$.

`shiftKeys` for Negative Offsets

As discussed before, handling the case where the `offset` is negative is tricky due to the fact that the tree structure may have to change after shifting. For example, if the right child of a node gets shifted by a large negative value such that it is no longer greater than the parent, the resulting tree will not hold the BST property.

We follow a similar intuition to Algorithm 3 to come up with an algorithm for the negative offset case. Specifically, we traverse down the tree and perform the key updates as before. However, after every update, we explicitly check for the BST property and fix the tree if it is violated. For that, in addition to the attributes we have discussed above, we maintain two more attributes in each node. Those are `minKey` and `maxKey` which represent the minimum and the maximum keys present in the subtree rooted by the corresponding node. Note that

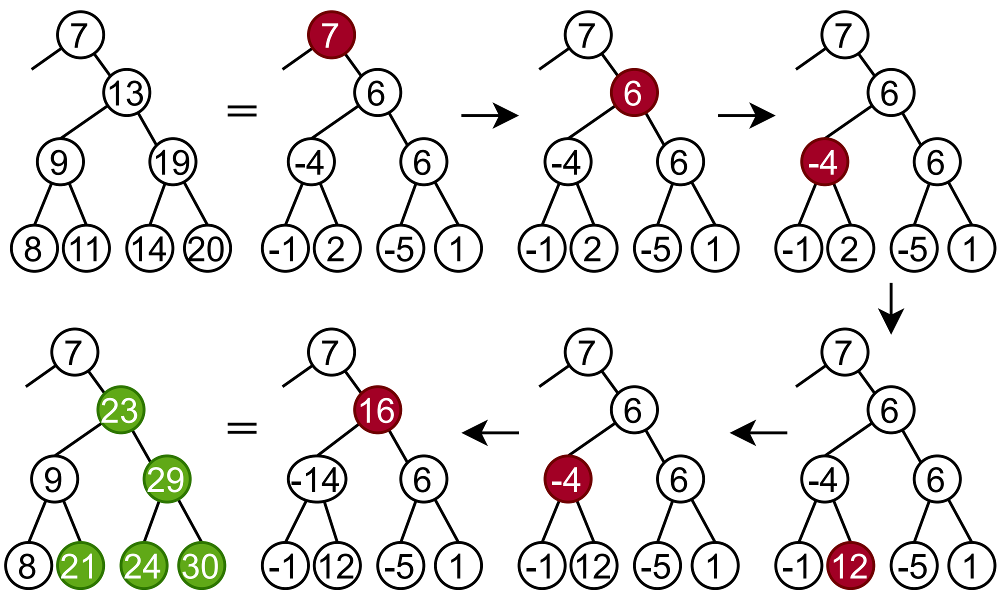


Figure 3.11. Example run of the Algorithm 3 which runs `shiftKeys(k=9, d=10)` on the given graph. This increments all the keys > 9 by 10. Two leftmost trees are not parent-relative trees and just shown for clarity.

these are based on the parent relative keys (not the actual keys). Although we do not use these attributes when performing `shiftKeys` with positive offset values, we need to make sure that we correctly maintain them during updates to the keys (Lines 7-8 and 12-13 in Algorithm 3).

This information can be used to figure out whether the key update resulted in a violation of the BST property. For example, when the recursive call of `shiftKeys` on the right subtree of a particular node has returned, we can check whether the `minKey` of that subtree is actually greater than the key of that node. If it turns out to be the case, then the BST property is preserved and the algorithm can proceed as usual. However, if the `minKey` is less than the key of the node, that means, the right subtree contains one or more keys that are less than the parent key which violates the BST property. In such cases, we need to fix the tree by explicitly restructuring nodes such that the nodes are in the correct places with respect to the BST property. The same idea can be applied to the left subtree case as well.

Algorithm 4 shows the implementation for the `shiftKeys` with a negative offset value. The code looks almost the same as the positive offset case except for the included checks to ensure the BST property is preserved (lines 8 and 12). The first case is when the current node has a key greater than the given k. In that case, the current node and all the nodes in the right subtree will get updated by the same offset value, hence, no violation arising from the right side. However, the left subtree may contain one or more nodes that are less than or equal to the given k, meaning their original keys remaining unchanged. Therefore, there is a possibility that those unchanged nodes having a larger key compared to the subtree root of which the key got decreased, violating the BST property. In our algorithm, this is detected by checking the `maxKey` of the left subtree and calling `fixTree` if the BST property is violated. The same intuition applies to the second case.

Algorithm 4 lines 18-25 shows the pseudo-code for the `fixTree` operator for fixing a subtree having a left subtree that violates the BST property. First, the subtree branch causing the violation of the BST property is removed from the original tree which results in a proper BST. Then, we simply iterate through all the removed nodes and re-insert them into the tree using the `add` operation. The semantics of `add` guarantees that the newly inserted nodes will be put into correct locations. The implementation for the right-side case follows the same idea.

Figure 3.12 shows how this algorithm works for an example input. We use actual keys in the figure instead of the parent-relative values to improve clarity. In this example, we shift the largest key of the tree by a large negative value so that the algorithm invokes `fixTree` at each step of the way up. First, `shiftKeys` will recursively traverse the tree to identify the keys that need to be shifted. Then, it shifts the key 5 and then returns back to key 19. Now, the right subtree contains a smaller key (i.e., $5 < 9$) and hence, violates the BST property at that node. Therefore, `fixTreeFromRight` is invoked where the right subtree (in this case just 5) is removed and then re-inserted back to the tree. This continues to happen until the root at which the resulting tree becomes correct.

In fact, this is the worst-case scenario for this algorithm. That is, this example requires traversing to the leaf level of the tree and performs `fixTree` at every level. To analyze the time complexity of the algorithm, we consider a tree with n nodes. Consider a similar scenario as the above example. First, one node will be inserted into a tree with two nodes.

Algorithm 4 Shift Keys Operation (for `d` < 0)

Result: All the keys $> k$ shifted by `d`

```
def shiftKeys(node, k, d):
  if (k < node.key):
    node.left = shiftKeys(node.left, k - node.key, d)
    node.key += d
    node.left.key -= d
    node.minKey -= d
    node.maxKey -= d
    if (node.key <= node.left.maxKey + node.key):
      return fixTreeFromLeft(node)
  else:
    shiftKeys(node.right, k - node.key, d)
    if (node.key >= node.right.minKey + node.key):
      return fixTreeFromRight(node)
  node.minKey = node.left.minKey + node.key
  node.maxKey = node.right.maxKey + node.key
  return node

def fixTreeFromLeft(tree):
  leftSubtree = tree.left
  tree.left = None
  tree.minKey = tree.key
  tree.sum -= leftSubtree.sum
  for (key, value) in leftSubtree:
    tree.add(key, value)
  return tree
```

Then, three nodes will be inserted into a tree with four nodes and so on. Therefore, we can derive the total time complexity as follows[1] (assuming balanced trees):

$$1 \times \log 2 + (1+2) \times \log 4 + (1+2+2^2) \times \log 8 + \ldots$$
$$= \Sigma_{\mathrm{i}=1}^{\log n}(2^{\mathrm{i}} - 1) * \mathrm{i} = O(n \log n)$$

[1]↑Summation was solved using `https://www.wolframalpha.com/`

Therefore, the overall time complexity of `shiftKeys` comes to $O(n \log n)$ for the negative offset case. This is not ideal since our PAI Map based approach can perform this operation in $O(n)$ time.

Special Case for `shiftKeys` with Negative Offsets

This worst-case time complexity is applicable for the general `shiftKeys` operation with an arbitrary offset value. However, in our context, we use this data structure to index aggregate values. For instance, in Example 3.5, we index the `rhs_sum` which corresponds to the summation of `volume` values of the `bids` relation. Moreover, we rely on `shiftKeys` when these aggregate values change as a result of tuple insertion (positive offset) or deletion (negative offset). A key realization is that, due to the monotonic nature of the aggregates, deletion of a tuple can only make two aggregate values (i.e., keys in the index) equal in the worst case. Therefore, we can simply extract the corresponding value and then delete the duplicate node from the right subtree. This can be done using the `delete` operation in the normal BST data structures that takes $O(\log n)$ time. Therefore, although the worst-case time complexity of the general `shiftKeys` is $O(n \log n)$, in the context of applying that to optimize nested aggregate queries, it becomes $O(\log n)$.

Therefore, there can only be one place where the BST property is getting violated, and hence, `fixTree` gets called only once in such a scenario. Even in this case, the worst-case time complexity of Algorithm 4 can be $O(n \log n)$. For example, if the value stored in the root and the leftmost value in the right subtree becomes equal, `fixTree` will remove the whole right subtree and re-insert all the nodes individually. This can lead to $\frac{n}{2} \times \log \frac{n}{2} = O(n \log n)$ time.

However, rather than using `fixTree` in this case, we can simply extract the corresponding value and then delete the duplicate node from the right subtree. This can be done using the `delete` operation in the normal BST data structures that takes $O(\log n)$ time. Therefore, although the worst-case time complexity of the general `shiftKeys` is $O(n \log n)$, in the context of applying that to optimize nested aggregate queries, it becomes $O(\log n)$.

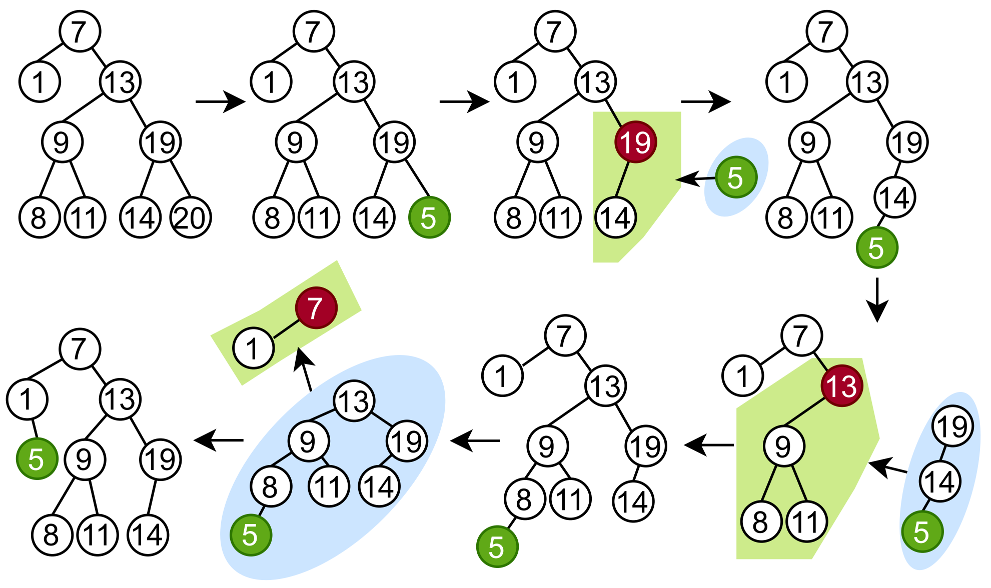


Figure 3.12. Example run of the Algorithm 4 for `shiftKeys(k=19, d=-15)` which represents the worst-case input. Actual keys are shown instead of parent-relative values to improve clarity. Green node is the node that gets its key shifted. Red nodes are places where `fixTree` is called for BST property violation. In such cases, nodes in the blue background are removed and re-inserted to the subtree in the green background.

Balanced Trees

Whenever tree-based data structures are used in practice, it is vital to make the trees balanced to achieve good runtime performance (e.g., TreeMap implementation in Java uses Red-Black Trees [76]). Similarly, we implement a Left-Leaning Red-Black Tree [77] structure and make the necessary changes to the relevant operations (e.g., rotations) to make sure our added attributes (e.g., `minKey`, `maxKey`, `sum`) are maintained properly.

In summary, in this section, we have designed a new tree-based index structured called Relative Partial Aggregate Index (RPAI) trees that supports all the required operations `get`, `put`, `add`, `getSum`, and `shiftKeys` in logarithmic time. We note that we used binary trees in our discussion and implementation, but the same principles would apply to B-trees [78] as well.

3.5 Incrementalizing Algorithm and Implementation

In the previous section, we have introduced PAI Maps and RPAI Trees and demonstrated how building such index structures on aggregates can improve the performance of correlated nested aggregate queries. In this section, we generalize our approach and present algorithms for incrementalizing different types of aggregate queries with correlated nested subqueries in their join predicates.

3.5.1 Supported Queries

We specifically target aggregate queries that contain correlated or uncorrelated aggregate subqueries in their join predicates. We construct a formal grammar (shown in Figure 3.13) to represent such queries in a concise manner instead of relying on the general relational algebra which might make the representation more verbose.

$$\begin{aligned}
AggrQ &\rightarrow Aggr_{[cols]}(AggrFunc, Relations, Predicates)\\
AggrQ &\rightarrow Aggr_{[cols]}(AggrFunc, Relations, Predicates)\\
AggrFunc &\rightarrow AggrFunc\ \mathbf{op}\ AggrFunc\\
AggrFunc &\rightarrow (SUM|COUNT|AVERAGE|MIN|MAX)f(cols)\\
Relations &\rightarrow Relation \mid Relation, Relation\\
Relation &\rightarrow Q \mid R\\
Predicates &\rightarrow Predicate \mid Predicate\ (AND|OR)\ Predicate\\
Predicate &\rightarrow Value\ \theta\ Value\\
Value &\rightarrow Value\ \mathbf{op}\ Value\\
Value &\rightarrow Const \mid Col \mid Aggr_{[]}(AggrFunc, Relations, Predicates)\\
\theta &\rightarrow\ > \mid >= \mid < \mid <= \mid =\\
\mathbf{op} &\rightarrow\ + \mid - \mid \times \mid \div
\end{aligned}$$

Figure 3.13. Formal grammar representing the set of supported queries.

Any given query mainly contains four parts. First, there is a top-level aggregate function that corresponds to the final aggregate value that needs to be computed. This can be any arbitrary function that uses aggregates of any computations on columns (e.g., `SUM(r.A) +`

`COUNT(r.B)`). Second, there is a set of relations (base tables or other subqueries) that are joined in the query. Third, there is a set of predicates (that are connected by `AND` or `OR`). Each predicate can contain nested aggregate queries, constant values, or columns from one of the join relations. Finally, the query specifies a set of group-by columns for which the final aggregate values need to be grouped. Shown below is the query in Example 3.5.

$$
\begin{aligned}
q1 &= Agg_{[]}(SUM(b.price \times b.volume), (bids\ b), q2 < q3)\\
q2 &= Agg_{[]}(SUM(b1.volume), (bids\ b1), \emptyset)\\
q3 &= Agg_{[]}(SUM(b2.volume), (bids\ b2), b2.price \leq b.price)
\end{aligned}
$$

We also define a utility function $free$ that finds all the columns that are referred within q and that are not from relations used inside the query ($free_r$ represents the subset corresponding to relation r). That is, in the case of correlated subqueries, this will be the set of correlated columns. Similarly, we define *bound* to retrieve the rest of the columns used in predicates. For example, in the query above $free_{bids}(q1) = \emptyset$, $free_{bids}(q3) = \{price\}$, $bound_{bids}(q1) = \emptyset$, and $bound_{bids}(q3) = \{price\}$. Then, we define another utility $extractPredVals$ function that extracts the predicate values, given a query. For example $extractPredVals(q1) = \{q2, q3\}$.

We only focus on aggregate queries that may contain nested aggregate queries in their join predicates since these are the type of queries other systems fail to handle efficiently (as discussed in Section 3.1). For cases where such queries occur as subqueries of other larger queries, our approach can be used incrementalize the respective portion whereas the rest of the query can be handled by already existing approaches (e.g., DBToaster).

3.5.2 General Incrementalization Algorithm

We first look at a general incrementalization algorithm that works for arbitrary queries and has better asymptotic runtime properties compared to existing approaches. In the subsequent sections, we will look at several specific commonly occurring query patterns where we further optimize using PAI Maps and RPAI Trees. We will use the query in Example 3.5 to illustrate each step.

Execution Model

We follow a similar approach to DBToaster for the query execution model. Specifically, given a query and a set of base relations (i.e., data stream sources), we first identify a set of maps that needs to be maintained to compute the final result incrementally. Then, we construct update triggers for each relation that would maintain those maps upon new tuple arrivals (or deletions). Whenever a new tuple arrives, the corresponding trigger will be called and the final result is computed after updating the indexes (similar to Figure 3.4 and 3.8).

Initializing the required maps

Although we follow a similar execution model to that of DBToaster, we do not rely on delta rules (that does not work for nested aggregates) to construct the required maps. Instead, we come up with a new algorithm to find the required indexes for queries with nested aggregates.

Our algorithm works on a simple intuition. For an aggregate query with correlated subquery predicates, whenever a new tuple arrives, two things happen. First, if the base relation corresponding to the new tuple is used in the outer query, then, there will be a new tuple for the outer relation. Hence, the predicates must be evaluated for this new tuple and if it satisfies the predicates, it should be considered in the final result. For instance, for VWAP (Example 3.5) an insertion to *bids* will result in a new tuple for the outer *bids b* relation for which the predicates must be evaluated. Second, the addition of this new tuple affects the nested aggregate predicate values of other outer tuples, hence, affecting their predicate evaluations. For example, in VWAP, insertion to *bids* will result in change of aggregate value of the left-side (because of new tuple in `b1`) and the right-side (because of new tuple in `b2`). Therefore, now for each tuple in the outer relation `b`, both sides of the predicate may have been changed, hence, need to be reevaluated to construct the result.

Algorithm 5 Identifying the required maps

```
def identifyMaps(Q):
 freeIters = {} # all free cols used in predicates
 freeAndBoundIters = {} # all free and bound
 predFreeMaps = {} # maps for freeVar -> sum
 predBoundMaps = {} # maps for boundVar -> sum
 resMaps = {} # maps for freeAndBound -> finalRes

 for Ri in Q.relations:
  freeIters[Ri] = Set()
  freeAndBoundIters[Ri] = bound(Q, Ri)

 for Vi in extractPredVals(Q):
  if (isCorrelatedAggr(Vi)):
   for Ri in Q.relations:
    freeIters[Ri].add(free(Vi, Ri))
    # new Map for free(Vi, Ri) -> aggrSumVi
    predFreeMaps[Vi][Ri] = new Map
    # new Map for bound(Vi, Ri) -> aggrSumVi
    predBoundMaps[Vi][Ri] = new Map

 for Ri in Q.relations:
  for finalRes in requiredFinalRes(Q, Ri):
   # new Map for freeBound -> finalRes
   resMaps[Ri][finalRes] = new Map
```

For any predicate value that is a constant or an uncorrelated nested aggregate, the value will be fixed for all the tuples, hence, can be computed independently. For example, the left side of the VWAP query will be a fixed value for all the outer tuples. Moreover, in the case where the value simply refers to a computation of columns from the outer tables, tuples from the outer relation would not have an impact on the other tuples. However, in the other case, where the nested aggregate query is correlated to the outer query, the addition of new tuples could change predicate evaluations of other outer tuples as well. We need to maintain index structures to efficiently keep track of these changes and compute the result without recomputing from scratch. Specifically, for each of the correlated nested aggregate predicates, we create two separate maps that map to the nested aggregate value from free and bound columns separately. Moreover, we construct a set of indexes that maps from the union of bound and free variables (off all predicate subqueries) to the final result aggregate. In the subsequent subsections, we will see how these maps are used to compute the final result. For any nested aggregate predicate value that contains multi-level nesting, we apply this approach recursively and initialize the relevant maps.

Algorithm 5 shows how the required maps for a query are identified. Specifically, for each of the correlated nested aggregate predicates, we create two separate maps that map to the nested aggregate value from free and bound columns separately (Line 17 and 19). Finally, we construct a set of indexes that maps from the union of bound and free variables to the final result aggregate. Note that there can be multiple final result aggregates (Line 24) based on the query (e.g., `SUM(a.price) + SUM(a.volume)` would need to maintain the two aggregates separately). In the following two subsections, we will see how some of these maps act as auxiliary indexes to maintain several other maps that are used to compute the final result. For any nested aggregate predicate value that contains multi-level nesting, we apply this approach recursively.

Consider the VWAP query. The left side in the predicate is not a correlated query, therefore the aggregate value can be independently maintained (e.g., in variable `lhs_sum`). For the right side aggregate subquery, both free and bound variables are the same $\{price\}$. Therefore, we need to create two maps mapping price to the aggregate sum (e.g., `freeMapRhs: price` $\rightarrow$ `rhs_sum` and `boundMapRhs: price` $\rightarrow$ `rhs_sum`). Finally, we need a map that maps $free \cup bound$ to the final aggregate value (e.g., `resMap: price` $\rightarrow$ `resSum`). These are shown in Figure 3.14, lines 1-4. We will walk through the rest of the code and observe how these maps are maintained and used in the next subsection.

Creating Update Triggers

Once the required maps are constructed, we need to generate triggers for each base relation that maintain these maps under updates. Algorithm 6 shows how such a trigger is created for relation R_i. First, we iterate through free and bound column values of all relations except R_i. This is done because the predicate values (hence, the respective indexes) of all

Algorithm 6 Creating the update triggers

```
1  def trigger_Ri(t):
2   for vR1 in freeAndBoundIters[R1]:
3    for vR2 in freeAndBoundIters[R2]:
4     # skip Ri
5     ....
6     for vRn in freeAndBoundIters[Rn]:
7    values = {vR1, vR2, .., vRn}
8    for Vi in extractPredValues(Q, Ri):
9     if (isCorrelatedAggr(Vi)):
10     # update the bound maps
11     predBoundMaps[Vi][Ri].add(t,
12                              Vi.aggrSum(values, t))
13    # update affected aggr values
14    for key in predFreeMaps[Vi][Ri].get(values):
15     if Vi.evaluatePredicate(values, t, key):
16      predFreeMaps[Vi][Ri].add(values + key,
17                          Vi.aggrSum(values, t, key))
18      # find the aggr value for the newTuple
19     if not predFreeMaps[Vi][Ri].exists(values + t):
20      aggr = 0.0
21      for key in predBoundMaps[Vi][Ri].get(values):
22       if Vi.evaluatePredicate(values, key, t):
23        aggr += Vi.aggrSum(values, key, t)
24      predFreeMaps[Vi][Ri].update(values + t, aggr)
25   # update the result maps
26   for reqRes in resMaps[Ri]:
27     resMaps[Ri][reqRes].add(values + t,  compute(reqRes, t))
28
```

those relations only depend on the respective free and bound column values. Then, there are mainly three maps that need to be updated.

First, the addition of a new tuple changes the bound maps of any predicate that has columns from R_i as bound variables. Lines 8-11 show how this update is performed. Specifically, we iterate through all the predicate values that use columns from R_i and check whether it is a correlated subquery. Note that this and other similar checks have to be done only once (i.e., during trigger generation) as at runtime, the constructed triggers are specialized for a given query. Then, we retrieve the corresponding map and update the relevant entry. Here, we directly pass `t` as a key to `add` and omit the code for extracting the correct key (i.e., values of bound columns in this case). For VWAP (Figure 3.14), `boundMapRhs` is updated in line 9.

Second, we need to update the aggregate values (in free maps) that are affected by the addition of the new tuple to the inner relations (shown in lines 14-17). We iterate through the keys in the corresponding free map and evaluate the predicate for the new tuple. If it satisfies the predicate, the aggregate value is affected, hence, the map is updated. Here, we pass `key` and `t` for the values of free and bound columns respectively as we are updating the change caused by the addition of the new tuple to the inner relation (i.e., bound). For VWAP, this is done in lines 11-13 in Figure 3.14.

Next, we need to compute the aggregate value for the new tuple in the outer relation. This can be computed by iterating through the keys in the corresponding bound maps and evaluating the predicate and accumulating the aggregate value (lines 19-24). Here, we should pass `key` and `t` for the values of bound and free columns respectively (i.e., opposite to the above case) because now we are computing the aggregate value for the new tuple in outer relation (i.e., free in the context of subquery). Figure 3.14 lines 15-20 shows this for VWAP.

Finally, we need to update all the result maps corresponding to the relation R_{i}. This can be done by simply iterating through all the result maps of R_{i} and updating the maps with the corresponding value (Lines 28-30). Here, `values + t` refers to appending `t` to `values`. For the VWAP query, this is done in line 22 in Figure 3.14.

Computing the final result

After updating all the maps, the next step is to compute the final result using the updated maps. For this, we iterate through all the free and bound column values for all the relations and evaluate the query predicate. When evaluating the nested aggregate values, we can directly use the free maps that we have created before to get the corresponding aggregate value without doing any more computations. Then for keys that satisfy all the query predicates, the corresponding results maps are queried to get the aggregate sum.

```
 1 lhs_sum = 0         # maintains lhs_sum
 2 freeMapRhs = Map()  # price -> rhs_sum
 3 boundMapRhs = Map() # price -> Sum(volume)
 4 resMap = Map()      # price -> Sum(price * vol)
 5 def on_new_bids(t):
 6   # update lhs_sum
 7   lhs_sum = 0.75 * t.X * t.volume
 8   # update bound maps
 9   boundMapsRhs.add(t, t.X*t.volume)
10   # update affected aggr values
11   for price in freeMapRhs:
12   if t.price <= price:
13   freeMapRhs.add(price, t.X*t.volume)
14   # find the aggr value for the new tuple
15   if not t.price in freeMapRhs:
16   aggr = 0.0
17   for price in boundMapRhs:
18   if price <= t.price:
19   aggr += boundMapRhs.get(price)
20   freeMapRhs.update(t.price, aggr)
21   # update resMap
22   resMap.add(t.price, t.X*t.price*t.volume)
23   # compute the result
24   res = 0.0
25   for price in resMap:
26   if lhs_sum < freeMapRhs.get(price):
27     res += resMaps.get(price)
28   return res
```

Figure 3.14. Code generated for incrementally computing the VWAP query (Example 3.5) using the General Incrementalization Algorithm (Section 3.5.2)

Figure 3.14, lines 26-30 shows how the final result is computed for the VWAP query. The overall time complexity of this approach is $O(n^k)$ where n is the cardinality of relations and k is the number of relations in the query. In comparison, DBToaster can take up to $O(n^k \times n^k)$ for processing queries of this structure.

Limitations

The algorithms discussed above assumes the aggregates functions are *streamable* [69]. That is, the aggregate values can be updated by having information about the current aggregate and the new value (e.g., `SUM`, `COUNT`, `AVERAGE`, etc.). For other types of aggregates (e.g., `MIN/MAX`), our approach of storing the aggregate would only work for insertion-only updates. For deletions, we cannot recover the updated aggregate value for such aggregate types since it is impossible to recover the new value with just the previous aggregate value. One way to handle this is to keep a binary search tree of the data instead of storing just the aggregate value. Now, for tuple deletions, we can simply remove the corresponding value from the tree and retrieve the next maximum or minimum value in logarithmic time (assuming balanced trees). However, it is not clear how this can be generalizable beyond `MIN/MAX` to any arbitrary non-streamable monoid.

3.5.3 Partial Aggregate Index Optimization

The general algorithm above can improve incrementalization efficiency for a wide class of aggregate queries that contain correlated nested aggregates as predicates. In this section, we look at how we can further optimize certain classes of nested aggregate queries that appear commonly in practice (e.g., computing useful metrics in algorithmic trading [64]) using PAI Maps and RPAI Trees introduced in Section 3.4. As discussed in Section 3.3, a key idea to improve the incrementalization efficiency of nested aggregate queries is to build indexes that are indexed by aggregate values. Then, whenever a new tuple arrives, shift a single or a range of aggregate values (i.e., keys in these trees) that are affected and then use these updated indexes to compute the final result.

We do note that this approach only works for queries having a certain structure compared to the general algorithm which works for any general query. The main requirement for this approach to work efficiently is that the addition of a new tuple should only affect a single aggregate value or a single range of aggregate values per aggregate index. Whenever a range of aggregates needs to be shifted, we can use the `shiftKeys` method introduced in Section

3.4. For a query that is operated on a single base table (or subquery) R_1, it should have the following structure to use aggregate indexes to improve the incrementalization.

$$AggrQ_{[cols]}(AggrFunc, R_1, v_1\ \theta\ q_1)$$

Here, the $AggrQ$ should be a streamable aggregate function, v_1 is either a *const* or an $AggrQ$ with $free(v_1) = \emptyset$ (hence, can be independently maintained), and q_1 is an $AggrQ$. Note that v_1 and q_1 can be on opposite sides as well. A key characteristic of this query is that, for all the tuples from the outer relation, v_1 will have the same value (because not correlated). Therefore, if there is an aggregate index that maps the aggregate values of q_1 to the aggregate values required for the final query results, we can directly compute the query result. For example, if θ was '=', then we get the value of v_1 and query the aggregate index with the value of v_1 to get the result. For other cases, we can use `getSum` method to find the final aggregate values that are within the range implied by the predicate.

To maintain this aggregate index, the addition of a new tuple should change either a single aggregate value or a range of aggregates. This happens if either q_1 contains multiple conjunctive equality predicates (results in a single point update) or q_1 contains a single inequality predicate (results in an update of range of aggregates). For example, in the VWAP query (Example 3.5), predicate in the nested query is the inequality `b2.price <= b.price`, meaning that whenever a new tuple arrives for `b2`, all the aggregates values that have a `price` greater than the `price` of new tuple must be updated (by the same amount).

Shown below is the structure of queries with multiple relations that supports the aggregate index optimization.

$$AggrQ_{[cols]}(AggrFunc, R_1, \ldots R_n, v_1\ \theta\ q_{R_1} \ldots AND\ v_n\ \theta\ q_{R_n})$$

Here, $AggrFunc$ and $v_1, \ldots v_n$ has the same properties as before, and for q_{R_i}, it should be correlated only on columns from R_i (i.e., $free(q_{R_\mathrm{i}}) \subseteq R_\mathrm{i}.columns$). Similar to the general algorithm, we create free and bound maps for each of the correlated nested subqueries. Moreover, we create aggregate indexes that maps aggregate values (i.e., q_{R_i} values) to the required result sums.

Algorithm 7 Creating the update triggers

```
def trigger_Ri(t):
  tAggrVal = QRi.aggrVal(t)
  if predicateType(QRi) == "=":
  newAggr = boundMaps[Ri].get(t) + tAggrVal
  affectedAggr = freeMaps[Ri].get(t)
  # shift point aggregates
  for reqSum in requiredSums(Q, Ri):
  valToMove = resMaps[Ri].get(t)
  aggrMaps[Ri].add(affectedAggr, -valToMove)
  aggrMaps[Ri].add(affectedAggr+tAggrVal, valToMove)
  aggrMaps[Ri].add(newAggrSum, reqSum(t))
  resMaps[Ri].add(t, reqSum(t))

  elif predicateType(QRi) == "<=":
  newAggr = boundMaps[Ri].getSum(t)
  affectedAggr = freeMaps[Ri].getSum(QRi.boundComp(t))
  # shift all aggr >= affectedAggr
  for reqSum in requiredSums(Q, Ri):
  aggrMaps[reqSum].shiftKeysInclusive(
                            affectedAggr, reqSum(t))
  aggrMaps[reqSum].update(newAggr, reqSum(t))
  elif predicateType(QRi) == "<":
  ...
  ...
  # update the rest of the maps
  freeMaps[Ri].add(t, tAggrVal)
  boundMaps[Ri].add(t, tAggrVal)
```

Algorithm 7 shows how the trigger is generated for relation R_i. The code generated for each relation depends on the type of subquery predicates. The pseudocode only shows the case for equality and $\leq$ case, and rest of the cases follow a similar strategy. The crux of the algorithm is identifying the single aggregate value or range of aggregates that need to be shifted and performing the update.

Overhead of Identification

For our approach to be applied to a general incremental query processing setting, it is imperative to be able to recognize the queries that our optimizations can be applied. Given a query (or part of another query), we can first check whether the query computes an aggregate. Then, we can make use of the helper functions $free$ and *bound* to determine if our query fits the structure for applying the aggregate index optimization. Therefore, identification can be done as part of the query optimizer (e.g., pattern matching on the query tree) and would have a similar overhead. Once the eligible queries have been selected, there is a cost associated with creating the required maps and generating the triggers. Similar overheads exist for other systems that follow a similar execution model. For example, identifying the different predicates and their value types (e.g., Line 9-10 in Algorithm 6) is done in trigger generation time and would not have an impact on the runtime performance. Overall these initializations take a time linear to the size of the query (i.e., no exponential blowup).

Limitations

This approach only works for the queries that are in the structure mentioned above, and cannot apply generally for other types of aggregate queries. Moreover, since this optimization relies on indexing maps with aggregate values and shifting aggregate ranges, this only works with aggregates like `SUM`, `COUNT`, and `AVERAGE` and cannot be used for `MIN/MAX`.

3.5.4 Prototype Implementation

We implement a prototype of our approach in a high-level programming language, Scala. Scala's features such as first-class functions, case classes, and pattern matching make the implementation and management of the different types of trigger functions relatively convenient. Moreover, using existing techniques like Multi-Stage Programming (MSP) [33, 79, 80], we can eradicate the overhead of using such high-level features by generating specialized native code. This approach generally achieves orders of magnitude performance improvements [19, 20, 81]. However, this performance optimization is beyond the scope of this chapter, and

Table 3.1. Queries used for evaluation with their corresponding optimizations and asymptotic time complexity. GA - general algorithm, Aggr - aggregate index optimzation

Query	Optimizations		DBToaster	RPAI (Ours)
	GA	Aggr		
MST, VWAP, NQ1	✓	✓	$O(n^2)$	$O(\log n)$
PSP	✓	✓	$O(n)$	$O(\log n)$
SQ1, SQ2	✓	✗	$O(n^2)$	$O(n)$
NQ2	✓	✗	$O(n^3)$	$O(n \log n)$
TPC-H Q17	✓	✓	$O(n)$	$O(\log n)$
TPC-H Q18	✓	✗	$O(1)$	$O(1)$

hence, we only use the techniques presented in this chapter with the corresponding Scala implementation for experiments in Section 3.6.

Our implementation is done in the context of an in-memory incremental processing setting where we assume there is sufficient memory to hold the indexes and any other data required. We rely on the JVM for memory management. If this memory usage becomes a bottleneck, we can use techniques like MSP to compile our program into a specialized low-level code where we can rely on the operating system (i.e., paging) or custom memory management mechanisms. Moreover, the ideas presented in this chapter are not limited to the context of processing data streams but are generally applicable to any other incremental processing use cases (e.g., IVM in DBMS, approximate query processing [65], etc.).

3.6 Experimental Evaluation

In the previous sections, we have demonstrated how to use RPAI Trees and PAI Maps to achieve better asymptotic time complexities for queries with correlated nested aggregates. In this section, we run several benchmarks and measure the actual execution time for workloads of practical importance to ensure that the expected performance characteristics are indeed realized in practice.

3.6.1 Benchmark Environment

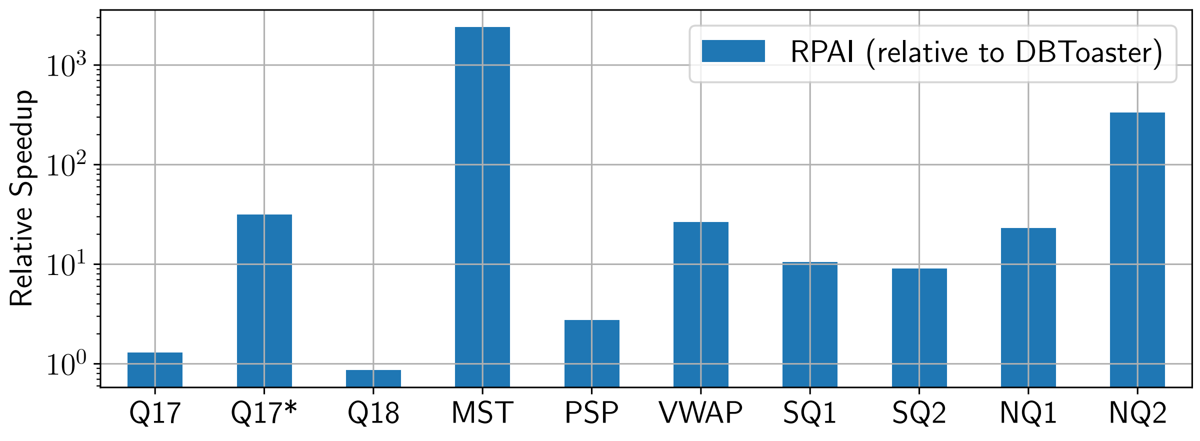
Data and Query Workload

There is no single established benchmark for evaluating the performance of correlated nested aggregate queries specifically. Therefore, we use two existing datasets with the addition of synthetic queries for our evaluation. The first workload consists of running algorithmic trading related queries on a stream of order book data (used in prior related work [1]). There are two types of order book entries; *bids* and *asks* each having *timestamp*, *broker_id*, *price* and *volume* as their attributes. We evaluate the performance for MST, PSP, and VWAP queries from the original benchmark and added SQ1, SQ2, NQ1, and NQ2 synthetic queries to evaluate different types of correlated nested aggregate queries. The second workload is using selected queries from the TPC-H benchmark adapted to an incremental processing setting (similar to [1, 68]). We specifically focus on Q17 and Q18, which contain nested aggregates.

one having a correlation with outer query. SQ1 and SQ2 are modified versions of the VWAP query. SQ1 makes the uncorrelated subquery into a correlated one by adding a predicate inside the nested subquery. SQ2 replaces the inequality inside the nested aggregate from having the same column on both sides to having an arbitrary computation.. Nested1 and Nested2 are also modified versions of VWAP that replaces the nested aggregate with another multi-nested aggregate query. Nested1's correlation is always to the immediate outer query whereas Nested2 is contains correlations to the outer-outer level.

System Environment

We run all our experiments on a NUMA machine with 4 sockets, 24 Intel(R) Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total) running Ubuntu 18.04.4 LTS. We use DBToaster 2.3 and use the generated Scala code in our experiments. Our implementation uses Scala 2.10.4. All the experiments are single-threaded and run on



Time (ms)	Q17	Q17*	Q18	MST	PSP	VWAP	SQ1	SQ2	NQ1	NQ2
DBToaster	42864	1097090	42410	168520	88	888	3967	1265	2957	202544
RPAI	33066	35041	46417	70	32	34	379	140	129	610

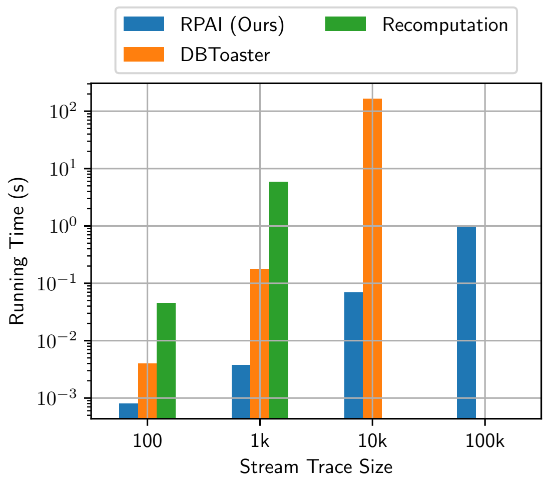
Figure 3.15. Relative execution time for queries compared to DBToaster. All queries except TPC-H Q18 have better performance.

a single-core. We use default JVM configurations for most cases except for TPC-H queries where we increase the max heap size to per-socket memory.

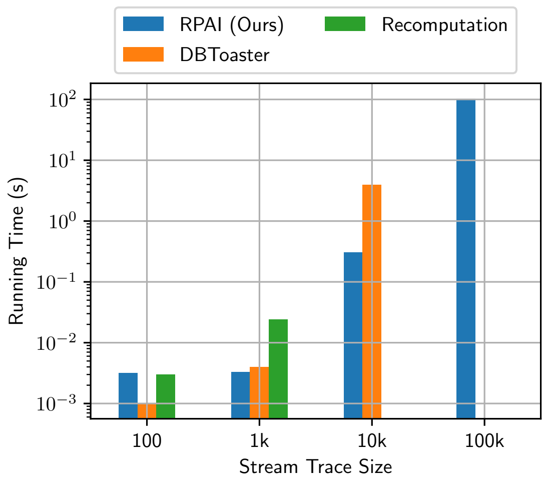
3.6.2 Evaluating Incrementalizing Performance

Query Properties

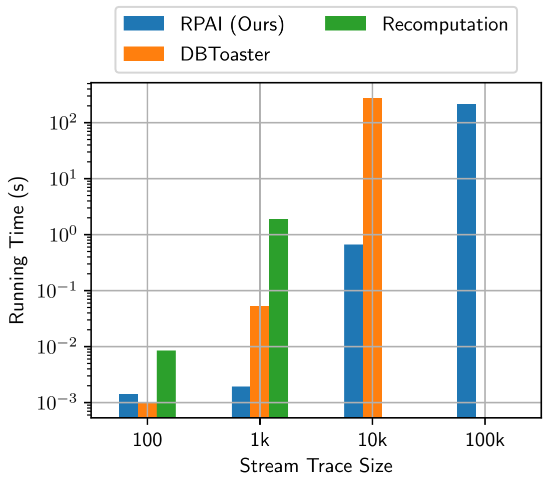
Table 3.1 summarizes the specific optimizations we perform on each query and the time complexity difference between our approach and DBToaster. MST and PSP are operating on both *asks* and *bids* and perform a cross join with inequality predicates. MST has four nested aggregates of which two are correlated. As DBToaster does not incrementalize correlated nested aggregates, it needs to iterate through records from both relations to compute those correlated subqueries. We incrementalize those queries using RPAI Maps. Therefore, whenever a new record arrives, we can compute the corresponding aggregate value and shift all the affected aggregate values in logarithmic time (as demonstrated in Section 3.5). PSP is similar to MST but contains join predicates on a column (`volume`) instead of a correlated nested aggregate. We follow a similar approach to incrementalize PSP in logarithmic time.



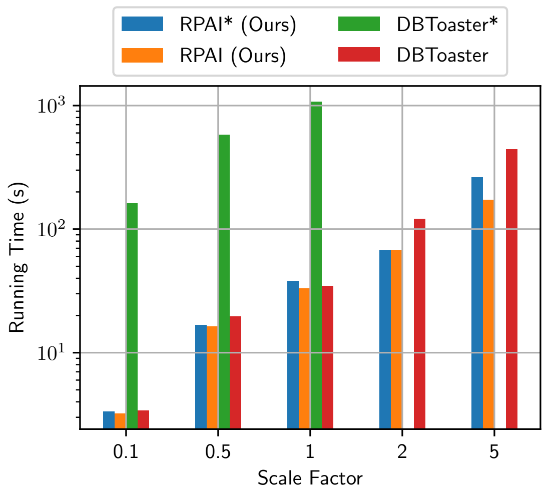
(a) MST



(b) SQ1



(c) NQ2



(d) Q17 (*for skewed dataset)

Figure 3.16. Analyzing the scalability of our algorithm for different queries. (d) contains results for both the uniform (default) and the non-uniform (augmented) versions of the TPC-H datasets.

We saw VWAP in Example 3.5. It contains two nested aggregates; one correlated with the outer query. SQ1 and SQ2 are modified versions of the VWAP query. SQ1 makes the uncorrelated subquery into a correlated one by adding a predicate inside the nested subquery. With this modification, both sides of the predicate become variable for a given outer record. Hence, the final result computation can no longer be done in logarithmic time using `getSum` operator and has to rely on the general algorithm (hence, $O(n)$). SQ2 changes the right-

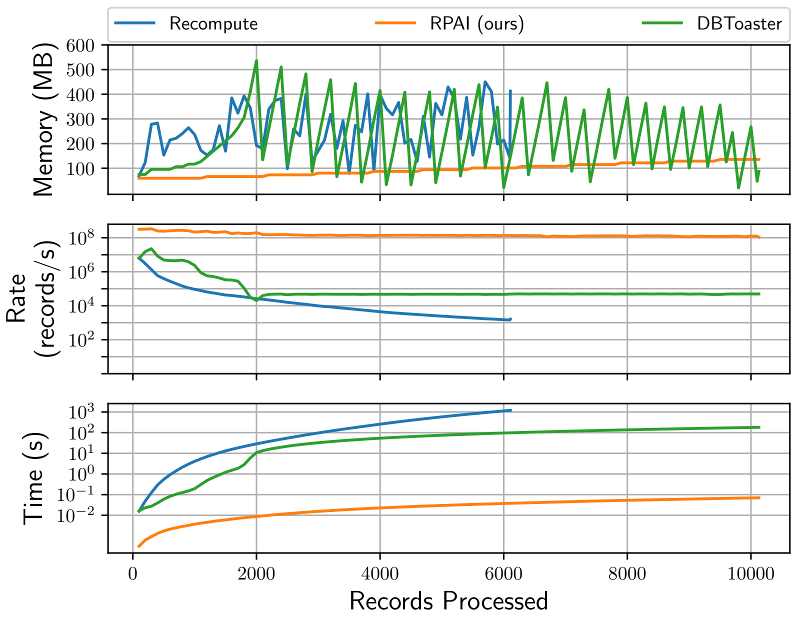


Figure 3.17. The memory footprint (top), rate of processing records (middle), and time (bottom) as incoming records get processed for MST query

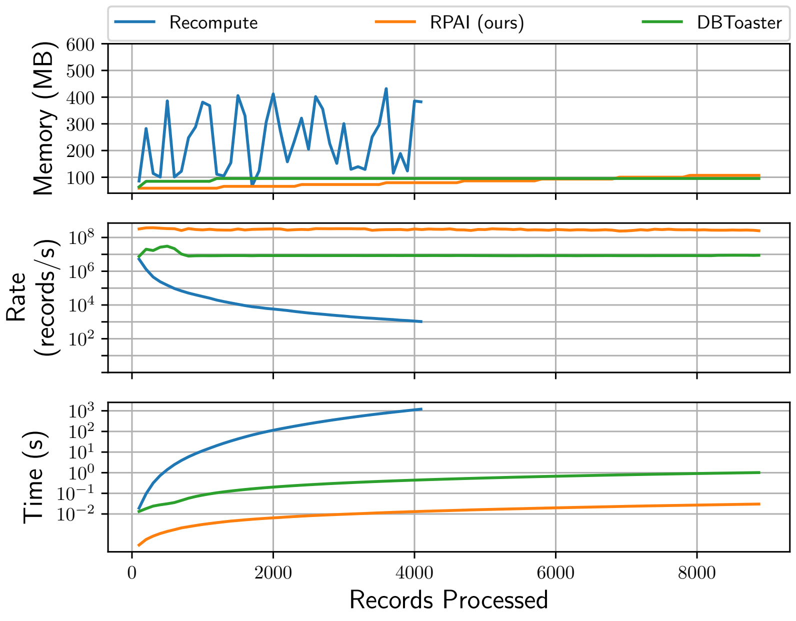


Figure 3.18. The memory footprint (top), rate of processing records (middle), and time (bottom) as incoming records get processed for VWAP query

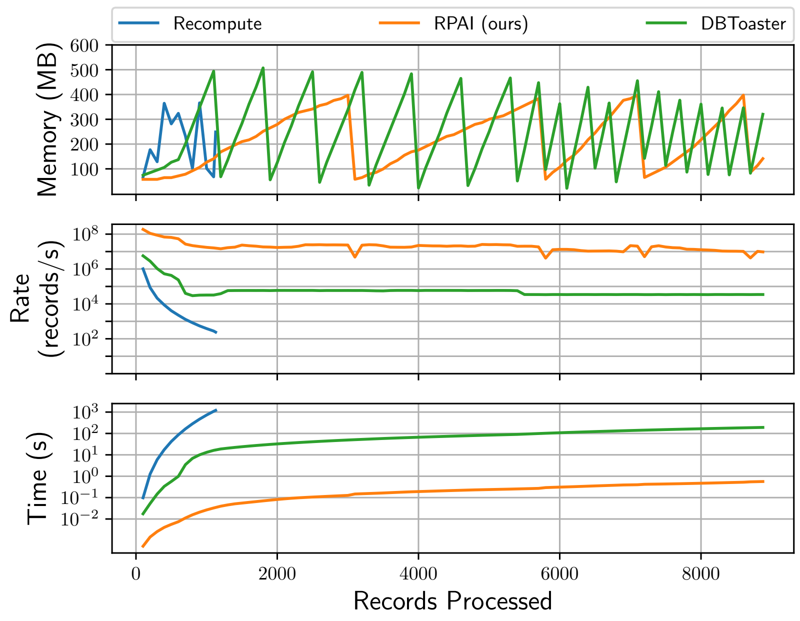


Figure 3.19. The memory footprint (top), rate of processing records (middle), and time (bottom) as incoming records get processed for NQ2 query

side inner predicate of VWAP to contain asymmetric computations in the two sides of the inequality.

NQ1 and NQ2 are also modified versions of VWAP that replace the nested aggregate with another multi-nested aggregate query. Specifically, NQ1 replaces the nested aggregate in VWAP with another correlated nested aggregate query like VWAP (i.e., containing 2-level nesting). NQ2 is similar but the replaced query also has a correlation to the outer query (i.e., the lowest level is correlated to the outermost query). NQ1 is handled by computing the delta of the new subquery independent of the outer query. Once we compute the delta, the rest of the computation is the same as VWAP incrementalization. For NQ2, we have to rely on the general algorithm for the outermost query. DBToaster uses three nested loops whereas we incrementalize this query in $O(n \log n)$ time.

TPC-H Q17 and Q18 contain a single correlated and uncorrelated nested-aggregate respectively. We can improve the efficiency of Q17 over DBToaster by using our approach. However, since Q18 is uncorrelated, both our implementation and DBToaster fully incrementalize the query, leaving the time complexity at the same level. We added Q18 to demonstrate that our approach still achieves competitive performance for other types of aggregate queries.

Performance Analysis

Figure 3.15 shows the relative execution time for all queries. In this experiment, we run all the queries on a trace of stream of the same size (10k records for finance queries, Scale Factor 1 for Q17 and Q18). We observe that most queries show a significant speedup (up to more than 1100 ×) which is in line with the expected performance gains due to lower asymptotic time complexity. Our implementation has a similar performance to DBToaster for Q17 and Q18. This can be partly attributed to the efficiency of specialized internal data structures (e.g., specialized hash-maps) generated by DBToaster. Although it explains the performance of Q18, it still cannot explain why DBToaster scales at a similar rate to our implementation for different sizes of datasets for Q17, despite having a higher time complexity (Figure 3.16d). Consider Q17 shown in Figure 3.20:

```sql
SELECT SUM(l.extendedprice) / 7.0 AS avg_yearly
FROM   lineitem l, part p
WHERE  p.partkey = l.partkey AND  p.brand = 'Brand#23'
AND  p.container = 'WRAP BOX'
AND  l.quantity < (
SELECT 0.2 * AVG(l2.quantity) FROM lineitem l2
WHERE l2.partkey = p.partkey)
```

Figure 3.20. TPC-H Query 17

Our implementation incrementalizes this by maintaining an RPAI Tree (`quantity` → `avg_yearly`) and the aggregate value of the nested aggregate query for each distinct `partkey`. Whenever, a new `lineitem` arrives, we update the corresponding index and the aggregate value, and compute the change to `avg_yearly` using `getSum(rhs_sum)`.

In the case of DBToaster, whenever a new `lineitem` arrives, first it updates the nested-aggregate sum (similar to us) and it loops over all the `lineitem`s that have the same `partkey` and evaluate the predicate and updates the `avg_yeary` change. However, rather than iterating over *all* the line items with the same part key, they are only iterating over line items with unique `quantity` values. This is done by a multi-level index that maintains partial sums for each unique `quantity` per unique `partkey` for `lineitem` records (i.e., `partkey` $\rightarrow$ `quantity` $\rightarrow$ `avg_yearly`). This works well for datasets like TPC-H where the data is uniformly distributed. Therefore, although the worst-case time complexity is O(n), due to the uniformity of the data, the updates can be performed in a very small fraction of the total records. We note that the amount of computation is still lower in our implementation compared to this optimized strategy as our TreeMap also maintains unique `quantity` values. To validate our hypothesis, we analyzed the behavior of DBToaster against a skewed dataset. We augmented the TPC-H data generation framework to generate skewed data. As expected, the performance gap grows from 1.3× to more than 30× (Q17* in Figure 3.15). Therefore, although this is an extremely useful optimization for performing simple nested aggregates on uniform datasets, it does not work well in other scenarios. Moreover, it only works for nested subqueries correlated on equalities.

Figure 3.16 analyses the scalability of our approach over the stream trace size. The workload for the first three queries (i.e, MST, SQ1, and NQ2) is selected from different sizes of stream traces from the original finance dataset [1]. For the last figure, we use the TPC-H dataset of scale factors 0.1, 0.5, 1, 2, 5 (100MB, 500MB, 1GB, 2GB, and 5GB respectively), both the uniform and the skewed version. In some cases, for smaller datasets (up to 1k in finance, SF=1 in TPC-H), our approach performs competitively or worse than both recomputation and DBToaster. This can be attributed to the fact that our approach initializes more index structures and for smaller workloads, maintenance of these indexes can outweigh the performance benefits. However, as the workload size increases, our approach performs significantly better and scales well (as expected due to better asymptotic time complexity).

Figures **??** shows how the runtime performance characteristics change as the stream of data gets processed. Specifically, we analyze how the memory usage, rate of processing records, and time varies over the number of records processed. The rate of processing records generally drops over time as more data is manipulated in indexes and iteration spaces become larger. We can see our implementation consistently has a better rate compared to DBToaster due to being able to perform the updates efficiently. We can also observe that for some cases the memory usage contains spikes at regular intervals. These correspond to garbage collection (GC) triggered by the Java Virtual Machine (JVM). This is also evident from the fact that there is a decline in the rate of processing records at the same point in time. The memory usage just after a GC run is closer to the starting memory in many cases, indicating that the working set size does not grow notably over time. For all the queries, we can see that these GC cycles happen more frequently in DBToaster compared to our implementation, implying better allocation rates. We note that these behaviors may only be applicable for languages like Scala where there is a managed runtime and may see different behavior for runtimes with explicit memory management.

3.7 Related Work

Traditional Incremental View Maintenance (IVM) [82–86] approaches deal with maintaining materialized views on updates to underlying base tables. Generally, these approaches do not focus on performing real-time analytics on fast data streams, and hence, are not optimized for low latency. DBToaster [1, 64, 68] applies the idea of IVM in a recursive manner [87] to maintain a view using a hierarchy of materialized views. Some prior works present algorithms and data structures for incrementalization of specific types of queries. DYN [69] and IDYN [70, 88] are two approaches that focus on acyclic Conjunctive Queries (CQ) with equality and inequality join predicates respectively and AJU [71] focuses on acyclic foreign-key joins. None of these approaches efficiently incrementalize queries with correlated nested aggregate sub-queries when correlation removal [89] is not possible.

Incremental query processing algorithms are used in many other use cases including approximate query processing [65], progressive data warehouses [90, 91] and intermittent query processing [67]. Some works investigate how to balance the resource consumption for view maintenance and latency [66, 67, 92–94] and improve overall performance in multi-query settings by sharing computations across concurrent queries [92, 95–97]. However, many of the incrementalization algorithms presented in these works either do not efficiently incrementalize complex queries such as correlated nested aggregates or are restrictive due to their application-specific requirements (e.g., produces approximate results, does not support tuple deletions, etc.). Therefore, such algorithms are not directly applicable to cases where perfectly accurate results are needed to be produced as a stream. Since the algorithms presented in this chapter are relatively general, we believe our approach can be applied in these settings to improve their performance.

Stream processing systems [62, 98–100] process data streams and incrementally maintain the query outputs. These systems are mainly designed for queries with window semantics. Naiad (Timely Dataflow [TD]) [101], Differential Dataflow (DD) (build on top of TD) [102] are two frameworks that allow scalable incremental computation. Materialize [103] is an IVM engine built on top of both TD and DD. We believe that the algorithms and the data structures presented in this chapter can be implemented in those systems for efficient incrementalization of nested aggregates.

TreeMap is a data structure found in the standard libraries of many programming languages [75, 104], and of course, tree indexes are standard in any RDBMS [78]. Fenwick Trees [74] and Segment Trees [73] are two tree-based data structures that support operations similar to `getSum` in logarithmic time. However, none of them have support for efficiently shifting key ranges. To our knowledge, the data structure we proposed for RPAI Trees is the first to support both `getSum` and key shifts (`shiftKey`) in logarithmic time.

3.8 Conclusions

We have introduced a novel algorithm and indexing structures called PAI Maps and RPAI Trees for efficiently incrementalizing complex nested-aggregate queries with correlations. We have presented a new data structure to realize these index structures efficiently and demonstrated the performance gains in both asymptotic time complexity and actual execution time. We believe the ideas presented in this chapter can be applied to other incremental execution settings to improve the performance of these types of queries.

4. FLAN: AN EXPRESSIVE AND EFFICIENT DATALOG COMPILER FOR PROGRAM ANALYSIS

Portions of this chapter will appear in *Flan: An Expressive and Efficient Datalog Compiler for Program Analysis, Proc. ACM Program. Lang. 8, POPL, Article 86 (January 2024)* [105].

4.1 Introduction

Datalog has experienced a resurgence in popularity due to its high expressivity and ease of use in various applications. Its simple and intuitive declarative nature makes it readily applicable in a range of domains, including business analytics [106], graph analysis [107–109], declarative networking [110], binary dissambly [111], and declarative program analysis [2, 112–116]. In particular, in the field of declarative program analysis, Datalog has proven to be an invaluable tool for dealing with intricate analyses that would otherwise necessitate thousands of lines of imperative code. Specifically, numerous program analysis problems can be formulated as a form of fixed-point computations, which is precisely what Datalog is designed to handle.

While declarative program analysis simplifies matters significantly, earlier approaches utilizing Datalog were not as scalable and performant as their imperatively written counterparts. Datalog compilers [2, 117–119] have been developed to close this performance gap by generating specialized code for a given Datalog program. Among such Datalog compilers, Soufflé [2] stands out as one of the most comprehensive systems with many person-years of engineering investment, having numerous features such as fast parallel specialized data structures [120–122], automatic index selection [123], join order optimization [124], among others.

Although Soufflé generates specialized code for a specific Datalog program leveraging the idea of Futamura projections [38] (resulting in impressive performance), their style of code generation, as discussed in Section 4.2, sacrifices some potential for specialization and, hence, performance. Moreover, Soufflé exists largely as a monolithic, closed system. It accepts Datalog programs as input and produces specialized C++ code, which is then compiled into an independent binary. However, as identified in previous research [4, 117], many Datalog programs, especially in the context of program analysis, do not exist in isolation, but rather as components of larger programs written in more comprehensive programming languages. While Soufflé offers mechanisms to interface with the generated binary, it suffers from the *two language problem* where declarative analyses are written in a custom Datalog dialect and further computations not supported by Soufflé must be written in other languages with data stored in a common format. Moreover, this lack of interoperability with a host language becomes a bottleneck when efforts are made to incorporate recently proposed Datalog extensions like user-defined lattices [3, 117], SMT constraints [4], etc.

At the opposite end of the spectrum, languages like Flix [125] support first-class Datalog constraints and seamless embedding of Datalog logic within a general-purpose host language. This approach provides greater flexibility, allowing users to create modular Datalog programs that can be composed in various ways depending on the specific task. Additionally, it becomes feasible to enrich Datalog with features such as user-defined Lattices [3], since the end user can employ a full-fledged language to define these data structures and associated operations. However, existing systems following this approach lack support for specialized code generation (akin to Soufflé), resulting in significantly lower performance when compared.

In this work, we aim to bridge the gap in designing Datalog engines that are both *flexible* (akin to Flix) and *performant* (akin to Soufflé). One might contend that this could be achieved by augmenting the textual frontend of Soufflé-like systems with capabilities akin to those of Flix. However, such an undertaking would be tantamount to constructing an entire programming language from the ground up, complete with a comprehensive type system and more. Additionally, significant effort would be necessary to expand the compiler backend in order to accommodate these new features, owing to the intricacies inherent to the backend engine.

In this chapter, we investigate if it is possible to architect a Datalog system with three key characteristics: (1) generating fully specialized code, a critical factor for optimal performance; (2) creating a flexible backend that can easily accommodate a variety of features and evaluation strategies; (3) creating a flexible frontend capable of seamless interaction with a full-fledged programming language, potentially using its features to enrich the frontend with various extensions.

In pursuing these objectives, we turned our focus towards established generative programming tools such as Lightweight Modular Staging (LMS)/Scala [21], BuildIt/C++ [126], and AutoGraph/Python [127]. These tools have demonstrated success in constructing various high-performance systems with runtime code generation capabilities [19, 81, 127–130]. We selected LMS, one of the most well-established tools in this space, and used it to develop Flan — a Datalog compiler fully embedded in Scala that is capable of producing fully specialized code for any given Datalog program.

A key realization is that these tools (1) provide programmable, fine-grained code generation that allows full specialization of programs (2) facilitates seamless interoperability with host languages (e.g., Scala) in the frontend. In Flan, we leverage this interoperability with Scala to enrich our frontend with various features such as polymorphic rules, higher-order relations [131], user-defined lattices, and so on, by creating an embedding for Datalog in Scala (Section 4.5). Flan's implementation resembles a high-level Datalog interpreter implemented using multiple high-level abstractions (Section 4.3), while, LMS effectively dismantles these abstractions during code generation, thereby eliminating any related runtime costs. Flan's design features a streamlined operator interface which enables effortless support for a host of features including user-defined aggregates, user-defined functions (UDFs), stratified negation, and more (Section 4.4.2). Moreover, this allows support for multiple execution strategies, such as multi-way and binary joins (Section 4.4.3), and different index structures like BTrees and Hash indexes (Section 4.4.4), with minimal effort. While the rest of this chapter presents the system implementation within the Scala/LMS context, we believe that the same can be accomplished within other similar frameworks and languages like Python/AutoGraph and C++/BuildIt (technically, any language with operator overloading).

Evaluation We evaluate the performance of Flan using a set of benchmarks commonly employed in similar studies. Benchmarks include a relatively simple points-to analysis (formulated as relations) on synthetic data, a comprehensive program analysis benchmark from Doop [115] applied to a selection of DaCapo benchmark programs [132], and another involving lattice-based reasoning [117]. We compare against various established Datalog engines, including Soufflé, Crepe [118], Ascent, and Flix. Flan consistently demonstrates competitive performance in comparison to state-of-the-art systems, achieving speedups in the range of 1.4× to 12.5×.

Contributions

- We elucidate the rationale behind our choice of employing generative programming, specifically, LMS, as the basis for constructing Flan (Section 4.2).
- We review essential background and demonstrate the construction of a simple Datalog interpreter from scratch in Scala (Section 4.3).
- We illustrate how to effortlessly transform the Datalog interpreter into a compiler that generates fast, specialized code by utilizing LMS (Section 4.4.1). We demonstrate the integration of various extensions such as constraints, UDFs, negations, and aggregations through streamlined abstractions (Section 4.4.2). We highlight Flan's backend flexibility by adding support for multiple join evaluation strategies and index structures (Sections 4.4.3 and 4.4.4).
- We showcase the Datalog compiler can be seamlessly embedded into a full programming language, capitalizing on existing features of the language (e.g., type system, abstractions) for enabling composable, polymorphic, higher-order Datalog programs. Moreover, we demonstrate how this simplifies the process of enriching Datalog with features such as user-defined lattices (Section 4.5).
- We compare our engine with state-of-the-art Datalog engines such as Soufflé, Ascent, Crepe, and Flix across a diverse range of benchmarks. Flan consistently delivers competitive or superior performance in each benchmark (Section 4.6).

In Section 4.7, we examine the relevant literature, followed by drawing conclusions and discussing potential future work in Section 4.8.

4.2 Why Generative Programming?

In the previous section, we outlined our goal of creating a Datalog engine that delivers performance on par with existing compilers through specialized code generation while preserving the ability to extend both the frontend and backend. This section delves into these concerns, articulating the potential of a generative programming-based approach to effectively address these needs.

Specialization In the context of Datalog engines, specialization refers to the process of generating specific code tailored for the semi-naive evaluation of rules of a given program. There are various methods to achieve this specialization. Soufflé, a highly performant and well-regarded Datalog compiler in the field of program analysis, employs template metaprogramming for this purpose. Consider the code generation logic for a simple fixpoint loop operation. Figs. 4.1 and 4.2 present the corresponding logic in Soufflé and Flan, respectively. Soufflé's code generation is driven by an imperative IR derived from the given Datalog program. Specifically, it involves concatenating a set of pre-written, operator-specific, stringified code templates like the one in Fig. 4.1.

Granularity of Specialization This approach indeed attains a level of specialization, yet it leaves potential for additional specializationand thus, performanceunexploited. Specifically, the translation happens at the granularity of the query operators (e.g., fixed points, search, insert, etc.). Hence, a lot of abstractions (index structures, etc.) remain in the generated code, which is harder for the downstream C/LLVM compilers to reason about. There are several approaches to push this specialization further. One approach is to use full progressive lowering using an extensive compiler with multiple IRs, as seen in tools like DBLAB [36] or potential multiple MLIR dialects [133]. However, this becomes only a partial solution if we need easy extensibility, as adding features would require notable changes to this IR, lowering passes, and so on. Alternatively, fine-grained programmable code generation using tools like LMS (Scala), BuildIt (C++), or AutoGraph (Python) could be a preferable choice.

[2] https://github.com/souffle-lang/souffle/blob/9aca1614a8865476abd681f17544dc9032dbb186/src/synthesiser/Synthesiser.cpp#L557-L566

```
// codegen logic for Loop
void visit_(/* elided */) {
  PRINT_BEGIN_COMMENT(out);
  out << "iter = 0;\n";
  out << "for(;;) {\n";
  dispatch(loop.getBody(), out);
  out << "iter++;\n";
  out << "}\n";
  out << "iter = 0;\n";
  PRINT_END_COMMENT(out);
}
```

Figure 4.1. Code generation logic for fixed point loop in Soufflé. Uses non-hygienic string code templates that are syntactically ill-formed.[2]

```
// next-stage variables
var iter = 0   // : Rep[Int]
var cond = ... // : Rep[Boolean]

// staged loop since next-stage cond
while (cond) {
  /*
  logic for loop body
  */
  cond = /* update condition */
  iter += 1
}
```

Figure 4.2. Flan's fixed point loop code. Scala essentially serves as a macro system, ensuring code generation is syntactically well-formed and hygienic.

Lightweight Modular Staging For instance, LMS provides a way of controling this specialization through types. Specifically, LMS introduces the concept of `Rep` types, representing computations that occur in the next stage and should, therefore, appear in the generated code. Computations (including control flow, etc.) that take place on regular types (e.g., `Int`, `String`, etc.) are evaluated at the current stage. Below, we present a power function written in LMS (left) and the corresponding specialized code (generated in

C) for `power(b, 5)` (right). Notice that `b` is a `Rep`-typed variable, indicating that it is a next stage variable, and hence, any computations performed on `b` (in this case, multiplications) should appear in the generated code.

```
// Scala LMS code
def power(b: Rep[Int], n: Int): Rep[Int] =
  if (n == 0) 1
  else b * power(b, n-1)
```

The Scala program on above is a program written using LMS's `Rep`-types and LMS will automatically partially evaluate the program with respect to the given value of `n` and produce the residual program as shown below.

```
// Specialized code for power(b, 5)
// generated using LMS
int power5(int b) {
  return b * b * b * b * b
}
```

LMS operates by evaluating the program as a standard program for regular-typed values, while constructing a graph-like IR for operations involving `Rep` values [134]. This IR goes through multiple optimizations like dead-code elimination (DCE), code motion, etc. [33, 135] and generates code in the target language.

This offers us a convenient means of controlling the level of specialization. For instance, if we were to utilize an off-the-shelf hash table for implementing our hash-based indexes needed for rule evaluation, we could employ `Rep[HashMap[K,V]]`. This implies that the `HashMap` abstraction will be present in the generated code (e.g., HashMap from the standard library). In fact, as discussed in Section 4.4.4, we adopt this strategy to incorporate Soufflé's BTree index structure [120] into Flan. However, if we had, instead, used a current stage abstraction, e.g., a `HashIndex` that uses `Rep[Array[V]]` internally, then this abstraction would not appear in the generated code. Instead, fully specialized versions of each method invocation (insert, contain, etc.), that uses native arrays would be present in the final code. In our experiments (Section 4.6.3), we discovered that such a fully-specialized index implementation can be an order of magnitude faster than a library-based, generic, off-the-shelf implementation.

Principled Code Generation Another advantage of using an existing tool like LMS to perform runtime code generation is that the operator logic can be completely decoupled from code generation, and implemented in a regular manner. For instance, as is well known, relying on pre-written code templates (like in Fig. 4.1) tends to be brittle, as the developer must manipulate these string fragments to produce the final code. These fragments provides no guarantee of syntactic well-formedness [136] and hygiene [137], leaving room for inadvertent variable capturing and name conflicts (e.g., reusing the same name `iter` in another template during `dispatch` call). Handling even minor syntactic details, such as curly braces, requires specific consideration. While this may appear relatively tractable in a simple setting, maintaining and coordinating such templates becomes increasingly intricate as the number of templates grows and is fundamentaly at odds with extensibility. In contrast, a key distinction in Flan's case (Fig. 4.2) is the handling of variable scoping, typing, etc. by the host language (Scala, in our case).

Flexibility All this results in an implementation that mirrors a relatively simple interpreter, constructed in a high-level language employing high-level abstractions, as exhibited in Section 4.3. This architecture becomes the key in achieving backend flexibility, given that these high-level abstractions enable the creation of a streamlined operator interface that is flexible enough to facilitate the addition of various features like user-defined aggregates, functions, negations, constraints, and so on, (Section 4.4.2) alongside support for different indexing structures (tree or hash-based; Section 4.4.4) and evaluation strategies (binary or multi-way joins; Section 4.4.3).

While this approach gives us a flexible and efficient backend, the question of crafting a flexible frontend remains. One possibility is to solely depend on a textual frontend and incrementally add functionality/extensions as required. However, given the demand for full-programming language like extensions, this would eventually result in an effort similar to creating a new language from scratch. A key realization is that we can use the interoperability with the host language of tools like LMS to our advantage. Building on this concept, and inspired by lots of prior work on embedded logic DSLs, in Flan, we devised an embedding for Datalog using standard Scala (Section 4.5). This utilization of Scala infrastructure significantly reduces the necessary engineering effort. A key distinction between our embed-

`y := &x`	$\mathit{pointsTo}(y, x) :- \mathit{addressOf}(y, x).$	(4.1)
`y := z`	$\mathit{pointsTo}(y, x) :- \mathit{assign}(y, z), \mathit{pointsTo}(z, x).$	(4.2)
`y := *x`	$\mathit{pointsTo}(y, w) :- \mathit{load}(y, x), \mathit{pointsTo}(x, z), \mathit{pointsTo}(z, w).$	(4.3)
`*y := x`	$\mathit{pointsTo}(z, w) :- \mathit{store}(y, x), \mathit{pointsTo}(y, z), \mathit{pointsTo}(x, w).$	(4.4)

Figure 4.3. Simplified points-to analysis rules written in Datalog (used as a running example in Section 4.3)

ding and previous work lies in our integration with code generation — thus becoming an extensible, *compiled* embedded logic DSL. This strategy also reaps additional benefits, such as utilizing Scala's type system for automatic basic type checking, including appropriate use of variables in custom aggregates, user-defined functions and rules, etc.

4.3 A Datalog Interpreter

An initial approach to developing a Datalog engine could involve converting a Datalog program into a collection of relational queries to be run on an SQL query engine, as explored by Scholz *et al.* [138]. To implement the efficient semi-naive evaluation necessary for Datalog, we would need to: (1) represent each relation as three corresponding tables: '*base*', '*delta*', and '*next*' (2) establish an external driver loop to the query engine to enforce fixpoint semantics and manage records across the sub-relations. However, such trivial implementations usually perform more poorly than systems specifically designed for Datalog [2] unless special care is taken [139, 140]. The main challenge lies in creating and maintaining carefully selected indices (e.g., moving tuples from *next* to *delta*, etc.), and avoiding redundant computations arising from semi-naive evaluation. Nonetheless, it has been shown that building an optimized SQL engine can be done in 500 Lines of Code (LOCs) [81, 141]. Although this engine cannot be repurposed directly due to the limitations mentioned, it prompts the question: Can we adopt similar principles to develop an efficient Datalog interpreter?

In this work, we demonstrate that the answer is indeed yes! Inspired by Rompf and Amin [141]'s SQL engine, we will develop a succinct bottom-up Datalog interpreter in this

$$
\begin{aligned}
&c \in \mathsf{B},\ x \in \mathsf{Var}, R \in \mathsf{Rel}\\
t &::= c \mid x\\
a &::= R(t_1, \ldots, t_n)\\
r &::= a \mathrel{:-} a_1, \ldots, a_n\\
P &::= r_1, \ldots, r_n
\end{aligned}
$$

```scala
/* Auxiliary definitions for Relation */
case class Schema(fields: Seq[Field])
class Relation(val name: String, val schema: Schema)

/* AST definitions */
type BaseTy = Boolean   Int   String
enum Term:
  case Const(v: BaseTy)
  case Var(s: String)

case class Atom(relName: String, args: Seq[Term]):
  def rel: Relation = ...

case class Rule(head: Atom, body: Seq[Atom])

type Program = Seq[Rule]
```

Figure 4.4. Formal syntax for Datalog programs (top) and corresponding AST definition in Scala (bottom)

section, utilizing semi-naive evaluation as illustrated in standard database textbooks [142, Figure 3.6] [143, Algorithm 13.1.2]. For the sake of brevity, our initial focus will be the pure Datalog subset, excluding negations, aggregations, and other extensions. Nevertheless, the integration of these features is fairly straightforward and will be discussed in Section 4.4.2. We will use Scala 3 syntax throughout the chapter.

4.3.1 Datalog

A Datalog program consists of a set of clauses of the form $a :- a_1, a_2, \ldots, a_n$. Each clause is composed of a head atom a and body atoms a_{i}, where each atom possesses an associated arity n and a corresponding number of term arguments t_{i}, denoted as $R(t_1, \ldots, t_m)$. A term argument can be either a variable x or a constant c of a base type B. The head atom of a rule defines facts of a relation while body atoms form a conjunctive query over multiple relations constrained by a sequence of arguments. The predicate symbol R associated with the atom is referred to as the relation. There are primarily two types of relations: *Extensional* relations (EDB), the base relations that serve as the input and *intensional* relations (IDB), intermediate or output relations derived by applying the rules in the Datalog program.

We show the formal syntax and Scala abstract syntax definition in Fig. 4.4. The auxiliary definition `Relation` holds metadata of relations. A relation's facts are derived by the rules in which the relation appears in the head atom. A relation has a schema that defines its arity and the column names (`Field`s). The `rel` property performs name resolution and provides the relation an atom refers to.

Consider a basic program analysis task formulated using Datalog shown in Fig. 4.3 [144]. Specifically, this is a simplified version for Andersen-style pointer analysis [145], which aims to determine the objects a variable may potentially point to [144]. This analysis involves four EDB relations: *AddressOf*, *PointsTo*, *Load* and *Store*. We will use this as a running example in the rest of Section 4.3. The rules are reasonably clear and self-explanatory. For instance, rule 4.1 indicates that if `y` holds the address of `x` (i.e., `y := &x`), then `y` points to `x`.

The actual evaluation of the clauses relies on fixed-point semantics. In this process, the rules are applied to facts, initially starting from EDBs and subsequently with IDBs as they are derived, until the evaluation reaches a fixed point. This method is referred to as naive evaluation. However, it is inefficient due to the presence of numerous redundant computations. For instance, in rule 4.2, any tuples of *PointsTo* found prior to the previous iteration would not result in new tuples during the current iteration (since *Assign* is fixed). Consequently, it is more efficient to operate only on the tuples discovered in the immediate

```
/* Entry point of the interpreter */
def compute(prog: Program,
            edbs: Map[Relation, Seq[Record]],
            outputs: Seq[Relation]): Map[Relation, Array[Record]] =
  // construct dependency graph of relations, find
  // strongly-connected components, and topologicaly sort
  val strata = stratify(prog)

  // store: Stores tuples of relations
  val store: Store = Store()
  for ((rel, recs) <- edbs)
    // Load EDB relation tuples into store
    recs.foreach(store.insert(into=rel, _))

  // Compute IDB by evaluating rules
  for (rules <- strata)
    eval(rules)(store)

  // Collect outputs and return
  outputs.map(rel => (rel, store.records(rel))).toMap

/* Evaluating rules of a stratum */
def eval(rules: Program)(store: Store): Unit =
  // Relations of the stratum
  val relations = rules.map(_.head.rel).toSet
  // Split into simple and recursive, and add delta variants for
   recursive
  val (simpleRules, recursiveRules) = expandRules(rules)

  // Evaluate simple (non-recursive) rules
  for (rule <- simpleRules)
    for (record <- evalRule(rule))
      // Project join output and insert
      store.insert(into=rule.head.rel, record)

  // Evaluate recursive rules until fixed point
  while store.hasNextIteration(relations) do
    for (rule <- recursiveRules)
      for (record <- evalRule(rule)(store))
        // Project join output and insert
        store.insert(into=rule.head.rel, record)
```

Figure 4.5. Main interpreter loop that drives rule evaluation. `compute` partitions the relations and rules into strata, and call `eval` to evaluate each stratum.

$$\textit{pointsTo}.\mathsf{next}^1(y,x) \quad :- \ \textit{addressOf}(y,x) \qquad \text{(Rule 4.1 expansion)}$$

$$\begin{aligned}
\textit{pointsTo}.\Delta^{\mathrm{i}} &:= \textit{pointsTo}.\mathsf{next}^{\mathrm{i}-1} - \textit{pointsTo}^{\mathrm{i}-1} \\
\textit{pointsTo}^{\mathrm{i}} &:= \textit{pointsTo}^{\mathrm{i}-1} \cup \textit{pointsTo}.\Delta^{\mathrm{i}} && \text{(Rule 4.2 expansion)} \\
\textit{pointsTo}.\mathsf{next}^{\mathrm{i}}(y,x) &:- \ \textit{assign}(y,z), \textit{pointsTo}.\Delta^{\mathrm{i}}(z,x)
\end{aligned}$$

Figure 4.6. Rule expansion for rules 4.1 and 4.2 from Fig. 4.3 done by `expandRules` (called in Fig. 4.5 L 27). The first rule's expansion is simple and can be evaluated only once outside the fixpoint loop while the second one is recursive and is decomposed into a rule over delta relations.

previous iteration (referred to as the *delta*). This strategy of employing deltas for fixed-point computation is known as semi-naive evaluation and is regarded as the state-of-the-art evaluation algorithm for Datalog.

4.3.2 Stratification

We now proceed with the definition of the interpreter. The evaluation of a Datalog program P can be decomposed into the evaluation of subprograms P_{i} where $\{P_1, \ldots, P_n\}$ is a partitioning (stratification) of the rules in P [143]. The partitioning is obtained by computing the topological sort of the Strongly connected components (SCCs) of the (cyclic) dependency graph defined by the head relation and body relations of each rule. Each SCC i is a set of relations that are mutually recursive and P_{i} is defined as the set of rules defining any relation in SCC i. For instance, for our running example, there would be five strata: one for each EDB relation (*addressOf*, *assign*, *load*, and *store*) and another for the IDB relation *pointsTo*. The last stratum contains all four rules. In Fig. 4.5 we show `compute`, which is the entry point of our Datalog interpreter. First the program given as input is stratified through `stratify` (L 7), then records for each EDB are loaded into a `store` (L 11 and 13) and finally the interpreter evaluates the rules for each stratum in the topological order.

$$pointsTo(y, w) :- load(\overset{sym}{y}, \overset{ptr}{x}), pointsTo(\overset{src}{x}, \overset{dst}{z}), pointsTo(\overset{src}{z}, \overset{dst}{w}).$$

```
for (x <- uniqueValues(of=ptr, from=load,     having={}))
  for (z <- uniqueValues(of=dst, from=pointsTo, having={src: x}))
    for (w <- uniqueValues(of=dst, from=pointsTo, having={src: z}))
      for (y <- uniqueValues(of=sym, from=load,     having={ptr: x}))
        yield (y, w)
```

Figure 4.7. Top: Datalog rule 4.3 from Fig. 4.3 with column names for each relation shown on top of each variable. Bottom: Corresponding multi-way join nested loop.

4.3.3 Stratum Evaluation

Typically, a stratum comprises multiple rules, and the interpreter's core functionality lies in how these rules are evaluated, which corresponds to `eval` in Fig. 4.5. Stratum rules are partitioned into simple and recursive rules. Simple rules are non-recursive rules, i.e., the body does not mention any of the relations present in the current stratum, and therefore can be computed immediately outside of the fixpoint computation. Recursive rules, by contrast, need to be evaluated until a fixpoint is reached. The evaluation of a rule yields records that can then be stored as shown in L 33 and 40.

To implement semi-naive evaluation, recursive rules need to be expanded by `expandRules` in L 27 into rules operating on three sub-relations: *base*, *delta*, and *next*. The *base* sub-relation contains all records discovered prior to the current iteration, while the *delta* sub-relation includes only the records found in the previous iteration. The *next* sub-relation holds the records derived during the rule evaluations of the current iteration. Fig. 4.6 illustrates expansions for rules 4.1 and 4.2 in the points-to program from Fig. 4.3. For rule 4.1, the `expandRules` function results in one simple rule with only *addressOf* in the body, which can be evaluated once outside the fixpoint loop. For rule 4.2, it results in a rule with a join between *assign* and the delta relation of *pointsTo*. The notation := represents table updates performed by `store.hasNextIteration(relations)` (L 36), which computes the stable, next, and delta relations for the new iteration, as shown by the set operations.

```
 1 /* body: remaining body, iters: remaining order,
 2    env: current environment */
 3 def multiWayJoin(body: Set[Atom], iters: Seq[JoinOperand],
 4                 env: Map[Var, Value])
 5                 (store: Store) = new:
 6   def foreach(f: Map[Var, Value] => Unit): Unit =
 7     iters match
 8       case Seq(op, rest*) =>
 9         // lookup bound fields
10         val (filter,missingFields) = lookup(op.atom,env)
11         // iterate variable
12         for (value <-store.uniqueValues(
13                                   of=op.field,from=op.atom.rel,filter))
14           val newEnv = env + (op.arg -> value)
15           // remaining: has unbound columns
16           // ready: all columns bound
17           val (readyIncl,remaining) =
18             body.partition(lookup(_,newEnv)._2.isEmpty)
19           val ready = readyIncl - op.atom
20           if (ready.forall(
21                     a => store.contains(a.rel,lookup(a,newEnv)._1)))
22             // perform rest of join
23             multiWayJoin(remaining,rest,newEnv)(store).foreach(f)
24    case _ => f(env) // join completed
25
26 /* main entry-point for rule evaluation */
27 def evalRule(rule: Rule)(store: Store) = new:
28   def foreach(f: Record => Unit): Unit =
29     // compute the order to perform the join
30     val order = variableOrder(rule.body)
31     for (env <- multiWayJoin(rule.body.toSet, order, Map())(store))
32       // project to rule head
33       val (record, _) = lookup(rule.head, env)
34       f(record)
```

Figure 4.8. Code for evaluation of rules. First, the variable order is computed (`order`), followed by the invocation of `multiWayJoin` that iterates through variables in the specified order and produces the join output. Code in L 12 highlighted in `blue` corresponds to the nested loops in Fig. 4.7.

4.3.4 Rule Evaluation

Next, we will explore the evaluation of rules. Broadly speaking, Datalog rule evaluation strategies fall into two categories: binary joins and multi-way joins. While binary joins typically involve iterating through entire tuples from relations, multi-way joins function at the granularity of individual variables. Our full system, Flan is capable of accommodating both strategies with only slight modifications to the core evaluation logic (discussed in Section 4.4.3). For the sake of conciseness, this section will exclusively present and discuss the code for a multi-way join strategy.

As an example, let's consider rule 4.3 from Fig. 4.3, which performs a join across three relations predicated on the keys x and z (as depicted in Fig. 4.7). The bottom part of the figure illustrates an analogous 'for comprehension' that emulates the logic of join evaluation. `uniqueValues` yields the unique values of the field passed as argument to the parameter '`of`' from relation '`from`' where the remaining field are constrained by the '`having`' record (essentially a filter). For readers who are familiar with the Generic Join [146] algorithm, this may look similar but without the intersections. We discuss how a similar notion to intersections is achieved in this setting via explicit contains checks in Section 4.4.3, but omit it here for brevity.

For a given rule, the initial step involves computing the order in which variables (along with their corresponding relations) should be iterated. Upon establishing this order, the `uniqueValues` method can be invoked from the pertinent relations in the determined order, as illustrated in Fig. 4.7. It is important to note that while the sequence of iterating over the variables does not influence the correctness of the computation, it can have a substantial impact on the execution time of the query.

For completeness, the implementation for rule evaluation is shown in Fig. 4.8. The entry point is `evalRule`, which computes a `variableOrder` (L 30) and then calls `multiWayJoin` which corresponds to the evaluation of the nested for loops generating the unique values for each variable and introducing them in `env`. Specifically, L12-23 of Fig. 4.8 corresponds to evaluating the rule using a loop nest, such as Fig. 4.7. The loop nest iterates over each join variable, with relation lookups interspersed at appropriate places. `multiWayJoin` returns an object implementing `foreach` allowing to iterate over the joined records through a simple for comprehension. Finally the loop body `f` is called with the projection of the environment corresponding to the rule's head record.

Close attention should be paid to the conditional present in L 21: whenever the introduction of a variable of the rule completes one of the rule's body atoms, an contains check takes place, essential to produce the correct output. For example if we had picked *ptr* from *load* to yield the possible values of x and *src* from *pointsTo* for the values of z, then we must also check that (x, z) is present in *pointsTo* as shown below.

```
for (x <- uniqueValues(of=ptr, from=load,     having={}))
  for (z <- uniqueValues(of=src, from=pointsTo, having={}))
    if (pointsTo contains (x, z))
      ...
```

In Flan, we mimic the join order (computed by `variableOrder`) of left-associative binary joins with join variables iterated first. However it is straightforward to extend the system to support other query plans based on user provided hints or heuristics like cardinality/selectivity estimation.

4.3.5 Record Store

The last key remaining piece is the `Store` which will store the records that are produced during evaluation, ensure uniqueness of records, and enable fast retrieval of the unique values present in a specific field of the relation. Its implementation is shown in Fig. 4.9.

The `records` map contains the records of a given relation. The `insert` and `contains` method allow to perform insertions of a record into a relation (Fig. 4.5, L13, 33 and 40) and check the presence of a record in a relation (Fig. 4.8, L21), respectively. `hasNextIteration` checks whether a fixpoint has been reached (Fig. 4.5, L 36) and performs the $:=$ updates shown in Fig. 4.6. Finally `uniqueValues` iterates over the unique values of a column of a relation on the specific subset of records satisfying the filter (`having`), as discussed in Section 4.3.4 and shown in Fig. 4.8, L 12. The implementation of the `Store` presentend is deliberately simple and inefficient. We defer the discussion of design decisions of the `Store` to Section 4.4.4. In particular, we introduce the notion of `IndexedStore`s, which employs

```
case class Value(v: BaseTy)
case class Record(schema: Schema, values: Seq[Value])

class Store:
  // records for each relation are maintained in Sets
  val records =
   mutable.Map().withDefault((_:Relation) => Set[Record]())

  // inserting new records
  def insert(into: Relation, record: Record): Unit =
    records(into) = records(into) + record

  // checks existance of records
  def contains(rel: Relation, record: Record): Boolean =
    records(rel) contains record

  def hasNextIteration(relations: Set[Relation]): Boolean =
    var updated = false
    for (relation <- relations)
      // Perform the updates shown in Rule 4.2 expansion from Fig. 4.6
      records(relation.delta) =
          records(relation.next) diff records(relation.base)
      records(relation.base) =
          records(relation.base) union records(relation.delta)
      records(relation.next) = Set()
      updated  = records(relation.delta).nonEmpty
    updated // returns whether a fixed point has been reached

  def uniqueValues(of: Field, from: Relation,
                   having: Record): Iterable[Value] =
    records(from).filter(
           rec => rec(having.schema) == having.values).map(_(of))
```

Figure 4.9. Code for `Store`, which maintains tuples of relations and is used throughout the evaluation.

various index structures such as trees or hash tables to execute the required operations efficiently.

Combining all of these components results in a relatively simple Datalog interpreter. Although easy to construct, its performance is significantly inferior to that of a compiled engine generating specialized code. This is due to the substantial runtime interpretation overhead, such as the overheads associated with high-level abstractions and using generic data structures for indices. In Section 4.4, we will explore how to transform this basic interpreter into a compiler capable of producing efficient specialized code, requiring only minimal modifications to the original code.

4.4 Flan: Datalog Compiler

4.4.1 Deriving a Datalog Compiler from the Interpreter

In Section 4.3, we demonstrated how to construct a relatively simple Datalog interpreter in Scala. In this section, we explore how to transform our slow interpreter into a compiler that generates fast, specialized code with minimal modifications to the original interpreter. We will employ the concept of partial evaluation and Futamura projections [38] to achieve this goal.

Partial Evaluation The concept of partial evaluation [147] involves decomposing the evaluation process into two or more stages, often based on the availability of inputs, with each stage evaluated to generate a residual program that contains the logic for evaluating the remaining stages. Consider our interpreter as an example: it accepts a Datalog program and the actual input data (i.e., EDB) as inputs and produces an output.

```
output = interpreter(datalog_program, input)
```

In the context of program analysis, the same `datalog_program` implementing the analysis is applied to multiple different programs to be analyzed. Consequently, it is sensible to divide the interpreter into two stages. Initially, we partially evaluate the Datalog interpreter with respect to the given Datalog program, obtaining a staged interpreter specialized for that specific analysis. This process is facilitated by a program specializer (or partial evaluator). Ideally, the specialized code should eliminate any overheads associated with interpreting the Datalog program while maintaining the ability to handle dynamic input EDBs.

```
specialized = specializer(datalog_interpreter, datalog_program)
output = specialized(input)
```

The first Futamura projection [38] states that executing the interpreter produces the same output as evaluating the specialized (i.e., partially evaluated) code with the same input. Furthermore, it states that this process of specialization is analogous to compilation. In essence, the `specialized` code mentioned above would be equivalent to code generated by a Datalog compiler.

One way to achieve this specialization is to create a custom program transformation tailored to the use case (i.e., Datalog) that generates the required `specialized` code for a given input Datalog program. However, this can result in a notable engineering effort as it is essentially equivalent to writing a compiler from scratch. Instead, we can rely on an existing generative programming framework like LMS to do this in a more principled manner as discussed in Section 4.2.

Our Approach using LMS As mentioned in Section 4.2, LMS leverages a type-based approach to facilitate this stage distinction and specialization. To achieve this, it introduces the notion of `Rep`-types. `Rep`-typed variables (e.g., `Rep[Int]`, `Rep[Array[Long]]`, etc.) designate them as next-stage values. Consequently, any operations conducted on these variables would appear in the generated code for next stage. In contrast, operations on variables with regular types (e.g., `Int`, `Array[Int]`, etc.) are executed in the current stage. LMS takes regular Scala programs with `Rep`-type annotations as input, partially evaluates it using the Scala runtime, and subsequently generates specialized C code for the residual program.

In Fig. 4.10, we illustrate the modifications necessary to transform the interpreter, discussed in Section 4.3, into a compiler using LMS. The segments of code that remain unaltered are depicted in gray. First, we should identify static data available at staging, or in this case, code generation time. The primary available component is the Datalog program which contains the relation definitions along with their corresponding schema, input/output specifications, etc. Consequently, at staging time, we can compute the program's strata, topological order, recursive and non-recursive rules, join orders, and so on. Hence, in the LMS-based staged interpreter, computations pertaining to these aspects are carried out at staging time, leading to specialized code that eliminates any runtime overhead associated with these operations.

```
def stratify(prog: Program): Seq[Program] = ...
def compute(prog: Program,
            edbs: Map[Relation, Seq[Record]],
            outputs: Seq[Relation]): Map[Relation, RepBuffer] = ...
def eval(rules: Program)(store: Store): Unit = ...
def evalRule(rule: Rule)(store: Store) = ...
def multiWayJoin(body: Set[Atom], iters: Seq[JoinOperand], env: Map[Var,
    Value])
                (store: Store) = ...

case class Record(schema: Schema, values: Seq[Value])

case class Value(v: Rep[_])

class Store:        // staged store implementation
  val records =
    mutable.Map().withDefault((_: Relation) => RepIndexedBuffer())

  def insert(into: Relation, record: Record): Unit
  def contains(rel: Relation, record: Record): Rep[Boolean]
  def hasNextIteration(relations: Set[Relation]): Rep[Boolean]
  def uniqueValues(of: Field, from: Relation, having: Record): RepBuffer
```

Figure 4.10. Deriving a compiler from the interpreter via mixed-stage Store. Shown in gray are the previous definitions of the interpreter from Figs. 4.5 and 4.8 which are left unchanged. The only changes needed are: (1) `Value` s types are now `Rep[_]` instead of Scala base types, denoting second-stage values (2) the store is updated to an implementation of a second-stage store (we elide the implementation).

The sole piece of information not available at the staging time are the actual facts (i.e., records) of the EDB relations. Consequently, the values and data structures associated with these should be marked as next-stage values (i.e., `Rep` types). In particular, we update `Value` to contain `Rep[_]` values denoting next-stage values. This, along with other fixes related to type errors that arise from this change, are sufficient to convert our interpreter into a compiler that generates specialized code.

Finally we now need to use a staged version of `Set[Record]` in the records store (`Store` in Section 4.3.5, since now it contains next-stage values) and update its interface

```
/* evaluate UDFs, aggregates and return result */
def evalArg(/* ... elided ... */): Value =
  case v: Variable => env(v)
  case Constant(v) => Value(v)
  case UDFCall(str, args) => evalUdf(str, args)
  case agg: Aggregator => evalAggr(agg)

/* evaluate all ready constraints and return whether all satisfied */
def evalReady(ready: Seq[Litereal],
              env: Map[Variable, Value]): Rep[Boolean] =
  ready.filter{case _: Assignment => false case _ => true}.forall{
  case Negation(atom) => !store.contains(atom.rel, lookup(atom, env).
 _1)
  case a: Atom => store.contains(a.rel, lookup(a, env)._1)
  case bc: BinaryConstraint => evalBinaryConstraint(bc, env) }

```

Figure 4.11. `evalArg` and `evalReady` used in Fig. 4.12.

accordingly. For example, `contains` checks will now return `Rep[Boolean]` and be present in the residual program. The concrete implementation of `Set[Record]` is done through `RepIndexedBuffer`. We elide the implementation and defer a thorough discussion on implementation considerations in Section 4.4.4.

LMS readily offers staging support for most primitive operations involving `Rep`-typed values. For instance, LMS provides out-of-the-box support for assignments, comparisons, and other operations on primitive `Rep[T]` values. Additionally, language constructs such as `if`, `while`, and `for` involving next-stage values are transformed into their staged counterparts using macros [134]. For example, when a `Rep[Int]` is compared with another next-stage value or a regular integer, it results in a `Rep[Boolean]`. If a conditional depends on this boolean value, the corresponding if block will be lifted into a staged if, which will ultimately be included in the generated code.

```
/* process any ready components before moving to next join step */
def processReady(remaining: Seq[Literal], env: Map[Variable, Value])
                (f: (Seq[Literal], Map[Variable, Value]) => Unit)
                (store: Store) =
 // ready: components with all vars in env
 val (ready, nextRemaining) =
      remaining.partition(allVarsAvailable(_, env))

 // assignments that will introduce vars to env
 val readyAssignments =
      ready.collect { case a: Assignment => a }

 // process all ready constraints
 if (evalReady(ready, env))
  readyAssignments.headOption match
   case Some(assign) =>
    // process UDFs, aggregates, and put result into env, and repeat
    val value = evalArg(assign.right, env)
    processReady(nextRemaining ++
            readyAssignments.tail, env + (assign.left -> value))(f)
   case None =>
    f(nextRemaining, env)

def join(/* elided */) = new:
 def foreach(f: Map[Var, Value] => Unit): Unit =
  processReady(body, env) { (remaining, newEnv) =>
   // process any available UDFs, aggreagtes constraints, etc. prior to
   join step
   /* remain unchanged from Fig. 4.8 */
   iters match
    case Seq(op, rest*) => ...}
```

Figure 4.12. The required code updates in Flan to accommodate negations, constraints, UDFs, aggregates, etc. Components are processed in `processReady`, which utilizes `evalReady` and `evalArg` (shown in Fig. 4.11) for evaluating negations and constraints, and for managing constants, UDFs, and aggregates respectively.

4.4.2 Beyond Pure Datalog

Up to this point, we have developed an interpreter that supports pure Datalog and demonstrated how to transform it into a compiler that generates efficient specialized code. However, pure Datalog is not adequate for most practical program analysis tasks. For instance, the Doop analysis, one of the benchmarks we will examine in Section 4.6, employs various extensions such as stratified negation, aggregation, UDFs, constraints, and more. In this section, we will discuss how to incorporate these features into our Datalog compiler. Integrating these features is relatively straightforward because we can utilize high-level Scala capabilities to implement them (as we saw in Section 4.3) without concern for code generation or any runtime overhead associated with the abstractions.

Stratified Negation The concept of negation in Datalog serves to enforce that a given relation does not contain a specific value. Consider the following example of a rule with negation:

```
R(x, y) :- S(x, y), !T(x, y)
```

This rule states that we should insert all records from S that are *not* present in T to R. To ensure safety and unique fixed-point semantics, negation must be stratified, which means that negation can only be applied to a relation that appears in a previous stratum (in the topological order) [148]. During rule evaluation, negations are translated into simple contains checks. Specifically, when all variable bindings of a negated atom become available, we can perform a `contains` operation using the relevant index.

Aggregations, UDFs, and Constraints The example shown below presents a simple rule that incorporates a combination of features including aggregates, constraints, and User-Defined Functions (UDFs). The rule aims to find the count of objects that a variable may point to for variable names that match a given regex. Here, `match` is a UDF, `count {PointsTo(var, _)}` is an aggregate, and `match(..) = true` is a constraint.

```
PointsToCount(var, p_count):-
  Vars(var),
  match("<some-regex>", var) = true,
  p_count = count {PointsTo(var, _)},
```

The common trait in the evaluation of these extensions is that each component is processed as soon as the required variables become available in the environment. Thus, all we need to do is to modify the join logic to initially handle these 'ready' components before proceeding with the remainder of the join. To keep the logic for evaluating these extensions simple and general, we introduce the notion of rule canonicalization. Specifically, we rewrite the rules so that atoms in their body contain only variables, while other argument types are converted into variable assignments. Additionally, aggregates and UDF calls are hoisted into assignment operations using new variables. Shown below is the canonicalized version of the above query.

```
// canonicalized version of rule on left
PointsToCount(var, p_count):-
  Vars(var),
  #udf1 = true,
  p_count = #aggr1,
  #const1 := "<some-regex>",
  #udf1 := match(#const1, var),
  #agg1 := count {PointsTo(var, _)}
```

Figure 4.12 shows the updated `join` function that incorporates the `processReady` method for handling the extensions. It takes as arguments the current environment and the remaining body literals, partitioning them into two categories: `ready`, encompassing literals with all necessary variables present in the environment, and `nextRemaining`, which includes the rest. For instance, in the initial join call for the rule above, `#const1` qualifies as a ready literal as it doesn't require any additional environment variables. The function `evalReady` then executes any pending constraint checks (like negations, binary constraints), and only if all constraints pass, the evaluation of the rest of the rule proceeds. Note that `evalReady` yields a `Rep[Boolean]`, indicating that constraint evaluation occurs in the next stage and the logic will feature in the generated code.

We treat `Assignment`s distinctively; they introduce variables into the environment that could potentially render some remaining literals 'ready'. Therefore, we process one `Assignment` at a time, repeatedly employing the aforementioned process until all 'ready' components have been handled. We omit the code for evaluating binary constraints (`evalBinaryConstraint`) which simply looks at the constraint type and evaluate left and right hand side and compare based on the constraint. We also exclude the code for calling UDFs (`evalUDF`), which simply retrieves the corresponding UDF from the UDF map (where user can register their UDFs) and call it.

Flan fully supports user-defined aggregates. All users need to specify are an initial value for the aggregate and an update function, having type `(Value, Value) => Value`. The aggregate's body is evaluated using the same `join` function, starting with an initial environment where all free variables are bound by the corresponding outer variables. This also allows arbitrary nesting of aggregates. For brevity, we have chosen to exclude the code for `evalAggr(agg)` but it largely reuses already existing methods like `variableOrder` to determine the join order for aggregate sub query, `join` to perform the join, and `evalArg` for evaluating arguments (e.g., UDFs) used inside the sub query.

One significant advantage of maintaining high-level engine code in the interpreter style is the relative ease of adding new features, as seen above. In contrast, other existing approaches may necessitate more extensive modifications, such as augmenting their intermediate-level IRs, adding logic to their code generators, and more. We evaluate this ease of implementation in Section 4.6.2.

Incorporating functionality in this manner not only simplifies the process but also automates certain optimizations that other engines perform as separate passes. For instance, existing approaches rely on transformation passes on their imperative IR for optimizations such as hoisting aggregates, hoisting if blocks (or predicate pushdown), and collapsing filters [2]. In contrast, our interpreter inherently integrates these optimizations by design. For instance, aggregates, constraints, and UDFs are computed as soon as their variable bindings become available, and execution is short-circuited as early as possible.

4.4.3 Join Strategies

We initially outlined the implementation of multi-way joins in Section 4.3.4 when we introduced the code for our interpreter. In this section, we will delve further into the specifics, illustrating how we can facilitate both binary and multi-way joins seamlessly. We evaluate the performance of these strategies later in Section 4.6.3.

A core aspect of executing Datalog programs involves performing joins. Most existing Datalog engines rely on *relation-at-a-time* joins [2, 3, 119], which entail performing nested loop joins by iterating tuples from different relations at each level. This approach is analogous to binary joins in traditional relational database management systems (DBMS), with the key difference being that intermediate relations are not materialized. However, as well known in DBMS research [149], using this type of binary joins for multiple relations can be asymptotically suboptimal in some cases. An alternative approach is to use *variable-at-a-time* joins, which leads to a class of join algorithms called worst-case optimal joins (WCOJ) [109, 149–151]. In essence, the asymptotic complexity of these join algorithms are bounded by the worst-case output size of the *final* result, as opposed to the worst-case size of the *intermediate* results, as seen in binary joins.

The effectiveness of join strategies is influenced by several factors, including the cardinalities of relations, join selectivity, etc. Given this, prominent DBMS engines offer a range of join evaluation methods, using certain heuristics to select the most appropriate one [152, 153]. This underlines the necessity for Datalog engines to incorporate an array of built-in join strategies. Bearing this in mind, we implemented both join types in Flan. Given the abstractions previously defined, this addition was relatively simple.

We can make Flan seamlessly support both strategies by modifying a few interface APIs. First, we need to update the `uniqueValues(of: Field, from: Relation, filter: Record)` function to generate unique values across multiple columns simultaneously, rather than one column at a time. We achieve this by changing the '`of`' parameter's type from `Field` to `Seq[Field]`. Then, we update the `variableOrder` logic, which determines the sequence in which variables should be evaluated for the join. For variable-at-a-time joins, this yields a sequence of `JoinOperand`s, each containing a variable and the related field of atom used to enumerate the chosen variable's values. To accommodate binary joins, we merely need a variant of `variableOrder` that returns a sequence of `JoinOperand`s, each including an atom and *all* its associated fields necessary for iteration. This alteration implies that, instead of iterating one field at a time, we would iterate all required fields from an atom in a single join step. This enables the support of both join strategies, without any changes to the rest of the system code. Specifically, we have defined a `trait Strategy` that comprises an abstract `variableOrder` method. Then, we provide two traits`MultiWayJoinStrategy` and `BinaryJoinStrategy`that can be seamlessly mixed-in as needed to activate the required strategy.

Intersections This multi-way join evaluation strategy we saw in Section 4.3.4 closely resembles the Generic Join algorithm [150], with the notable exception of the use of 'intersections', which is crucial for achieving strong asymptotic guarantees and runtime performance. In Generic Join, the set of iterated values for a variable is determined by intersecting the sets of values from all relations containing that variable. We achieve a similar notion to intersections by performing checks to determine whether an introduced variable is unifiable in the rest of the rule body atoms, and short circuits the execution when it is not unifiable. In essence, when a variable is introduced, we examine the other relations to determine if that variable is present before proceeding with the remaining loop nest. We have incorporated these checks in both binary and multi-way join strategies, and our experiments affirm that their inclusion leads to noticeable runtime performance improvements.

Fused Traversals Variable-at-a-time iteration, as employed in multi-way joins, may introduce suboptimal looping structures under certain conditions. Specifically, consider a loop nest that consists of consecutive variable-iterating loops, where each of these loops iterate over variables from the *same* relation without any constraint checks or intersections between them. An example would be a simple rule like $R(a, b) :- S(a, b, c)$, which leads to a loop nest with two loops iterating over `a` and `b`, respectively. In such cases, the more efficient approach is to utilize a 'fused' traversal that combines these two loops, enabling iteration over both `a` and `b` with a single lookup. Our multi-way join strategy incorporates this traversal fusion as a plan-level optimization.

4.4.4 Indices: BTree and Hash Indexes

In Section 4.3.5, we constructed a generic implementation of `Store` without any indexing. However, this approach is not practical as lookups (i.e., `uniqueValues`) and checks (i.e., `contains`) are performance critical operations during execution. Consequently, it is essential to construct efficient indices to support these operations. There are several data structures at our disposal to build these indices. As discussed in prior work [106, 150–152], an indexing structure resembling a logical trie (each level representing a field) is necessary for variable-at-a-time joins. This can be realized through the use of an actual trie [151], or by employing alternative data structures such as hash tables [152] or trees that efficiently manage each level of the trie.

BTrees Our `Store` abstraction is agnostic to the backend index data structure, allowing us to select any suitable backend data structure as long as it supports the required operations. Datalog compilers like Soufflé is already equipped with fast index structure implementations, specifically tailored to Datalog, which have demonstrated impressive performance across a broad spectrum of workloads [120]. Hence, we have written a custom wrapper for Soufflé's BTree and integrated it into Flan to execute the lookups and checks efficiently. In this instance, as the data structure abstraction appears in the generated code, we use type `Rep [SouffleBTree]` for indices. It is important to note that this means the granularity of specialization would not be pushed beyond the BTree abstraction in the generated code.

Although our initial tree-based index implementation yielded good performance, we wanted to explore potential benefits of hash indexes due to their superior asymptotic performance (constant versus logarithmic). Jordan *et al.* [121] has shown that Soufflé BTrees consistently outperform standard library-based hash table indexes across diverse micro-benchmarks and full program analysis benchmarks. We also experimented with using library-based hash table implementations (using `Rep[HashIndex[K,V]]`), and observed similar performance behaviors to what they have demonstrated. A key realization is that these library-based data structures are overly generic (i.e., have room for more specialization), which consequently lead to significant runtime overheads as evident from our experiments in Section 4.6.3.

Fully Specialized Indices An alternative approach to achieve full specialization is to build our own data structures that operate directly on low-level `Rep[T]` values (e.g., `Rep[Array[T]]`), pushing the specialization granularity beyond the data structures like `HashIndex`, `BTree`, etc. We can still utilize high-level Scala data structures for this implementation, but staging will automatically erase all abstractions and operations performed on current stage values (i.e., non-`Rep` typed), resulting in the desired full specialization in the generated code. For instance, hash indices implemented in this way are transformed into sets of native C arrays in the generated code, where all operations (e.g., lookups, inserts, etc.) are inlined whenever they are invoked. This manner of generating fully-specialized code offers more opportunities for additional optimizations (e.g., CSE, loop fusion, etc.) by downstream general-purpose compilers (e.g., gcc -O3) compared to having generic function calls to perform the operations. Furthermore, this approach gives us greater control over the data structure implementation, enabling us to fine-tune it for our specific use case. For instance, instead of storing the actual full key of the filter, we can simply maintain an offset (i.e., tuple id) to the central record buffer that stores all the records for a given relation, resulting in memory savings (also useful in adding support for lattices as discussed in Section 4.5.2).

We implemented another performance-enhancement tweak, specifically aimed at optimizing the transfer of `next` tuples required upon conclusion of each fixed point iteration. The most common approach is to first insert the tuples from `next` into the `base` relations, swap `next` and `delta`, and clear `next` (for all indices) [2]. While this works well for tree-based indices, whose size depends on the actual data stored, our initial experiments suggested that, in some cases, it can result in poor performance for constant-size indices like hash indices, whose size does not directly depend on the actual element count. Specifically, we would end up clearing a large region of memory even if the previous iteration produced very few tuples. For instance, after performing n fixed-point loops, there would be `memset` calls at the end of each iteration (i.e., n sets of `memset` calls), regardless of how many `next` tuples were produced in the prior iteration. In some cases, these calls can become a bottleneck. In our hash indices, we have opted for a different approach, clearing the underlying data structures only when they are nearly full, thus amortizing the cost of clearing the buffers.

Now the question is, if we do not clear the delta hash tables, how to retrieve the delta for the current iteration (needed for semi-naive evaluation) because now the delta hash tables contain delta tuples for multiple previous iterations. To address this, we store the fixed point iteration number at which a given value was inserted within the hash index structure itself. This allows us to store values for deltas of all iterations in the same index and recover the last delta needed for semi-naive evaluation. Consequently, `next` simply becomes a watermark in the relation's record buffer, and we avoid clearing large memory regions that would consume a significant amount of time, particularly when there is a large number of fixed point iterations where each iteration takes a smaller amount of time (i.e., produces very few `next` tuples).

As discussed in Section 4.6, this finely-tuned, fully-specialized hash-based index demonstrated impressive results, consistently outperforming the tree-based index structure mentioned above. Additionally, akin to our approach with join strategies, this is also implemented in a 'modular' manner. We can easily incorporate the required index type by mixing-in the corresponding trait (`BTreeStore`, `HashStore`), without necessitating modifications to any other part of the codebase.

4.4.5 Identifying Required Indices

So far, we enriched Flan with prevalent Datalog extensions, demonstrated the ease of implementing different join strategies, and explored multiple index structures. Yet, the identification of necessary indexes for operations, a crucial element intersecting all these, remains. Most existing systems execute this step by conducting an analysis pass on their imperative IR [2, 117, 123], but in our case, we do not construct such an IR.

One feasible approach is to simply create all possible indexes for all relations and rely on the dead-code elimination (DCE) pass of LMS to discard any superfluous indexes in the final generated code. This strategy would function as expected because the LMS IR meticulously tracks the effects, thus enabling precise DCE for high-level data structures [135]. However, this approach may not be optimal since it looks at each operation locally, and does not consider the possibility of index sharing, hence, not minimizing index creation globally. Alternatively, we could implement custom logic that computes the necessary indices based on the evaluation strategy. However, in this case, we would have to write this analysis separately for each evaluation strategy and would need to modify it each time new features (e.g., fused traversals as discussed in Section 4.4.3) are introduced, leading to a possible increase in maintenance complexity.

In Flan, we opted for a different approach: we perform a 'dummy' evaluation pass of the program using a `DummyStore` in place of the standard `Store`. The `DummyStore` mirrors the `Store`'s API but, rather than performing actual evaluations, it tracks the types of operations performed on each relation. Further, we ensure it reifies, for instance, both branches of a conditional, by creating dummy nodes in the IR (which don't generate any code), and returning next-stage values (e.g., `Rep[Boolean]` for `contains`).

The process of gathering required indices via dummy evaluation happens in two stages. First, an instance of a `DummyStore` is created and the program is 'evaluated' using the provided execution strategy. Although this uses the actual evaluation logic (i.e., core Flan code), this is not an actual evaluation, instead, this only collects the required indices for each relation. For example, when encountering a call to `store.contains(R, record)`, it will note the need for an index to facilitate a contains check on relation R for the fields `record .schema`. Once this dummy evaluation completes, it will give us the set of required iter indexes (to support faster `uniqueValues`) and check indexes (to support faster `contains`) for each relation.

After recording the operations performed on each relation, we can then create the minimal set of indexes required for these operations taking into account the possibility of index sharing. Since this dummy evaluation pass simply 'runs' the program, it is adaptable to any evaluation strategy, join strategy, and so forth, making it unaffected by changes in other parts of the system.

4.4.6 Parallelization

We adopt an approach similar to Soufflé for parallelization [120]. Specifically, we partition the outermost loop and execute the remaining rule evaluation in parallel using `pthreads`. However, when a new tuple is found, it is crucial to ensure that inserts occur in a thread-safe manner. To achieve this, we augment our hash index implementation to support parallel inserts by utilizing a series of locks, with each lock corresponding to a region of the hash keys (similar to `StripedHashSet` described in [154]). Alternatively, we can achieve thread safety by relying on atomic operations, specifically, atomic fetch-add and compare-and-swap. However, a comprehensive discussion of this implementation is beyond the scope of this chapter and thus omitted. Furthermore, the experiments presented in Section 4.6 that involve parallel execution report the performance for the aforementioned lock-based implementation.

4.5 Cipolla: A Datalog Dialect as Embedded DSL in Scala

In Section 4.4, we have presented the core of Flan, elucidating its ability to yield high performance while offering a broad range of functionalities. Although the textual Datalog frontend we employed (akin to Soufflé) facilitates the use across a variety of cases, as indicated in Section 4.1, certain use cases mandate seamless interoperability with a fully-fledged programming language.

The declarative nature of Datalog and its fixpoint semantics make it ideal for writing program analyses in a concise and clearer manner than writing them in a general-purpose language. Moreover, when coupled with an efficient engine, these analysis can scale to extremely large programs and be easily parallelized [2, 115]. Nevertheless, a growing amount of works [3, 4, 131, 139] is showing that it is possible use Datalog in a more diverse set of analyses, when the language is extended with additional features. In particular, Madsen *et al.* [3] have identified the lack of support for UDFs and lattices as significant obstacles in applying Datalog to a wider range of dataflow analysis problems. Additionally, the lack of integration with existing ecosystems complicates and renders interacting with Datalog computations inefficient, necessitating serializing queries and input/output data back and forth. To overcome these, they propose Flix, a full-fledged programming language with *first-class* support for Datalog. Bembenek *et al.* [4] demonstrate that combining Datalog with Satisfiability-Modulo Theory (SMT) solvers enables writing more advanced and optimized SMT-based analyses. They propose Formulog, an extension of Datalog featuring a first-order fragment of ML and support for SMT formulas.

On one side we have highly-optimized Datalog query engines, where extensions can only be implemented by compiler writers. On the other, we move closer to general-purpose programming languages featuring Datalog-like declarative subsets, which prioritize modularity and abstraction at the expense of performance and require major engineering efforts to design and develop.

In this section, we reconcile these requirements, by embedding Datalog rules into LMS enabling abstraction without performance costs through *linguistic reuse* (technically, shallow) of host language (Scala) features. We will use concrete examples that have motivated the design of the Flix language [125] throughout the section.

4.5.1 Embedding in LMS Scala

One can use the interpreter/compiler presented in Section 4.4 as a separate process from the main program. In this case, whenever the main program needs to evaluate a Datalog query, it can invoke the Datalog compiler as an external solver, sending the program in textual format along with the required data. The engine then compiles and evaluates the query, producing the result and sending it back to the main program. But this has a lot of overhead due to the repeated back and forth communication. Writing the main program in LMS instead already gives us some benefits — evaluating a Datalog program can easily be done by calling the staged version of `compute` from Fig. 4.5 as `compute(parse(program), edbs, outputs)`. Code producing `edbs` and postprocessing the `outputs` can be optimized jointly with the staged interpreter code.

Yet, adding support for lattices and other extensions still requires designing additional syntactic constructs and typing rules. Instead, in Flan, we delegate these responsibilities to the host language, by providing an embedding of Datalog rules into Scala, similarly to Scalogno [155]. Fig. 4.13a shows how we designed this Datalog embedding (named Cipolla) in Scala that we use as a front end for Flan. An example Datalog program computing node reachability, as expressed through this embedding, is shown in Fig. 4.13b. In this setup, rules are formulated as functions wherein the arguments serve as the rule's metavariables, and the function bodies represent clauses. The `rel` combinator introduces relations of varying arities (`R1`, `R2`, etc). For instance, in Fig. 4.13b L 7, `rel { (src: Id, dst: Id)...` defines `path` as a relation comprising two fields, `src` and `dst`. The argument function must produce an atom (`Atm`). Atoms are formed by applying relations — binding the variables of the defining relation to other relations (e.g., L 9 `edge(src, dst)`), or by the disjunction ( ) and conjunction (`&&`) of other atoms. These can also include constraints as specified by `where` (examples involving such constraints are provided later). The `exists` function introduces existential variables (variables that only appear on the right-hand side of a Datalog rule)

```
class Atm: // body of rules
  def   ( o: Atm): Atm = ...
  def &&(o: Atm): Atm = ...
  def where(cs: Constraints∗): Atm = ...
type R1[T1] = Function1[T1, Atm]
type R2[T1, T2] = Function2[T1, T2, Atm]
// R3, R4, ...
// input relations
def input[T1: Typ, T2: Typ](fname: String): R2[T1, T2] = ...
// relations of different arities
def rel[T:Typ](f: T => Atm): R1[T] = ...
def rel[T1:Typ, T2:Typ](f: (T1, T2) => Atm): R2[T1, T2] = ...
... // other overloads
// introduction of vars
// (see Fig. 4.13b for an example)
def exists[T:Typ](f: T => Atm): Atm = ...
def exists[T1:Typ, T2:Typ](f: (T1, T2) => Atm): Atm = ...
... // other overloads
```

(a) Definitions for implementing Datalog-like declarative rules using standard Scala constructs.

```
.type Id

.decl edge(src: Id, dst: Id)
.input edge

.decl path(src: Id, dst: Id)
.output path

path(src, dst) :-
  edge(src, dst).
path(src, dst) :-
  path(src, node),
  edge(node, dst).
```

```
type Id = Rep[Int]
def edge: R2[Id, Id] =
  input("Edge.facts")
def path: R2[Id, Id] =
  rel { (src: Id, dst: Id) =>
    edge(src, dst)
    exists { (node: Id) =>
      edge(src, node) &&
        path(node, dst)
    }
  }
```

(b) Expressing `path` Datalog program (left) in our Scala embedding (right)

Figure 4.13. The core of our embedded DSL in Scala for writing Datalog programs, with (a) illustrating the definitions, and (b) displaying a sample program.

explicitly. Additionally, we employ a `Typ` typeclass to ensure that relations are only defined for types supported by LMS's `Rep` types. From this point forward, we omit this typeclass constraint. We provide a `query` function that reifies the query into the AST format we used earlier for our textual frontend (Fig. 4.4).

```
val reachable = query[Int, Int] { (src, dst) => reach.path(src, dst) }
```

This reification process is done by repurposing the LMS staging capabilities. Essentially, we stage the program expressed in our embedding and construct the corresponding LMS IR. However, instead of using LMS backend for code generation from this IR as we typically do, we extract the LMS IR and transform it into our Datalog AST representation. Once the AST is derived, the subsequent steps of the process follow the same as described in Section 4.4. Specifically, in the above code, `query` will carry out the fixpoint computation and produce a second-stage buffer that can be used to print or further process the pairs of reachable nodes.

This embedding already allows us to reap multiple benefits from a developer experience perspective. We inherently gain the perks of syntax highlighting, refactoring, and typing, facilitated by the host language's tooling. Scala's type system automatically checks for ill-typed variable bindings in atoms within the rules. Going a step further, we can reuse other features of the host language such as case classes, pattern matching, polymorphism, and abstract methods when defining our program, as we shall discuss below.

Polymorphism and Typeclasses The program below generalizes the `path` program to work on any data-type `T` as long as a typeclass for `Eq` is available. `Eq` signifies that values of type `T` should have an equality function enabling comparison with other `T` values, which is essential for unification during rule evaluation. For example, `PolyReachability[Int]` would give us an instance of reachability that operate on graphs with `Int` labeled edges. Another key thing to note below is that the `edge` relation is left abstract such that users can provide different facts produced by means other than loading from a file, e.g., result of a different rule.

```
trait PolyReachability[T: Eq] extends DatalogModule:
  def edge(src: T, dst: T): Atm
  def path: R2[T, T] = rel { (src: T, dst: T) =>
    edge(src, dst)
    exists[T] { node =>
      path(src, node) && edge(node, dst)
    }
  }
```

We achieve the benefits of mixing declarative programming with conventional functional programming as proposed by the Flix language [125] leveraging host language abstractions. Moreover, by using LMS's `Rep` types we achieve 'abstraction without regret' [21]: polymorphism and typeclasses are compiled away during staging and only code essential to the computation is generated.

Higher-order Relations Moving forward, it becomes evident that the definition of `path` is the transitive closure of `edge`. Therefore we can go ahead and rewrite our rules in terms of other higher-order rules, similarly to how it can be done in Datafun [131] and Scalogno [155].

```
def compose[A, B, C](r: R2[A, B], t: R2[B, C]): R2[A, C] =
  // Compose Rule: (simple join)
  // R(a,c) :- r(a,b), t(b,c)
  rel { (a: A, c: C) =>
    exists[B] { b => r(a, b) && t(b, c) }
  }
def transitive_closure[T](r: R2[T, T]): R2[T, T] =
  // defines transitive closure using compose (above)
  // tc(x, y) :- r(x, y)
  // tc(x, y) :- r(x, b), tc(b, y)
  rel { (x, y) =>
    r(x, y)
        compose(r, transitive_closure(r))(x, y)
  }
def edge = ...
// path: transitive closure of edge
def path = transitive_closure(edge)
```

The `compose` method becomes a shorthand for joining two relations and `transitive_closure` enables a more concise definition of `path`. Higher-order relations like the ones above enables us to rewrite rule 4.2 from the points-to example as follows.

```
def pointsTo = rel { (x, y) =>
  addressOf(x, y)                    /* rule (4.1) from Fig. (4.3) */
  compose(assign, pointsTo)(x, y)    /* rule (4.2) from Fig. (4.3),
                                        which is a simple join */
  ...
}
```

User-defined Functions Finally, we show how UDFs are integrated into the DSL. Given that we reside within Scala, users can create their UDFs using familiar Scala syntax. The following snippet shows an example where a UDF is used. Specifically, it shows a variant of the previously discussed 'path' program, wherein the `edge` has an extra attribute - `label`.

The UDF serves as a predicate constraint, determining whether a given edge, based on its label, should be included in the path.

```
trait PolyReachability[T, L] extends DatalogModule:
  def edge: R3[T, L, T]

  // abstract declaration of the UDF
  def pred: L => Rep[Boolean]

  // path(src, dst) :- edge(src, dst, lab), pred(lab) = true
  // path(src, dst) :- path(src, node), edge(node, lab, dst),
  //                   pred(lab) = true
  def path = rel { (src: T, dst: T) =>
    exists[L] { lab => edge(src, lab, dst) where pred(lab)}
    exists[T, L] { (node, lab) =>
    path(src, node) && edge(node, lab, dst)
    where pred(lab) }
  }
val reach = new PolyReachability[Int]:
  def edge = ...
  def pred = (lab: Rep[Float]) => lab > 0.5     // UDF implementation

// query
val reachable: Buffer[Int] = query { (dst: Id) => reach.path(2, dst) }
```

Since the query performed looks for paths from a specific `src` (2 in this case), at staging time, the concrete value of `src` will be used to generate the specialized code. Additionally the predicate function is inlined in the fixpoint loop in the generated code, minimizing function call overhead. As noted before, polymorphism is monomorphized during staging.

4.5.2 User-defined Lattices

With the aforementioned tight integration with the host language, adding extensions such as user-defined lattices becomes trivial. Essentially, we can use regular Scala classes to define lattices, regular Scala functions to define operations like least-upper bound for lattices, and regular Scala functions to define any UDFs operating on lattices. We examine a relatively simple example of using lattices alongside Datalog in Fig. 4.14 (left), which computes the shortest path from a given source node to all other nodes in the graph (taken from Flix documentation).

While it is possible to encode the same problem in regular Datalog (that uses the powerset lattice) followed by an aggregation, this approach would be significantly slower, as it computes all possible distance values for each pair rather than just the shortest distance.

[4]https://doc.flix.dev/lattice-semantics.html#using-lattice-semantics-to-compute-shortest-paths

```
 1 type Node = Rep[Int]
 2
 3 /* '@udd' indicating a lattice value */
 4 @udd
 5 case class D(x: Rep[Int])
 6
 7 /* defines lub - used for updates */
 8 val lat = new JoinLattice[D]:
 9   def lub = udf { (x: D, y: D) =>
10     val D(n1) = x
11     val D(n2) = y
12     D(n1 min n2)
13   }
14 /* UDF on lattice values */
15 def add = udf { (x: D, y: D) =>
16   val D(n1) = x
17   val D(n2) = y
18   D(n1 + n2)
19 }
20
21 def edge =
22   input[Node, Rep[Int], Node]("Edge.facts")
23
24 def dist(src: Node): R2[Node, D] =
25   rel { (dst, d) =>
26     dist(src, D(0))
27     exists { (d1: D, d2: Rep[Int], x: Node) =>
28       dist(x, d1) && edge(x, d2, dst) where
29         (d `=` add(d1, D(d2)))
30     }
31   }
```

Figure 4.14. Code on the left demonstrates the implementation of the shortest path using lattices and UDFs, identical to Flix.[4] Unlike Flix, however, we generate highly specialized code that erases all high-level abstractions such as case classes, pattern matching, extractors, etc. The generated code is shown in Fig. 4.15.

Instead, as demonstrated in Fig. 4.14 (left), we can encode this using a user-defined lattice (L 4) that maintains only the shortest path observed so far (similar to Flix). Note the use of `@udd` in L 4, a macro we employ to track lattice types. Specifically, the `value` field of `dist` will now be a lattice value, and instead of retaining all previously seen values, it will only store the smallest value (defined using the `lub` function in L 8) observed so far for a given '`to`' value. Flan backend is modified to handle lattice relations (i.e., ones having a `@udd` typed column) separately, and to call the corresponding `lub` function whenever new tuples

```
struct edge* edge_store = ...
/* index related data structures */
struct edge_ind* ind = ...
...

// fixed point loop
bool changed = true;
while (changed) {
  /* outer loops elided */
    // x, d1, d2, to in context
    // inlined udf
    int value = d1 + d2;

    // lattice update logic
    int oldValue = ...; // retrieve

    // inlined lub
    // values are unboxed
    int newValue = min(value, oldValue);

    if (oldValue != newValue) {
        // update the lattice ...
    }
}
```

Figure 4.15. An excerpt of the generated code for Fig. 4.14. Note that variables have been renamed for improved clarity.

are discovered. Moreover, whenever a new tuple triggers an update, we maintain these in our delta indices to facilitate semi-naive evaluation.

Fig. 4.14 (right) displays an excerpt from the code generated by Flan for this Datalog program. A crucial observation is that none of the abstractions used in defining the program on the left, such as case classes, object extractors, UDFs, and so on, are present in the generated code. Instead, it consists solely of operations on low-level primitive data types, native arrays, etc. ensuring that no runtime costs are incurred for the abstractions employed when defining the Datalog program. In Section 4.6.5, we evaluate the performance of programs that use these user-defined lattices and observe that the code generated by Flan is significantly faster compared to other existing systems.

In this example, we saw a scenario in which an unbounded lattice `D` is employed to represent the shortest distance, with $\perp$ implicitly signified by non-existence. However, there are situations where bounded lattices with an explicit representation of $\top$ are necessary. For instance, the strong-update program analysis benchmark [156] used in Section 4.6.5 makes use of the constant propagation lattice, encompassing $\perp$, a constant, and $\top$. In such cases, one option for users is to define a member within the case class (which represents the lattice) to indicate whether the current value is $\top$ or $\perp$. This would translate to a C struct representation in the generated code. Alternatively, users can use special values (e.g., `INT_MAX` for $\top$) to represent these.

4.6 Experiments

In this section, we first evaluate the engineering effort associated with Flan and then the performance of Flan in comparison to various state-of-the-art systems across a diverse range of benchmarks. Our primary objective is to demonstrate that the use of staging for transforming a high-level interpreter into a compiler successfully generates specialized code that achieves competitive performance with other sophisticated Datalog compilers.

Our evaluation consists of three primary benchmarks. First, we select an existing simple points-to analysis benchmark from Soufflé and use it to compare Flan against a variety of other established systems. Next, we delve into a more comprehensive program analysis benchmarks from the Doop framework [115]. Lastly, we explore a use case that combines lattice semantics with Datalog and evaluate Flan's performance against systems that support lattices.

4.6.1 Environment

We conduct all our experiments on a NUMA machine featuring 4 sockets, each with 24 Intel(R) Xeon(R) Platinum 8168 cores and 750GB RAM per socket (3 TB in total), running Ubuntu 18.04.4 LTS. For multi-threaded executions, we employ `numactl` to ensure that processes are allocated within the same NUMA node and utilize the nearest memory regions. Each experiment is run five times, and we report the average across those five runs. We did not observe any significant variance among each run for all systems, hence, we omitted the error bars from the plots. For all experiments we have verified that all systems produce the same result. We use the most recent releases of all baseline systems, specifically: Soufflé 2.3, Ascent 0.5.0, Crepe 0.1.8, and Flix 0.35.

4.6.2 Engineering Effort

Since we consistently emphasize Flan's implementation simplicity throughout this work, we now try to evaluate this aspect of Flan. While it is very hard to quantify the engineering effort scientifically, we have chosen lines of code (LOC) as a crude proxy for this evaluation. The table below illustrates the additional LOC required for each extension. 'Base' refers to the staged version of the interpreter initially introduced in Section 4.3, plus parsing, embedding (Section 4.5), other utilities like profiling (e.g., measuring per-rule time), logging, output verification, etc.

Base	+Features	+Join Strategies	+BTree	+HashIndex	**Total**
2197	+366	+130	+332	+1063	4088

The '+Features' case encompasses the integration of user-defined aggregates, UDFs, negations, constraints, etc. The added code pertains to the canonicalization process discussed in Section 4.4.2, and the logic highlighted in Fig. 4.12. This also includes the addition of support for user-defined aggregates, a feature recently added to Soufflé. Implementing this in Soufflé required over 1500 LOC modifications, impacting their declarative and imperative IRs, core engine mechanics, and code synthesizer [157]. In constrast, we accomplished the same functionality with fewer than 50 lines of high-level Scala code. It is important to note, however, that this comparison is not rigorously precise, as LOC do not always accurately reflect implementation complexities.

The '+Join Strategies' denotes the addition of support for multi-way and binary joins, achieved by creating two implementations of the `trait Strategy` with the `variableOrder` overridden, as detailed in Section 4.4.3. '+BTree' and '+HashIndex' refers to the LOC added to implement the respective index structures. As outlined in Section 4.4.4, for BTree, we simply created a wrapper around Soufflé's B-trees, whereas for HashIndex, we developed our own specialized version in Scala.

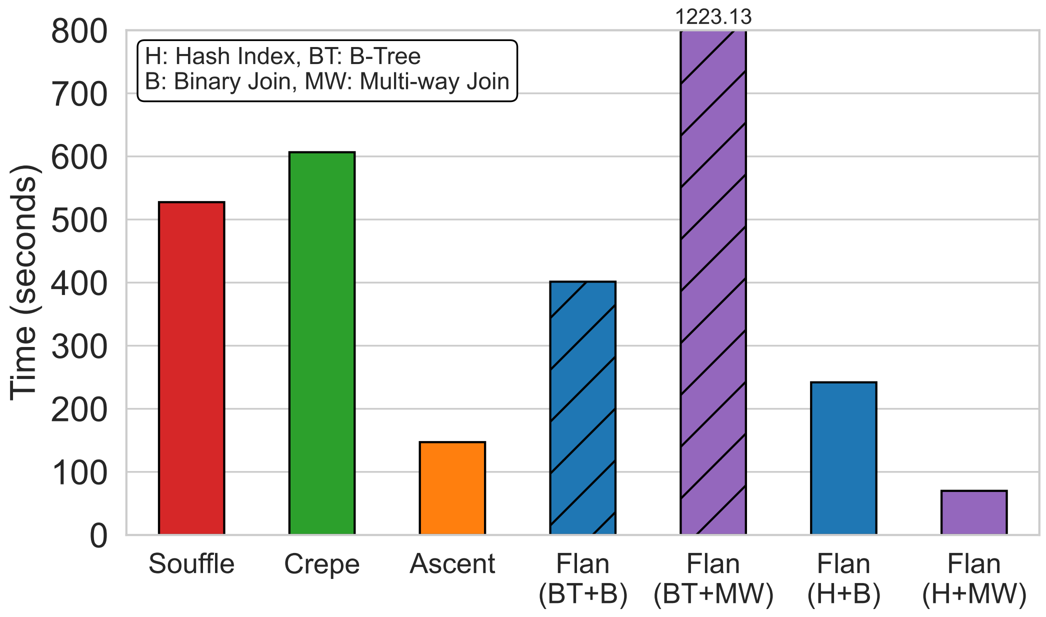
4.6.3 Simple Points-to Analysis

To evaluate performance relative to several other systems, we chose a relatively simple, openly available benchmark from Soufflé[5] that conducts a call-insensitive, field-sensitive points-to analysis using Datalog on a provided synthetic dataset. This analysis includes roughly 7,000 facts each in four EDB relations, two IDB relations, and five rules. We re-implemented these benchmarks in Ascent and Crepe, whereas for Flan and Soufflé, we directly input the corresponding Datalog file (Flan supports running Soufflé-style Datalog files directly). It is worth noting that this benchmark does not incorporate any extensions such as UDFs, aggregations, constraints, negations, or others.

Fig. 4.16 presents the benchmark results. Flan incorporates both binary and multi-way joins, while the rest of the baselines employ solely binary joins. Soufflé utilizes tree-based indices, which, compared to the hash-based indices used by all other systems, has a higher asymptotic runtime leading to longer execution times. Despite Crepe's use of hash-based indices, it lags noticeably behind Soufflé due to its reliance on generic hash table implementations, in contrast to Soufflé's data structures tuned for Datalog. Although Ascent generally uses binary joins, it employs a special optimization for certain rule types. This optimization transforms the outermost join steps into a form identical to multi-way joins. For instance, the performance-critical rules in this benchmark, which account for 53% and 44% of total time, contain only two atoms in the body (each with two variables), and performs a transitive closure. For these two rules, Ascent's optimization results in a join strategy identical to our multi-way joins, demonstrating a speedup of 3.25× over Soufflé.

The different execution strategies deployed in Flan showcase the distinct performance characteristics of each. For both strategies, our specialized hash-index offers substantial performance gains over their generic B-tree counterparts, yielding speedups of 17.5× and

[5]↑https://github.com/souffle-lang/benchmarks/tree/acd6b9ec5043109fc1e6674e759a86164634e70b/benchmarks/pointsto

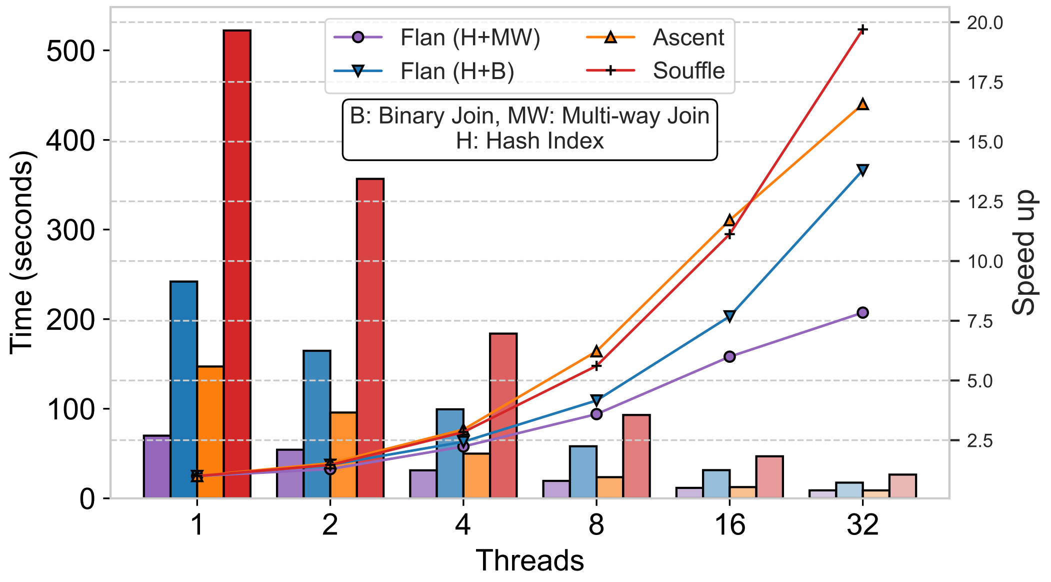


Time (s)	Soufflé	Crepe	Ascent	Flan			
				BT+B	**BT+MW**	**H+B**	**H+MW**
PointsTo	527.50	606.64	147.16	401.45	1223.13	241.96	**69.97**
Speedup	1.00	0.87	3.58	1.31	0.43	2.18	7.54

Figure 4.16. Performance comparison between Flan (our system, with different strategies) and various other Datalog compilers for a simple call-insensitive, field-sensitive points-to analysis, for single-threaded execution.

1.66×. For this benchmark, multi-way joins perform significantly better than binary joins. Upon closer inspection, we observed that the multi-way join strategy results in 23% fewer lookups compared to the binary join strategy (3,914,693 vs 5,126,129) for the performance-critical rules. Interestingly, the multi-way join case using B-trees notably underperforms compared to its binary join counterpart. This is because, despite a smaller number of lookups, the total cost of lookups is higher in the multi-way join case.

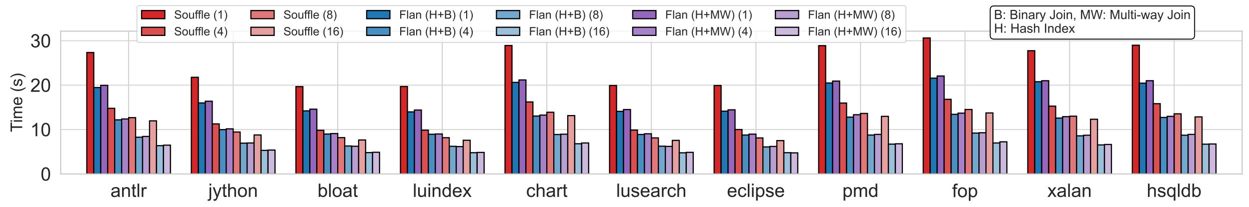
Although not shown in Fig. 4.16, we experimented with a variant of our multi-way join that used C++ standard library's unordered maps. The result was an execution time of over 30 minutes, an order of magnitude slower than our specialized hash-indexes. This disparity exemplifies the efficiency of having fully-specialized index implementations.



Time (s)	1	2	4	8	16	32
Flan (H+MW)	**69.97**	**54.27**	**31.29**	**19.48**	**11.67**	8.92
Flan (H+B)	241.96	164.68	99.37	58.12	31.49	17.53
Ascent	147.16	95.78	49.93	23.65	12.56	**8.88**
Soufflé	522.18	356.55	183.87	93.04	46.93	26.51

Figure 4.17. Performance of parallel execution for the same benchmark in Fig. 4.16, comparing Soufflé and Flan (for binary and multi-way joins). The lines represent the scaling of each system relative to its own single-threaded performance.

Parallel Scaling Fig. 4.17 shows the results for parallel execution performance. For this, we employed the same benchmark but varied the number of threads. Among prior systems, only Soufflé, Ascent, and Flan support parallel execution. Flan consistently outperforms Soufflé across all thread counts, achieving speedups ranging from 3× to 7.5× (compared to Soufflé with the same number of threads). Moreover, Soufflé requires 8 to 16 threads to match Flan's single-threaded performance. In comparison to Ascent, Flan demonstrates significantly better performance for up to 8 threads, achieving speedups of up to 2.1×, and remains competitive in 16 and 32 threads cases.



Time (s)	**antlr**	**jython**	**bloat**	**luindex**	**chart**	**lusearch**	**eclipse**	**pmd**	**fop**	**xalan**	**hsqldb**
Soufflé (1)	27.35	21.77	19.67	19.70	28.93	19.92	19.92	28.90	30.62	27.76	28.99
Flan (H+B) (1)	**19.46**	**15.99**	**14.21**	**13.98**	**20.62**	**14.10**	**14.14**	**20.48**	**21.57**	**20.78**	**20.44**
Flan (H+MW) (1)	19.95	16.38	14.59	14.40	21.17	14.51	14.45	20.92	22.06	21.00	21.01
Soufflé (4)	14.78	11.27	9.83	9.86	16.22	9.89	10.01	15.97	16.81	15.27	15.83
Flan (H+B) (4)	**12.19**	**9.98**	**8.99**	**8.94**	**13.04**	**8.89**	**8.77**	**12.80**	**13.43**	**12.59**	**12.72**
Flan (H+MW) (4)	12.38	10.16	9.10	9.00	13.25	9.05	8.97	13.34	13.70	12.91	12.98
Soufflé (8)	12.68	9.45	8.18	8.17	13.90	8.12	8.11	13.63	14.52	13.01	13.55
Flan (H+B) (8)	**8.27**	**6.93**	6.30	6.23	**8.89**	6.25	**6.09**	**8.76**	**9.21**	**8.59**	**8.73**
Flan (H+MW) (8)	8.45	6.98	**6.24**	**6.17**	8.96	**6.21**	6.20	8.91	9.28	8.72	8.91
Soufflé (16)	11.96	8.78	7.65	7.59	13.15	7.56	7.52	12.97	13.74	12.32	12.85
Flan (H+B) (16)	**6.37**	**5.31**	**4.81**	**4.79**	**6.80**	**4.77**	4.79	**6.72**	**6.98**	**6.54**	**6.71**
Flan (H+MW) (16)	6.48	5.39	4.88	4.85	6.98	4.87	**4.75**	6.80	7.24	6.64	6.75

Figure 4.18. Performance comparison of our system (Flan) with Soufflé for the micro analysis variant of the Doop program analysis toolchain. The value within parentheses indicates (index structure + join strategy), followed by the number of threads used. The analysis is conducted on eleven programs from the DaCapo benchmark suite. The table presents the execution times for each scenario.

4.6.4 Doop: Points-to Analysis of Java Programs

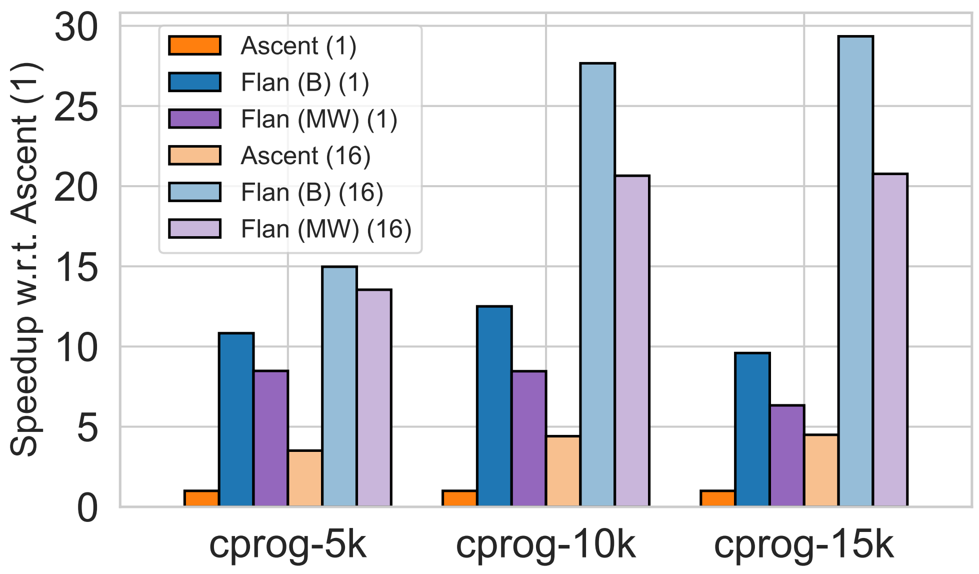
We also investigated how each system's performance scales in relation to its single-threaded execution (depicted by a line graph in Fig. 4.17). All systems demonstrate good scalability as the number of threads increases, with Soufflé scaling to 22.6×, Ascent to 16.5×, and Flan reaching 13.8× (binary) and 7.8× (multi-way) at 32 threads. It is important to note that despite Soufflé's better scalability, it remains slower in absolute terms. Soufflé's better scalability can be attributed to their highly efficient concurrent index structures [122] tailored for Datalog. Moreover, as identified by McSherry *et al.* [158], systems with higher overall overhead tend to scale better due to the parallelization of overheads, which is likely another reason for Soufflé's superior scaling. Ascent uses the data parallelism-library Rayon [159] for parallel execution, which uses a work stealing model. For index structures, Ascent uses the DashMap library [160], which is a highly-efficient concurrent hash table implementation in Rust. This allows Ascent to scale well as the number of threads are increased. In contrast, as discussed in Section 4.4.6, Flan's parallel data structures are currently based on relatively simple lock-based segmented hash tables. We believe the performance scalability can be further improved by utilizing more advanced concurrent index structures as we discussed in Section 4.4.6.

In the previous section, we evaluated performance for a relatively simple points-to analysis benchmark. In this section, we examine a more comprehensive benchmark, specifically choosing the micro analysis from the Doop program analysis benchmark suite. This analysis consists of 389 relations (including 109 EDB relations) and 300 rules, employing various Datalog extensions such as aggregation, negation, UDFs, and constraints. As done in prior similar studies [161, 162], we run this analysis on a selected set of programs from the DaCapo benchmark suite [132, 163]. Doop extracts the facts for the EDB relations used in the analysis from these programs, which can then be used to perform the analysis using Datalog.

We evaluate the performance of Flan only against Soufflé, as none of the other systems directly support the analysis logic emitted by Doop or the various extensions used in the analysis logic. We evaluate both single-threaded performance and parallel performance, running on 4, 8, and 16 threads. We observe similar performance traits to those of the prior experiment. Both join variants of Flan utilizing specialized hash indices consistently outperform Soufflé. In particular, binary joins demonstrate an average speedup of 1.45× (across all cases) with a peak speedup of 1.97×. Meanwhile, multi-way joins show an average speedup of 1.42× and achieve a maximum speedup of 1.91×. It is worth noting that the binary join strategy in Flan differs slightly from Soufflé's since we also perform intersections to short-circuit the loop nest as early as possible (as discussed in Section 4.4.3). Although not included, we also conducted benchmarks on a variant of binary joins that does not perform these intersections, which resulted in an average speedup of 1.2× and a maximum speedup of 1.6× compared to Soufflé.

4.6.5 Strong-Update Points-to Analysis using Lattice Semantics

In this section, we evaluate the performance of a program analysis benchmark employing lattice semantics. More specifically, we selected the strong-update points-to analysis, as introduced by Lhoták and Chung [156], which combines elements of flow-insensitive and flow-sensitive analysis. This is achieved by employing a singleton-set lattice to propagate singleton



Time (s)	**cprog-5k**	**cprog-10k**	**cprog-15k**
Ascent (1)	2.99	52.19	268.15
Flan (B) (1)	**0.28**	**4.17**	**27.95**
Flan (MW) (1)	0.35	6.17	42.36
Ascent (16)	0.85	11.84	59.66
Flan (B) (16)	**0.20**	**1.89**	**9.14**
Flan (MW) (16)	0.22	2.53	12.91

Figure 4.19. Performance evaluation for performing a strong-update analysis (implemented using user-defined lattices) on three C programs. The left plot shows the normalized execution time with respect to Ascent single-threaded time. The table on the right presents the actual execution times for each case.

sets in a flow-sensitive manner. Our evaluation replicates the declarative implementation outlined in the work by Madsen *et al.* [3] (Figure 4 in that paper), which was originally implemented using Flix. We have re-implemented this analysis both in Flan and Ascent.

We evaluate the performance of this benchmark on three sample C programs each having 5k, 10k, and 15k instructions, from which input relations `AddrOf`, `Copy`, etc. are extracted. Ascent has previously demonstrated speedups of several orders of magnitude compared to Flix in similar benchmarks [117]. Consequently, we omit the Flix numbers for this experiment.

Fig. 4.19 illustrates the performance of Flan compared to Ascent for this benchmark. Specifically, it shows the normalized execution time with respect to Ascent's single-threaded execution time for both Flan and Ascent under single-threaded and 16-threaded execution settings. As in previous experiments, we report Flan's performance for both the multi-way and binary join strategies. In single-threaded execution, Flan demonstrates significant speedups, with the multi-way strategy achieving a maximum speedup of 8.5× and an average speedup of 7.8×. The binary join strategy in Flan attains a maximum speedup of 12.5× (average speedup of 11×) compared to the single-threaded performance of Ascent. When scaling up to 16 threads, Flan maintains its superior performance, with the multi-way join strategy delivering a maximum speedup of 4.7× (average speedup of 4.4×). Meanwhile, the binary joins in Flan achieves a maximum speedup of 6.5× (average speedup of 5.7×) when compared to their 16-threaded Ascent counterparts.

In this particular example, apart from the differences in join strategies as we discussed in previous subsections, the other main difference between Ascent and Flan is the specialization granularity. Ascent enables specialization for compiling the given Datalog rules, but any abstractions associated with defining lattices, UDFs, etc. will still be present in the generated code. Conversely, as seen in Fig. 4.14, Flan fully specializes the program, including any arbitrary programming logic surrounding the Datalog logic. In this case, this includes abstractions related to user-defined lattices and the corresponding UDFs. As a result, the generated code in Flan does not contain any of these abstractions or UDFs (inlined), thereby eliminating any runtime cost associated with them.

The superior performance of Flan's binary joins compared to multi-way joins in this specific benchmark can be attributed to the higher selectivities of variable intersections. In essence, this translates to a reduced amount of filtering or short-circuiting at each level within the loop nest in the multi-way case. As a consequence, a higher number of total lookups is conducted compared to the binary join case. It is important to note that this behavior can be highly dependent on the specific program and input data.

In summary, in this section, we have observed that Flan achieves competitive performance, and in some cases, significantly outperforms existing state-of-the-art systems across various types of workloads. This validates our claim that staging can be utilized to transform a high-level Datalog interpreter into a compiler that generates specialized code that achieves the same level of performance as other sophisticated Datalog compilers with only a fraction of the engineering cost.

4.7 Related Work

Datalog for Program Analysis Datalog has been a popular choice for declarative program analysis for quite some time [112–115, 164–169]. bddbddb employs Binary Decision Diagrams (BDDs) for evaluating Datalog rules; however, this representation is only effective for specific problem structures, and its performance is highly dependent on variable ordering. Soufflé [2] is a high-performance Datalog engine that synthesizes specialized C++ code for a given Datalog program. As one of the most mature and widely used Datalog systems, Soufflé offers a broad range of features, including specialized parallel data structures [120–122], provenance [161], automatic index selection [123], incremental execution [170], join order optimization [124], and more. Owing to its maturity, Soufflé has been used not only in popular program analysis toolchains like Doop [115, 144] but also in other domains such as binary disassembly [111] and decompilation of smart contracts [171]. However, as discussed in Sections 4.1 and 4.2, there are several limitations in Soufflé's code generation approach. RecStep [140] is a Datalog engine built on top of the relational query engine QuickStep [172] and achieves competitive performance on diverse workloads including graph analytics and program analysis. However, similar to Soufflé, it lacks an expressive front end.

Several recent works have followed in Soufflé's footsteps in developing Datalog compilers. Crepe [118] and Ascent [117] are Datalog compilers embedded in Rust using macros and employing quasi-quotations for code generation of a given Datalog program. Eclair [119] is a Datalog compiler that translates Datalog programs into LLVM IR [30]. However, all these code generation approaches suffer from similar limitations to a certain degree as Soufflé (e.g., having to manage virtual registers when generating LLVM IR, writing quoted code fragments, as opposed to writing high-level interpreter-style code). Differential Datalog [139] (built atop Differential Dataflow [102]) and IncA [173] are Datalog engines that support incremental execution. Notably, IncA is tailored towards program analysis and has support for lattices [174, 175]. Morever, IncA DSL has some superficial similarities to our Scala embedded DSL. Flan currently lacks support for incremental execution, an avenue we plan to explore in future work. Drawing upon previous work that maps e-matching [176, 177] into conjunctive queries [178], EggLog [179, 180] unifies equality saturation with Datalog-style fixed point computations by baking in the notion of equivalence into Datalog relations. Exploring how to expand Flan to support equality saturation in a similar manner is an interesting future research direction. Both Zhang *et al.* [178] and EggLog employ Generic Joins [146, 150], a type of multi-way join algorithm that dynamically picks the atom with lowest cardinality for variable iteration which leads to worst-case optimal joins.

BYODS [181], an extension to Ascent, offers a means to create user-defined data structures for storing relations which also capture properties (e.g., transitivity) of the relations. They have demonstrated significant performance improvements for certain types of program analyses using this approach. We believe that a similar notion can be realized in Flan in a similar way since Flan programs are fully interoperable with Scala, enabling users to employ Scala to define and integrate such custom data structures. However, it is important to note that, we have not evaluated the plausibility or effectiveness of this approach in our current work.

Datalog Extensions Pure Datalog is not sufficiently expressive to cover a wide variety of declarative program analysis tasks, and many existing systems propose various extensions. Flix [3, 125] supports Datalog as first-class values in a relatively comprehensive programming language and supports lattices beyond the powerset lattice, allowing more expressive analysis like StrongUpdate [156] analysis and, IFDS and IDE algorithms [182, 183]. Datafun [131, 184], another language that provides support for lattices also has the ability to track monotonicity via types. Formulog [4] extends Datalog with the ability to interact with a functional programming language and the capability to use SMT constraints in rules, offloading them to a solver as necessary. These approaches greatly enhance Datalog's power and applicability across various domains. However, most of these systems do not leverage techniques like specialized code generation, potentially sacrificing performance. We believe that Flan-style compiler construction, closely resembling interpreters, would make it easier for such extensions to achieve their optimal performance. Functional IncA [185] proposes the use of a functional programming with set-based fixpoint-semantics as a frontend for Datalog. In this model, Datalog serves as an IR, with programs written in the functional language translated into Datalog and resolved through pre-existing solvers. This approach contrasts with Flan's approach, where the 'general-purpose' aspect of the computation is processed using the (general-purpose) host language, with the Datalog solver being specifically employed for fixed point computations.

Query Compilation Our work is primarily inspired by prior research in relational query compilation. Earlier such approaches [186, 187] relied on operator templates (similar to Soufflé) and generated code by concatenating these templates based on the query plan. Hyper [24], on the other hand, utilized the programmatic LLVM API [30], achieving significant performance gains but at the expense of a more low-level implementation. LB2 [19, 20] (an extension of [81, 141]) was based on generative programming [21] and achieved the same level of performance while keeping the operator interface simple. We draw inspiration from these works and adapt the same ideas to Datalog-based program analysis.

4.8 Conclusion

We introduced Flan, a Datalog compiler constructed by partially evaluating a high-level Datalog interpreter implemented in Scala. Utilizing the existing generative programming framework, LMS, we generate *fully* specialized code, a feature that existing compilers lack, but paramount to performance. Capitalizing on its seamless interoperability with Scala, we constructed a flexible frontend that leverages Scala's capabilities to add extensions. Moreover, we devised a streamlined operator interface that effortlessly facilitates a variety of evaluation strategies and index structures.

One of the main limitations of Flan currently is we perform only very limited plan-level optimizations based on very simple heuristics. For instance, variable ordering of multi-way joins are currently derived from a left associative binary join plan based on the rule specification and by default uses hash indices. Users do have the option to specify their preferred join types and index types using global flags that applies these changes to all rule evaluations. However, much more sophisticated analysis and query planning could be done, for example, to choose between multi-way and binary joins, and for picking good variable orderings, or for picking tree-based or hash-based indices and so on at the granularity of each rule. For instance, as demonstrated in our evaluation in Section 4.6, the most effective evaluation strategy (e.g., multi-way or binary joins), often varies from one benchmark to another.

Flan achieves a commendable level of performance even in the absence of such clever query planning. We believe that Flan could be coupled with any existing well-studied query planning approach (e.g., using cardinality estimates to pick the index and join types, etc.) [123, 124] to improve performance further. Moreover, it is possible to add support for *dynamic* optimizations (e.g., adaptive join order optimization) using techniques such as on-stack replacement as demonstrated in prior work [188].

Another promising line of future work is on integrating SMT constraints into Flan in the style of Formulog [4] that would unlock a wider range of static analysis including symbolic execution. Moreover, GenSym [128, 189], a state-of-the-art symbolic execution engine built using LMS, already equips LMS with functionalities to interface with SMT solvers. Consequently, much of the essential infrastructure is already in place. This integration would enable Flan to effectively function as a 'compiled Formulog' with minimal additional engineering effort. We plan to explore the feasibility and the effectiveness of this approach in our future work.

5. RHYME: A DATA-CENTRIC EXPRESSIVE QUERY LANGUAGE FOR NESTED DATA STRUCTURES

Portions of this chapter will appear in *Rhyme: A Data-Centric Expressive Query Language for Nested Data Structures in Proceedings of Practical Aspects of Declarative Languages 26th International Symposium, PADL 2024, London, UK, January 15-16, 2024* [190].

5.1 Introduction

Declarative programming represents a paradigm in which users articulate *what* computation needs to be performed, without the explicit specification of the procedural steps required for its execution. Declarative programming languages find application across a diverse array of domains. Notable examples include SQL, employed for data querying and manipulation, Datalog [191], used for data querying as well as in domains like declarative program analysis [112, 113, 192] and binary decompilation [**ddissam**], Einstein notation (or similar domain specific languages [193]) for expressing tensor computations mathematically, and GraphQL [194] for data querying within the context of web application front-ends, and so on.

In practical scenarios where diverse paradigms of workloads are combined (e.g., data frames + tensors), the necessity arises to employ multiple query languages in tandem. Each of these languages interfaces with the respective engines tasked with handling individual workloads. However, this approach is inherently inefficient, both from a performance perspective, due to the reliance on multiple isolated backends, and from a programmer productivity standpoint, as it necessitates learning and maintaining code written in multiple query languages. In this work, we address this challenge by introducing a unified query language, named Rhyme, that comprises a general substrate capable of accommodating a wide array of different use cases.

Rhyme takes inspiration from many existing declarative languages including GraphQL, JQ [195], XQuery [196], JSONPath [197], Einstein notation, Datalog, recent functional logic programming languages like Verse [198], etc. Rhyme is designed to serve as an expressive lan-

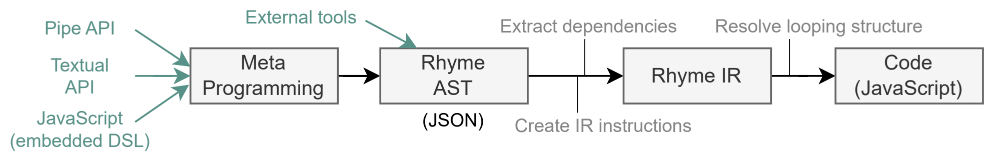


Figure 5.1. End-to-end workflow of Rhyme, with green markers indicating various entry points leading to the common entry point, Rhyme AST. The Rhyme AST can be constructed directly from external tools or via metaprogramming using different APIs. This AST serves as the basis for generating an IR (with dependencies), driving subsequent code generation.

guage for high-level data manipulation, enabling the querying and transformation of nested data structures (e.g., JSON, tensors) and producing nested structures as output. There are several key defining characteristics of Rhyme. First, Rhyme adopts a query syntax that closely mirrors existing object notation, meaning that queries are essentially expressed as JSON objects. Second, Rhyme is designed in a way that permits query optimization and code generation via the construction of an intermediate representation (IR). This IR contains loop-free and branch-free code with dependencies that implicitly capture the program structure. Third, Rhyme is compositional and easy to meta-program, recognizing that data transformation queries are typically used as part of larger programs and are often generated programmatically.

Fig. 5.1 provides an overview of the end-to-end workflow of Rhyme. The central point of entry into this workflow is the Rhyme AST (Section 5.2.5). Notably, this AST is represented in JSON format, hence serializable, enabling the ability to be exported/imported from various other environments. We implement Rhyme as an embedded DSL in JavaScript (JS), which constructs the Rhyme AST from Rhyme queries. Additionally, alternative interfaces can be used, such as pipes (Section 5.3.1), or entirely textual inputs that can be processed by a parser to construct the same AST. Once the AST is constructed, it gets transformed into Rhyme IR. During this transformation, declarative query operators are mapped to IR instructions along with their dependencies. Subsequently, this IR is used to analyze the looping structure and generate the final code.

```
// input dataset
let data = [
  {key: "A", val: 10},
  {key: "B", val: 15},
  {key: "A", val: 25}
]
// query
sum('data.∗.val')
// AST constructed from the query:
{agg: 'sum', param: 'data.∗.val'}
// result: 50
```

```
let tmp = {}
// ??= is assign if null operator
tmp[0] ??= 0
for (let star in data)  {
  tmp[0] += data[star]['val']
}
return tmp[0]
```

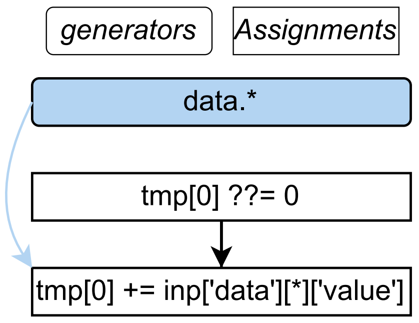


Figure 5.2. A query computing the sum of all values. Query expression is shown in top left, the constructed IR is shown in bottom, and the final generated code is shown in top right.

Figs. 5.2 to 5.4 shows this end-to-end workflow using several example queries. Specifically, the query is shown on the left, alongside its AST representation, the IR in the center, and the generated code on the right. While the comprehensive exploration of these components will be the focus of subsequent sections in this chapter, we offer an introductory overview of each element here.

Consider the query in Fig. 5.2. In Rhyme, `data.∗.val` is called a *path* expression, a notation inspired by JSONPath [197]. The ∗ symbol serves as an iterator, facilitating iteration through the `val` values, while the aggregator `sum` calculates the sum of these iterated values. Rhyme's IR (shown in the center) consists of two main types of operators. First, it has *generators* (represented as *rounded* rectangles), which represents iterators that enumerate a list of items (ultimately translated into loops in the generated code). In our example,

```
// query
{
  total: sum('data.∗.val'),
  'data.∗.key': sum('data.∗.val')
}
// AST constructed from the query:
{
  total:
    {agg: 'sum',
    param: 'data.∗.val'},
  'data.∗.key':
    {agg: 'sum',
    param: 'data.∗.val'}
}
// result:
//  { total: 50,
//    'A': 35,
//    'B': 15 }
```

```
let tmp = {}
tmp[0] ??= {}
tmp[0]['total'] ??= 0

for (let star in data)  {
  tmp[0]['total'] +=
   data[star]['val']
  tmp[0][data[star]['key']] ??= 0
  tmp[0][data[star]['key']] +=
   data[star]['val']
}
return tmp[0]
```

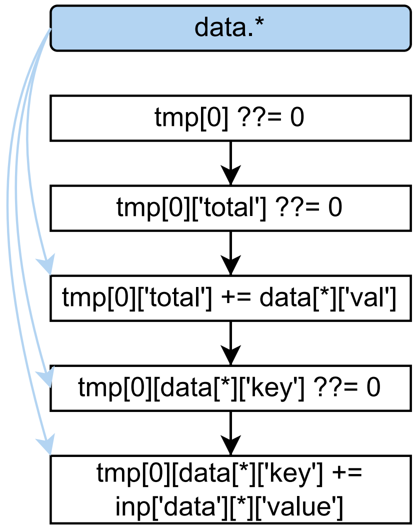


Figure 5.3. A query computing sum of all values (`total`) and sum per each key. Query expression is shown in top left, the constructed IR is shown in bottom, and the final generated code is shown in top right.

`data.∗` is a generator, iterating values from the data object. Second, the IR has *assignments* (represented as rectangles), comprising the computations required for executing the query.

```
// query
{
  'data.∗A.key':
   div(sum('data.∗A.val'),

   sum('data.∗B.val'))
}
// AST constructed from the query:
{
  'data.∗.key': {
    path: 'div',
    param: [
      {agg: 'sum',
      param: 'data.∗A.val'},
      {agg: 'sum',
      param: 'data.∗B.val'}
    ]
  }
}

// result:
// {'A': 0.7, 'B': 0.3}
```

```
let tmp = {}
tmp[1] ??= {}
tmp[2] ??= 0

// loop hoisted!
for (let starB in data) {
  tmp[2] += data[starB]['val']
}

tmp[0] ??= {}
for (let starA in data) {
  tmp[1][data[starA]['key']] ??= 0;
  tmp[1][data[starA]['key']] +=
   data[starA]['val']
  tmp[0][data[starA]['key']] =
   tmp[1][data[starA]['key']] /
   tmp[2]
}
return tmp[0]
```

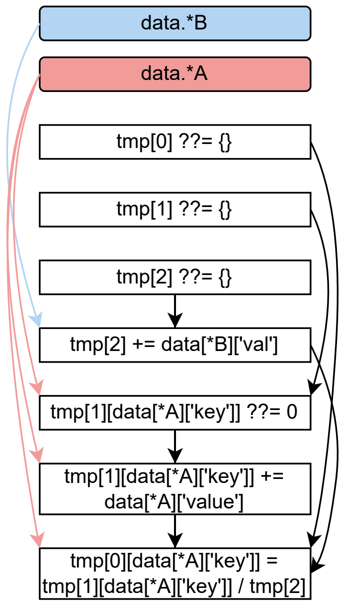

data.*B
data.*A
tmp[0] ??= {}
tmp[1] ??= {}
tmp[2] ??= {}
tmp[2] += data[*B]['val']
tmp[1][data[*A]['key']] ??= 0
tmp[1][data[*A]['key']] += data[*A]['value']
tmp[0][data[*A]['key']] = tmp[1][data[*A]['key']] / tmp[2]

Figure 5.4. A query computing key-specific relative aggregate proportions. Query expression is shown in top left, the constructed IR is shown in bottom, and the final generated code is shown in top right.

In the case of our first query, the initialization logic `tmp[0] ??= 0` and the subsequent sum computation `tmp[0] += ...` are assignments.

Notably, the IR operates without the need for explicit control flow constructs. Instead, the program's structure is implicitly inferred through dependencies. For our example query, the assignment `tmp[0] += ...` has dependencies to both the initializer (because initialization must come first) and the generator (because its operand is the iterated value). Additionally, it is worth noting that computations are performed on temporary state variables (`tmp[0]` in the first example). This approach, utilizing intermediate temporaries, draws inspiration from works such as RPAI [63] and DBToaster [1]. These systems utilize such state variables to maintain values for various sub-queries of the main query, which are then used to compute the final result. Finally, the IR is translated into JS code (and compiled using `eval()`), taking dependencies into account to extract the program structure and performing optimizations as part of this transformation process.

Fig. 5.3 and Fig. 5.4 illustrate two additional queries executed on the same dataset. Specifically, in Fig. 5.3, we compute both the total sum of all values (similar to the query in Fig. 5.2) and the sum of values per key, effectively a group-by sum operation. A key characteristic of Rhyme lies in the contextual interpretation of expressions such as `sum(data.*.val)`. That is, the semantics of this expression differs depending on its context. This is similar to local unification semantics in Verse [198]. For instance, when it is nested within `{data.*.key: ...}`, the expression signifies a group-by sum operation. Conversely, if it is not nested within an iterator (i.e., ∗), it calculates the aggregate over the entire set of values. This distinction is exemplified in the query presented in Fig. 5.4. Here, `sum(data.*A.val)` is nested within `{data.*A.key: ...}`, signifying a group-by aggregate operation. Conversely, `sum(data.*B.val)` computes a total aggregate, as it is not nested within a `*B` iterator.

Additionally, Fig. 5.4 also demonstrates an optimization that occurs at the IR level. Specifically, even though the `sum(data.*B.val)` appears as a nested sub-query in the original query, the Rhyme backend can determine that the generated `*B` can be hoisted out as a separate loop by analyzing the IR dependencies. This is essentially similar to sub-query hoisting that happens at the logical plan level in other traditional query optimizers.

These examples provide a broad overview of Rhyme's functionalities and its syntactic structure. In the subsequent sections of this chapter, we will delve deeper into the syntax and capabilities of Rhyme and discuss the process of IR construction and code generation. Moreover, the previous example queries focused on Rhyme's capability to express analytical queries on JSON objects. However, Rhyme covers an even broader spectrum of use cases, including the ability to express tensor computations and declaratively specify visual components (e.g., tables, charts) of web applications.

Our specific contributions are as follows.

- We introduce the syntax of Rhyme, showcasing the ability to express common data manipulation operators such as selections, group-bys, joins, user-defined functions (UDFs), and others (Section 5.2).
- We highlight the versatility of Rhyme across various use cases, including the expression of visual elements in web applications (e.g., tables, charts using SVG), declarative tensor computations (akin to Einsum), and alternative 'pipe' APIs via metaprogramming (Section 5.3).
- We elucidate the process of lowering queries into an IR that features loop-free and branch-free code, with dependencies implicitly representing the program structure. Then, we illustrate how this IR facilitates code generation by constructing the optimal program structure from dependencies (Section 5.4).
- We evaluate the performance of Rhyme on several JSON analytics workloads to demonstrate the effectiveness of our code generation approach (Section 5.5).

We discuss related work in Section 5.6, followed by conclusions and potential future research directions in Section 5.7.

5.2 The Rhyme Query Language

In the previous section, we saw Rhyme in action for a set of relatively simple queries. In this section, we will introduce the syntax of Rhyme, illustrating how it facilitates the

```
      Ident ::= [a-zA-Z_][a-zA-Z0-9_]*          Atom ::= Ident | Num | Var
        Num ::= [0-9]+                           Path ::= Atom ( . Atom )*
        Var ::= * Ident                          Expr ::=  Path
   ScalarOp ::= get | apply | plus | minus                | Expr ScalarOp Expr
              | div | fdiv | times | mod                  | ReductionOp ( Expr )
ReductionOp ::= sum | count | max | min                   | [ Expr ]
              | first | last | array                      | { ( Path : Expr )* }
                                                Query ::= Expr
```

Figure 5.5. Syntax for expressing Rhyme queries.

expression of common data manipulation operations like selections, aggregates, group-bys, and so on. The formal grammar is shown in Fig. 5.5. The rest of this section illustrates how each of these components are used to express different kinds of queries. To improve understanding, we will employ a running illustrative example dataset, as depicted below. The dataset contains populations of several major cities, along with the respective country. Our chosen dataset is deliberately kept simple, devoid of intricate nested structures. However, Rhyme has the capacity to seamlessly query nested JSON data in the same way. We will see such examples later in Section 5.3.

```
let data = [
  {country: "Japan",  city: "Tokyo",      population: 14},
  {country: "China",  city: "Beijing",    population: 22},
  {country: "France", city: "Paris",      population: 3},
  {country: "UK",     city: "London",     population: 9},
  {country: "Japan",  city: "Osaka",      population: 3},
  {country: "UK",     city: "Birmingham", population: 2}
]
```

5.2.1 Basics

First we will look at how to perform several basic query operations on the aforementioned dataset. For instance, if we want to select a particular key of the dataset at a given index, we can use the following syntax.

```
'data.2.country'              // result: France
{first : 'data.0.country'}    // result: {first: Japan}
```

Several key attributes of Rhyme can be observed from the aforementioned examples. Here, the reference `data` refers to the dataset object, and can be simply indexed through integer indices. Furthermore, specific keys can be selected by specifying the desired key names (e.g., `.country`). Notably, Rhyme offers the convenience of the familiar JS-like syntax for constructing structured output from extracted values, as exemplified in the second instance.

While this form of explicit indexing into the array can be useful for several use cases, generally, queries involve some form of iterating over the dataset. Rhyme offers this capability through the ∗ operator, serving as an implicit iteration operator. Moreover, as we saw in Fig. 5.4, we can perform controlled iteration with multiple generator symbols (e.g., `*A`, `*B`, etc.). In fact, these generator symbols behave like logic variables in Datalog, Prolog, and other logic programming languages as we see in Section 5.3.2. Below, we present three example queries that leverage iterators and compute aggregates over the iterated values.

```
['data.*.city']              // result: [Tokyo, Beijing, ..., Birmingham]
sum('data.*.population')     // result: 53
max('data.*.population')     // result: 22
```

These queries are self-explanatory in nature. In the first example, we illustrate a scenario where an array can be constructed from the values obtained through iteration, employing the `[...]` syntax. Moreover, users can compute aggregates over the iterated values using the relevant aggregate functions, such as `sum`, `max`, and so forth. As discussed previously, these queries can be used as parts of object construction logic and combined flexibly, as shown below.

```
{ total: sum('data.*.population'),
  highest: max('data.*.population') }

// result:
//  {total: 53, highest: 22}
```

5.2.2 Group By

Another vital query operator, especially relevant to JSON-style objects, is the group-by query. Rhyme offers an intuitive means of implicitly expressing group-bys. The following query exemplifies this, grouping records based on the `country` attribute and subsequently calculating the total population for each group:

```
{ 'data.*.country': sum('data.*.population') }
// result: {Japan: 17, China: 22, France: 3, UK: 11}
```

Here, specifying `{data.*.country: ...}` as the key implies that any iteration carried out within this key utilizing the same iterator (∗) is performed for records with each unique value of `country` separately. The next example shows how to use this form of grouping to compute aggregates at different levels. It computes the total population of all records, breaks it down by country, and subsequently computes the population proportion of each city with respect to total population:

```
let query = {
  // total population
  total: sum('data.*.population'),
  'data.*.country': {
    // population per country
    total: sum('data.*.population'),
    // population proportion (per each city)
    'data.*.city': div(sum('data.*.population'),
                       sum('data.*A.population'))}
}
// result: {total: 53.
//          Japan: {total: 17, Tokyo: 0.26, Osaka: 0.06},
//          ...}
```

In the given query, the `sum('data.*.population')` at each query level computes distinct results: total population, total population per country, and population per city, respectively. If we had multiple population values for a given city (e.g., county data), then the last aggregation would compute the per city sum. Since the `sum('data.*A.population')` is not nested within a `*A` key, it performs a total aggregation, which sums all the population values. It is worth noting that our implementation employs calls like `div` for some operators since it functions as an embedded DSL in JS. In a textual frontend, the query could appear even more concise, using standard operators like `/` for division.

5.2.3 Join

Joins are another fundamental operator in data querying. To illustrate how joins work in Rhyme, consider the following new dataset named `other`, which includes information about the `region` to which each country belongs:

```
let other = [
  {country: "Japan",          region: "Asia"},
  {country: "China",          region: "Asia"},
  {country: "France",         region: "Europe"},
  {country: "UK",             region: "Europe"},
]
```

Now, consider a scenario where we aim to compute aggregate population values based on regions. For this, we must perform a join between our original `data` and this new `other` object to acquire the corresponding region for each country. The following Rhyme query illustrates how this is expressed.

```
// create a mapping of country -> region
let countryToRegion = {
  'other.*O.country': 'other.*O.region'
}
// Note - Use of "-" and keyval is because  JS enforces
// JSON keys to be strings and our key is a var
let query = {
  '-': keyval(get(countryToRegion, 'data.*.country'), {
    total: sum('data.*.population')
      'data.*.country' : sum('data.*.population')
    })
}
// result: {Asia: {total:39, Japan:17, China:22},
//          Europe: {total:14, France:3, UK:11}}
```

Here, we use a distinct query (`countryToRegion`) to retrieve the corresponding region for a given country. The main query conducts a group-by based on the `region` (retrieved using `get`) first, followed by another group-by based on `country`, ultimately computing the desired aggregates. We use `'-'` in our implementation due to JS's requirement that JSON keys be represented as strings. Consequently, specifying `get(countryToRegion, 'data.*.country')` as the key directly is not possible. Hence, we introduce `keyval(<key>, <value>)` as a workaround that allows arbitrary arguments to be used as a key. It is worth noting that in a textual frontend, such workarounds would not be necessary.

5.2.4 User-defined Functions

Rhyme allows using user-defined functions (UDFs) written in JS seamlessly with the queries. Consider a simple query where we want to obtain the percentage population per each country from our dataset.

```
let udf = {
  // computes the percentage (and format)
  formatPercent: v => (v*100).toFixed(2) + "%"
}
let query = {
  'data.*.country':
    apply(udf.formatPercent, div(sum('data.*.population'),
                                 sum('data.*A.population')))
}
// result: {Japan: 32.05%, China: 41.50%, France: 5.66%, UK: 20.75%}
```

We have defined the UDF `formatPercent` that, given a proportion value, computes the percentage and adds a % sign at the end. We can then use `apply` in the query to call this UDF to convert the proportions to percentages.

5.2.5 Rhyme AST

As we saw in Fig. 5.1, all the queries above construct a Rhyme AST representation which serves as the basis for the dependency analysis and IR construction. This AST representation is in JSON format, and closely mirrors the query JSON structure. The difference is, all the calls to reducers like `sum`, `count`, etc. and other operations like `plus`, `minus`, etc. will be translated to explicit objects components with `agg` (or `path`) and the corresponding arguments. `agg` is for reducers (which are stateful), and `path` is for operations simply performs a lookup and some computation. For instance, Fig. 5.4 (left) shows how `sum` is translated to `agg` and `param`, and `div` is translated to `path` and param.

This AST representation serves as the entry point for our compiler backend, and gets translated into an IR, as elucidated in Section 5.4. Moreover, as depicted in Fig. 5.1, the creation of this AST is not limited to the aforementioned JS embedding. Instead, it can be constructed using different APIs, (e.g., Fluent API introduced in Section 5.3.1), or a textual frontend with a parser, and so on.

5.3 Case Studies

5.3.1 Fluent API

Rhyme's query frontend (JSON) we saw in Section 5.2 describes the query using the structure of the computed *result*. However, sometimes, it is more natural to start from the structure of the *input*, and specify a sequence of transformation steps. Given that our frontend is embedded in JS, we can use metaprogramming to layer a LINQ-style [199] pipeline API on top.

To illustrate the advantages of such an interface, consider a simple task borrowed from Advent of Code 2022 [200]. The task involves processing a sequence of values partitioned into chunks, each containing multiple values. The objective is to calculate the sum of values for each chunk and subsequently identify the maximum sum among those computed. To begin, let us examine how the Rhyme query appears when utilizing the familiar JSON-style API for data parsing and computation. First, we define several user-defined functions (UDFs) to assist with data parsing. The role of each UDF is simple and self-explanatory.

```
let input = '100,200,300400500,600700,800,9001000' // sample input
// some UDFs for parsing the data
let udf = {
  'splitPipe' : x => x.split(''),
  'splitComma': x => x.split(','),
  'toNum'         : x => Number(x)
}
```

Shown below is the Rhyme query responsible for executing the required computation, with comments provided alongside to elucidate each section of the query (numbered for clarity):

```
let query = max(get({
  // 5. find maximum among group sums
  '*chunk': sum(
    // 4. group-by chunk and compute sum
    apply('udf.toNum',
      // 3. convert each string number to a number object
      get(apply('udf.splitComma',
        // 2. split by comma to get numbers of each chunk
        // 1. split into chunks
        get(apply('udf.splitPipe', '.input'), '*chunk')),
        '*line')))
  },'*'))
```

Function `apply()` is used to apply UDFs to arguments; `get()`, when used with an unbounded generator symbol (e.g., `*chunk`), binds the iterator to the object in the first argument. For example, in Line 5, `get(..., '*chunk')` binds the iterator `*chunk` to the result of splitting the output by the pipe symbol. While this approach works as intended and yields the correct results, for these kinds of workloads, it is more natural to think starting from the input instead of the output structure. In such cases, Rhyme's 'fluent' interface offers an alternative way to express this query concisely as shown below.

```
let query =
  pipe('.input')
    // 1. split into chunks (and bind to *chunk)
    .map('udf.splitPipe').get('*chunk')
    // 2. split by comma to get numbers of each chunk
    .map('udf.splitComma').get('*line')
    // 3. convert each string number to a number object
    .map('udf.toNum')
    // 4. group-by chunk and compute sum
    .sum().group('*chunk').get('*')
    // 5. find maximum among group sums
    .max()
```

Here, we specify the query as a sequence of transformation steps on the input. This high-level fluent API essentially functions as a metaprogramming layer that generates an equivalent Rhyme AST as before. The `pipe()` function creates a `Pipe` object equipped with methods `sum`, `max`, and so on, all of which return a `Pipe`. The `map()` function, similar to `apply()` mentioned earlier, is employed to apply a UDF, and `group()` is used to perform a group-by (i.e., `e.group(x)` is `{x : e}`).

5.3.2 Tensor Expressions

Rhyme provides an elegant framework for expressing tensor computations, drawing inspiration from the Einstein summation (Einsum) notation frequently employed in tensor frameworks and Einops [201]. The Einsum notation offers a concise means of articulating tensor computations. For instance, $\mathrm{i}k, k\mathrm{j} \rightarrow \mathrm{ij}$ specifies a standard matrix product that takes two two-dimensional tensors (i.e., matrices A and B), and yields a third tensor (say C), as the result, computed as $C_{\mathrm{ij}} = \sum_k A_{\mathrm{i}k} \times B_{k\mathrm{j}}$. Likewise, complex tensor computations involving multiple n-dimensional tensors can be specified using such declarative expressions.

Rhyme provides a similar way to express tensor computations in a declarative fashion. To perform tensor computations, we rely on using the notion of unbounded iterators in Rhyme. Specifically, in prior cases, we explicitly specified the data source from which we iterate, as seen in constructs like `data.*A`. However, if instead we only specify the iterator as the key, Rhyme's backend automatically determines the appropriate data source by examining the query body. For instance, when we have a query like `{*i: sum(times(A.*i, B.*i))}`, the backend selects either `A` or `B` as the data source for iteration. Subsequently, the generated code ensures that the iterated values exist in both `A` and `B`. This concept essentially parallels the notion of unification in logic programming, and more specifically narrowing in functional logic programming[198, 202].

To demonstrate how tensor computations are expressed in Rhyme, let's consider some examples. While Rhyme accommodates tensors in various nested formats, for the sake of simplicity, we will consider a scenario where we represent tensors using JS Arrays, as illustrated below:

```
let A = [[1, 2], [3, 4]]; let B = [[1, 2, 3], [4, 5, 6]]
```

Shown below are a set of example tensor computations expressed in Rhyme. Einsum notation counterparts (which closely mirrors Rhyme query structure) are shown in parentheses for each example.

One benefit of having a unified query language for both data manipulation and tensor computations is the ability to handle combined workloads efficiently. To illustrate this, consider a simplified version of computing a city's 'crime index' (taken from [16]). We first

```
// tensor transpose (ij->ji)
{'*j': {'*i': 'B.*i.*j'}}
// sum of all elements (ij->)
sum('B.*i.*j')
// column sum (ij->j)
{'*j': sum('B.*i.*j')}
// row sum (ij->i)
{'*i': sum('B.*i.*j')}
// dot product (vector-vector)
   (i,i->)
sum(times('vecA.*i', 'vecB.*i'))
```

```
// matrix multiplication (ik,kj->ij)
{'*i': {'*j': sum(
  times('A.*i.*k', 'B.*k.*j')) }}
// Hadamard product (ij,ij->ij)
{'*i': {'*j': times('A.*i.*j',
   'B.*i.*j') }}
// general tensor contraction (n-d
   Tensors)
// e.g., pqrs,tuqvr->pstuv
{'*p':{'*s':{'*t':{'*u':{'*v':sum(
  times('T1.*p.*q.*r.*s',
   'T2.*t.*u.*q.*v.*r')) }}}}}
```

select a set of features of cities (e.g., population, adult population and number of robberies), followed by a dot product with a predefined weight vector.

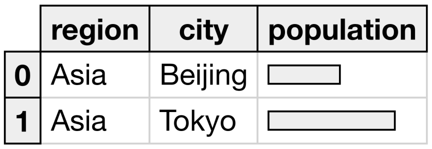
```
let cityVec = { 'data.*.city':
        ['data.*.pop', 'data.*.adultPop', 'data.*.numRobs'] }
let weightVec = [1.0, 1.0, -2000.0] // weight of each feature
let crimeIndex = {    // dot product
  'data.*.city': sum(times(
        'weightVec.*i',
        get(get(cityVec, 'data.*.city'), '*i')))
} // computes the crime index for each city
```

We can specify both the data manipulation component (i.e., the projection) and the tensor computation (i.e., dot product) within the same query language, and we generate a unified code for the combined task. While we kept the example simple for brevity, this has the potential to optimize practical intricate workloads that combine data processing with tensor computations.

5.3.3 Declarative Visualizations

Since Rhyme is embedded within JS, we can extend its capabilities by introducing a means to *declaratively* specify the visual components of websites using a similar structural approach. This allows the seamless integration of data querying logic with the corresponding data visualization logic, such as creating tables. This is enabled by a special key called `'$display'`. To illustrate this, consider the following example query, alongside its corresponding output:

```
{
  '$display': 'table'            // display data in a table
  rows: [0], cols: [1], // row:data index, col:keys
  data: [
    {region:"Asia",city:"Beijing",
      "population":{'$display':"bar",value:40}},
    {region:"Asia",city:"Tokyo",
      "population":{'$display':"bar",value:70}}
  ]
}
```



	region	city	population
0	Asia	Beijing	
1	Asia	Tokyo	

The query above produces the visualization shown on the right. Specifically, it declaratively specifies to display a table that has `data` as the underlying data. This data can be some raw JSON or another Rhyme query. Notice that we can mix, and manipulate these components as valid values inside Rhyme queries, and compose them as necessary. For instance, the progress bars are manipulated as values in the query. These visualizations can take various forms, including standard DOM elements like `h1`, `p`, or high-level components like `table`, `bar`, `select`, and more, as well as SVG objects.

To exemplify the practical utility of this approach, consider a scenario involving the visualization of data related to mobile phone supplier warehouses. Imagine the raw data is represented as an array of JSON objects, each featuring attributes such as warehouse, product, model, and quantity. Before delving into the main query, shown below is a helper query. This auxiliary query calculates the sum of quantities, generates a formatted percentage total, and presents this information as a progress bar displaying the corresponding percentage.

```
let computeEntry = {
  'Quantity': sum('data.∗.quantity'),
  'Percent': apply('udf.formatPercent',
   div(sum('data.∗.quantity'),sum('data.∗B.quantity'))),
  'Bar Chart': {
    '$display': 'bar',
    value: apply('udf.percent', div(sum('data.*.quantity'),
   sum('data.*B.quantity')))
  }
}
```

As previously discussed, the semantics of these aggregations varies depending on the calling context. For instance, when invoked within the context of `{'data.*.warehouse': ...}`, the aggregates computed on ∗ iterators are grouped based on the `warehouse` attribute. Below, we present a query that leverages the above sub query and visualizes our data in a pivot table. This query performs aggregations at multiple levels and displays each level of aggregate in a single table as shown on the right. Naturally, this repeated structure could be abstracted further into a single operation such as `rollup('data.*', [model, product, warehouse], computeEntry)`.

We can similarly use Rhyme to visualize data in different types of charts using SVG graphics. Regarding how this visualizations are handled in the backend, we start by building the necessary helper functions to create these components (tables, bars, etc.) programmatically. Then, the backend is augmented to use these functions whenever a `'$display'` is encountered. Our ultimate goal of these kinds of integrations is to enable users to use Rhyme to build interactive dashboards and CRUD applications directly from a single query.

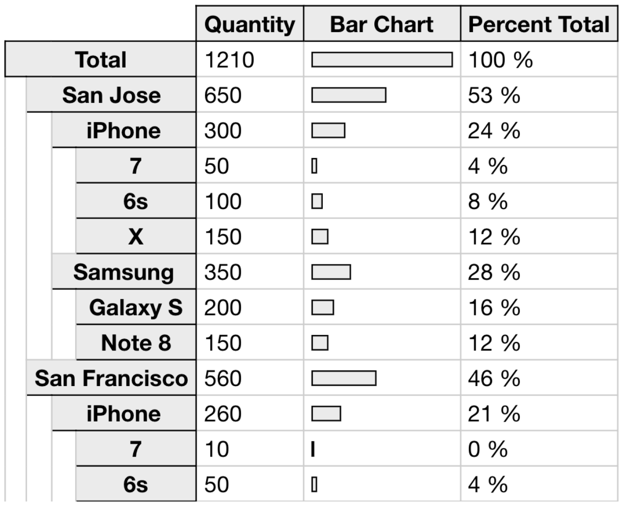
5.4 IR and Code Generation

Up to this point, we have provided an introduction to the syntax of Rhyme and explored its versatility across various domains. In Fig. 5.1 and Figs. 5.2 to 5.4, we gained a preliminary understanding of how Rhyme queries are transformed into an IR, which subsequently serves as the basis for generating optimized code. In this section, we will delve into a detailed discussion of the IR structure and the code generation process.

5.4.1 IR Structure

As discussed in Section 5.1 (Figs. 5.2 to 5.4), the IR structure of Rhyme consists of two primary types of instructions: *generators* and *assignments*. Generators correspond to iterators responsible for enumerating input or intermediate nested objects, and these gen-

```
let query = {
  '$display': 'table',
  ...
  data: { Total: {
    props: computeEntry,        // total aggregate
    children: {'data.*.warehouse': {
      props: computeEntry,      // warehouse-level aggr
      children: 'data.*.product': {
        props: computeEntry,    // product-level aggr
        children: {
          'data.*.model':       // model-level aggr
              computeEntry
        }
  ...
}
```



	Quantity	Bar Chart	Percent Total
Total	1210	▭	100 %
San Jose	650	▭	53 %
iPhone	300	▭	24 %
7	50	▯	4 %
6s	100	▯	8 %
X	150	▭	12 %
Samsung	350	▭	28 %
Galaxy S	200	▭	16 %
Note 8	150	▭	12 %
San Francisco	560	▭	46 %
iPhone	260	▭	21 %
7	10	|	0 %
6s	50	▯	4 %

erators are transformed into loops in the generated code. Assignments, on the other hand, encompass any form of computation that updates or initializes an intermediate or output state.

Rhyme queries inherently exhibit nested iterating structures that could be simply translated into a series of nested loop structures in the generated code. However, performing this transformation naively and enforcing the 'program structure' implied by the user query would lead to missed optimization opportunities like hoisting computations and loops that are independent of outer loops, common sub-query/expression elimination, and more. Therefore, rather than naively transforming queries and directly imposing the program structure, we extract a set of generators, iterators, and their dependencies during IR construction. While the IR does not explicitly capture the program structure, the optimal program structure can be derived from an analysis of these dependencies.

5.4.2 Constructing the IR

The dependency structure is relatively straightforward. As demonstrated in the generated code snippet in Figs. 5.2 to 5.4, we utilize objects such as `tmp[0]`, `tmp[1]`, and so on, to maintain intermediate results required for computing the final query result. Assignment operators have these temporaries as operands, creating data dependencies in the process. Generators can also iterate values from these temporaries, and in such cases, we introduce similar dependencies for the generators. Similarly, when a generator symbol is used in an assignment or another generator, a dependency is added. To illustrate how dependencies are created, consider the following assignment instruction extracted from Line 14 in Fig. 5.4:

```
tmp[0][data[starA]['key']] = tmp[1][data[starA]['key']] / tmp[2]
```

This instruction relies on `tmp[0]`, `tmp[1]`, and `tmp[2]` as operands. As a result, this instruction is associated with the last (write) operations of all three temporaries as dependencies, as depicted in the IR visualization presented in Fig. 5.4. Furthermore, since it employs the `*A` iterator, it also exhibits a dependency on the corresponding generator.

The final missing piece in our lowering process is determining the appropriate IR instruction from our query AST. This is done distinctly for reduction operators (those with a `key` in the AST) and other operators (those with a `path` in the AST). For reduction operators, which require the management of state, we introduce stateful temporary variables indexed by the current grouping path. In this context, 'path' refers to the pertinent 'parent' grouping keys within the query nest. In contrast, other types of operators are simpler to handle. We retrieve the operands and subsequently create an instruction that performs the desired computation. For example, for `plus`, we retrieve the left and right operands and create a binary operation utilizing the `+` operator.

5.4.3 Code Generation

Once the IR is constructed for a query, the next step is to generate the final code taking into account the instruction dependencies. Specifically, it entails determining where to insert loops, how to nest loops within one another, where to place assignment instructions, and so on. We will use the query from Fig. 5.4 as a running example for this section.

The first step is computing two auxiliary relations: `tmpInsideLoop` and `tmpAfterTmp`. These relations track which temporaries should be scheduled inside particular loops and which temporaries should be scheduled after certain other temporaries. This can be done by analyzing assignment to assignment dependencies and generator to assignment dependencies.

The next step involves computing the relation `tmpAfterLoop` based on the two relations computed above. Specifically, if we determined that a temp variable `t2` should be scheduled after another temp variable `t1` (i.e., `tmpAfterTmp[t2][t1]`), which resides inside a loop `l` (with the condition that `t2` itself is not within loop `l`), then this implies that `t2` should be scheduled after the loop `l`. For instance, in our sample query, `tmp[0]` should be scheduled after `*A`.

Subsequently, as the final analysis step before code generation, we determine `loopAfter-Loop` and `loopInsideLoop`. These essentially help identify how the loops should be scheduled. In particular, if we ascertain that a given temp `t` should be scheduled inside both loops `l1` and `l2`, it implies that `l1` and `l2` should belong to the same loop nest. Conversely, if we determined that for a particular temp `t`, it resides within loop `l2`, and we also know that `t` should be scheduled after another loop `l1`, then this indicates that the loop `l2` should be scheduled after `l1` (provided they are not part of the same loop nest).

Once the analysis steps are completed, we proceed with the code generation process. Our approach to code generation draws inspiration from the IR scheduling algorithm utilized in Lightweight Modular Staging (LMS) [80, 135]. In particular, we schedule generators and assignments in an 'outside-in' fashion, commencing with the outer loops before progressing to the inner ones.

Since we did not have program control structures enforced from the front end, this code scheduling mechanism freely schedules assignments and generators in an optimal manner.

For instance, any generator that does not have dependencies to the 'outer query' would be hoisted and scheduled as a separate query instead of repeating the computation multiple times inside a nested loop.

5.5 Experiments

In this section, we conduct a performance evaluation of our current Rhyme implementation, comparing it against two established JSON processing systems: JQ [195] and Rumble [203], the latter of which utilizes Spark for distributed processing. We acknowledge that systems like Rumble are primarily designed for large-scale, cluster execution and may not exhibit optimal performance in a single-node, single-threaded context. Nevertheless, we consider it a valuable baseline for comparison.

5.5.1 Experimental Setup

We run three queries on a simple synthetic dataset comprising 1 million records of JSON objects. Each object in the dataset contains two string keys, `key1` and `key2`, as well as an integer `value`. The first query calculates the sum of all `value`s, the second query performs an aggregate sum after grouping by the `key1`, and the third query computes a two-level aggregate using `key1`, then `key2`.

We run all the experiments using a single thread, on a NUMA machine with 4 sockets, 24 Intel(R) Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total) running Ubuntu 18.04.4 LTS. We have used JQ v1.6, Rumble v1.21.0, and Node v18.18.0 (for running our JS code). All experiments are run five times, reporting the mean execution time.

5.5.2 Results

Across the three queries, Rhyme demonstrates the best performance. This can be attributed to its ability to generate optimized JS code tailored to a specific query, which significantly reduces overhead compared to the general execution engines employed by sys-

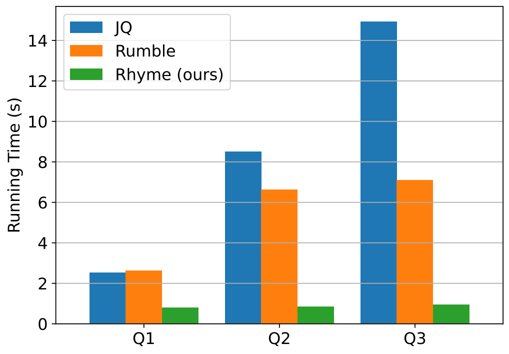


Figure 5.6. Running time (left) for JQ, Rumble, and Rhyme for three different queries

tems like Rumble. JQ performs the worst, as it lacks any form of 'query planning' and executes queries naively without optimizations such as loop fusion.

While these results highlight the potential performance gains achievable through Rhyme's code generation capabilities, it is important to note that this benchmark does not provide a comprehensive analysis that contains the full spectrum of representative cases in JSON analytics. Such a comprehensive evaluation is deferred to future research.

5.6 Related Work

There are several query languages designed for working with semi-structured data like JSON, each with its own focus and strengths. JSONiq [204, 205] is a notable query language explicitly tailored for JSON data, borrowing most of its syntax from XQuery [196] (e.g., FLWOR expressions). Zorba [206] and RumbleDB [203] are examples of engines that support JSONiq, with RumbleDB using Spark [39] as a backend, leveraging the scalability of Spark for execution. AsterixDB, designed for semi-structured data, employs AQL [207] and SQL++ [208] as its query languages. GraphQL [194], on the other hand, is widely used in web application development for querying data from backend services. Most of these languages are specifically targeted towards large-scale JSON analytics workloads and are not expressive enough to support cases like the ones in Sections 5.3.1 to 5.3.3. While we take inspiration from these languages for the design of the language, Rhyme is designed to handle various forms of nested data (e.g., tensors), and it offers support for efficient code generation and optimizations through its IR.

Rhyme's path expressions are inspired by JSONPath [197] (a descendent of XPath). Although not discussed extensively in this chapter, it is possible to extend Rhyme to support the full set of JSONPath operators, which include conditions, recursion, etc. Rhyme also borrows many ideas from functional logic programming languages like Verse [198], Curry [202], miniKanren [209] and Scalogno [155], and adapts them into a new data-centric declarative language.

5.7 Conclusions and Future Work

In this chapter, we introduced Rhyme, a new query language tailored for high-level data manipulation. We illustrated how Rhyme's design facilitates query optimization and code generation through the construction of an IR. This IR comprises loop-free branch-free code, with program structure implicitly captured by dependencies. Throughout the chapter, we demonstrated the versatility of Rhyme by showcasing its applicability in expressive data manipulations, tensor computations, manipulation of visual aspects, and so on in a declarative manner. It is worth noting that Rhyme is still in its early developmental stages, and we are excited about exploring various avenues of interesting future work.

Incrementality While not extensively covered in this chapter, the concept of utilizing intermediate temporaries within the generated code is inspired by prior works in incremental execution, such as DBToaster [1] and RPAI [63]. An immediate focus of our future work involves introducing support for incremental execution. Notably, our generated code is inherently designed to be 'incremental-friendly'. This implies that we have the capability to generate code akin to update triggers, which are invoked whenever a modification is made to the dataset. Specifically, instead of dense loops used in the current version, update triggers generate 'sparse' loops in the sense that they iterate only over the deltas.

Another dimension of incrementality involves managing query changes. This entails finding ways to accommodate changes in queries while maximizing the utilization of previously computed temporaries and sharing state across multiple queries. Such an approach will be useful in interactive applications, where users can dynamically modify their queries.

Performance While the ability to generate JS for browser-based execution is undeniably valuable, there are specific scenarios where optimizing performance becomes paramount. In such cases, the generation of low-level, specialized C code becomes imperative to eliminate any potential overhead associated with managed runtimes. A substantial body of prior research has already demonstrated the efficacy of such compilation mechanisms [19, 20, 81, 210]. Furthermore, it is feasible to leverage existing compiler infrastructures such as LMS [80] or MLIR [133] for streamlined handling of tasks like IR construction, dependency analysis, and the eventual generation of highly specialized low-level code.

6. CONCLUSION

This dissertation explored challenges in modern compilation-based data processing systems. Specifically, it identified several competing demands resulting from changes in hardware, the accumulation of large volumes of diverse data, the need for near-real-time data processing, and the expanding scope of data analytics beyond traditional relational processing and more into workloads that combine multiple paradigms.

This dissertation first tackled the challenge of multi-paradigm workloads from two angles. Firstly, it proposed an efficient post-hoc integration of individual systems using generative programming via the construction of common intermediate layers. This approach preserved the best-of-breed performance of individual workloads while achieving state-of-the-art performance for combined workloads. Secondly, it introduced a high-level query language capable of expressing various workload types, acting as a general substrate to implement combined workloads. This allowed the generation of optimized code for end-to-end workloads through the construction of an intermediate representation (IR).

The dissertation then shifted its focus to data processing systems used for incremental view maintenance (IVM). While existing IVM systems achieved high performance through compilation and novel algorithms, they had limitations in handling specific query classes. Notably, they were incapable of handling queries involving correlated nested aggregate subqueries. To address this, it proposed a novel indexing scheme based on a new data structure and a corresponding set of algorithms that fully incrementalized such queries. This approach resulted in substantial asymptotic speedups and order-of-magnitude performance improvements for workloads of practical importance.

Finally, the dissertation explored efficient and expressive fixed-point computations, with a focus on Datalog—a language widely used for declarative program analysis. Although existing Datalog engines relied on compilation and specialized code generation to achieve performance, they lacked the flexibility to support extensions required for complex program analysis. This dissertation introduced a new Datalog engine built using generative programming techniques that offered both flexibility and state-of-the-art performance through specialized code generation.

REFERENCES

[1] C. Koch *et al.*, “Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views,” *VLDB J.*, vol. 23, no. 2, pp. 253–278, 2014.

[2] B. Scholz, H. Jordan, P. Subotic, and T. Westmann, “On fast large-scale program analysis in datalog,” in *CC*, ACM, 2016, pp. 196–206.

[3] M. Madsen, M. Yee, and O. Lhoták, “From datalog to flix: A declarative language for fixed points on lattices,” in *PLDI*, ACM, 2016, pp. 194–208.

[4] A. Bembenek, M. Greenberg, and S. Chong, “Formulog: Datalog for smt-based static analysis,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 141:1–141:31, 2020.

[5] S. Abeysinghe, F. Wang, G. Essertel, and T. Rompf, “Architecting intermediate layers for efficient composition of data management and machine learning systems,” *arXiv preprint arXiv:2311.02781*, 2023.

[6] M. Jasny, T. Ziegler, T. Kraska, U. Röhm, and C. Binnig, “DB4ML - an in-memory database kernel with machine learning support,” in *SIGMOD Conference*, ACM, 2020, pp. 159–173.

[7] M. E. Schüle, H. Lang, M. Springer, A. Kemper, T. Neumann, and S. Günnemann, “In-database machine learning with SQL on gpus,” in *SSDBM*, ACM, 2021, pp. 25–36.

[8] X. Feng, A. Kumar, B. Recht, and C. Ré, “Towards a unified architecture for in-rdbms analytics,” in *SIGMOD Conference*, ACM, 2012, pp. 325–336.

[9] L. Chen, A. Kumar, J. F. Naughton, and J. M. Patel, “Towards linear algebra over normalized data,” *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1214–1225, 2017.

[10] M. Schleich, D. Olteanu, M. A. Khamis, H. Q. Ngo, and X. Nguyen, “A layered aggregate engine for analytics workloads,” in *SIGMOD Conference*, ACM, 2019, pp. 1642–1659.

[11] J. M. Hellerstein *et al.*, “The madlib analytics library or MAD skills, the SQL,” *CoRR*, vol. abs/1208.4165, 2012.

[12] A. Vaswani *et al.*, “Attention is all you need,” in *NIPS*, 2017, pp. 5998–6008.

[13] Apache, *Apache arrow*, https://arrow.apache.org.

[14] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *OSDI*, USENIX Association, 2016, pp. 265–283.

[15] A. Paszke *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *NeurIPS*, 2019, pp. 8024–8035.

[16] S. Palkar *et al.*, “A common runtime for high performance data analysis,” in *CIDR*, www.cidrdb.org, 2017.

[17] H. Pirk, O. R. Moll, M. Zaharia, and S. Madden, “Voodoo - A vector algebra for portable database performance on modern hardware,” *Proc. VLDB Endow.*, vol. 9, no. 14, pp. 1707–1718, 2016.

[18] A. K. Sujeeth *et al.*, “Delite: A compiler architecture for performance-oriented embedded domain-specific languages,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, 134:1–134:25, 2014.

[19] R. Y. Tahboub, G. M. Essertel, and T. Rompf, “How to architect a query compiler, revisited,” in *SIGMOD Conference*, ACM, 2018, pp. 307–322.

[20] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, “Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data,” in *OSDI*, USENIX Association, 2018, pp. 799–815.

[21] T. Rompf and M. Odersky, “Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls,” in *GPCE*, ACM, 2010, pp. 127–136.

[22] F. Wang, D. Zheng, J. M. Decker, X. Wu, G. M. Essertel, and T. Rompf, “Demystifying differentiable programming: Shift/reset the penultimate backpropagator,” *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, 96:1–96:31, 2019.

[23] F. Wang, J. M. Decker, X. Wu, G. M. Essertel, and T. Rompf, “Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming,” in *NeurIPS*, 2018, pp. 10 201–10 212.

[24] T. Neumann, “Efficiently compiling efficient query plans for modern hardware,” *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 539–550, 2011.

[25] T. T. P. Council., *TPC-H Version 2.15.0.*

[26] D. Amodei *et al.*, “Deep speech 2 : End-to-end speech recognition in english and mandarin,” in *ICML*, ser. JMLR Workshop and Conference Proceedings, vol. 48, JMLR.org, 2016, pp. 173–182.

[27] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size,” *CoRR*, vol. abs/1602.07360, 2016.

[28] J. M. Decker *et al.*, “Snek: Overloading python semantics via virtualization,” https://www.cs.purdue.edu/homes/rompf/papers/decker-preprint201907.pdf.

[29] S. Palkar *et al.*, “Evaluating end-to-end optimization for data analytics applications in weld,” *Proc. VLDB Endow.*, vol. 11, no. 9, pp. 1002–1015, 2018.

[30] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis and transformation,” San Jose, CA, USA, Mar. 2004, pp. 75–88.

[31] W. Taha and T. Sheard, “Metaml and multi-stage programming with explicit annotations,” *Theor. Comput. Sci.*, vol. 248, no. 1-2, pp. 211–242, 2000.

[32] W. Taha, “A gentle introduction to multi-stage programming, part II,” in *GTTSE*, ser. Lecture Notes in Computer Science, vol. 5235, Springer, 2007, pp. 260–290.

[33] T. Rompf *et al.*, “Optimizing data structures in high-level programs: New directions for extensible compilers based on staging,” in *POPL*, ACM, 2013, pp. 497–510.

[34] M. Armbrust *et al.*, “Spark SQL: relational data processing in spark,” in *SIGMOD Conference*, ACM, 2015, pp. 1383–1394.

[35] PostgreSQL, *Postgresql*, https://github.com/postgres/postgres.

[36] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch, “How to architect a query compiler,” in *SIGMOD Conference*, ACM, 2016, pp. 1907–1922.

[37] F. Wang and T. Rompf, "A language and compiler view on differentiable programming," in *ICLR (Workshop)*, OpenReview.net, 2018.

[38] Y. Futamura, "Partial evaluation of computation process-an approach to a compiler-compiler," *Systems, Computers, Controls*, vol. 25, pp. 45–50, 1971.

[39] M. Zaharia *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud*, USENIX Association, 2010.

[41] M. J. Fischer, "Lambda-calculus schemata," *LISP Symb. Comput.*, vol. 6, no. 3-4, pp. 259–288, 1993.

[42] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, Austin, TX, vol. 445, 2010, pp. 51–56.

[43] M. Odersky and M. Zenger, "Scalable component abstractions," in *OOPSLA*, ACM, 2005, pp. 41–57.

[44] T. Rompf, "Reflections on LMS: exploring front-end alternatives," in *SCALA@SPLASH*, ACM, 2016, pp. 41–50.

[45] Kaggle, *Kaggle developer survey*, www.kaggle.com/headsortails/what-we-do-in-the-kernels-a-kaggle-survey-story, 2018.

[46] A. F. Agarap, "Deep learning using rectified linear units (relu)," *CoRR*, vol. abs/1803.08375, 2018.

[47] D. Elliott, S. Frank, K. Sima'an, and L. Specia, "Multi30k: Multilingual english-german image descriptions," in *VL@ACL*, The Association for Computer Linguistics, 2016.

[48] Dask Development Team, *Dask: Library for dynamic task scheduling*, https://dask.org, 2016. [Online]. Available: https://dask.org.

[49] A. K. Sujeeth *et al.*, “Composition and reuse with compiled domain-specific languages,” in *ECOOP*, ser. Lecture Notes in Computer Science, vol. 7920, Springer, 2013, pp. 52–78.

[50] A. Shaikhha, M. Schleich, A. Ghita, and D. Olteanu, “Multi-layer optimizations for end-to-end data analytics,” in *CGO*, ACM, 2020, pp. 145–157.

[51] K. J. Brown *et al.*, “Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns,” in *CGO*, ACM, 2016, pp. 194–205.

[52] S. Palkar and M. Zaharia, “Optimizing data-intensive computations in existing libraries with split annotations,” in *SOSP*, ACM, 2019, pp. 291–305.

[53] K. Karanasos *et al.*, “Extending relational query processing with ML inference,” in *CIDR*, www.cidrdb.org, 2020.

[54] A. Kumar, M. Jalal, B. Yan, J. F. Naughton, and J. M. Patel, “Demonstration of santoku: Optimizing machine learning over normalized data,” *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1864–1867, 2015.

[55] A. Kumar, J. F. Naughton, and J. M. Patel, “Learning generalized linear models over normalized data,” in *SIGMOD Conference*, ACM, 2015, pp. 1969–1984.

[56] D. Olteanu and M. Schleich, “F: regression models over factorized views,” *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1573–1576, 2016.

[57] M. Schleich, D. Olteanu, and R. Ciucanu, “Learning linear regression models over factorized joins,” in *SIGMOD Conference*, ACM, 2016, pp. 3–18.

[58] D. Olteanu and M. Schleich, “Factorized databases,” *SIGMOD Rec.*, vol. 45, no. 2, pp. 5–16, 2016.

[59] S. Li, L. Chen, and A. Kumar, “Enabling and optimizing non-linear feature interactions in factorized linear algebra,” in *SIGMOD Conference*, ACM, 2019, pp. 1571–1588.

[60] D. Petersohn *et al.*, “Towards scalable dataframe systems,” *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2033–2046, 2020.

[61] X. Meng *et al.*, “Mllib: Machine learning in apache spark,” *J. Mach. Learn. Res.*, vol. 17, 34:1–34:7, 2016.

[62] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink™: Stream and batch processing in a single engine,” *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[63] S. Abeysinghe, Q. He, and T. Rompf, “Efficient incrementialization of correlated nested aggregate queries using relative partial aggregate indexes (RPAI),” in *SIGMOD Conference*, ACM, 2022, pp. 136–149.

[64] O. Kennedy, Y. Ahmad, and C. Koch, “Dbtoaster: Agile views for a dynamic data management system,” in *CIDR*, www.cidrdb.org, 2011, pp. 284–295.

[65] K. Zeng, S. Agarwal, and I. Stoica, “Iolap: Managing uncertainty for efficient incremental OLAP,” in *SIGMOD Conference*, ACM, 2016, pp. 1347–1361.

[66] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin, “Thrifty query execution via incrementability,” in *SIGMOD Conference*, ACM, 2020, pp. 1241–1256. DOI: 10.1145/3318464.3389756. [Online]. Available: https://doi.org/10.1145/3318464.3389756.

[67] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin, “Intermittent query processing,” *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1427–1441, 2019. DOI: 10.14778/3342263.3342278. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1427-tang.pdf.

[68] M. Nikolic, M. Dashti, and C. Koch, “How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates,” in *SIGMOD Conference*, ACM, 2016, pp. 511–526.

[69] M. Idris, M. Ugarte, and S. Vansummeren, “The dynamic yannakakis algorithm: Compact and efficient query processing under updates,” in *SIGMOD Conference*, ACM, 2017, pp. 1259–1274.

[70] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner, “Conjunctive queries with inequalities under updates,” *Proc. VLDB Endow.*, vol. 11, no. 7, pp. 733–745, 2018.

[71] Q. Wang and K. Yi, “Maintaining acyclic foreign-key joins under updates,” in *SIGMOD Conference*, ACM, 2020, pp. 1225–1239.

[72] K. Tan, C. H. Goh, and B. C. Ooi, "Progressive evaluation of nested aggregate queries," *VLDB J.*, vol. 9, no. 3, pp. 261–278, 2000.

[73] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars, *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008, pp. 231–236.

[74] P. M. Fenwick, "A new data structure for cumulative frequency tables," *Softw. Pract. Exp.*, vol. 24, no. 3, pp. 327–336, 1994.

[75] Oracle. "Treemap (java platform se 8)." (Jun. 2021), [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html.

[76] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *FOCS*, IEEE Computer Society, 1978, pp. 8–21.

[77] R. Sedgewick. "Left-leaning red-black trees." (2008), [Online]. Available: https://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf.

[78] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," in *SIGFIDET Workshop*, ACM, 1970, pp. 107–141.

[79] W. Taha and T. Sheard, "Metaml and multi-stage programming with explicit annotations," *Theor. Comput. Sci.*, vol. 248, no. 1-2, pp. 211–242, 2000.

[80] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls," *Commun. ACM*, vol. 55, no. 6, pp. 121–130, 2012.

[81] T. Rompf and N. Amin, "A SQL to C compiler in 500 lines of code," *J. Funct. Program.*, vol. 29, e9, 2019.

[82] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *SIGMOD Conference*, ACM Press, 1993, pp. 157–166.

[83] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Data Eng. Bull.*, vol. 18, no. 2, pp. 3–18, 1995.

[84] T. Griffin and L. Libkin, "Incremental maintenance of views with duplicates," in *SIGMOD Conference*, ACM Press, 1995, pp. 328–339.

[85] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *ICDE*, IEEE Computer Society, 1995, pp. 190–200.

[86] R. Chirkova and J. Yang, "Materialized views," *Found. Trends Databases*, vol. 4, no. 4, pp. 295–405, 2012.

[87] C. Koch, "Incremental query evaluation in a ring of databases," in *PODS*, ACM, 2010, pp. 87–98.

[88] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner, "Efficient query processing for dynamically changing datasets," *SIGMOD Rec.*, vol. 48, no. 1, pp. 33–40, 2019.

[89] C. A. Galindo-Legaria and M. Joshi, "Orthogonal optimization of subqueries and aggregation," in *SIGMOD Conference*, ACM, 2001, pp. 571–581.

[90] Z. Wang *et al.*, "Grosbeak: A data warehouse supporting resource-aware incremental computing," in *SIGMOD Conference*, ACM, 2020, pp. 2797–2800. DOI: 10.1145/3318464.3384708. [Online]. Available: https://doi.org/10.1145/3318464.3384708.

[91] Z. Wang *et al.*, "Tempura: A general cost-based optimizer framework for incremental data processing," *Proc. VLDB Endow.*, vol. 14, no. 1, pp. 14–27, 2020. DOI: 10.14778/3421424.3421427. [Online]. Available: http://www.vldb.org/pvldb/vol14/p14-wang.pdf.

[92] D. Tang, Z. Shang, W. W. Ma, A. J. Elmore, and S. Krishnan, "Resource-efficient shared query execution via exploiting time slackness," in *SIGMOD Conference*, ACM, 2021, pp. 1797–1810. DOI: 10.1145/3448016.3457282. [Online]. Available: https://doi.org/10.1145/3448016.3457282.

[93] J. Zhou, P. Larson, and H. G. Elmongui, "Lazy maintenance of materialized views," in *VLDB*, ACM, 2007, pp. 231–242. DOI: https://dl.acm.org/doi/10.5555/1325851.1325881. [Online]. Available: http://www.vldb.org/conf/2007/papers/research/p231-zhou.pdf.

[94] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey, "Algorithms for deferred view maintenance," in *SIGMOD Conference*, ACM Press, 1996, pp. 469–480. DOI: 10.1145/233269.233364. [Online]. Available: https://doi.org/10.1145/233269.233364.

[95] F. McSherry, A. Lattuada, M. Schwarzkopf, and T. Roscoe, “Shared arrangements: Practical inter-query sharing for streaming dataflows,” *Proc. VLDB Endow.*, vol. 13, no. 10, pp. 1793–1806, 2020.

[96] J. Karimov, T. Rabl, and V. Markl, “Ajoin: Ad-hoc stream joins at scale,” *Proc. VLDB Endow.*, vol. 13, no. 4, pp. 435–448, 2019.

[97] J. Karimov, T. Rabl, and V. Markl, “Astream: Ad-hoc shared stream processing,” in *SIGMOD Conference*, ACM, 2019, pp. 607–622.

[98] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *SOSP*, ACM, 2013, pp. 423–438.

[99] T. Akidau *et al.*, “Millwheel: Fault-tolerant stream processing at internet scale,” *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, 2013.

[100] A. Arasu *et al.*, “STREAM: the stanford data stream management system,” in *Data Stream Management*, ser. Data-Centric Systems and Applications, Springer, 2016, pp. 317–336.

[101] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: A timely dataflow system,” in *SOSP*, ACM, 2013, pp. 439–455.

[102] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, “Differential dataflow,” in *CIDR*, www.cidrdb.org, 2013.

[103] M. Inc. “Materialize: Event streaming database for real-time applications.” (2021), [Online]. Available: https://materialize.com/.

[104] Scala. “Treemap (scala standard library).” (Jun. 2021), [Online]. Available: https://www.scala-lang.org/api/current/scala/collection/mutable/TreeMap.html.

[105] S. Abeysinghe, A. Xhebraj, and T. Rompf, “Flan: An expressive and efficient datalog compiler for program analysis,” *Proc. ACM Program. Lang.*, vol. 8, no. POPL, 2024.

[106] M. Aref *et al.*, “Design and implementation of the logicblox system,” in *SIGMOD Conference*, ACM, 2015, pp. 1371–1382.

[107] J. Seo, S. Guo, and M. S. Lam, “Socialite: An efficient graph query language based on datalog,” *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1824–1837, 2015.

[108] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, “Big data analytics with datalog queries on spark,” in *SIGMOD Conference*, ACM, 2016, pp. 1135–1149.

[109] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, “Emptyheaded: A relational engine for graph processing,” *ACM Trans. Database Syst.*, vol. 42, no. 4, 20:1–20:44, 2017.

[110] B. T. Loo *et al.*, “Declarative networking: Language, execution and optimization,” in *SIGMOD Conference*, ACM, 2006, pp. 97–108.

[111] A. Flores-Montoya and E. M. Schulte, “Datalog disassembly,” in *USENIX Security Symposium*, USENIX Association, 2020, pp. 1075–1092.

[112] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989, pp. 984–987.

[113] T. W. Reps, “Solving demand versions of interprocedural analysis problems,” in *CC*, ser. Lecture Notes in Computer Science, vol. 786, Springer, 1994, pp. 389–403.

[114] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, “Using datalog with binary decision diagrams for program analysis,” in *APLAS*, ser. Lecture Notes in Computer Science, vol. 3780, Springer, 2005, pp. 97–118.

[115] M. Bravenboer and Y. Smaragdakis, “Strictly declarative specification of sophisticated points-to analyses,” in *OOPSLA*, ACM, 2009, pp. 243–262.

[116] N. Allen, B. Scholz, and P. Krishnan, “Staged points-to analysis for large code bases,” in *CC*, ser. Lecture Notes in Computer Science, vol. 9031, Springer, 2015, pp. 131–150.

[117] A. Sahebolamri, T. Gilray, and K. K. Micinski, “Seamless deductive inference via macros,” in *CC*, ACM, 2022, pp. 77–88.

[118] E. Zhang, *Ekzhang/crepe: Datalog compiler embedded in rust as a procedural macro*, https://github.com/ekzhang/crepe, 2020.

[119] L. Tielen, *Eclair-lang*, https://github.com/luc-tielen/eclair-lang, 2023.

[120] H. Jordan, P. Subotic, D. Zhao, and B. Scholz, “A specialized b-tree for concurrent datalog evaluation,” in *PPoPP*, ACM, 2019, pp. 327–339.

[121] H. Jordan, P. Subotic, D. Zhao, and B. Scholz, “Brie: A specialized trie for concurrent datalog,” in *PMAM@PPoPP*, ACM, 2019, pp. 31–40.

[122] H. Jordan, P. Subotic, D. Zhao, and B. Scholz, “Specializing parallel data structures for datalog,” *Concurr. Comput. Pract. Exp.*, vol. 34, no. 2, 2022.

[123] P. Subotic, H. Jordan, L. Chang, A. D. Fekete, and B. Scholz, “Automatic index selection for large-scale datalog computation,” *Proc. VLDB Endow.*, vol. 12, no. 2, pp. 141–153, 2018.

[124] S. Arch, X. Hu, D. Zhao, P. Subotic, and B. Scholz, “Building a join optimizer for soufflé,” in *LOPSTR*, ser. Lecture Notes in Computer Science, vol. 13474, Springer, 2022, pp. 83–102.

[125] M. Madsen, J. Starup, and O. Lhoták, “Flix: A meta programming language for datalog,” in *Datalog*, ser. CEUR Workshop Proceedings, vol. 3203, CEUR-WS.org, 2022, pp. 202–206.

[126] A. Brahmakshatriya and S. P. Amarasinghe, “Buildit: A type-based multi-stage programming framework for code generation in C++,” in *CGO*, IEEE, 2021, pp. 39–51.

[127] D. Moldovan *et al.*, “Autograph: Imperative-style coding with graph-based performance,” in *MLSys*, mlsys.org, 2019.

[128] G. Wei, O. Bracevac, S. Tan, and T. Rompf, “Compiling symbolic execution with staging and algebraic effects,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 164:1–164:33, 2020.

[129] G. Wei, Y. Chen, and T. Rompf, “Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 126:1–126:32, 2019.

[130] A. Brahmakshatriya and S. P. Amarasinghe, “Graphit to CUDA compiler in 2021 LOC: A case for high-performance DSL implementation via staging with buildsl,” in *CGO*, IEEE, 2022, pp. 53–65.

[131] M. Arntzenius and N. R. Krishnaswami, “Datafun: A functional datalog,” in *ICFP*, ACM, 2016, pp. 214–227.

[132] S. M. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[133] C. Lattner *et al.*, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[134] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky, "Scala-virtualized: Linguistic reuse for deep embeddings," *High. Order Symb. Comput.*, vol. 25, no. 1, pp. 165–207, 2012.

[135] O. Braevac *et al.*, "Graph irs for impure higher-order languages: Making aggressive optimizations affordable with precise effect dependencies," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, 236:1–236:31, 2023.

[136] W. Taha and T. Sheard, "Multi-stage programming with explicit annotations," in *PEPM*, ACM, 1997, pp. 203–217.

[137] E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. F. Duba, "Hygienic macro expansion," in *LISP and Functional Programming*, ACM, 1986, pp. 151–161.

[138] B. Scholz, K. Vorobyov, P. Krishnan, and T. Westmann, "A datalog source-to-source translator for static program analysis: An experience report," in *ASWEC*, IEEE Computer Society, 2015, pp. 28–37.

[139] L. Ryzhyk and M. Budiu, "Differential datalog," in *Datalog*, ser. CEUR Workshop Proceedings, vol. 2368, CEUR-WS.org, 2019, pp. 56–67.

[140] Z. Fan, J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, and J. M. Patel, "Scaling-up in-memory datalog processing: Observations and techniques," *Proc. VLDB Endow.*, vol. 12, no. 6, pp. 695–708, 2019.

[141] T. Rompf and N. Amin, "Functional pearl: A SQL to C compiler in 500 lines of code," in *ICFP*, ACM, 2015, pp. 2–9.

[142] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I* (Principles of computer science series). Computer Science Press, 1988, vol. 14.

[143] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases.* Addison-Wesley, 1995, ISBN: 0-201-53771-0.

[144] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, pp. 1–69, 2015.

[145] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, Citeseer, 1994.

[146] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: New developments in the theory of join algorithms," *SIGMOD Rec.*, vol. 42, no. 4, pp. 5–16, 2013.

[147] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation* (Prentice Hall international series in computer science). Prentice Hall, 1993.

[148] A. K. Chandra and D. Harel, "Horn clauses queries and generalizations," *J. Log. Program.*, vol. 2, no. 1, pp. 1–15, 1985.

[149] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," in *FOCS*, IEEE Computer Society, 2008, pp. 739–748.

[150] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *J. ACM*, vol. 65, no. 3, 16:1–16:40, 2018.

[151] T. L. Veldhuizen, "Triejoin: A simple, worst-case optimal join algorithm," in *ICDT*, OpenProceedings.org, 2014, pp. 96–106.

[152] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, "Adopting worst-case optimal joins in relational database systems," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 1891–1904, 2020.

[153] Y. R. Wang, M. Willsey, and D. Suciu, "Free join: Unifying worst-case optimal and traditional joins," *Proc. ACM Manag. Data*, vol. 1, no. 2, 150:1–150:23, 2023.

[154] J. Vu, "The art of multiprocessor programming by maurice herlihy and nir shavit," *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 52–53, 2011.

[155] N. Amin, W. E. Byrd, and T. Rompf, "Lightweight functional logic meta-programming," in *APLAS*, ser. Lecture Notes in Computer Science, vol. 11893, Springer, 2019, pp. 225–243.

[156] O. Lhoták and K. A. Chung, "Points-to analysis with efficient strong updates," in *POPL*, ACM, 2011, pp. 3–16.

[157] J. Henry, *User defined aggregate*, https://github.com/souffle-lang/souffle/pull/2282/files, 2022.

[158] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" In *HotOS*, USENIX Association, 2015.

[159] Rayon, *Rayon: A data parallelism library for rust*, https://github.com/rayon-rs/rayon, 2022.

[160] J. Wejdenstål, *Dashmap*, https://github.com/xacrimon/dashmap, 2022.

[161] D. Zhao, P. Subotic, and B. Scholz, "Debugging large-scale datalog: A scalable provenance evaluation strategy," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 2, 7:1–7:35, 2020.

[162] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis, "Porting doop to soufflé: A tale of inter-engine portability for datalog-based analyses," in *SOAP@PLDI*, ACM, 2017, pp. 25–30.

[163] S. M. Blackburn *et al.*, "The DaCapo Benchmarks: Java benchmarking development and analysis (extended version)," Tech. Rep. TR-CS-06-01, 2006, http://www.dacapobench.org.

[164] W. C. Benton and C. N. Fischer, "Interactive, scalable, declarative program analysis: From prototype to implementation," in *PPDP*, ACM, 2007, pp. 13–24.

[165] E. Hajiyev, M. Verbaere, and O. de Moor, "*codeQuest:* scalable source code queries with datalog," in *ECOOP*, ser. Lecture Notes in Computer Science, vol. 4067, Springer, 2006, pp. 2–27.

[166] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *PLDI*, ACM, 2004, pp. 131–144.

[167] M. S. Lam *et al.*, "Context-sensitive program analysis as database queries," in *PODS*, ACM, 2005, pp. 1–12.

[168] K. Hoder, N. S. Bjørner, and L. M. de Moura, "μZ- an efficient engine for fixed points with constraints," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 457–462.

[169] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *Datalog*, ser. Lecture Notes in Computer Science, vol. 6702, Springer, 2010, pp. 245–251.

[170] D. Zhao, P. Subotic, M. Raghothaman, and B. Scholz, "Towards elastic incrementalization for datalog," in *PPDP*, ACM, 2021, 20:1–20:16.

[171] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *ICSE*, IEEE, 2019, pp. 1176–1186.

[172] J. M. Patel *et al.*, "Quickstep: A data platform based on the scaling-up approach," *Proc. VLDB Endow.*, vol. 11, no. 6, pp. 663–676, 2018.

[173] T. Szabó, S. Erdweg, and M. Voelter, "Inca: A DSL for the definition of incremental program analyses," in *ASE*, ACM, 2016, pp. 320–331.

[174] T. Szabó, M. Voelter, and S. Erdweg, "Incal: A dsl for incremental program analysis with lattices," in *Proceedings of the International Workshop on Incremental Computing (IC)*, 2017.

[175] T. Szabó, G. Bergmann, S. Erdweg, and M. Voelter, "Incrementalizing lattice-based program analyses in datalog," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 139:1–139:29, 2018.

[176] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–29, 2021.

[177] L. M. de Moura and N. S. Bjørner, "Efficient e-matching for SMT solvers," in *CADE*, ser. Lecture Notes in Computer Science, vol. 4603, Springer, 2007, pp. 183–198.

[178] Y. Zhang, Y. R. Wang, M. Willsey, and Z. Tatlock, "Relational e-matching," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–22, 2022.

[179] Y. Zhang *et al.*, "Better together: Unifying datalog and equality saturation," *CoRR*, vol. abs/2304.04332, 2023.

[180] P. Zucker, "Logging an egg: Datalog on e-graphs," ser. EGRAPHS 2022, San Diego, CA, USA, 2022, pp. 1–6.

[181] A. Sahebolamri, L. Barrett, S. Moore, and K. Micinski, "Bring your own data structures to datalog," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, 264:1–264:26, 2023.

[182] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, ACM Press, 1995, pp. 49–61.

[183] S. Sagiv, T. W. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theor. Comput. Sci.*, vol. 167, no. 1&2, pp. 131–170, 1996.

[184] M. Arntzenius and N. Krishnaswami, "Seminave evaluation for a higher-order functional language," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 22:1–22:28, 2020.

[185] A. Pacak and S. Erdweg, "Functional programming with datalog," in *ECOOP*, ser. LIPIcs, vol. 222, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 7:1–7:28.

[186] K. Krikellas, S. Viglas, and M. Cintra, "Generating code for holistic query evaluation," in *ICDE*, IEEE Computer Society, 2010, pp. 613–624.

[187] C. Freedman, E. Ismert, and P. Larson, "Compilation in the microsoft SQL server hekaton engine," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 22–30, 2014.

[188] G. M. Essertel, R. Y. Tahboub, and T. Rompf, "On-stack replacement for program generators and source-to-source compilers," in *GPCE*, ACM, 2021, pp. 156–169.

[189] G. Wei *et al.*, "Compiling parallel symbolic execution with continuations," in *ICSE*, IEEE, 2023, pp. 1316–1328.

[190] S. Abeysinghe and T. Rompf, "Rhyme: A data-centric expressive query language for nested data structures," in *PADL*, ser. Lecture Notes in Computer Science, Springer, 2024.

[191] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, 1989.

[192] H. Jordan, B. Scholz, and P. Subotic, "Soufflé: On synthesis of program analyzers," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 9780, Springer, 2016, pp. 422–430.

[193] N. Vasilache *et al.*, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018.

[194] GraphQL, *A query language for your api*, https://graphql.org/, Accessed: 2023-09-27.

[195] *Jq manual*, https://jqlang.github.io/jq/manual/, Accessed: 2023-09-27.

[196] *Xquery 3.1: An xml query language*, https://www.w3.org/TR/xquery-31/, Accessed: 2023-09-27, 2017.

[197] S. Goessner, *Jsonpath - xpath for json*, https://goessner.net/articles/JsonPath/, Accessed: 2023-09-27, 2007.

[198] L. Augustsson *et al.*, "The verse calculus: A core calculus for deterministic functional logic programming," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, 2023.

[199] E. Meijer, B. Beckman, and G. M. Bierman, "LINQ: reconciling object, relations and XML in the .net framework," in *SIGMOD Conference*, ACM, 2006, p. 706.

[200] *Advent of code 2022*, https://adventofcode.com/2022/day/1, Accessed: 2023-09-27.

[201] A. Rogozhnikov, "Einops: Clear and reliable tensor manipulations with einstein-like notation," in *ICLR*, OpenReview.net, 2022.

[202] M. Hanus, "Functional logic programming: From theory to Curry," in *Programming Logics*, ser. Lecture Notes in Computer Science, vol. 7797, Springer, 2013, pp. 123–168.

[203] I. Müller, G. Fourny, S. Irimescu, C. B. Cikis, and G. Alonso, "Rumble: Data independence for large messy data sets," *Proc. VLDB Endow.*, vol. 14, no. 4, pp. 498–506, 2020.

[204] *Jsoniq*, https://www.jsoniq.org/, Accessed: 2023-09-27, 2018.

[205] D. Florescu and G. Fourny, "Jsoniq: The history of a query language," *IEEE Internet Comput.*, vol. 17, no. 5, pp. 86–90, 2013.

[206] *Zorba*, https://www.zorba.io/, Accessed: 2023-09-27, 2018.

[207] *The asterix query language (aql)*, https://asterixdb.apache.org/docs/0.9.8/aql/manual.html, Accessed: 2023-09-27.

[208] K. W. Ong, Y. Papakonstantinou, and R. Vernoux, "The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases," *CoRR*, vol. abs/1405.3631, 2014.

[209] W. E. Byrd, "Relational programming in minikanren: Techniques, applications, and implementations," Ph.D. dissertation, Indiana University, 2009.

[210] G. Wei, Y. Chen, and T. Rompf, "Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 126:1–126:32, 2019.