Compilation and Code Optimization for Data Analytics

THÈSE N^{O} 8762 (2018)

PRÉSENTÉE LE 31 AOÛT 2018

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE THÉORIE ET APPLICATIONS D'ANALYSE DE DONNÉES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Amir SHAIKHHA

acceptée sur proposition du jury:

Prof. V. Kuncak, président du jury
Prof. C. Koch, directeur de thèse
Prof. V. Tannen, rapporteur
Dr D. Vytiniotis, rapporteur
Prof. M. Odersky, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2018

To my mother and father,
who have always supported me,
no matter how flawed and ungrateful I was.

Acknowledgements

First of all, I would like to thank Prof. Christoph Koch. Christoph was more than a Ph.D. advisor to me. I will never forget countless deadline nights that he was staying awake with us. He taught us not to be satisfied with low standards and what is the meaning of high-quality research. Apart from that, he provided me with great opportunities for collaborations and gave me the freedom to work on really interesting projects.

I am also grateful to Prof. Martin Odersky. Before accepting to be in my Ph.D. thesis committee, I had the honor of having him as my master thesis supervisor. Being a member of the Scala group (LAMP) was indeed one of the best opportunities that I had during my master studies at EPFL.

I would also like to thank Prof. Val Tannen who have kindly accepted to be a reviewer of my Ph.D. thesis. Also, I am grateful to Dr. Dimitrios Vytiniotis for not only being in my Ph.D. thesis committee but also a great collaborator. Dimitrios was one of my mentors during an internship that I did at Microsoft Research Cambridge (MSRC). I would like to also thank Dr. Andrew Fitzgibbon, Dr. Don Syme, and Dr. Simon Peyton Jones for the valuable lessons they taught me and the intriguing discussions we had during my internship at MSRC.

My Ph.D. was supported by NCCR MARVEL and a Google Ph.D. fellowship. I would like to thank Christoph, Martin, and Simon for their support in my Google Ph.D. fellowship bid.

During my Ph.D., I had the chance to be in the DATA lab. Aleksandar, Daniel L., Immanuel, and Milos have been great colleagues. Simone, our secretary, was always patient and supportive during the 6 years that I was in the DATA lab. Yannis, Mohammad D., Mohammed, and Lionel have been great lab members and collaborators, and we shared many enjoyable deadline nights together. Vojin was my master thesis supervisor and my first research collaborator. I am really grateful to him for being very patient with the (super-)inexperienced and immature version of me. Finally, I would like to thank Lionel for translating my thesis abstract to French.

Also, I would like to thank several bachelor/master students/interns, from whom I have learned a lot: Mohsen, Parand, Stefan, Daniel E., Lewis, Michal, Matthieu, Robin, Kevin, Laurent, and Khayyam.

Being on the second floor of the BC building gave me the opportunity to share very good moments with the greek gang: Stella, Matt, Eleni, and Manos. Also, I had the chance to have several awesome Iranian friends close by, with whom I had amazing coffee breaks: Mohammad Yaghini, Ahmad Agha, Ehsan Mohammadpour, Farnood, Fatemeh Q., and Sharareh.

Being outside my home country would not be possible without an awesome Iranian community. I would like to thank little Helma and Hessam (Ebrahim and Fereshteh), little Amir Ali

(Meysam and Hoda), little Zeynab (Pedram and Fatemeh S.), little Samieh (Fazel and Fatemeh R.), little Dorsa and Diana (Vahid and Maryam), little Bahar and Saleh (Mostafa and Malihe), little Fatemeh Goli (Amirhossein and Negar), Seyed Mohsen, Mojgan, Amin, Mahshid, Mohammad, and Rozhin. I would also like to thank Fatemeh Gh. and her family without whom I would not have been half the man I am now.
The last year of my Ph.D. was one the most challenging (if not the most) years of my life. I am really grateful to many friends for being there for me. To name a few, I would like to thank: Ashkan, Arman, Ehsan Mansouri, Mortez, Reza, Zhaleh, Fatemeh N., Hesam, Farnaz F., Farnaz E., Aida, Bahar, Niloofar P., Abolfazl, Amir Aminifar, and Niloofar Momeni.
Last but not least, I would like to thank my family for their endless love and support. My sister Nazila has always been there for me during the difficult moments of my life. Parinaz has always been a great source of energy by sending me the photos and videos of her kids. I could have never believed that my little sister Nazanin would become so mature by the end of my Ph.D. studies.
I would like to dedicate this thesis to my parents who have always prayed for me and supported me in every moment of my life.

Lausanne, May 2018 Amir Shaikhha

Abstract

The trade-offs between the use of modern high-level and low-level programming languages in constructing complex software artifacts are well known. High-level languages allow for greater programmer productivity: abstraction and genericity allow for the same functionality to be implemented with significantly less code compared to low-level languages. Modularity, object-orientation, functional programming, and powerful type systems allow programmers not only to create clean abstractions and protect them from leaking, but also to define code units that are reusable and easily composable, and software architectures that are adaptable and extensible. The abstraction, succinctness, and modularity of high-level code help to avoid software bugs and facilitate debugging and maintenance.

The use of high-level languages comes at a performance cost: increased indirection due to abstraction, virtualization, and interpretation, and superfluous work, particularly in the form of tempory memory allocation and deallocation to support objects and encapsulation. As a result of this, the cost of high-level languages for performance-critical systems may seem prohibitive.

The vision of *abstraction without regret* argues that it is possible to use high-level languages for building performance-critical systems that allow for *both* productivity and high performance, instead of trading off the former for the latter. In this thesis, we realize this vision for building different types of data analytics systems. Our means of achieving this is by employing compilation. The goal is to compile away expensive language features – to compile high-level code down to efficient low-level code.

Key words: High-level programming languages, Domain-specific languages, Program synthesis, Query processing, Generative programming, Optimizing compilers, Abstraction without regret, Database optimization, Data analytics.

Résumé

Les compromis entre l'utilisation de langages de programmation modernes de haut niveau et ceux de bas niveau dans la construction d'artefacts logiciels complexes sont bien connus. Les langages de haut niveau permettent une plus grande productivité des programmeurs : l'abstraction et la généricité permettent d'implémenter la même fonctionnalité avec beaucoup moins de code que dans les langages de bas niveau. La modularité, l'orientation objet, la programmation fonctionnelle et les systèmes de types puissants permettent aux programmeurs non seulement de créer des abstractions aux frontières bien définies, mais aussi de définir des unités de code réutilisables et facilement composables, et des architectures logicielles adaptables et extensibles. L'abstraction, la concision et la modularité du code de haut niveau aident à éviter les bogues logiciels et facilitent le débogage et la maintenance.

L'utilisation de langages de haut niveau a un coût au niveau des performances : augmentation de l'indirection due à l'abstraction, à la virtualisation et à l'interprétation, et travail superflu, notamment sous forme d'allocation de mémoire temporaire et de désallocation pour supporter les objets et l'encapsulation. En conséquence, le coût des langages de haut niveau pour les systèmes à performance critique peut sembler prohibitif.

La vision de *l'abstraction sans regret* soutient qu'il est possible d'utiliser des langages de haut niveau pour construire des systèmes à performance critique qui permettent à *la fois* la productivité et la haute performance, au lieu de devoir troquer l'un pour l'autre. Dans cette thèse, nous réalisons cette vision pour la construction de différents types de systèmes d'analyse de données. Notre moyen d'y parvenir est de recourir à la compilation. L'objectif est de faire disparaître à la compilation les fonctionnalités coûteuses du langage pour transformer le code de haut niveau en code de bas niveau efficace.

Mots clefs : Langages de programmation de haut niveau, Langage dédié, synthèse de programmes, traitement des requêtes, programmation générative, compilateurs d'optimisation, abstraction sans regret, optimisation de bases de données, analyses de données.

Contents

List of Figures

List of Tables

1 Introduction

The trade-offs between the use of modern high-level and low-level programming languages in constructing complex software artifacts are well known. High-level languages allow for greater programmer productivity: abstraction and genericity allow for the same functionality to be implemented with significantly less code compared to low-level languages. Modularity, object-orientation, functional programming, and powerful type systems allow programmers not only to create clean abstractions and protect them from leaking, but also to define code units that are reusable and easily composable [260], and software architectures that are adaptable and extensible. The abstraction, succinctness and modularity of high-level code help to avoid software bugs and facilitate debugging and maintenance.

The use of high-level languages comes at a performance cost: increased indirection due to abstraction, virtualization, and interpretation, and superfluous work, particularly in the form of temporary memory allocation and deallocation to support objects and encapsulation.

As a result of this, the cost of high-level languages for performance-critical systems may seem prohibitive. Nevertheless, we have recently witnessed a shift towards the use of high-level programming languages for systems development. Examples include the Singularity Operating System [161], and the Spark [377] and DryadLINQ [374] frameworks for distributed data processing. These approaches collide with the traditional wisdom which calls for using low-level languages like C for building high-performance systems, and the authors of these works find that the advantages of high-level programming languages in combination with their creative contributions offset the performance penalties. Yet, we have not seen this trend to take root in data analytics systems, specially databases where systems continue to be written in low-level languages.

The vision of *abstraction without regret* [287] for database systems [203, 204] argues that it is possible to use high-level languages for building database systems that allow for *both* productivity and high performance, instead of trading off the former for the latter (cf. Figure 1.1). In this thesis, we realize this vision for building data analytics systems.

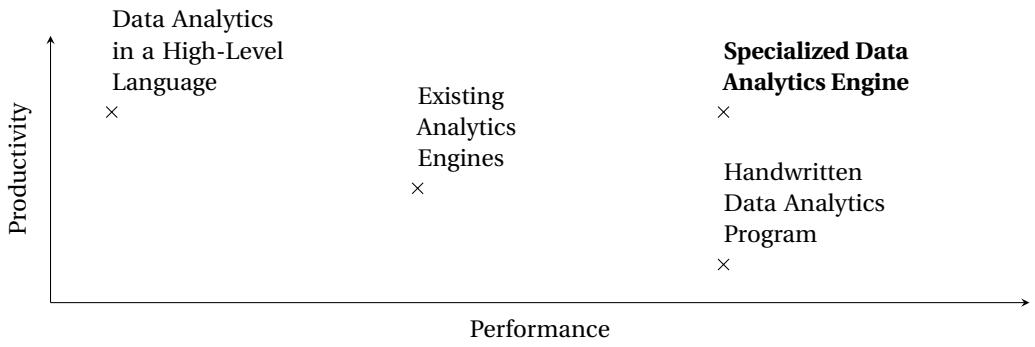


Figure 1.1 – Comparison of the performance/productivity trade-off for all approaches presented in this thesis.

Nowadays, data analytics goes beyond the "simple" analytics workloads; data scientists often use sophisticated statistical and machine learning models to gain insights into data [144]. These "complex" analytics workloads impose new challenges which are not addressed by the traditional relational database systems [318, 319].

In this thesis, we present data analytics systems for both traditional simple analytics workloads, expressed by languages inspired by relational algebra such as SQL, and complex analytics workloads, expressed by languages inspired by linear algebra such as MATLAB and R. All of the systems presented in this thesis are implemented in Scala, a high-level programming language. Our means of achieving high performance for these systems is employing compilation. The goal is to compile away expensive language features – to compile high-level code down to efficient low-level code.

This thesis is derived from the following papers, which are published in or are under submission to conferences and journals in programming languages (PL) and databases (DB) communities:

- Amir Shaikhha, Yannis Klonatos, Christoph Koch
  Building Efficient Query Engines in a High-Level Language [300]
  ACM Transactions on Database Systems (TODS), 2018

- Amir Shaikhha, Mohammad Dashti, Christoph Koch
  Push versus Pull-Based Loop Fusion in Query Engines [298]
  Journal of Functional Programming (JFP), 2018

- Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, Christoph Koch
  How to Architect a Query Compiler [301]
  SIGMOD, 2016

- Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, Dimitrios Vytiniotis

Destination-Passing Style for Efficient Memory Management [299]
FHPC, 2017

- Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, Dimitrios Vytiniotis, Christoph Koch
  Efficient Differentiable Programming in a Functional Array-Processing Language
  Under submission, 2018

- Amir Shaikhha, Mohammed ElSeidy, Daniel Espino, Stefan Mihaila, Christoph Koch
  Synthesis of Incremental Analytics
  Under submission, 2018

Building the compilation infrastructure of the systems used in this thesis resulted in the following papers published in programming languages venues, but are not included in this thesis:

- Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, Martin Odersky
  Yin-Yang: concealing the deep embedding of DSLs [177]
  GPCE, 2014

- Lionel Parreaux, Amir Shaikhha, Christoph Koch
  Quoted staged rewriting: a practical approach to library-defined optimizations [268]
  GPCE, 2017 **(Best Paper Award)**

- Lionel Parreaux, Antoine Voizard, Amir Shaikhha, Christoph Koch
  Unifying analytic and statically-typed quasiquotes [270]
  POPL, 2017

- Lionel Parreaux, Amir Shaikhha, Christoph Koch
  Squid: type-safe, hygienic, and reusable quasiquotes [269]
  Scala, 2017

Finally, the techniques and insights presented in this thesis have contributed to building the underlying systems behind the following publications, which are also not included in this thesis:

- Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, Amir Shaikhha
  DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views [205]
  VLDBJ, 2014

- Mohammad Dashti, Sachin Basil John, Amir Shaikhha, Christoph Koch
  Transaction Repair for Multi-Version Concurrency Control [82]
  SIGMOD, 2017

- Mohammad Dashti, Sachin Basil John, Thierry Coppey, Amir Shaikhha, Vojin Jovanovic, Christoph Koch
  Compiling Database Application Programs [83]
  http://arxiv.org/pdf/1807.09887v1.pdf, 2018

1.1 Background

The DB community has developed many techniques for optimizing data analytics tasks which are usually expressed in a declarative language (such as SQL). We give a brief overview of these techniques.

Algebraic Optimizations. Relational database management systems on relational algebra to expose their declarative query interfaces to the end users. Thanks to its mathematical nature, this algebra enables many opportunties for optimizations, such as join reordering and pushing down selections and projections.

Materialization. Data analytics programs can produce several intermediate results during their execution. The intermediate results that are used several times can be *materialized* [61] to avoid recomputation of the same value. There are also cases where datasets evolve through changes that are small relative to the overall dataset size. Recomputing data analytics on every slight dataset change is far from efficient. *Incremental View Maintenance* [39, 205, 138] (IVM) is a technique for computing only the required changes to the query instead of recomputing the whole query from scratch.

Pipelining. In many cases, the intermedate results of a data analytics program are used only once. In such cases materializing such collections of elements incurs additional runtime and memory cost. Database systems use *operator pipelining* for streaming data items through query operators, which removes the need for storing such unnecessary intermediate collections.

Memory Management. The input data and the final output data are stored in a storage medium, such as main memory, HDD, or SSD. A data analytics engine needs to transfer data through the memory hierarchy for performing the actual computation. Additionally, data analytics systems need additional memory for other purposes such as intermediate data-structures [145]. Data analytics systems can rely on garbage collectors for allocating/deallocating the memory, which can have a performance overhead. Instead, these systems can use in-memory data-structures maintaining contiguous memory space, called *memory pools*, which avoid the overhead caused by garbage collectors.

Query Planning. There are different algorithmic and data-structure choices for each relational algebra construct. For example, for implementing a join operator, one can use a hash-table data-structure (i.e., the hash-join operator) or a B-Tree data-structure (i.e., the index-nested loop join). Choosing an appropriate data-structure for different query operators and materil-

ization decisions [125] is performed by the *query optimizer* component of a database system. Query optimization is used for finding an optimized execution plan for a given query.

Differentiation. The scientific programming community has developed various techniques for computing the derivative of a given function. Automatic differentiation (AD) [187] is a technique for automatically computing the derivative of a given program. This makes AD an essential component for optimization algorithms such as different variations of the gradient descent algorithm, which are the core of many current machine learning frameworks.

All the mentioned techniques are usually performed during the runtime of the data analytics engines. The programming languages (PL) and the compilers communities have a massive literature on optimizing programs by moving the computation from run time to compilation time and performing static analysis on them during compilation. Frameworks such as Theano [334] and Tensorflow [5] are examples of data analytics frameworks performing optimizations before the actual execution of the data analytics tasks.

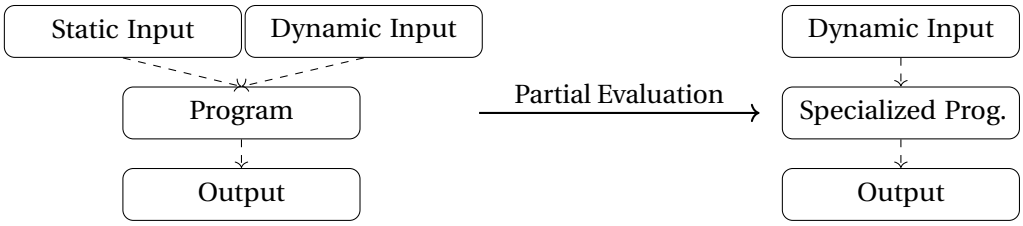
Compiler Optimizations. Optimizing compilers include many optimizations for improving the run-time performance of a given program. Among these optimizations, the *partial evaluation* [116] technique specializes a given program with its static input, so that it produces a specialized program that for a given dynamic input, produces the same output as the original program (cf. Figure 1.2a). Basically, the key benefit of using partial evaluation is pushing a computation that is supposed to be performed during the run time to the compilation time. In programming languages, this idea can be used for specializing an interpreted program with its interpreter for deriving a compiled program, which is known as the first Futamura projection (cf. Figure 1.2b). Furthermore, by specializing a partial evaluator with an interpreter, one can derive a compiler, which is known as the second Futamura projection (cf. Figure 1.2c).

Similary, one can use the idea of partial evaluation to specialize a data analytics program with its data processing engine in order to derive a speciliazied data analytics engine (cf. Figure 1.2d). This specialization process can be performed in various stages resulting in various types of specialized data analytics engines.

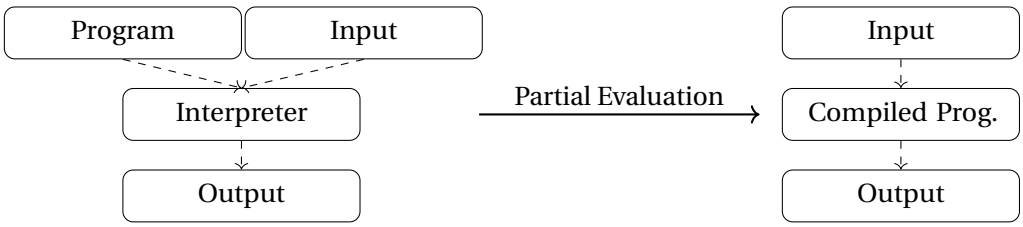
Program Analysis. In many cases, we do not have the actual data needed for running the program. Instead, we only know some properties about the data, and we are only interested in reasoning about similar properties of the output. The PL community have used the *abstract interpretation* [70] technique for performing various types of reasoning for programs.

1.2 Thesis Contributions

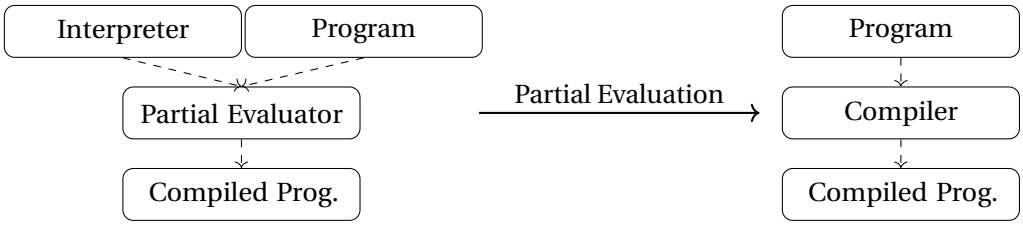
In this thesis, we show how these multi-disciplinary techniques can be used for building various types of data analytics systems (cf. Figure 1.3). More specifically, the contributions of this thesis are summarized as follows:



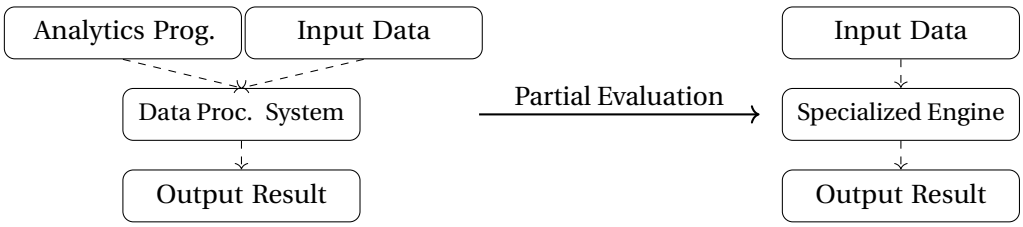
(a) Generic partial evaluation for specializing a program with its static input data, resulting in a specialized program.



(b) Specializing an interpreter with a program resulting in a compiled program, which is known as the first Futamura projection.



(c) Specializing a partial evaluator with an interpreter resulting in a compiler, which is known as the second Futamura projection.



(d) Specializing a data analytics system with a given analytics program resulting in a specialized analytics engine.

Figure 1.2 – Partial evaluation and its application for various cases.

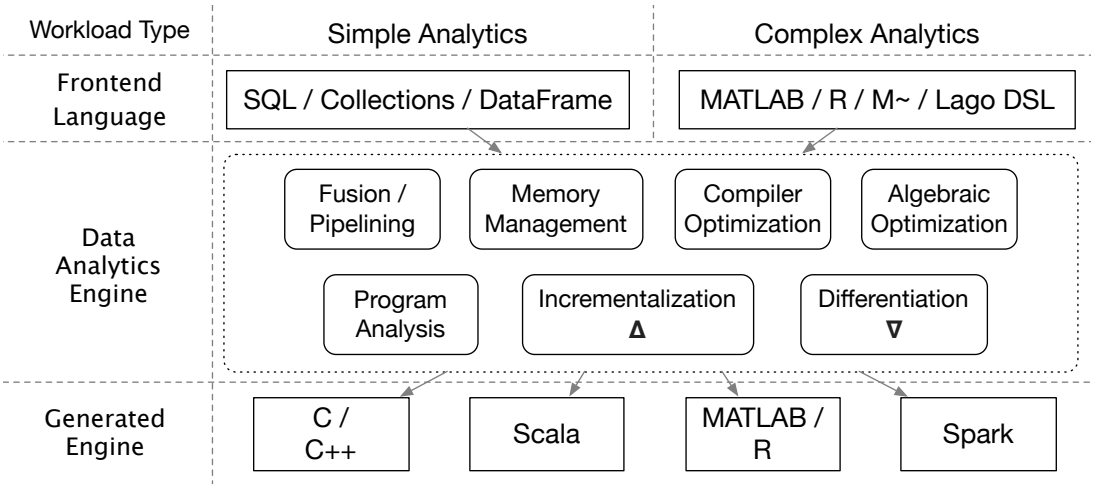


Figure 1.3 – Overall design for data analytics systems presented in this thesis.

- We present the LegoBase system, which uses the idea of partial evaluation in various stages for specializing a high-level implementation of an analytical database system with system parameters, schema, query, or even data (Section 2.2).

- We present various domain-specific languages for building compliation-based analytical database systems (Section 3.3) and more complex data analytics platforms (Section 5.2 and Section 7.3.2).

- We present various transformations for optimizing data analytics expressed in the presented high-level domain-specific languages (Section 2.3, Section 3.5, Section 6.4, and Section 7.3.3).

- In particular, we present the operator pipelining technique from the DB community (Section 4.2), and the loop fusion technique from the PL community (Section 4.3 and Section 5.2.3). We also show how these two techniques are connected, and based on this connection, we propose a new approach for building pipelined query engines (Section 4.4).

- Additionally, we present a new technique for efficient memory management when translating high-level complex data analytics to low-level C code (Section 5.3).

- We show how our designed languages and the associated optimizations can be used for making complex analytical tasks such as differentiable programming more efficient (Section 6.6).

- We show how the idea of abstract interpretation from the PL community (Section 7.4) and the idea of Incremental View Maintenance (IVM) from the DB community (Section 7.2) can be combined for efficient complex incremental analytics.

- We present the Alchemy framework, a compiler-compiler for generating DSL compilers from their implementation in Scala (Section 8.4).
- We present the Systems Compiler (SC), a compiler framework written in the Scala programming language used for optimizing the data anlaytics engines presented in this thesis (Section 8.5).

1.3 Thesis Outline

This thesis is structured as follows:

Simple Analytics. We focus on the realization of the abstraction without regret vision, for ad-hoc main-memory analytical database systems (query engines):

- We show the usage of compilation techniques for producing efficient query processing engines from their implementation in the Scala programming language in Chapter 2.
- We show a principled way of building compiler-based data management systems in Chapter 3. More specifically, we propose to use a stack of multiple DSLs on different abstraction levels through step-wise lowering. This makes query compilers easier to build and extend.
- We show the parallels between pipelined query engines and loop fusion techniques in functional languages in Chapter 4.

Complex Analytics. We go beyond the workloads supported by database query engines. We show the usage of compilation techniques for more complex analytical workloads required for machine learning and computer vision tasks:

- We present a high-level functional array-processing language, a linear-algebra-based language, and a technique for efficiently compiling to low-level C code in Chapter 5. More specifically, we focus on an efficient memory management technique for the generated C code.
- We show how the languages presented in Chapter 5 can be used for producing efficient differentiated code in Chapter 6.
- We show how a similar linear-algebra-based language can be used for incremental processing of advanced analytical workloads in Chapter 7.

Compiler Framework. We show the design of the Alchemy and SC frameworks in Chapter 8. We have used these compilation frameworks for building several data processing systems, including the ones presented in this thesis.

We review the related work in Chapter 9. Finally, in Chapter 10 we conclude the thesis and show future directions based on the results presented in this thesis.

2 Efficient and High-Level Query Engine

> *The hardest thing is to go to sleep at night, when there are so many urgent things needing to be done. A huge gap exists between what we know is possible with today's machines and what we have so far been able to finish.*
> – Donald Knuth

Abstraction without regret refers to the vision of using high-level programming languages for systems development without experiencing a negative impact on performance. A database system designed according to this vision offers both increased productivity and high performance, instead of sacrificing the former for the latter as is the case with existing, monolithic implementations that are hard to maintain and extend.

In this chapter, we realize this vision in the domain of analytical query processing. We present LegoBase, a query engine written in the *high-level* programming language Scala. The key technique to regain efficiency is to apply *generative* programming: LegoBase performs source-to-source compilation and optimizes database systems code by converting the high-level Scala code to specialized, low-level C code. We show how generative programming allows to *easily* implement a wide spectrum of optimizations, such as introducing data partitioning or switching from a row to a column data layout, which are difficult to achieve with existing low-level query compilers that handle *only* queries. We demonstrate that sufficiently powerful abstractions are essential for dealing with the complexity of the optimization effort, shielding developers from compiler internals and decoupling individual optimizations from each other.

2.1 Introduction

We have made a number of key choices:

- We achieve compilation by *generative* programming [330], a technique that allows for the programmatic removal of abstraction overhead through source-to-source compilation. We have implemented a new compiler framework, SC, to achieve this (cf. Chapter 8).

- The implementation of LegoBase, our database system, is direct, succinct, straightforward, and in a sense intentionally naive to be true to the idea of a high-level implementation.
- Systems programming techniques and performance-oriented refinements are applied by SC via code transformations.

  We have developed a library of code transformations that aim to represent the skill set of an experienced systems programmer.[1] These are applied to the LegoBase code by SC to create high-performance code.
- The database system can run stand-alone, albeit inefficiently – without being passed through SC. This means that there are neither lifted engine code[2] nor facilities for code generation in the code base of LegoBase. In particular, the query engine inside the database system is implemented as a query interpreter, which is automatically lifted to a query compiler by SC. (This is known as the *second Futamura projection* in the compilers literature [116].)
- We use Scala, a functional, object-oriented, strongly-typed programming language to implement all of our software artifacts: LegoBase, SC, and the transformations.

Some of these points need to be detailed and justified further.

SC is a compiler infrastructure for domain-specific languages (DSLs) *embedded* [158] in Scala. Simply speaking, such DSLs are fragments of Scala defined by excluding certain language features, data structures, or libraries. Scala is an impure language that allows to express both high-level and low-level programs.

All our code artifacts, including the LegoBase codebase, are written in Scala DSLs that SC is able to process. SC reads in a number of artifacts, such as (parts of) the LegoBase code base, schema, or queries, and emits code for, or executes, a performance-optimized database system. It must be emphasized again, since this is central, that SC is *not limited* to processing a query plan language that is essentially relational algebra as most database systems do. A Scala query plan DSL (Scala with query operator calls) is among the DSLs we work with, but SC can compile and optimize potentially all of LegoBase. One place outside the query engine that currently profits from this is the automatic specialization of storage structures to schema and workload characteristics. In future work, this could be applied to other aspects of database systems, such as the specialization of concurrency control techniques to workloads, or coordination avoidance in distributed database systems [23, 293].

Our transformations can be categorized into optimizing and lowering transformations. We have found that, starting with code in a high-level DSL, by iterating between optimization and

[1] We hope that with the experience of further systems implementations, we will be able to harden this set of transformations into a stable and reusable library.

[2] This refers to code in an intermediate representation, such as an abstract syntax tree.

lowering to lower-level DSLs (see Figure 3.3), we are able to express a large variety of optimizations easily by having them work on the DSL on the abstraction level most natural to them. This is consistent with the proposal of [301]. Programs in our lowest-level Scala DSL, C.Scala, are compositions of operators corresponding to constructs of the C programming language and its core libraries and can be directly stringified to C code, which can be executed outside Scala's virtual-machine-based runtime system, guaranteeing unimpeded performance.

Our compiler infrastructure is based on partial evaluation and can be used to build database systems that employ compilation at a variety of stages, or even in multiple stages – allowing, for instance, the creation of a query compiler at the end of development, and further specialization at the installation, administration, or query processing stages; eagerly, or just in time – specializing the query compiler with system parameters, schema, a query, or even data.

Throughout this article, we use the Scala language. The use of an object-oriented, functional, and strongly-typed programming language is key to achieving the productivity in the development of LegoBase that we report in this article. Functional programming languages have long been known to be particularly well suited for productively implementing compilers, and arguably the language feature most responsible for this is pattern matching, which is absent from Java8 and C++11, both recent additions to the family of functional programming languages. Pattern matching was indispensable for making the implementation of SC as well as our code analyses and transformations manageable for us. Some of the key features of SC such as its powerful type-safe quasiquotation mechanism internally depend on the combination of advanced genericity, support for mix-in composition, a powerful macro system, and a very powerful type system (with dependent types) that in this flavor currently only co-exist in Scala. A port to a number of other functional languages (the better known are Haskell, F# with quotations [327], and OCaml[3]) is likely possible but not straightforward, and would require further original research.

In addition to SC, LegoBase, and the code transformers as discussed above, this article makes the following contributions.

- We demonstrate the ease of use of the new SC compiler for optimizing system components that differ significantly in structure and granularity of operations. We do so by presenting (i) the optimizations applied to the LegoBase query engine and (b) the *high-level* compiler interfaces that database developers need to interact with when coding optimizations. We show that the design and interfaces of SC allow for a number of desirable properties for the LegoBase optimizations. These are expressed as library components, providing a *clean* separation from the base code of LegoBase (e.g. that of query operators), but also from each other. This is achieved, (as explained in Section 2.2) by applying them in multiple, *distinct* optimization phases. Optimizations are (a) adjustable to the characteristics of workloads and architectures, (b) configurable, so that

[3] More precisely, MetaOCaml [328].

they can be turned on and off on demand and (c) composable, so that they can be easily chained but also so that higher-level optimizations can be built from lower-level ones.

For each optimization, we present the *domain-specific* conditions that need to be satisfied in order to apply it (if any) and possible trade-offs (e.g. improved execution time versus increased memory consumption). We examine which categories of database systems can benefit from applying each of our optimizations by providing a classification of the LegoBase optimizations.

- We perform an experimental evaluation in the domain of analytical query processing using the TPC-H benchmark [343]. We show how our optimizations can lead to a system that has performance competitive to that of a standard, commercial in-memory database called DBX (which does not employ compilation) and the code generated by the query compiler of the HyPer database [255]. In addition, we illustrate that these performance improvements do not require significant programming effort as even complicated optimizations can be coded in LegoBase with only a few hundred lines of code. We also provide insights into the performance characteristics and trade-offs of individual optimizations. We do so by comparing major architectural decisions as fairly as possible, using a shared codebase that only differs by the effect of a single optimization. Finally, we demonstrate that our compilation approach incurs negligible overhead to query execution.

The rest of this thesis is organized as follows. Section 2.2 presents the overall design of LegoBase and SC. Section 2.3 gives an in-depth presentation of our optimizing code transformers. Section 2.4 presents our experimental evaluation. Finally, Section 2.5 concludes.

2.2 Architecture and System Design

In this section, we present the design of the LegoBase system. First, we describe the overall system architecture of our approach (Section 2.2.1). Then, we describe in detail how LegoBase uses the SC compiler (Section 2.2.2) as well as how we efficiently convert the high-level database system Scala code (not just that of individual operators) to optimized C code for each incoming query (Section 2.2.3). We give concrete code examples of what our physical query operators, physical query plans, and compiler interfaces look like.

2.2.1 Overall System Architecture

The overall system architecture of LegoBase is shown in Figure 2.1. The architecture of the system is based on the idea of *partial evaluation* [172]. In this technique, a program is considered as a mapping between the input data and output data, where the input data is divided into two distinct sets of *static* and *dynamic* inputs[4]. Partial evaluation transforms

[4] This is standard terminology in the PL and compilers communities.

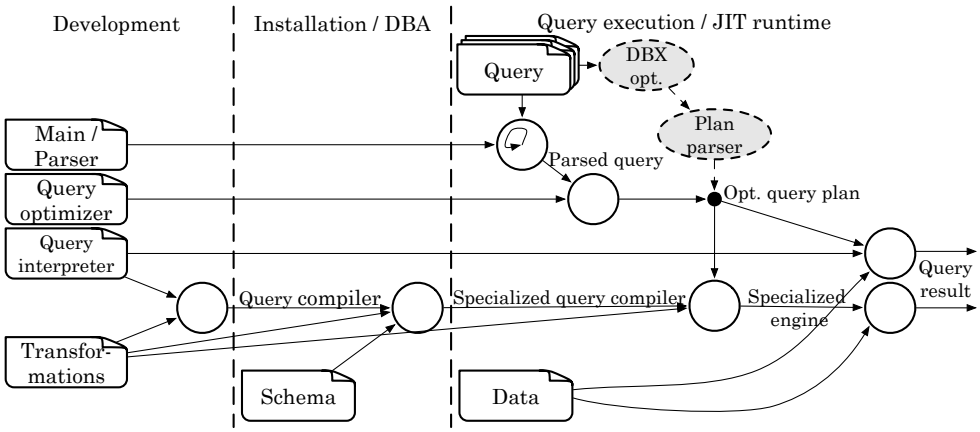


Figure 2.1 – Overall system architecture. The documents represent the inputs, the circles represent the partial evaluation, and the arrows represent the flow of state.

a given program into a specialized program, which only accepts the dynamic input of the original program. This specialized program returns the same output as the original program by specializing the parts which are only dependent on the static input. Typically, we expect the partial evaluator to make as much computation progress with the program as possible given that only the static part of the input is available, and the output program captures the remaining computation still to be done once the dynamic input becomes available.

Based on this definition, a partial evaluator can transform an interpreter given the input program as its static input, into a compiled version of the input program. This process is known as the first Futamura projection in the compilers literature [116]. Also, we can partially evaluate a partial evaluator given an interpreter as its static input (The dynamic input is the input program to be passed to the interpreter), into a compiler, a process known as the second Futamura projection [116]. Note that an evaluator is a special case of a partial evaluator where all the input data is static.

Partial evaluation can be used to build database systems that employ compilation at a variety of stages, or even in multiple stages. In Figure 2.1, we assume three stages:

- **Development stage.** At this stage, the query interpreter and the transformations provided by the database system developer are passed into SC, in order to lift the query interpreter into a query compiler (the second Futamura projection).
- **Installation/DBA stage.** At this stage, the schema of the relations and system configuration parameters are provided or modified, and the constructed query compiler from the previous stage is partially evaluated with this information. The result is a specialized, further optimized query compiler which is specific to the provided schema and configuration parameters (the first Futamura projection).
- **Query execution/JIT stage.** At this stage, each incoming SQL query is passed through a

query optimizer in order to get an optimized query plan which describes the physical query operators needed to process this query. Then, LegoBase parses this optimized plan and constructs an intermediate representation for it, which can be either interpreted or compiled into a specialized query engine (the first Futamura projection) once the user executes it.

However, one can consider additional stages. As an example, in an analytical data processing system, one can assume a separate data loading stage. In this stage, the data is loaded into the appropriate storage structures. Additionally, in this stage one can load data into pre-grouped data structures to be used in queries that can profit from this – an idea that is heavily used in data cubes.

We have not yet implemented a query optimizer.[5] Thus, for our evaluation, we choose the query optimizer of a commercial, in-memory database system, called DBX.[6]

(Partial) evaluation is shown in Figure 2.1 as circles. Even though one could use the partial evaluator in many different stages and for a different set of inputs, we have presented only a reasonable subset of such possibilities.

Next, we give more details about the alternative database systems that can be built at each of three stages presented in Figure 2.1.

Query interpretation. LegoBase can use the produced query plan representation and the input data in order to interpret the query. To do so, we use a library of operator implementations (written in Scala). Our operators (and their implementations) are composable, in the sense that they expose a unified interface so that they can be easily chained. This unified interface can either follow the Volcano model [128] (for a pull-based query engine) or the producer/consumer model [121, 255] (for a push-based query engine, the interface of which is given in the appendix. Figure 2.2 presents an example of how query plans and operators are written in LegoBase, respectively. That is, the Scala code example shown in Figure 2.2a loads the data, builds a functional tree from operator objects and then starts executing the query by passing the elements through these operators.[7] It is important to note that operator implementations like the one presented in Figure 2.2b are exactly what one would write for an

[5] For this work, we consider traditional query optimization (e.g. determining join order) as an orthogonal problem and instead focus more on experimenting with the different optimizations that can be applied *after* query optimization. This means that the focus of the transformations in LegoBase is to specialize the already provided algorithms and query plans, and *not* choose a *different* algorithm or query plan (e.g. join order) in their place. Hence, if a query optimizer chooses a query plan with a worse run time complexity, it would not be currently be possible for SC to compensate this run time complexity difference.

[6] The choice of the DBX query optimizer results from a collaboration with the manufacturer of the commercial system. We plan to implement the classical functionality of a query optimizer by a set of transformers to make LegoBase self-contained. We expect that performance can profit from including the transformer pipeline, i.e. orders of transformation, in the artifact to be optimized. This is future research.

[7] The current prototype of LegoBase targets the optimization of analytical queries. Hence, LegoBase currently assumes that data are loaded only *once* in the beginning, before any query is submitted by the users, and that updates are not taking place in the system.

```
1 def Q6() {
2   val lineitemTable = loadLineitem()
3   val scanOp = new ScanOp(lineitemTable)
4   val startDate = parseDate("1996-01-01")
5   val endDate = parseDate("1997-01-01")
6   val selectOp = new SelectOp(scanOp)
7     (x =>
8       x.L_SHIPDATE >= startDate &&
9       x.L_SHIPDATE < endDate &&
10      x.L_DISCOUNT >= 0.08 &&
11      x.L_DISCOUNT <= 0.1 &&
12      x.L_QUANTITY < 24
13    )
14  val aggOp = new AggOp(selectOp)
15    (x => "Total")
16    ((t, agg) => { agg +
17      (t.L_EXTENDEDPRICE * t.L_DISCOUNT)
18    })
19  val printOp = new PrintOp(aggOp)(
20    kv => printf("%.4f\n", kv.aggs(0))
21  )
22  printOp.open
23  printOp.next
24 }
```

(a)

```
1 class AggOp[B](child:Operator, grp:Record=>B,
2           aggFuncs:(Record,Double)=>Double*)
3  extends Operator {
4    val hm = HashMap[B, Array[Double]]()
5    def open() { parent.open }
6    def process(aggs:Array[Double], t:Record){
7      var i = 0
8      aggFuncs.foreach { aggFun =>
9        aggs(i) = aggFun(t, aggs(i))
10       i += 1
11     }
12   }
13   def consume(tuple:Record) {
14     val key = grp(tuple)
15     val aggs = hm.getOrElseUpdate(key,
16        new Array[Double](aggFuncs.size))
17     process(aggs, tuple)
18   }
19   def next() : Record = {
20     hm.foreach { pair => child.consume(
21       new AGGRecord(pair._1, pair._2)
22     ) }
23   }
24 }
```

(b)

Figure 2.2 – Example of a query plan and an operator implementation in LegoBase. The SQL query used as an input here is actually Query 6 of the TPC-H workload. The operator implementation presented here uses a Push interface [255].

interpreted query engine that does not involve compilation at all. However, without further optimizations, this engine cannot match the performance of existing databases: it consists of generic data structures (e.g. the one declared in line 4 of Figure 2.2b) and involves expensive memory allocations on the critical path,[8] both properties that can significantly affect performance.

Query compilation. LegoBase can use the specialized query compiler produced by SC in the previous stages, in order to specialize the code base of the query engine with respect to the given query plan. SC further specializes the code of the database system on the fly (including the code of individual operators, all data structures used as well as any required auxiliary functions), and progressively optimizes the code using our transformations (described in detail in Section 2.3). For example, it optimizes away the `HashMap` abstraction and transforms it to efficient low-level constructs (Section 2.3.2). In addition, SC utilizes *query-specific* information during compilation. For instance, it will inline the code of all individual operators and, for the example of Figure 2.2b, it automatically unrolls the loop of lines 8-11, since the number of aggregations can be statically determined based on how many aggregations the input SQL query has. Such fine-grained optimizations have a significant effect on performance, as

[8] Note that such memory allocations are not always explicit (i.e. at object definition time through the **new** keyword). For instance, in line 15 of Figure 2.2b, the HashMap may have to expand (in terms of allocated memory footprint) and be reorganized by the Scala runtime in order to more efficiently store data for future lookup operations. We discuss this issue and its consequences to performance further later in this thesis.

they improve branch prediction. Finally, our system generates the optimized C code of the specialized query engine,[9] which is compiled using any existing C compiler.[10] We then return the query results to the user.

Note that, although Figure 2.1 demonstrates many different possibilities for building database systems, in our experiments we use the implementation based on generated C code for a specialized engine of a particular query. This engine is generated using a query compiler which is specialized with respect to a particular schema.

2.2.2 The SC Compiler Framework

As we will later discuss in Chapter 8, the SC compiler was designed from the beginning so that it allows developers to have full control over the optimization process without exporting compiler internals such as code generation templates. It does so by delivering sufficiently powerful programming abstractions to developers like those afforded by modern high-level programming languages. The SC compiler and all optimizations are written in Scala, with its rich language features that support productivity.

In contrast with low-level compilation frameworks like LLVM – which express optimizations using a low-level, compiler-internal intermediate representation (IR) that operates on the level of registers and basic blocks – programmers in SC specify the result of a program transformation as a high-level, *compiler-agnostic* Scala program. As we will shown in Section 8.5.2, SC offers two *high-level* programming primitives named `analysis` and `rewrite` for this purpose, which are illustrated in Figure 2.3a and which analyze and manipulate statements and expressions of the input program, respectively, by using the pattern matching feature of Scala.[11] By expressing optimizations at a high level, our approach enables a user-friendly way to describe these domain-specific optimizations that humans can easily identify, without imposing the need to interact with compiler internals.[12] We use this optimization interface to provide database-specific optimizations as a library and to aggressively optimize our query engine.

To allow for maximum flexibility and expressive power to ultimately generate as efficient code as possible, developers must be able to easily experiment with different optimizations and op-

[9] In this work, we choose C as our code-generation language as this is the language traditionally used for building high-performance database systems. However, SC is not particularly aware of C and can be used to generate programs in other languages as well (e.g. optimized Scala). The generated C code can either be executed in a stand alone manner, or alternatively be dynamically linked into a runtime system.

[10] We use the CLang frontend of LLVM [214] for compiling the generated C code in our evaluation.

[11] For example, the second rule in Figure 2.3a, detects a **`while`** loop expression, and binds the entire expression, the condition, and the body to the variables `loop`, `cond`, and `body`, respectively. Note that, the fourth rewrite rule removes the matched expression. Hence, there is no need to specify any right-hand-side statement for the corresponding pattern matching expression.

[12] Of course, every compiler needs to represent code through an intermediate representation (IR). The difference between SC and other optimizing compilers is that the IR of our compiler is completely hidden from developers: both the input source code and all of its optimizations are written in plain Scala code, which is then translated to an internal IR through Yin-Yang [177].

```
analysis += statement {
  case sym -> code"new MultiMap[_, $v]"
    if isRecord(v) => allMaps += sym
}
analysis += rule {
  case loop @ code"while($cond) $body" =>
    currentWhileLoop = loop
}

rewrite += statement {
  case sym -> (code"new MultiMap[_, _]")
    if allMaps.contains(sym) =>
      createPartitionedArray(sym)
}
rewrite += remove {
  case code"($map: MultiMap[Any, Any])
    .addBinding($elem, $value)"
      if allMaps.contains(map) =>
}
rewrite += rule {
  case code"($map: MultiMap[Any, Any])
    .addBinding($elem, $value)"
      if allMaps.contains(map) =>
      /* Code for processing add Binding */
}
```

(a)

```
pipeline += OperatorInlining
pipeline += SingletonHashMapToValue
pipeline += ConstantSizeArrayToValue
pipeline += ParamPromDCEAndPartiallyEvaluate
if (settings.partitioning) {
  pipeline += PartitioningAndDateIndices
  pipeline += ParamPromDCEAndPartiallyEvaluate
}
if (settings.hashMapLowering)
  pipeline += HashMapLowering
if (settings.stringDictionary)
  pipeline += StringDictionary
if (settings.columnStore) {
  pipeline += ColumnStore
  pipeline += ParamPromDCEAndPartiallyEvaluate
}
if (settings.dsCodeMotion) {
  pipeline += HashMapHoisting
  pipeline += MallocHoisting
  pipeline += ParamPromDCEAndPartiallyEvaluate
}
if (settings.targetIsC)
  pipeline += ScalaToCLowering
// else: handle other languages, e.g. Scala
pipeline += ParamPromDCEAndPartiallyEvaluate
```

(b)

Figure 2.3 – (a) An example of the analysis and rewrite APIs of SC. The examples here are taken from various actual LegoBase optimizations and they are, thus, not self-contained. (b) The SC transformation pipeline used by LegoBase. Details for the listed optimizations are presented in Section 2.3.

timization orderings (depending on the characteristics of the input query or the properties of the underlying architecture). In SC, developers do so by explicitly specifying a *transformation pipeline*. This is a straightforward task as SC transformers act as black boxes, which can be plugged in at any stage in the pipeline. For instance, for the default transformation pipeline of LegoBase, shown in Figure 2.3b, Parameter Promotion, Dead Code Elimination and Partial Evaluation are all applied at the end of each of the custom, domain-specific optimizations. The transformation pipeline takes parameters, which allow to turn off certain optimizations for certain queries (e.g. *settings.partitioning* in Figure 2.3b) at demand as well as specify which optimizations should be applied *only* for specific hardware platforms.

Even though it has been advocated in previous work [290] that having multiple transformers can cause phase-ordering problems, our experience is that system developers can easily rise to the challenge of specifying a suitable order of transformations as they design their system and its compiler optimizations. As we show in Section 2.4, with a relatively small number of transformations we can get a significant performance improvement in LegoBase.

SC already provides many generic compiler optimizations like function inlining, common subexpression and dead code elimination, constant propagation, scalar replacement, partial evaluation, and code motion. In this work, we extend this set to include DBMS-specific optimizations (e.g. using the popular columnar layout for data processing). We describe these

optimizations in more detail in Section 2.3.

2.2.3 Multiple Abstraction Levels

Database systems comprise many components of significantly different nature and functionality, thus typically resulting in very big code bases. To optimize those in a productive way, developers must be able to express new optimizations without having to modify either (i) the base code of the system or (ii) previously developed optimizations. Compilation techniques based on template expansion do not scale to the task, as their single-pass approach forces developers to support multiple code transformers with different optimization roles (such as pipelining and data-structure specialization [301]) at the same time, which makes their debugging and development complicated.

To this end, the SC compiler is built around the principle that, instead of using template expansion to directly generate low-level code from a high-level program in a *single* macro expansion step, an optimizing compiler should instead **progressively lower the level of abstraction** until we reach the lowest level of representation, and only then generate the final, low-level code.

Each level of abstraction and all associated optimizations operating in it can be seen as independent modules, enforcing the principle of *separation of concerns*. Higher levels are more declarative, thus allowing for increased productivity, while lower levels are closer to the underlying architecture, thus making it easy to perform low-level performance tuning. For example, optimizations such as join reordering are only feasible in higher abstraction levels (where the operator objects are still present in the code), while register allocation decisions can only be expressed in the lowest abstraction level[13]. This design provides the nice property that generation of the final code becomes a trivial stringification of the lowest level representation.

More precisely, in order to reach the abstraction level of C code (the lowest level representation for the purposes of this thesis), transformations include multiple *lowering* steps that *progressively* map Scala constructs to (a set) of C constructs. Most Scala abstractions (e.g. objects, classes, inheritance) are optimized away in one of these intermediate stages (for example, hash maps are converted to arrays through the domain-specific optimizations described in Section 2.3), and for the remaining constructs (e.g. loops, variables, arrays) there exists a one-to-one correspondence between Scala and C. SC already offers such lowering transformers for an important subset of the Scala language. For example, classes are converted to structs,

[13] As we already discussed, LegoBase is a query engine designed for *in-memory* processing. This means that our design so far is focused on the use case where all data fit in main memory (including all memory space required for optimizations like horizontal partitioning etc). When one targets data that does not fit in memory, one needs to additionally optimize the access patterns for I/O. Such optimizations, which would occur at the lowest levels of abstraction, are not currently provided by the LegoBase system, but there is nothing in the design of either the query engine or the optimizing compiler that forbids expressing such optimizations. In fact, we have experimented with optimizing the initial data-loading of LegoBase by replacing naive calls to *fscanf* (which result in low-performance due to the increasing number of seeks) with corresponding calls to memory-mapped files obtained through the *mmap* function. It is our belief that the presented solution definitely allows for the easy extension of the transformation pipeline with I/O optimizations.

strings to arrays of bytes, etc.

This way of lowering does not require any modifications to the database code or effort from database developers other than just specifying in SC how and after which abstraction level custom data types and abstractions should be lowered. More importantly, such a design allows developers to create new abstractions in one of their optimizations, which can in turn be optimized away in subsequent optimization passes.

Finally, there are two additional implementation details of our source-to-source compilation from Scala to C that require special mentioning.

First, the final code produced by LegoBase, with all optimizations enabled, does not require library calls. For example, all collection data structures like hash maps are converted in LegoBase to primitive arrays (Section 2.3.2). However, we view LegoBase as a platform for easy experimentation of database optimizations. As a result, our architecture must also be able to support traditional collections as a library and convert, whenever necessary, Scala collections to corresponding ones in C (in the experiments of this article, using those provided by GLib [333]).

Second, and more importantly, the two languages handle memory management in a totally different way: Scala is garbage collected, while C has explicit memory management. Thus, when performing source-to-source compilation from Scala to C, we must take special care to free the memory that would normally be garbage collected in Scala in order to avoid memory overflow. This is a hard problem to solve automatically, as garbage collection may have to occur for objects allocated outside the DBMS code, e.g. for objects allocated inside the Scala libraries. For the scope of this work, we follow a conservative approach and make, whenever needed, allocations and deallocations explicit in the Scala code. We also free the allocated memory after each query execution.

We give more details on designing an appropriate DSL stack for query compilers in Chapter 3.

2.3 Compiler Optimizations

In this section, we present examples of compiler optimizations in six domains:[14] (a) inter-operator optimizations for query plans, (b) transparent data-structure modifications, (c) changing the data layout, (d) using string dictionaries for efficient processing of string operations, (e) domain-specific code motion, and, finally, (f) traditional compiler optimizations like

[14] Note that the optimizations that can be supported by our system are not limited to the ones presented in this section; one can define additional transformations using the transformation API provided by SC in order to support further optimizations. More specifically, one can assume a generic API for a particular component with several implementations. Then, the system can choose the best specialized implementation based on the context of a particular workload. However, as the focus of this thesis is mainly on the adhoc in-memory analytical query processing, we have only focused on optimizations related to the query engine component. Nevertheless, we plan to use a similar approach for optimizing other components of the database system, such as the buffer management, storage management, transaction processing, and concurrency control in future work.

```
def Q12() {
  val ordersScan = new ScanOp(loadOrders())
  val lineitemScan = new ScanOp(loadLineitem())
  val lineitemSelect = new SelectOp(lineitemScan)(record =>
    record.L_RECEIPTDATE >= parseDate("1994-01-01") &&
    record.L_RECEIPTDATE < parseDate("1995-01-01") &&
   (record.L_SHIPMODE == "MAIL" || record.L_SHIPMODE == "SHIP") &&
    record.L_SHIPDATE < record.L_COMMITDATE && record.L_COMMITDATE < record.L_RECEIPTDATE
  )
  val jo = new HashJoinOp(ordersScan, lineitemSelect) // Join Predicate and Hash Functions
    ((ordersRec,lineitemRec) => ordersRec.O_ORDERKEY == lineitemRec.L_ORDERKEY)
    (ordersRec => ordersRec.O_ORDERKEY)(lineitemRec => lineitemRec.L_ORDERKEY)
  val aggOp = new AggOp(jo)(t => t.L_SHIPMODE) // L-SHIPMODE is the Aggregation Key
    ((t, agg) => {
      if (t.O_ORDERPRIORITY == "1-URGENT" || t.O_ORDERPRIORITY == "2-HIGH") agg + 1 else agg
    },
    (t, agg) => {
      if (t.O_ORDERPRIORITY != "1-URGENT" && t.O_ORDERPRIORITY != "2-HIGH") agg + 1 else agg
    })
  val sortOp = new SortOp(aggOp)((x, y) => x.key - y.key)
  val po = new PrintOp(sortOp)(kv => {
    printf("%s|%.0f|%.0f\n", kv.key, kv.aggs(0), kv.aggs(1))
  })
  po.open
  po.next
}
```

Figure 2.4 – Example of an input query plan (TPC-H Q12). We use this query to explain various characteristics of our domain-specific optimizations in Section 2.3.

dead code elimination. The purpose of this section is to demonstrate the ease-of-use of our methodology: that by programming at the high-level, such optimizations are easily expressible without requiring changes to the base code of the query engine or interaction with compiler internals. Throughout this section we use, unless otherwise stated, Q12 of TPC-H, shown in Figure 2.4, as a guiding example in order to better illustrate various important characteristics of our optimizations.

2.3.1 Inter-Operator Optimizations – Eliminating Redundant Materializations

Consider a query in which a join is followed by a group-by, where the grouping of the aggregation and the hashing of the join are both performed on the same attribute. In this example, the generated code includes two materialization points: (a) at the group-by and (b) when materializing the left side of the join. However, there is no need to materialize the tuples of the aggregation in two different data structures as the aggregations can be immediately materialized in the data structure of the join. Such *inter-operator* optimizations are hard to express using *template-based* compilers. Alternatively, one can implement an additional operator (known as *groupjoin* [244]) in the query engine. In contrast to these two approaches, in our system we use high-level programming to easily pattern match on the operators, as we describe next.

By expressing optimizations at a high level, LegoBase can optimize code across operator

```
 1 def optimize(op: Operator): Operator = op match {
 2   case joinOperator@HashJoinOp(aggOp:AggOp, rightChild, joinPred, leftHash, rightHash) =>
 3     new HashJoinOp(aggOp.leftChild, rightChild, joinPred, leftHash, rightHash) {
 4       override def open() {
 5         // leftChild is now the child of aggOp (relation S)
 6         leftChild foreach { t =>
 7           // leftHash hashes according to the attributes referenced in the join condition
 8           val key = leftHash(aggOp.grp(t))
 9           // Get aggregations from the hash map of HashJoin
10           val aggs = hm.getOrElseUpdate(key, new Array[Double](aggOp.aggFuncs.size))
11           // Process all aggregations using the original code of Aggregate Operator
12           aggOp.process(aggs,t)
13         }
14       }
15     }
16   case op: Operator =>
17     op.leftChild = optimize(op.leftChild)
18     op.rightChild = optimize(op.rightChild)
19   case null => null // Operators with only one child have rightChild set to null.
20 }
```

Figure 2.5 – Removing redundant materializations by high-level programming (here between a group by and a join). This code follows the semantics of a Volcano-style engine, but the optimization can be similarly applied to a push engine. The code of the Aggregate Operator is given in Figure 2.2b.

interfaces; we can treat operators as Scala objects and match specific optimizations to certain chains of operators. Here, we can completely remove the aggregate operator and merge it with the join, thus eliminating the need of maintaining two data structures. The code of this optimization is shown in Figure 2.5.

This optimization operates as follows. First, we call the optimize function, passing it the top-level operator as an argument. The function then traverses the tree of Scala operator objects, until it encounters a proper chain of operators to which the optimization can be applied to. As shown in line 2 of Figure 2.5, for this example this chain is a hash-join operator connected to an aggregate operator. When this pattern is detected, a new HashJoinOp operator object is created, that is *not* connected to the aggregate operator, but instead to the child of the latter (first function argument in line 3 of Figure 2.5). As a result, the materialization point of the aggregate operator is completely removed. However, we must still find a place to (a) store the aggregate values and (b) perform the aggregation. For this purpose we use the hash map of the hash join operator (line 10), and we just call the corresponding function of the Aggregate operator (line 12), respectively. The rest of join-related processing remains unchanged.

We observe that this optimization is programmed on the same level of abstraction as the rest of the query engine: as normal Scala code. This fact demonstrates that when coding optimizations at a high level of abstraction (e.g. to optimize the operators' interfaces), developers no longer have to worry about low-level concerns such as code generation (as is the case with existing approaches) – these concerns are simply addressed by later stages in the transformation pipeline. Both these properties raise the productivity provided by our system, showing the merit of developing database systems using high-level programming languages.

2.3.2 Data-Structure Specialization

Data-structure optimizations contribute significantly to the complexity of database systems, as they tend to be heavily specialized to be workload, architecture and (even) query-specific. Our experience with the PostgreSQL database system reveals that there are many distinct implementations of the memory page abstraction and B-trees. These versions are *slightly* divergent from each other, suggesting that the optimization scope is limited. However, this situation contributes to a *maintenance nightmare* as in order to apply any code update, many different pieces of code have to be modified.

Implementing data-structure specialization in existing template-based query compilers is difficult, due to their low-level nature. Thanks to the unified high-level interface provided by SC, our approach can be used to optimize the database systems' Scala code, and not only the operator interfaces, enabling various degrees of specialization in data structures, as has been previous shown in [290].

In this article, we demonstrate such possibilities by explaining how our compiler can be used to: 1. Optimize the data structures used to hold in memory the data of the input relations, 2. optimize Hash Maps which are typically used in intermediate computations like aggregations, and, finally, 3. automatically infer and construct indices for SQL attributes of `date` type. We do so in the next three sections.

Data Partitioning

Optimizing the structures that hold the data of the input relations is an important form of data-structure specialization, as such optimizations generally enable more efficient join processing. This is true even for multi-way, join-intensive queries. In LegoBase, we perform data partitioning when loading the input data. We analyze this optimization, the code of which can be found in the appendix, next.

In LegoBase, developers can annotate the primary and foreign keys of their input relations at schema definition time. Using this information, our system then creates optimized data structures for those relations as follows.

First, for each input relation that has a *single-attribute* primary key, LegoBase creates a 1D-array which is accessed through the primary key specified for that relation. This is a usually a straightforward task, as the primary keys frequently have values in the range of $[1...\#num\_tuples]$. However, even when the primary key is not in a continuous value range, LegoBase currently aggressively trades-off system memory for performance, and stores the

```
1 // Sequential accessing for the ORDERS table (since it has smaller size)
2 for (int idx = 0 ; idx < ORDERS_TABLE_SIZE ; idx += 1) {
3   int O_ORDERKEY = orders_table[idx].O_ORDERKEY;
4   struct LINEITEMTuple* bucket = lineitem_table[O_ORDERKEY];
5   for (int i = 0; i < counts[bucket]; i+=1)
6     // process bucket[i] -- a tuple of the LINEITEM table
7 }
```

Figure 2.6 – Using primary and foreign keys in order to generate code for high-performance join processing. The underlying storage layout is that of a row-store for simplicity. The `counts` array holds the number of elements that exist in each bucket.

input data in a sparse array[15]. For our running example, LegoBase creates a 1D array for the ORDERS table, indexed through the O_ORDERKEY attribute.

Second, for composite primary keys as well as for foreign keys, LegoBase replicates and repartitions the data of the corresponding input relations to form multiple two-dimensional arrays, each indexed by one such key, where each bucket holds all tuples having a particular value for that key[16]. We resolve the case where the composite primary/foreign key is not in a contiguous value range by trading-off system memory, in a similar way to how we handle single-attribute primary keys. For the example of Q12, LegoBase creates four partitioned tables: one for the foreign key of the ORDERS table (O_CUSTKEY), one for the composite primary key of the LINEITEM table (as described above), and, finally, two more for the foreign keys of the LINEITEM table (on L_ORDERKEY and L_PARTKEY/L_SUPPKEY respectively).

Observe that for relations that have multiple foreign keys, not all corresponding partitioned input relations need to be kept in memory at the same time, as an incoming SQL query may not need to use all of them. To decide which partitioned tables to load, LegoBase depends on the physical query execution plan (e.g. referenced attributes and join conditions), but also on simple to estimate statistics, like cardinality estimation of the input relations. For example, for Q12, out of the two partitioned, foreign-key data structures of LINEITEM, our optimized code uses only the partitioned table on L_ORDERKEY, as there is no reference to L_PARTKEY or L_SUPPKEY in the query.

These partitioned data structures can be used to significantly improve join processing, as they allow to quickly extract matching tuples for a join between two relations on a primary-foreign key relationship. This is best illustrated through Q12 and the join between the LINEITEM

[15] It is also possible to define a transformation in order to introduce a dictionary for densifying the domain (c.f. Section 2.3.4). With this technique, one can reduce the memory footprint required compared to the current approach of LegoBase for sparse keys. However, this dictionary-based transformation performs a dictionary lookup for the original column during a hash join, which introduces additional overhead compared to the approach currently followed by LegoBase. This dictionary lookup can be avoided if the original column is not needed for the other parts of the query.

[16] Creating a 1D array is not possible for composite primary and foreign keys, as we would need to *uniquely* hash *all* attributes of such a key. Deriving such a hash function in full generality would require knowledge of the whole dataset in advance. More importantly, calculating the hash would introduce additional computation on the critical path, leading to a significant negative impact on performance.

and ORDERS tables. For this query, LegoBase (a) infers that the ORDERKEY attribute represents a primary-foreign key relationship and (b) uses statistics to derive that ORDERS is the smaller of the two tables. By utilizing this information, LegoBase can generate the code shown in Figure 2.6 to directly get the corresponding bucket of LINEITEM (by using the value of the ORDERKEY attribute), thus avoiding the processing of a possibly significant number of LINEITEM tuples.

LegoBase uses this approach for multi-way joins as well, to completely eliminate the overhead of intermediate data structures. This results in significant performance improvement as a number of expensive system calls responsible for the tuple copying between these intermediate data structures is completely avoided.

With this optimization, LegoBase can remove most of the overhead of using generic data structures for join processing. However, there are still some hash maps remaining in the generated code. These are primarily hash maps used for calculating aggregations and hash maps for joins on attributes that are *not* represented by a primary/foreign key relationship. In these cases, LegoBase lowers these maps to two-dimensional arrays as we discuss in our hash map lowering optimization in the next section.

Optimizing Hash Maps

By default, LegoBase uses GLib [333] hash tables for generating C code out of the HashMap constructs of the Scala language. Close examination of these generic hash maps in the baseline implementation of our operators (e.g. in the Aggregation of Figure 2.2b) reveals the following three main abstraction overheads.

First, for every *insert* operation, a generic hash map must allocate a container holding the key, the corresponding value, as well as a pointer to the next element in the hash bucket. This introduces a significant number of expensive memory allocations on the critical path. Second, hashing and comparison functions are called for every *lookup* in order to acquire the correct bucket and element in the hash list. These function calls are usually virtual, causing significant overhead on the critical path. Finally, the data structures may have to be resized *during runtime* in order to efficiently accommodate more data. These resizing operations can become a significant bottleneck, especially for long-running, computationally expensive queries.

Next, we resolve all these issues with our compiler using schema and query knowledge, without changing a single line of the base code of the operators that use these data structures, or the code of other optimizations. This property shows that our approach, which is based on using a high-level compiler API, is practical for specializing database systems. The transformation, shown in Figure 2.7, is applied during the *lowering* phase of the compiler (Section 2.2.3), where high-level Scala constructs are mapped to low-level C constructs. The optimization lowers Scala HashMaps to native C arrays and inlines the corresponding operations, by using the

following three observations:

1. For our workloads, the information stored on the key is *usually* a subset of the attributes of the value. Thus, generic hash maps store redundant data. To avoid this, whenever a functional dependency between key and value is detected, we convert the hash map to a native array that stores only the values, and not the associated key (lines 2-11). Then, since the inserted elements are anyway chained together in a hash list, we provision for the next pointer when these are first allocated (e.g. at data loading, *outside the critical path*[17]). Thus, we no longer need the key-value-next container and we manage to reduce the amount of memory allocations significantly.

2. Second, the SC compiler offers function inlining for any Scala function out-of-the-box. Thus, our system can automatically inline the body of the hash and equal functions (lines 20 and 23 of Figure 2.7). This significantly reduces the number of function calls (to almost zero), considerably improving query performance.

3. Finally, to avoid costly maintenance operations on the critical path, we preallocate all the necessary memory space that *may* be required for the hash map during execution. This is done by specifying a size parameter when allocating the data structure (line 3). Currently, we obtain this size by performing worst-case analysis on the query. Database statistics can make this estimation very accurate, as we show in Section 2.4, where we evaluate the memory consumption of LegoBase in more detail.

For our running example, the aggregation array, created in step 1 above, is accessed using the integer value obtained from hashing the `L_SHIPMODE` string. Then, the values located into the corresponding bucket of the array are checked one by one, in order to see if the value of `L_SHIPMODE` exists and if a match is found, the aggregation entries are updated accordingly, or a new entry is initialized otherwise.

Finally, we note that data-structure specialization is an example of intra-operator optimization and, thus, each operator can specialize its own data-structures by using similar optimizations as the one shown in Figure 2.7.

Automatically Inferring Indices on Date Attributes

Assume that an SQL query needs to *fully* scan an input relation in order to extract tuples belonging to a particular year. A naive implementation would simply execute an **`if`** condition for each tuple of the relation and propagate that tuple down the query plan if the check was satisfied. However, it is our observation that such conditions, as simple as they may be, can have a pronounced negative impact on performance, as they can significantly increase the total number of CPU instructions executed in a query.

[17] The transformer shown in Figure 2.7 is applied only for the code segment that handles basic query processing. There is another transformer which handles the provision of the next pointer during data loading.

```
1 class HashMapToArray extends RuleBasedTransformer {
2   rewrite += rule {
3     case code"new HashMap[K, V]($size, $hashFunc, $equalFunc)" => {
4       // Create new array for storing only the values
5       val arr = code"new Array[V]($size)"
6       // Keep hash and equal functions in the metadata of the new object
7       arr.attributes += "hash" -> hashFunc
8       arr.attributes += "equals" -> equalFunc
9       arr // Return new object for future reference
10    }
11  }
12  rewrite += rule {
13    case code"($hm: HashMap[K, V]).getOrElseUpdate($key, $value)" => {
14      val arr = transformed(hm) // Get the array representing the original hash map
15      // Extract functions
16      val hashFunc = arr.attributes("hash")
17      val equalFunc = arr.attributes("equals")
18      code"""
19        // Get bucket
20        val h = $hashFunc($value) // Inlines hash function
21        var elem = $arr(h)
22        // Search for element & inline equals function
23        while (elem != null && !$equalFunc(elem, $key))
24          elem = elem.next
25        // Not found: create new elem / update pointers
26        if (elem == null) {
27          elem = $value
28          elem.next = $arr(h)
29          $arr(h) = elem
30        }
31        elem
32      """
33    }
34  }
35  // Lowering of remaining operations is done similarly
36 }
```

Figure 2.7 – Specializing HashMaps by converting them to native arrays. The corresponding operations are mapped to a set of primitive C constructs.

Thus, for such cases, LegoBase uses the aforementioned partitioning mechanism in order to automatically create indices, at data loading time, for all attributes of `date` type. It does so by grouping the tuples of a date attribute based on the year, thus forming a two-dimensional array where each bucket holds all tuples of a particular year. This design allows to immediately skip, at query execution time, all years for which this predicate is incorrect. That is, as shown in Figure 2.8, the **`if`** condition now just checks whether the first tuple of a bucket is of a particular year and if not the whole bucket is skipped, as *all* of its tuples have the same year and, thus, they *all* fail to satisfy the predicate condition.

These indices are particularly important for queries that process large input relations, whose date values are uniformly distributed across years. This is the case, for example, for the LINEITEM and ORDERS tables of TPC-H, whose date attributes are always populated with values ranging from 1992-01-01 to 1998-12-31.

```
// Sequential scan through table
for (int idx=0 ; idx<TABLE_SIZE ; idx+=1) {

 if (table[idx].date >= "01-01-1994" &&
     table[idx].date <= "31-12-1994")
    // Propagate tuple down the query plan
}
```

(a) Original, naive code

```
// Sequential scan through table
for (int idx=0 ; idx<NUM_BUCKETS ; idx+=1) {
  // Check only the first entry
  if (table[idx][0].date >= "01-01-1994" &&
      table[idx][0].date <= "31-12-1994")
      // Propage all tuples of table[idx]
}
```

(b) Optimized code

Figure 2.8 – Using `date` indices to speed up selection predicates on large relations.

2.3.3 Changing Data Layout

An important trade-off in databases is between row and column stores [317, 142, 3]. The central contrasting point between these two approaches is the *data layout*, i.e. the way data is organized and grouped together. By default LegoBase uses a row layout, since this intuitive data organization facilitates fast development of the relational operator implementations. However, we quickly noted the benefits of using a column layout for efficient data processing. One solution would be to go back and redesign the whole query engine; however this misses the point of our compiler framework. In this section, we show how the transition from row to column layout can be expressed as an optimization[18].

The optimization of Figure 2.9 performs a conversion from an array of records (row layout) to a record of arrays (column layout), where each array in the column layout stores the values for *one* attribute. The optimization overwrites the default lowering for arrays, thus providing the new behavior. As with all our optimizations, *type information* determines the applicability of an optimization: here it is performed only if the array elements are of record type (lines 3,13,26). Otherwise, this transformation is a NOOP (e.g. an array of Integers remains unchanged).

Consider, for example, an update to an array of records (`arr(n) = v`), where **v** is a record. We know that the *new* representation of `arr` will be a record of arrays (column layout), and that `v` has the same attributes as an element of `arr`. So, for each of these attributes we extract the corresponding array from `arr` (line 18) and field from `v` (line 19); then we perform the update on the extracted array (line 19) using the same index.

This optimization also reveals another benefit of using an optimizing compiler: developers can create *new* abstractions in their optimizations, which will be in turn optimized away in *subsequent* optimization passes. For example, the first rule of Figure 2.9 results in *record reconstruction* by extracting the individual record fields from the record of arrays (lines 5-7) and then building a new record to hold the result (line 8). This intermediate record can be *automatically* removed using dead code elimination (DCE), as shown in Figure 2.10. Similarly, if SC can statically determine that some attribute is never used (e.g. by having all queries given in advance), then this attribute will just be an unused field in a record, which the optimizing compiler will be able to optimize away (e.g. attribute L2 in Figure 2.10).

[18] Changing the data layout does not mean that LegoBase becomes a full-fledged column store. There are other important aspects which we do not yet handle, and which we plan to investigate in future work.

```
1 class ColumnarLayoutTransformer extends RuleBasedTransformer {
2   rewrite += rule {
3     case code"new Array[T]($size)" if typeRep[T].isRecord => typeRep[T] match {
4       case RecordType(recordName, fields) => {
5         val arrays =
6           for((name, tp: TypeRep[Tp]) <- fields) yield
7             name -> code"new Array[Tp]($size)"
8         record(recordName, arrays)
9       }
10    }
11  }
12  rewrite += rule {
13    case code"(arr:Array[T]).update($idx,$v)" if typeRep[T].isRecord => typeRep[T] match {
14      case RecordType(recordName, fields) => {
15        val columnarArr = transformed(arr) // Get the record of arrays
16          for((name, tp: TypeRep[Tp]) <- fields) {
17            code """
18              val fieldArr: Array[Tp] = record_field($columnarArr, $name)
19              fieldArr($idx) = record_field($v, $name)
20            """
21          }
22      }
23    }
24  }
25  rewrite += rule {
26    case code"(arr:Array[T]).apply($index)" if typeRep[T].isRecord => typeRep[T] match {
27      case RecordType(recordName, fields) => {
28        val columnarArr = transformed(arr) // Get the record of arrays
29        val elems = for((name, tp: TypeRep[Tp]) <- fields) yield {
30          name -> code """
31            val fieldArr: Array[Tp] = record_field($columnarArr, $name)
32            fieldArr($index)
33          """
34        }
35        record(recordName, elems)
36      }
37    }
38  }
39 }
```

Figure 2.9 – Changing the data layout (from row to column) expressed as an optimization. Scala's `typeRep` carries type information, which is used to differentiate between Array[Rec] and other non-record arrays (e.g. an array of integers).

We notice that the transformation described in this section does not have any dependency on other optimizations or the code of the query engine. This is because it is applied in the optimization phase that handles *only* the optimization of arrays. This separation of concerns leads, as discussed previously, to a significant increase in productivity as, for example, developers that tackle the optimization of individual query operators do not have to worry about optimizations handling the data layout.

2.3.4 Introducing Dictionaries

LegoBase can improve the performance of queries by introducing dictionaries. These dictionaries contain a mapping to a new attribute, with better performance characteristics. Instead

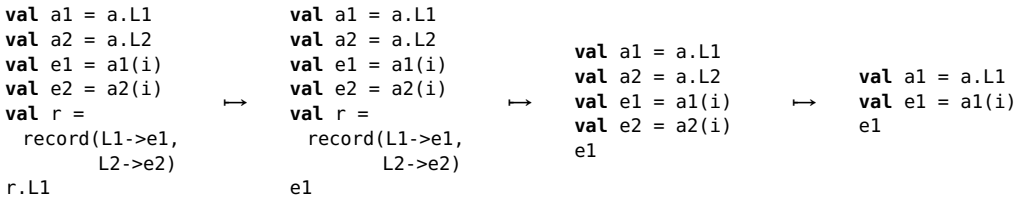


Figure 2.10 – Dead-code elimination (DCE) can remove intermediate materializations, e.g., row reconstructions when using a columnar layout. Here a is a record of arrays (columnar layout) and i is an integer. The records have only two attributes *L1* and *L2*. The notation `L1->v` associates the label (attribute name) L1 with value v.

of storing a relation[19] $R(A,B)$ in the database, we:

- introduce a *suitable*[20] domain A',
- store a relation $M(A,A')$ and a dictionary (subsequently also called M) from A to (potentially a set of) A' to speed up accesses when A is bound, and
- store a relation $R'(A',B)$

such that $R(A,B)$ can be defined as a view

$$R(A,B) \;:=\; \pi_{A,B}(M(A,A') \bowtie R'(A',B)).$$

Next, we give two use cases of this. In the first case, the dictionary introduction is an automatic optimization, while in the second, it is a schema design choice. LegoBase implements a transformer that expands such views using their definition, in all cases automatically.

String Dictionaries

Operations on non-primitive data types, such as strings, incur a high performance overhead. Typically, a function call is required, and most operations need to execute loops to process the encapsulated data. For example, *strcmp* needs to iterate over the underlying array of characters, comparing one character from each of the two input strings on each iteration. Thus, such operations significantly affect branch prediction and cache locality.

For strings, we use String Dictionaries [35] to remove their abstraction overhead. One dictionary is maintained for every attribute of String type, which generally operates as follows. First, at data loading time, each string value of an attribute is mapped to an integer value. Then, at query execution time, string operations are mapped to their integer counterparts, as shown in Table 2.1. This mapping allows to significantly improve the query execution performance, as it

[19] We show a binary relation here, but the generalization is obvious.

[20] In the two use cases, A' is chosen differently, once as identifiers for A values and once as row ids for R.

completely eliminates underlying loops and, thus, significantly reduces the number of CPU instructions executed.

LegoBase uses String Dictionaries[21] [35] to remove the abstraction overhead of strings. Our system maintains one dictionary for every attribute of String type. At data loading time, each string value of an attribute is mapped to an integer value. This value corresponds to the index of that string in a single linked-list holding the *distinct* string values of that attribute. The list basically constitutes the dictionary itself. In other words, each time a string appears for the first time during data loading, a unique integer is assigned to it; if the same string value reappears in a later tuple, the dictionary maps this string to the previously assigned integer. String operations are mapped to their integer counterparts, as shown in Table 2.1, thus significantly improving query execution performance. For our running example, LegoBase compresses the attributes L_SHIPMODE and O_ORDERPRIORITY by converting the six string equality checks into corresponding integer comparisons.

Special care is needed for string operations that require ordering. For example, Q2 and Q14 of TPC-H need to perform the *endsWith* and *startsWith* string operations with a constant string, respectively. This requires that we utilize a dictionary that maintains the data in order; that is, if $string_x < string_y$ lexicographically, then $Int_x < Int_y$ as well. To do so, we take advantage of the fact that in LegoBase all input data is already materialized, and thus we can first compute a list of lexicographically sorted distinct values, and then make a second pass over the data to assign integer values to the string attribute. By doing so, the constant string is then converted to a $[start, end]$ range, by iterating over the list of distinct values and finding the first and last strings which start or end with that particular string, as shown in Table 2.1[22]

It is important to note that string dictionaries significantly improve query execution performance, but have negative impact on the performance of data loading. In addition, string dictionaries can actually degrade performance when they are used for primary keys or for attributes that contain many distinct values (as in this case the string dictionary significantly increases memory consumption). In such cases, LegoBase can be configured so that it does not use string dictionaries for those attributes, through proper usage of the optimization pipeline described in Section 2.2.

Densification of Domains

In some workloads, the values of certain integer columns partition into multiple number domains. This means that queries that select only some partitions, creating sparsity, cannot benefit from optimizations relying on the dense nature of the values, such as the data partitioning transformation shown in Section 2.3.2. The join on columns with such properties

[21] Like all optimizations, this is an option that can be turned off. We experiment with both cases.

[22] In addition, there is another special case where the string attributes need to be tokenized on a word granularity. This happens, for example, in Q13 of TPC-H. Such queries need to perform the *indexOfSlice* string operation, where the slice represents a word. LegoBase provides a *word-tokenizing* string dictionary that contains all words in the strings instead of the string attributes themselves to handle such cases.

String Operation	C code	Integer Operation	Dictionary Type
equals	strcmp(x, y) == 0	x == y	Normal
notEquals	strcmp(x, y) != 0	x != y	Normal
startsWith	strncmp(x, y, strlen(y)) == 0	x>=start && x<=end	Ordered
indexOfSlice	strstr(x, y) != NULL	N/A	Word-Token

Table 2.1 – Mapping of string operations to integer operations through the corresponding type of string dictionaries. Variables x and y are strings arguments which are mapped to integers. The rest of string operations are mapped in a similar way.

suffers from bad locality. To solve this issue, one can introduce (and store in the database) a dictionary containing a mapping into a dense domain.

As an example, consider the relations `Vehicle(vid, model)` and `V_Driver(vid, name)` for which we would like to answer the following query: $\sigma_{\texttt{model}=c}(\texttt{Vehicle} \bowtie \texttt{V\_Driver})$. The selection based on the attribute `model` results in sparse values for the (key) attribute `vid` in relation `Vehicle`, leading to bad data locality for the join of the two relations. To improve data locality, one can introduce the dictionary `Dict(model,vid')`. This dictionary maps a model name to a dense set of values `vid'` identifying the vehicles of that model. In the data loading phase, instead of storing the original relations in main memory, the relation `V_Driver'(vid', name)` and the dictionary `Dict` are stored. During query processing, instead of the initial query, the following query is evaluated: $\sigma_{\texttt{model}=c}(\texttt{Dict}) \bowtie \texttt{V\_Driver'}$. In this case, the values of the attribute `vid'` are dense, leading to better data locality for the join with `V_Driver'`.

Note that this use case does *not* arise in TPC-H, and thus our experimental evaluation.

2.3.5 Domain-Specific Code Motion

Domain-Specific code motion includes optimizations that remove code segments that have negative impact on query execution performance from the critical path and instead executes the logic of those code segments during data loading. Thus, the optimizations in this category trade increased loading time for improved query execution performance. There are two main optimizations in this category, described next.

Hoisting Memory Allocations

Memory allocations can cause a significant degradation in query execution performance. LegoBase can completely eliminate such allocations from the critical path, by taking advantage of type information available in each SQL query, as described next.

At query compilation time, information is gathered regarding the data types used throughout

an incoming SQL query. This is done through an analysis phase, where the compiler collects all `malloc` nodes in the program, once the latter has been lowered to the abstraction level of C code. This is necessary to be done at this level, as high-level programming languages like Scala provide implicit memory management, which the SC optimizing compiler cannot currently optimize. The obtained types correspond either to the initial database relations (e.g. the LINEITEM table of TPC-H) or to types required for intermediate results, such as aggregations. Based on this information, SC initializes memory pools during data loading, one for each type.

Then, at query execution time, the corresponding `malloc` statements are replaced with references to those memory pools. We have observed that this optimization significantly reduces the number of CPU instructions executed during query evaluation, and significantly contributes to improving cache locality. This is because the memory space allocated for each pool is contiguous and, thus, each cache miss brings useful records to the cache (this is not the case for the fragmented memory space returned by the `malloc` system calls).

Finally, the size of the memory pools is estimated by performing worst-case analysis on a given query. However, we have confirmed that the statistics that LegoBase collects during data loading are accurate enough so that these pools do not unnecessarily create memory pressure.

Hoisting Data-Structure Initialization

The proper initialization and maintenance of any data structure needed during query execution generally require specific code to be executed in the critical path. This is typically true for data structures representing some form of *key-value* stores, as we describe next.

Consider the case of TPC-H Q12, for which a data structure is needed to store the results of the aggregate operator. Then, when evaluating the aggregation during query execution, we must check whether the corresponding key of the aggregation has been previously inserted in the aggregation data structure. In this case, the code must check whether a specific value of O_ORDERPRIORITY is already present in the data structure. If so, it would return the existing aggregation. Otherwise, it would insert a new aggregation into the data structure. This means that *at least one* **`if`** condition must be evaluated for *every* tuple that is processed by the aggregate operator. We have observed that such **`if`** conditions, which exist purely for the purpose of data-structure initialization, significantly affect branch prediction and overall query execution performance.

LegoBase provides an optimization to remove such data-structure initialization from the critical path by utilizing domain-specific knowledge. For example, LegoBase takes advantage of the fact that aggregations can usually be statically initialized with the value zero, for each corresponding key. To infer all these possible key values (i.e. infer the *domains* of these attribute), LegoBase utilizes the statistics collected during data loading for the input relations. Then, at query execution time, the corresponding **`if`** condition mentioned above no longer needs to be evaluated, as the aggregation already exists and can be accessed directly. We have observed

that, by removing code segments that perform *only* data-structure initialization, branch prediction is improved and the total number of CPU instructions executed is significantly reduced as well.

Observe that this optimization depends on the ability to predict the possible key values in advance, during data loading. This may not always be possible, as is the case when the key is a result of an intermediate operator deeply nested in the query plan. However, workloads like TPC-H mostly use attributes of the original relations to access data structures, attributes whose value range can be accurately estimated during data loading through statistics. In addition, for TPC-H queries, the key value range is very small, typically up to a couple of thousand sequential key values[23]. Under these two conditions, it becomes feasible to remove initialization overheads and the associated unnecessary computations.

2.3.6 Traditional Compiler Optimizations

In this section, we present a number of traditional compiler optimizations that originate mostly from work in the PL community. Most of them are generic in nature, and, thus, they are offered out-of-the-box by the SC compiler.

Removal of Unused Relational Attributes

In Section 2.3.3 we mentioned that LegoBase provides an optimization for removing relational attributes that are not accessed by a particular SQL query, assuming that this query is known *in advance*. For example, the Q12 running example references eight relational attributes. However, the relations LINEITEM and ORDERS contain 25 attributes in total. LegoBase avoids loading these unnecessary attributes into memory at data loading time. It does so by analyzing the input SQL query and removing the set of unused fields from the record definitions. This reduces memory pressure and improves cache locality.

Removing Unnecessary Let-Bindings

The SC compiler uses the Administrative Normal Form (ANF) when generating code. This simplifies code generation for the compiler. However, it has the negative effect of introducing many unnecessary intermediate variables. To improve upon this situation, SC uses a technique first introduced by the programming language community [323], which removes any intermediate variable that satisfies the following three conditions: the variable (a) is set only once, (b) has no side effects, and, finally, (c) is initialized with a single value (and thus its initialization does not correspond to executing expensive computations). SC then replaces any appearance of this variable later in the code with its initialization value. We have observed that since the variable initialization time may happen significantly earlier in the code than its

[23] A notable exception is TPC-H Q18 which uses O_ORDERKEY as a key, which has a sparse distribution of key values. LegoBase generates a specialized data structure for this case.

actual use, this does not allow for this optimization to be performed by low-level compilers like LLVM.

Finally, our compiler applies a technique known as *parameter promotion* or *scalar replacement*. This optimization removes *structs* whose fields can be flattened to local variables. This optimization has the effect of removing a memory access from the critical path as the field of a struct can be referenced immediately without referencing the variable holding the struct itself. As a result, the number of memory accesses occurring during query execution is significantly reduced.

Fine-grained Compiler Optimizations

Finally, there is a category of fine-grained compiler optimizations that are applied last in the compilation pipeline. These optimizations target optimizing very small code segments (even individual statements) under particular conditions. We briefly present three such optimizations next.

First, SC can transform arrays to a set of local variables. This lowering is possible only when (a) the array size is statically known at compile time, (b) the array is relatively small (to avoid increasing register pressure) and, finally, (c) the index of every array access can be inferred at compile time (otherwise, the compiler is not able to know to which local variable an array access should be mapped to).

Second, the compiler provides an optimization to change the boolean condition `x && y` to `x & y` where `x` and `y` both evaluate to boolean and the second operand does not have any side effect. According to our observations, this optimization can significantly improve branch prediction when the aforementioned conditions are satisfied.

Finally, the compiler can be instructed to apply tiling to **`for`** loops whose ranges are known at compile time (or can be accurately estimated).

It is our observation that all these fine-grained optimizations (as described above), which can be typically written in less than a hundred lines of code, can help to improve the performance of certain queries. More importantly, since they have very fine-grained granularity, their application does not introduce additional performance overheads.

2.4 Experimental Evaluation of LegoBase

In this section, we evaluate the realization of the abstraction without regret vision in the domain of analytical query processing. After briefly presenting our experimental platform, we address the following topics and open questions related to the LegoBase system:

1. How well can general-purpose compilers, such as LLVM or GCC, optimize query engines?

We show that these compilers ultimately fail to detect many high-level optimization opportunities and, thus, they perform poorly compared to our system (Section 2.4.2).

2. Is the code generated by LegoBase competitive, performance-wise, to (a) traditional database systems and (b) query compilers based on template expansion? We show that by utilizing query-specific knowledge and by extending the scope of compilation to optimize the *entire* query engine, we can obtain a system that significantly outperforms both alternative approaches (Section 2.4.3).

3. We experimentally validate that the source-to-source compilation from Scala to efficient, low-level C binaries is necessary as even highly optimized Scala programs exhibit a considerably worse performance than C (Section 2.4.4).

4. What insights can we gain by analyzing the performance improvement of individual optimizations? Our analysis reveals that important optimization opportunities have been so far neglected by compilation approaches that optimize *only* queries. To demonstrate this, we compare architectural decisions as fairly as possible, using a shared codebase that only differs by the effect of a single optimization (Section 2.4.5).

5. How much are the overall memory consumption and data loading speed of our system? These two metrics are of importance to main-memory databases, as a query engine must perform well in both directions to be usable in practice (Section 2.4.6).

6. We analyze the amount of effort required when programming query engines in LegoBase and show that, by programming in the abstract, we can derive a fully functional system in a relatively short amount of time and coding effort (Section 2.4.8).

7. We evaluate the compilation overheads of our approach. We show that SC can efficiently compile query engines even for the complicated, multi-way join queries typically found in analytical query processing (Section 2.4.9).

2.4.1 Experimental Setup

Our experimental platform consists of a server-type x86 machine equipped with two Intel Xeon E5-2620 v2 CPUs running at 2GHz each, 256GB of DDR3 RAM at 1600Mhz and two commodity hard disks of 2TB storing the experimental datasets. The operating system is Red Hat Enterprise 6.7. For all experiments, we have disabled huge pages in the kernel, since this provided better results for all tested systems (described in more detail in Table 2.2). For compiling the generated programs throughout the evaluation section, we use version 2.11.4 of the Scala compiler and version 3.4.2 of the CLang front-end for LLVM [214], with the default

[24] We note that according to the TPC-H specification rules, a database system can employ data partitioning (as described in Section 2.3.2) and still be TPC-H compliant. This is the case when all input relations are partitioned on *one and only one* primary or foreign key attribute across all queries. The LegoBase(TPC-H/C) configuration of our system follows exactly this partitioning approach, which is also used by the HyPer system (but in contrast to SC, partitioning in HyPer is not expressed as a compiler optimization).

System	Description	Compiler optimizations	TPC-H compliant	Uses query-specific info
DBX	Commercial, in-memory DBMS	No compilation	Yes	No
Compiler of HyPer	Query compiler of the HyPer DBMS	Operator inlining, push engine	Yes	No
LegoBase (Naive)	A naive engine with the minimal number of optimizations applied	Operator inlining, push engine	Yes	No
LegoBase (TPC-H/C)	TPC-H compliant engine	Operator inlining, push engine, data partitioning	Yes[24]	No
LegoBase (StrDict/C)	Non TPC-H compliant engine with some optimizations applied	Like above, plus String Dictionaries	No	No
LegoBase (Opt/C)	Optimized push-style engine	All optimizations of this thesis	No	Yes
LegoBase (Opt/Scala)	Optimized push-style engine	All optimizations of this thesis	No	Yes

Table 2.2 – Description of all systems evaluated in this chapter. Unless otherwise stated, all generated C programs of LegoBase are compiled to a final C binary using CLang. All listed LegoBase engines and optimizations are written with *only* high-level Scala code, which is then optimized and compiled to C or Scala code with SC.

optimization flags set for both compilers. For the Scala programs, we configure the Java Virtual Machine (JVM) to run with 192GB of heap space, while we use the GLib library (version 2.38.2) [333] whenever we need to generate generic data structures in C.

For our evaluation, we use the TPC-H benchmark [343]. TPC-H is a data warehousing and decision support benchmark that issues business analytics queries to a database with sales information. This benchmark suite includes 22 queries with a high degree of complexity that express most SQL features. We use all 22 queries to evaluate various design choices of our methodology. We execute each query five times and report the average performance of these runs. Unless otherwise stated, the scaling factor of TPC-H is set to 8 for all experiments. It is important to note that the final generated optimized code of LegoBase (configurations LegoBase(Opt/C) and LegoBase(Opt/Scala) in Table 2.2) employs materialization (e.g. for the date indices) and, thus, this version of the code does *comply* with the TPC-H implementation rules. However, we also present a TPC-H compliant configuration, LegoBase(TPC-H/C), for comparison purposes. A brief presentation of the TPC-H schema and queries can be found in Appendix C.

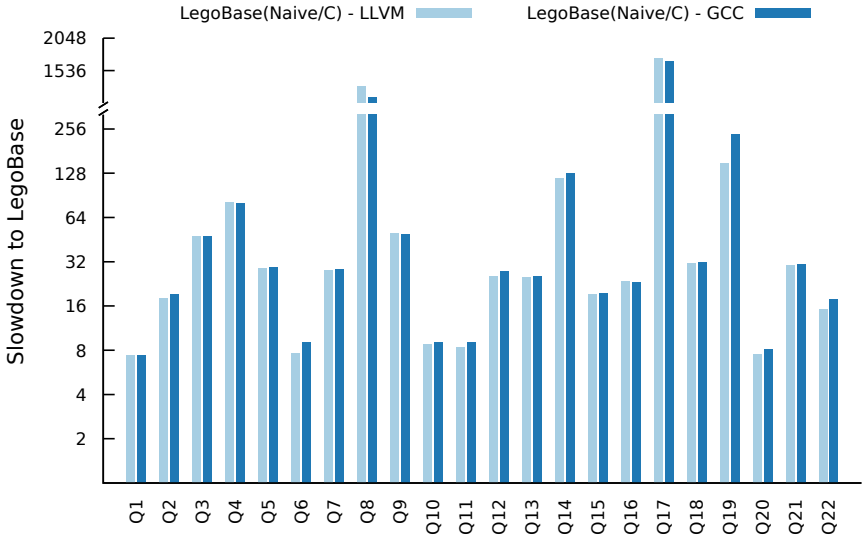


Figure 2.11 – Performance of a push-style engine compiled with LLVM and GCC. These engines are generated using *only* operator inlining. The baseline is the performance of the optimal generated code, LegoBase(Opt/C), with all optimizations enabled.

As a reference point for most results presented here, we use a commercial, in-memory, row-store database system called DBX, which does not employ compilation. We assign 192GB of DRAM as memory space in DBX, and we use the DBX-specific data types instead of generic SQL types. As described in Section 2.2, LegoBase uses query plans from the DBX database. We also use the query compiler of the HyPer system [255] (w) as a point of comparison with existing query compilation approaches. Since parallel execution is not yet possible at the time of writing for LegoBase, all systems have been forced to single-threaded execution, either by using the execution parameters some of them provide or by manually disabling the usage of CPU cores in the kernel configuration.

2.4.2 Optimizing Query Engines Using General-Purpose Compilers

First, we show that low-level, general-purpose compilation frameworks, such as LLVM, are not adequate for efficiently optimizing query engines. To do so, we use LegoBase to generate an *unoptimized* push-style engine with only operator inlining applied, which is then compiled to a final C binary using LLVM. We choose this engine as a starting point since it allows the underlying C compiler to be more effective when optimizing the whole C program (as the presence of procedures may otherwise force the compiler to make conservative decisions or miss optimization potential during compilation[25]).

As shown in Figure 2.11, the achieved performance is very poor: the unoptimized query engine,

[25] [301] presents an easy-to-follow example and an analysis of why general-purpose compilers need to operate in this fashion.

LegoBase(Naive/C)–LLVM, is significantly slower for all TPC-H queries, requiring more than 16× the execution time of the optimal LegoBase configuration in most cases. This is because frameworks like LLVM cannot automatically detect all optimization opportunities that we support in LegoBase (as described thus far in this thesis). This is because either (a) the scope of an optimization is too coarse-grained to be detected by a low-level compiler or (b) the optimization relies on domain-specific knowledge that general-purpose optimizing compilers such as LLVM are not aware of.

In addition, as shown in the same figure, compiling with LLVM does not *always* yield better results compared to using another traditional compiler like GCC[26]. We see that LLVM outperforms GCC for *only* 15 out of 22 queries (by 1.09× on average) while, for the remaining ones, the binary generated by GCC performs better (by 1.03× on average). In general, the performance difference between the two compilers can be significant (e.g. for Q19, there is a 1.58× difference). We also experimented with manually specifying optimizations flags to the two compilers, but this turns out to be a very delicate and complicated task as developers can specify flags which actually make performance worse. We argue that it is instead more beneficial for developers to invest their effort in developing high-level optimizations, like those presented in this thesis.

2.4.3 Comparing LegoBase with Previous Systems

Next, we compare our approach – which compiles the *entire* query engine and utilizes *query-specific* information – with the compiler of the HyPer database [255]. HyPer performs template expansion through LLVM in order to inline the relational operators of a query executed on a push engine[27]. The results are presented in Figure 2.12.

We perform this analysis in two steps. First, we generate a query engine that (a) does not utilize any query-specific information and (b) adheres to the implementation rules of the TPC-H workload. Such an engine represents a system where data are loaded *only once*, and all optimizations are applied before any query arrives (as happens with HyPer and any other traditional DBMS). We show that this LegoBase configuration, titled LegoBase(TPC-H/C), has performance competitive to that of the HyPer database system, and that efficient handling of string operations is essential in order to have the performance of our system match that of HyPer. Second, we show that by utilizing query-specific knowledge and performing aggressive materialization and repartition of input relations based on multiple attributes, we can generate a query engine, titled LegoBase(Opt/C), which significantly outperforms existing approaches. Such an engine corresponds to systems that, as discussed previously in Section 2.4.1, have all queries or data known in advance.

[26] For this experiment, we use version 4.4.7 of the GCC compiler.

[27] We also experimented with *another* in-memory DBMS that compiles SQL queries to native C++ code on the fly. However, we were unable to configure the system so that it performs well compared to the other systems. Thus, we omit its results from this chapter.

To begin with, Figure 2.12 shows that by using the query compiler of HyPer, performance is improved by 6.4× on average compared to DBX. To achieve this performance improvement, HyPer uses a push engine, operator inlining, and data partitioning. In contrast, the TPC-H-compliant configuration of our system, LegoBase(TPC-H/C), which uses the same optimizations, has an average execution time of only 4.4x the one of the DBX system, across all TPC-H queries. The main reason behind this significantly slower performance is, as we mentioned above, the inefficient handling of string operations in LegoBase(TPC-H/C). In this version, LegoBase uses the `strcmp` function (and its variants). In contrast, HyPer uses the SIMD instructions found in modern instructions sets (such as SSE4.2) for efficient string handling [41], a design choice that can lead to significant performance improvement compared to LegoBase(TPC-H/C). To validate this analysis, we use a configuration of our system, called LegoBase(StrDict/C), which additionally applies the string dictionary optimization. This configuration is no longer TPC-H-compliant (as it introduces an auxiliary data structure), but is still does not require query-specific information. We notice that the introduction of this optimization is enough to make LegoBase(StrDict/C) match the performance of HyPer: the two systems have *only* a 1.06× difference in performance.

Second, Figure 2.12 also shows that by using all other optimizations of LegoBase (as they were presented in Chapter 2.3), which are not performed by the query compiler of HyPer, we can get a total 45.4× performance improvement compared to DBX with all optimizations enabled. This is because, for example, LegoBase(Opt/C) uses query-specific information to remove unused relational attributes or hoist out expensive computation (thus reducing memory pressure and decreasing the number of CPU instructions executed) and aggressively repartitions input data on multiple attributes (thus allowing for more efficient join processing). Such optimizations result in improved cache locality and branch prediction, as shown in Figure 2.13. More specifically, there is an improvement of 1.68× and 1.31× on average for the two metrics, respectively, between DBX and LegoBase. In addition, the maximum, average and minimum difference in the number of CPU instructions executed in HyPer is 3.76×, 1.61×, and 1.08× that executed in LegoBase. These results prove that the optimized code of LegoBase(Opt/C) is competitive, performance wise, to both traditional database systems and query compilers based on template expansion.

Note that in several cases DBX has a significantly lower branch misprediction rate compared to LegoBase. To further investigate why this is the case, we performed some experiments using TPCH query 6 as an example. Our results show that the source of this difference is due to the pull vs. push-based nature of the query engines. In our experiments, we found that the pull-based engine (Volcano style [128]) has a better branch misprediction rate (3%) than a push-based engine (5%, producer/consumer style [255]). However, the absolute number of branches in a pull-based engine for this query is almost twice as many as the number of branches in a push-based engine. The absolute number of mispredicted branches is almost equal in both engines (12.9M vs 12.8M, for the pull and push engines, respectively). This means that half the branches in a pull-based engine, which are responsible for checking whether the relation is fully scanned, are in most cases correctly predicted. Hence, although a pull-based

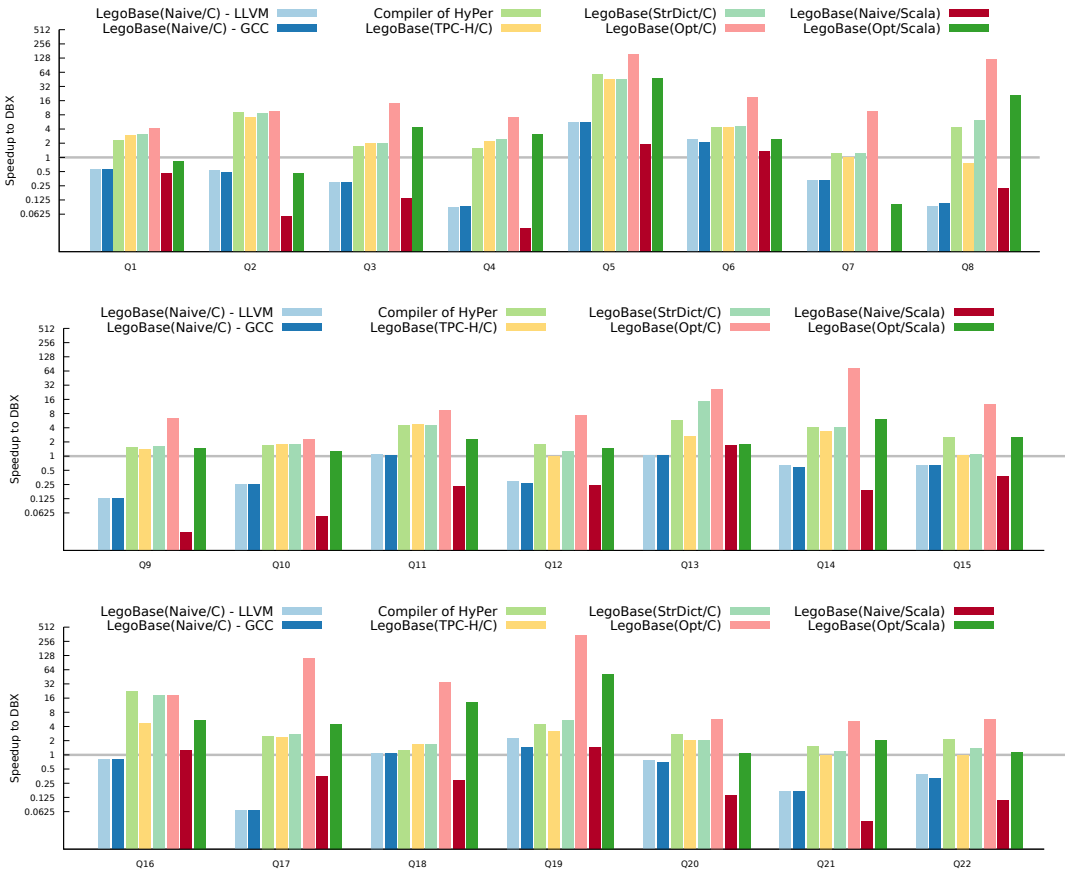


Figure 2.12 – Performance comparison of various LegoBase configurations (C and Scala programs) with the code generated by the query compiler of [255]. The baseline for all systems is the performance of the DBX commercial database system. The absolute execution times for this figure can be found in Appendix A.

engine may have a lower branch misprediction rate, it does not mean that the performance is better. This is further justified by seeing the similar trend of HyPer and LegoBase in terms of branch misprediction, as both are push-based engines. A detailed comparison of push and pull-based engines is can be found in Chapter 4.

Finally, we note that we plan to investigate even more aggressive and query-specific data-structure optimizations in future work. Such optimizations are definitely feasible, given the easy extensibility of the SC compiler.

2.4.4 Source-to-Source Compilation from Scala to C

Next, we show that source-to-source compilation from Scala to C is necessary in order to achieve optimal performance in LegoBase. To do so, Figure 2.12 also presents performance results for both a naive and an optimized Scala query engine, named LegoBase(Naive/Scala)

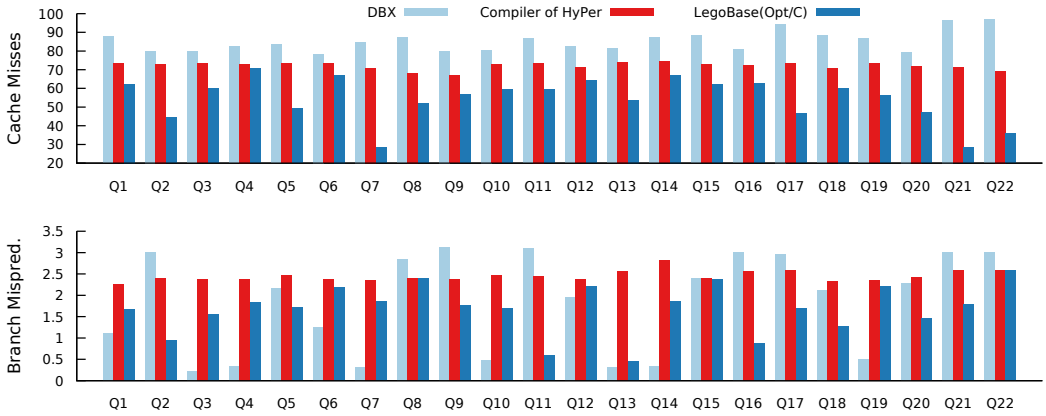


Figure 2.13 – Percentage of cache misses and branch mispredictions for DBX, HyPer and LegoBase(Opt/C) for all 22 TPC-H queries.

and LegoBase(Opt/Scala), respectively. First, we notice that the optimized generated Scala code is significantly faster than the naive counterpart, by 26.4×. This shows that extensive optimization of the Scala code is essential in order to achieve high performance. However, we also observe that the performance of the optimized Scala program cannot compete with that of the optimized C code, and is on average 10× slower.

Profiling information gathered with the *perf*[28] profiling tool of Linux reveals the following three reasons for this behavior: (a) Scala causes an increase to branch mispredictions, by 1.8× compared to the branch mispredictions in C, (b) The percentage of LLC misses is 1.3× to 2.4× those in Scala, and more importantly, (c) The number of CPU instructions executed in Scala is 6.2× the one executed by the C binary. Of course, these inefficiencies are to a great part due to the Java Virtual Machine and not specific to Scala[29]. Note that the optimized Scala program is competitive to DBX (especially for non-join-intensive queries, e.g. queries that have less than two joins): for 19 out of 22 queries, LegoBase(Opt/Scala) outperforms the commercial DBX system. This is because we remove all abstractions that incur significant overhead for Scala. For example, the performance of Q18, which builds a large hash map, is improved by 40× when applying the data-structure specialization provided by SC.

[28] https://perf.wiki.kernel.org/index.php/Main_Page.

[29] A publication from Google [160] comparing C++, Java, Go, and Scala seems to verify this hypothesis. In this work, the authors show how important it is to adequately optimize the garbage collection (GC) mechanism of the JVM by manually configuring its parameters. However, not only this work goes as far as to use *custom* JVM flags, but also, in our experience, tuning the GC is an equally delicate task as tuning a traditional, general-purpose C compiler. For example, the `+UseCompressedOops` GC flag improves the performance of Q16 (by 1.23×), but negatively affects the performance of Q6 (by 1.27×). In addition, this work also suggests that there are a number of language features and constructs of the Scala programming language that can significantly affect performance. For instance, the SC optimizing compiler generates *for-comprehensions* for Scala. Yet, the comparative study of Google suggests that it is better, performance wise, to use the `foreach` construct of Scala. We plan to explore such optimization opportunities for the generated Scala code and the JVM in future work.

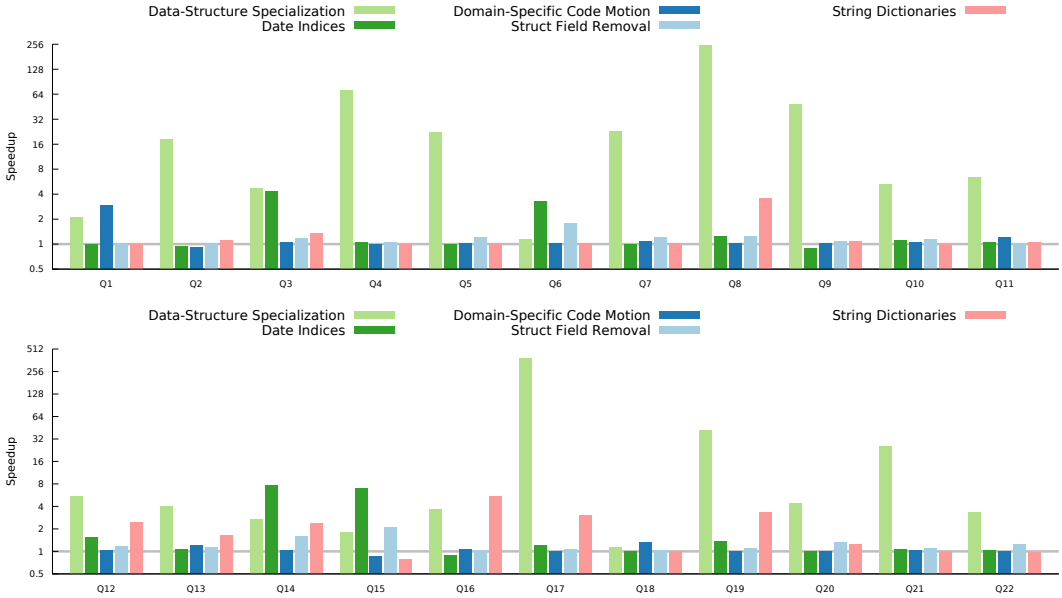


Figure 2.14 – Impact of different LegoBase optimizations on query execution time. The baseline is an engine that does not apply this optimization.

2.4.5 Impact of Individual Compiler Optimizations

In this section, we provide additional information about the performance improvement expected when applying one of the compiler optimizations of LegoBase. These results, illustrated in Figure 2.14, aim to demonstrate that significant optimization opportunities have been ignored by existing compilation techniques that handle only queries.

To begin with, we can clearly see in this figure that the most important transformation in LegoBase is the data-structure specialization (presented in Sections 2.3.2 and 2.3.2). This form of optimization is not provided by existing approaches, as data structures are typically precompiled in existing database systems. We see that, in general, when data-structure specialization is applied, the generated code has an average performance improvement of 30× (excluding queries Q8 and Q17 where the partitioning optimization facilitates skipping the processing of the majority of the tuples of the input relations). Moreover, we note that the performance improvement is not directly dependent on the number of join operators or input relations in the query plan. For example, join-intensive queries such as Q5, Q7, Q8, Q9, Q21 obtain a speedup of at least 22× when applying this optimization. However, the single-join queries Q4 and Q19 also receive similar performance benefit to that of multi-way join queries. This is because query plans may filter input data early on, thus reducing the need for efficient join data structures. Thus, selectivity information and analysis of the whole query plan are essential for analyzing the potential performance benefit of this optimization. Note that, for similar reasons, date indices (Section 2.3.2) allow to avoid unnecessary tuple processing and thus lead to increased performance for a number of queries (such as Q3, Q14, and Q15).

For the domain-specific code motion and the removal of unused relational attributes optimizations, we observe that they both improve performance, by 1.12× and 1.21×, respectively on average for all TPC-H queries. This improvement is not be as pronounced as that of other optimizations of LegoBase (like the one presented above). However, it is important to note that they both significantly reduce memory pressure, thus allowing the freed memory space to be used for other optimizations, such as the partitioning specialization, which in turn provide significant performance improvement. Nevertheless, these two optimizations – which are not provided by previous approaches (since they depend on query-specific knowledge) – can provide considerable performance improvement by themselves for some queries. For example, for TPC-H Q1, performing domain-specific code motion leads to a speedup of 2.96×, while the removal of unused attributes gives a speedup of 2.11× for Q15.

Moreover, the same figure evaluates the speedup we gain when using string dictionaries. We observe that for the TPC-H queries that perform a number of expensive string operations, using string dictionaries always leads to improved query execution performance: this speedup ranges from 1.06× to 5.5×, with an average speedup of 2.41×[30]. We also note that the speedup this optimization provides depends on the characteristics of the query. More specifically, if the query contains string operations on scan operators, as is the case with Q8, Q12, Q13, Q16, Q17, and Q19, then this optimization provides a greater performance improvement than when string operations occur in operators appearing later in the query plan. This is because, TPC-H queries typically filter out more tuples as more operators are applied in the query plan. Stated otherwise, operators being executed in the last stages of the query plan do not process as many tuples as scan operators. Thus, the impact of string operations is more pronounced when such operations take place in scan operators.

It is important to note that using string dictionaries comes at a price. First, this optimization increases the loading time of the query. Second, this optimization requires keeping a dictionary between strings and integer values, a design choice which requires additional memory. This may, in turn, increase memory pressure, possibly causing a drop in performance. However, it is our observation that, based on the individual use case and data characteristics (e.g. number of distinct values of a string attribute), developers can easily detect whether it makes sense performance-wise to use this optimization or not. We also present a more detailed analysis of the memory consumption required by the overall LegoBase system later in this chapter.

Then, the benefit of applying operator inlining (not shown) varies significantly between different TPC-H queries and ranges from a speedup of 1.07× up to 19.5×, with an average performance improvement of 3.96×. The speedup gained from applying this optimization depends on the complexity of the execution path of a query. This is a hard metric to visualize, as the improvement depends not only on *how many* operators are used but also on their type, their position in the overall query plan and how much each of them affects branch prediction and cache locality. For instance, queries Q5, Q7 and Q9 have the same number of

[30]The rest of the TPC-H queries (Q1, Q4, Q5, Q6, Q7, Q10, Q11, Q15, Q18, Q21, Q22) either did not have any string operation or the number of these operations on those queries was negligible.

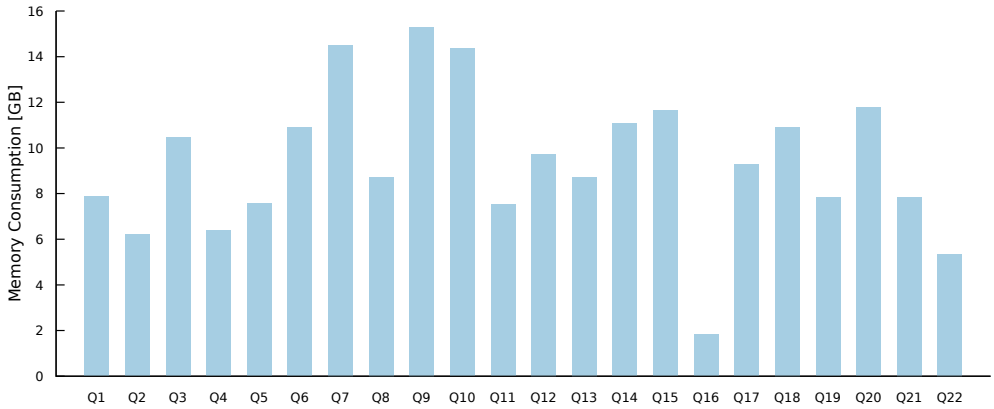


Figure 2.15 – Memory consumption of LegoBase(Opt/C) for the TPC-H queries.

operators, but the performance improvement gained varies significantly, by 10.4×, 1.4× and 7.5×, respectively. In addition, it is our observation that the benefit of inlining depends on which operators are being inlined. This is an important observation, as for very large queries, the compiler may have to choose which operators to inline (e.g. to avoid the code not fitting in the instruction cache). In general, when such cases appear, we believe that the compiler framework should merit inlining joins instead of simpler operators (e.g. scans or aggregations).

Finally, for the column layout optimization (not shown), the performance improvement is generally proportional to the percentage of attributes in the input relations that are actually used. This is expected as the benefits of the column layout are evident when this layout can "skip" loading into memory a number of unused attributes, thus significantly reducing cache misses. Synthetic queries on TPC-H data referencing 100% of the attributes show that, in this case, the column layout actually yields no benefit, and it is slightly worse than the row layout. Actual TPC-H queries reference 24% - 68% of the attributes and, for this range, the optimization gives a 2.5× to 1.05× improvement, which degrades as more attributes are referenced.

2.4.6 Memory Consumption and Overhead on Input Data Loading

Figure 2.15 shows the memory consumption of LegoBase(Opt/C) for all TPC-H queries. We use Valgrind for memory profiling as well as a custom memory profiler, while the JVM is always first warmed up. We make the following observations. First, the allocated memory is at most twice the size of the input data for all TPC-H queries (16GB of memory for 8GB of input data for all relations), while the *average* memory consumption is only 1.16× the total size of the input relations. These results suggest that our approach is usable in practice, as even for complicated, multi-way join queries the memory used remains relatively small. The additional memory requirements come as a result of the fact that LegoBase aggressively repartitions input data in many different ways (as was described in Section 2.3.2) in order to achieve

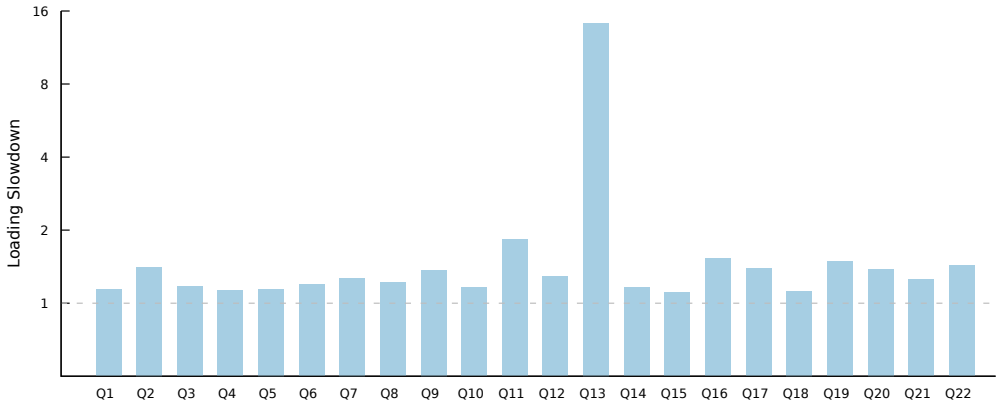


Figure 2.16 – Slowdown of input data loading occurring from applying all LegoBase optimizations to the C programs of the TPC-H workload (scaling factor 8).

optimal performance. Second, when all optimizations are enabled, LegoBase consumes less memory than the total size of the input data, for a number of queries. For instance, Q16 consumes merely 2GB for all necessary data structures. This behavior is a result of removing unused attributes from relational tables as well as of compressing attributes of string type when loading the input data. In general, it is our observation that memory consumption grows linearly with the scaling factor of the TPC-H workload.

In addition, we have mentioned before that applying our compiler optimizations can lead to an increase in the loading time of the input data. Figure 2.16 presents the total slowdown on input data loading when applying all LegoBase optimizations to the generated C programs (LegoBase(Opt/C)) compared to the loading time of the unoptimized C programs (LegoBase(Naive/C)). We observe that the total time spent on data loading, across all queries and with all optimizations applied, is *not* (excluding Q13 which applies the word-tokenizing string dictionary) more than 1.5× that of the unoptimized, push-style generated C code. This means that while our optimizations lead to significant performance improvement, they do not affect the loading time of input data significantly (there is an average slowdown of 1.88× including Q13). Based on these observations, as well as the absolute loading times presented in Appendix A, we can see that the additional overhead of our optimizations is not prohibitive: it takes in average less than a minute for LegoBase to load the 8GB TPC-H dataset, repartition the data, and build all necessary auxiliary data structures for efficient query processing.

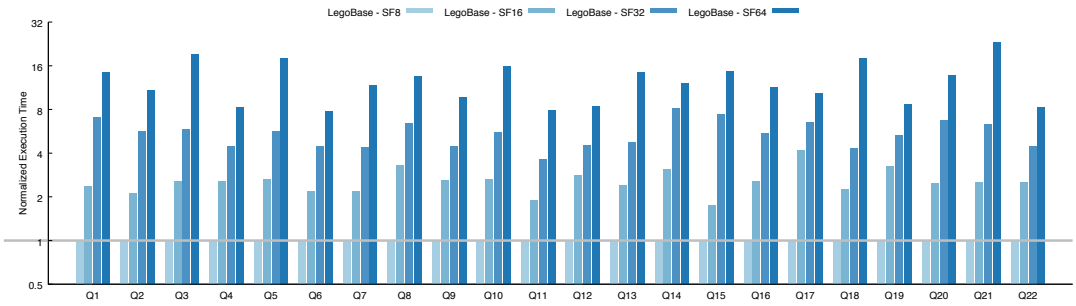


Figure 2.17 – The normalized query execution time for various scaling factors. The baseline is the execution time of the most optimized LegoBase using scaling factor 8 for each query.

2.4.7 Scalability Evaluation

Figure 2.17 shows the normalized query execution time of the most optimized generated code of LegoBase for different scaling factors for all TPC-H queries.[31]

Based on this experiment, we observe an almost linear increase in the query execution time, as we increase the size of input data, for most TPC-H queries. Note also that queries that perform aggressive data-structure repartitioning in LegoBase, such as Q21, currently do not scale linearly due to the increased memory pressure caused by the repartitioning. We plan to investigate implementing additional optimizations to handle these cases in future work.

In general, this experiment shows that LegoBase can scale to handle big datasets, as long as there is a sufficient amount of memory available for storing the original data, as well as the auxiliary repartitioned data.

2.4.8 Productivity Evaluation

An important point of this thesis is that the performance of query engines can be improved without much programming effort. Next, we present the productivity/performance evaluation of our system, which is summarized in Table 2.3.

We observe three things. First, by programming at the high level, we can provide a fully functional system with a small amount of effort. Less development time was spent on debugging the system, thus allowing us to focus on developing new useful optimizations. Development of the LegoBase query engine alongside the domain-specific optimizations required, including debugging time, eight months for only two programmers. However, the majority of this effort was invested in building the new optimizing compiler SC (27K LOC) rather than developing

[31] As we explained in the previous section, the total memory consumption can reach up to twice the size of the input data. In addition, the input data is stored in a Linux ramdisk (as explained in Section 2.4.1), which by construction consumes memory space equal to the size of the input data. Hence, it is not currently possible for us to perform experiments for scaling factor 128 or higher. This is because a scaling factor of 128 would, for example, require a total of up to 384 GB of memory (128GB for the ramdisk, plus 256GB for LegoBase, while our experimental platform has a total RAM size of 256GB).

Data-Structure Partitioning	505
Automatic Inference of Date Indices	318
Memory Allocation Hoisting	186
Column Store Transformer	184
Constant-Size Array to Local Vars	125
Flattening Nested Structs	118
Horizontal Fusion	152
Scala Constructs to C Transformer	793
Scala Collections to GLib Transformer	411
Scala Scanner Class to mmap Transformer	90
Other miscellaneous optimizations	≈ 200
Total	2930

Table 2.3 – Lines of code of several transformations in LegoBase with the SC compiler.

the basic, unoptimized, query engine itself (1K LOC).

Second, each optimization requires only a few hundred lines of high-level code to provide significant performance improvement. More specifically, for ≈3000 LOC in total, LegoBase is improved by 45.4× compared to the performance of DBX, as we described previously. Source-to-source compilation is critical to achieving this behavior, as the combined size of the operators and optimizations of LegoBase is around 40 times less than the generated code size for all 22 TPC-H queries written in C.

Finally, from Table 2.3 it becomes clear that new transformations can be introduced in SC with relative small programming effort. This becomes evident when one considers complicated transformations like those of Automatic Index Inference and Horizontal Fusion[32] which can both be coded for merely ≈500 lines of code. We also observe that around half of the code-base required to be introduced in SC concerns converting Scala code to C. However, this is a naïve task to be performed by SC developers, as it usually results in a one-to-one translation between Scala and C constructs. More importantly, this is a task that is required to be performed only *once* for each language construct, and it needs to be extended *only* as new constructs are introduced in SC (e.g. those required for custom data types and operations on those types).

2.4.9 Compilation Overheads

Finally, we analyze the compilation time for the optimized C programs of LegoBase(Opt/C) for all 22 TPC-H queries. Our results are presented in Figure 2.18, where the y-axis corresponds to

[32]To perform a decent loop fusion, the short-cut deforestation is not sufficient. Such techniques only provide *vertical* loop fusion, in which one loop uses the result produced by another loop. However, in order to perform further optimizations one requires to perform *horizontal* loop fusion, in which different loops iterating over the same range are fused into one loop [29, 124]. A decent loop fusion is still an open topic in the PL community [325, 71, 121].

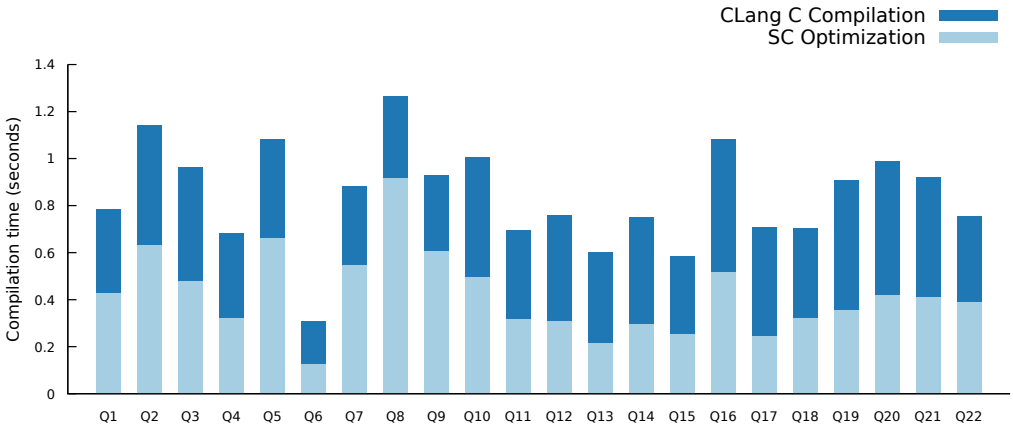


Figure 2.18 – Compilation time (in seconds) of all LegoBase(Opt/C) programs.

the time to (a) optimize an incoming query in our system and generate the C code with SC, and, (b) the time CLang requires before producing the final C executable.

We see that, in general, all TPC-H queries require less than 1.2 seconds of compilation time. We argue that this is an acceptable compilation overhead, especially for analytical queries like those in TPC-H that are typically known in advance and which process huge amounts of data. In this case, a compilation overhead of some seconds is negligible compared to the total execution time. This result proves that our approach is usable in practice for quickly compiling *entire* query engines written using high-level programming languages. To achieve these results, special effort was made so that the SC compiler can quickly optimize input programs. More specifically, our progressive lowering approach allows for quick application of optimizations, as most of our optimizations operate on a relatively small number of language constructs, thus making it easy to quickly detect which parts of the input program should be modified at each transformation step, while the rest of them can be quickly skipped. In addition, we observe that the CLang C compilation time can be significant. This is because, by applying all the domain-specific optimizations of LegoBase to an input query, we get an increase in the total program size that CLang receives from SC.

Finally, we note that if we generate Scala code instead of C, then the time required for compiling the final optimized Scala programs is 7.2× that of compiling the C programs with LLVM. To some extent this is expected as calling the Scala compiler is a heavyweight process: for every query compiled there is significant startup overhead for loading the necessary Scala and Java libraries. By just optimizing a Scala program using optimizations written in the same level of abstraction, our architecture allows us to avoid these overheads, providing a much more lightweight compilation process.

2.5 Conclusions

LegoBase is a new analytical database system currently under development. In this thesis, we presented the current prototype of the query execution subsystem of LegoBase. Our approach suggests using high-level programming languages for DBMS development without having to pay the associated abstraction penalty. This vision has been previously called *abstraction without regret*. The key technique to admit this productivity/efficiency combination is to apply generative programming and source-to-source compile the high-level Scala code to efficient low-level C code. We demonstrate how state-of-the-art compiler technology allows developers to express database-specific optimizations naturally at a high level as a library and use it to optimize the database systems code. In LegoBase, programmers need to develop just a few hundred lines of *high-level* code to implement techniques and optimizations that result in significant performance improvement. All these properties are very hard to achieve with existing compilers that handle *only* queries and which are based on template expansion. Our experiments show that LegoBase significantly outperforms both a commercial in-memory database system as well as an existing query compiler.

3 Modular Query Compiler

A language that doesn't have everything is actually easier to program in than some that do.
– Dennis M. Ritchie

This chapter studies architecting query compilers. The state of the art in query compiler construction is lagging behind that in the compilers field. We attempt to remedy this by exploring the key causes of technical challenges in need of well founded solutions, and by gathering the most relevant ideas and approaches from the PL and compilers communities for easy digestion by database researchers. All query compilers known to us are more or less monolithic template expanders that do the bulk of the compilation task in one large leap. Such systems are hard to build and maintain. We propose to use a stack of multiple DSLs on different levels of abstraction with lowering in multiple steps to make query compilers easier to build and extend, ultimately allowing us to create more convincing and sustainable compiler-based data management systems. We attempt to derive our advice for creating such DSL stacks from widely acceptable principles. We have also re-created the LegoBase query engine following these ideas and report on this effort.

3.1 Introduction

Query compilation has been with us since the dawn of the relational database era: IBM's System R employed query compilation in its very first prototype, but this approach was quickly abandoned in favor of query *interpretation* [53]. Recently, query compilation has returned to the limelight, with commercial systems such as StreamBase, IBM Spade, Microsoft's Hekaton, Cloudera Impala, and MemSQL employing it. Academic research has also intensified [134, 10, 202, 208, 255, 203, 204, 205, 199, 352, 74, 252, 185, 19].

We can argue that, despite all this recent work, the state of the art in the design of query compilers lags behind the programming languages and compilers research field. To the best of our knowledge, all existing query compilers are *template expanders* at heart. A template

expander is a procedure that, simply speaking, generates low-level code in one direct macro expansion step. While a query interpreter *calls* an operator implementation used inside a query plan, the template expander essentially *inlines* the operator code in the plan, for each operator, to obtain low-level code for the entire plan. We restrict our study to the actual compiler component of a possibly larger data management system. For instance, a system which first parses SQL into query plans and optimizes these Selinger-style, then feeds such plans to a compiler which generates LLVM bytecode, which is in turn lowered to machine code by LLVM, is still a template expander if that core compiler component – into which all of the DBMS engineering team's compiler efforts went – is sufficiently primitive, even if more than two languages and abstraction levels are present in the system as a whole.

Template expansion is a robust and intellectually accessible concept, but it has a number of drawbacks. The System R team reports that query compilation was abandoned in favor of interpretation since query compiler code was hard to maintain (cf. [53]). More fundamentally, template expanders make it impractical to support a range of sophisticated optimizations since multiple code transformers (with different optimization roles) have to be composed and inlined in all possible ways and orderings, causing a code size explosion in compiler code bases. Furthermore, template expanders make cross-operator code optimization hard to implement [199][1].

To illustrate the above-mentioned code explosion further, consider the example of a template expander that is to support two transformations: 1) pipelining (i.e. removing the need to materialize intermediate results between query operators) and 2) data-structure specialization (i.e. adapting the definition of a data structure to the particular context in which it is used). In order to perform these two optimizations together, one has to implement every combination of their respective cases. For example, if each optimization handles 3 different cases, one has to create a template expander with 9 cases to handle their combination. In general, the code complexity grows exponentially with the depth of the stack of desired transformations. Figure 3.1a illustrates this code explosion.

System R's initial query compiler as well as Hekaton are confirmed template expanders (cf., [53] and a private communication with the Hekaton team). While academic work makes numerous contributions to the practice of query compilers, it is fair to say that creating a query compiler that produces highly optimized code is a formidable challenge due to the needs to work with various Domain-Specific Languages (DSLs), understand their optimization potentials, and build on the wealth of algorithmic and systems results created by the research over multiple decades. The database community can profit from further tools and techniques from the compilers field.

In this chapter, we provide, to the best of our knowledge, the first principled methodology for building query compilers.

[1] The key contribution of [255] is to show how a push operator interface can eliminate the need for cross-operator fusion transformers, making template expanders produce faster code.

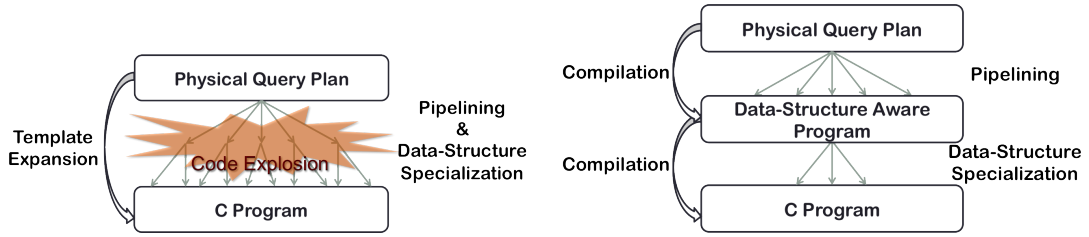
We create tools and techniques to increase the *modularity* of query compiler components, to manage the complexity of these systems. Instead of using template expansion to directly generate low-level code from a high-level query plan, we propose **progressively lowering the level of abstraction** until we reach the lowest level, and only then generating low-level code. Each level of abstraction and each associated optimization can be seen as independent modules, enforcing the principle of *separation of concerns*. There are two kinds of code transformations, optimizations (where the source and target DSLs are the same) and lowerings. By supporting optimizations on every abstraction level, we obtain real power – moving to a lower-level DSL tends to expand the code size and there is a tradeoff between the search space for optimizations and the granularity of the DSL. Optimizations such as join reorderings are only feasible in high-level DSLs, while register allocation decisions can only be expressed in very low-level DSLs. We propose to use a stack of multiple internal DSLs, one for each such abstraction layer. We attempt to derive our advice for creating such DSL stacks from easily acceptable principles.

Returning to the above example, we can define a third intermediate abstraction level, a *data-structure aware DSL*, to place between the source and target languages. Pipelining transforms a high-level query plan to that intermediate language which has explicit constructs for the operations on the hash-table and list data structures. Then, data-structure specialization transforms the program from this language to low-level code by using an appropriate implementation of each data structure based on its context. Using this intermediate DSL and stepwise lowering, there is no longer any need to consider every combination of the two optimizations, as they no longer interfere by manipulating the same expression. As a result, the complexity of the query compiler code base is more manageable, as demonstrated in Figure 3.1b.

Creating a sophisticated query compiler is a challenging undertaking, and there are a number of results from the PL and compilers research communities that can help. We explore the key technical challenges in need of well founded solutions, and attempt to gather the most relevant ideas and solution approaches from the PL and compilers communities for easy digestion by database researchers. Specifically, we look at the choice of intermediate languages, how to maximize the separation of concerns among compiler optimizations and lowerings, implementation design choices, and several transformations expressible using this approach.

Throughout this chapter, we use the Scala language for our examples and for embedding our DSLs, but nowhere does this work specifically depend on this choice of programming language.

This is an atypical work in that our main contributions are hard to experimentally validate. Our insights are based on building four distinct compiler-based data management systems over the past seven years, but creating query compilers with a maximally shared codebase for the multitude of alternatives discussed in this thesis is beyond the resources of any research group.



(a) Pipelining and data-structure specialization are applied concurrently. Hence, we should apply brute-force to all combinations, resulting in code explosion.

(b) Pipelining and data-structure specialization are applied sequentially. Hence, there is no need to consider all combinations, resulting in managed code base.

Figure 3.1 – Handling concurrent optimizations in template expansion and progressive compilation approaches.

Instead, we have re-created a well-known query compiler [199] following our ideas – our DSL stack with stepwise lowering – and report on this effort. We report on the productivity of creating this compiler using a suitable DSL compiler framework [100], and show that we are able to implement all the optimizations of [199] (and more), obtaining at least comparable and often better performance.

3.2 Overall Design Principles

3.2.1 Background

Most database management systems to date use high-level DSLs such as SQL to express queries. Those queries are transformed into optimized physical query plans, and then passed to an engine that interprets them. This approach pays the cost of interpretation overhead, and many low-level optimization opportunities are missed (e.g. function inlining), because they are not expressible in the high-level DSL of query plans. These sources of overhead may become especially significant for queries working with in-memory data, where computation is increasingly CPU-bound (rather than I/O-bound). To circumvent these limitations, query compilation aims to generate low-level code from the high-level query plans, allowing specific optimizations to be applied at this low-level representation[2].

However, by directly generating low-level code, we also miss optimization opportunities that are both not available at the high level, and hard to express at the low level – mainly because

[2] In this work, we consider compilation as a form of *partial evaluation* [172]. A compiler-based approach can always delay some of this partial evaluation to runtime, in which case the nature of the system moves, at least in some aspect, from compilation to interpretation. A strict choice of either one or the other at design time is not, however, necessary. More concretely, there are scenarios where compilation staging, just-in-time compilation, or even interpretation are desirable: if some essential piece of information that can substantially speed up evaluation is not available at compile time, it is better to delay this partial evaluation until that information is available. The classical example is probably statistics for query optimization. Even a compilation-based query engine cannot afford to go without a query optimizer.

the locality of code patterns to match is lost. For example, loop fusion (i.e. replacing multiple loops with a single one) is not expressible in query plans, as there is no notion of loop at this abstraction level. Also, it is proven to be NP-complete [80, 192] and impractical in imperative languages such as C [114], the usual target of query compilers.

We propose the introduction of intermediate abstraction levels (referred to as *intermediate DSLs*) for expressing such optimizations. This has already been done in other domains, such as signal transforms and linear algebra. In these domains, we can cite the Σ-SPL [114] and Σ-LL [309] intermediate languages, which have been developed in Spiral [277] for expressing loop fusion optimizations.

In this work, we show that by using several abstraction levels and by doing *step-wise lowering* across them, we can express powerful optimizations (such as pipelining and data-structure synthesis) that are hard or impossible to express in existing query compilation approaches. The resulting set of DSLs, ordered from higher level to lower level, is referred to as the *DSL stack*.

3.2.2 Choosing The DSLs

Although the design space for a DSL stack may seem overwhelming, there are several constraints that make some design choices impractical, or even infeasible. These directions can be discarded. Among these constraints, there is an important principle for designing intermediate abstraction levels related to the expressive power of programming languages, in the sense of Felleisen [108]:

> **Expressibility principle:** *Any program written in a given DSL should be expressible in any of the lower-level DSLs as well. Therefore, by lowering the level of abstraction from the former to the latter, one should retain the same expressive power or gain more.*

Note that the converse does not need to hold: a program in a low-level DSL does *not* need to be expressible in higher-level DSLs. Based on the expressibility principle, we can start designing an appropriate DSL stack. For that, we need to answer the following questions: 1) What abstraction levels (DSLs) do we need? 2) On which abstraction level(s) should we put each transformation?

We propose a methodology that naturally answers both questions at the same time: to design a DSL stack, one should start from the simplest stack possible (a high-level DSL that maps directly to a low level one), then iteratively examine the desired transformations one by one and see if the existing DSL stack is sufficient for it or if a new abstraction level needs to be introduced.

Every transformation requires an input program in a *source language* and produces a program in a *target language*. If the source language and the target language are the same, the transformation is called an *optimization*, whereas if the target language is at lower level, it is called a *lowering* transformation. We will see in the next section why the source language is never going to be lower-level than the target language.

Optimizations are subject to the well-known *phase-ordering problem* [342]: most optimizations can produce opportunities for other optimizations that apply to the same language, but it is not clear how to order them optimally. This problem is still a topic of research in the Programming Language community, and no definitive answer has been formulated yet.[3] To mitigate this, we recursively apply optimizations inside the same abstraction level until we reach a fixed point,[4] where either no more optimizations can be applied or the application of an optimization does not yield structurally different code. On the other hand, lowering transformations are not subject to this issue, because they change the abstraction level of the program so previous optimizations are no longer applicable.

Note that lowering transformations should always be applicable, irrelevant of the input program. Otherwise, the transformation chains would be broken. In contrast, optimizations may or may not be applicable, depending on the information and patterns found in the program being optimized.

3.2.3 Constructing The Stack

For the DSL stack to be well-formed and to maximize the reuse of transformations, we need to ensure that transformations are not redundant. Each lowering transformation translates a DSL to the next lower-level DSL, and each high-level DSL program is, through a sequence of lowerings, eventually mapped to the target language. This requirement is summarized in the following principle:

Transformation cohesion principle: *Between any two DSLs of different abstraction levels, there should be a unique path of lowering transformations translating programs in the higher-level DSL into programs in the lower-level one.*

Notice that this does not prevent having several different high-level front-end DSLs, or different low-level target languages. On the other hand, the principle implies that there can be no transformations from a given DSL a to a higher-level DSL b, as there would also need to be a path down from b to a, creating a loop, and thus an infinity of lowering paths from b to a, violating the principle.

[3] *Online* (or *local*) transformations do help removing ordering problems for a certain class of optimizations [173]. We provide facilities (cf. [50]) to encode them in our framework, but this is out of the scope of the current chapter.

[4] Special care needs to be taken in the design of optimizations to ensure that iteratively applying them leads to termination.

Let us consider the case where we need to introduce a new abstraction level, which mainly happens when we have more than one lowering transformation between two particular DSLs. In such a case, we have to split either the source language or the target language into two separate languages. The new intermediate language should follow the expressibility principle, interpolating between the level above and below. Furthermore, the affected transformations should update their source or target languages: optimizations on the source and target languages, and the lowering transformations from the source to the target language. This update in the existing lowering transformations is necessary as, otherwise, the transformation cohesion principle would be violated.

Going back to our working example, consider a query compiler with SQL as the query language, C as the target language, and pipelining and data-structure specialization as transformations. First, we start from a two-level DSL stack, as shown in Figure 3.1a.

Second, we consider where to place the pipelining transformation. The information required for checking the applicability of pipelining is available in SQL. However, it is very hard to check such opportunities in a low-level language like C. Expressing a pipelined program is not possible in SQL. This requires the expressibility of a lower language, like C in this case. Hence, pipelining is a lowering transformation from SQL to C.

Finally, we examine the existing DSL stack (which is still two levels till now) for adding data-structure specialization. Similar to pipelining, this transformation is a lowering from SQL to C. This is because we need the high-level information available in SQL, but also a way to explicitly represent data structures (not possible in SQL). We now have two lowering transformations from SQL to C. This means we should break one of these languages into two languages to modularize these two transformations. In this case, we break C into two languages: 1) Data-structure-aware C, which is an extension to the standard C language with specific constructs for the data structures of a query engine, and 2) The standard C language. Data-structure specialization can now use data-structure-aware C as its source language and low-level C as its target language. At last, all transformations which had C as their source or target language should be updated accordingly. In this case, pipelining should update its target language in order to follow the transformation cohesion principle. Pipelining can be expressed in data-structure-aware C as well. Hence, the pipelining transformation is now a lowering transformation between SQL and data-structure-aware C. The new DSL stack is shown in Figure 3.1b.

In the next section, we further discuss the design space as well as compilation concerns for different DSL stacks.

Paradigm	Advantages
Declarative	✓ Concise programs ✓ Small search space of equivalent programs ✓ Simple to analyze and verify ✓ Simple to parallelize
Imperative	✓ Efficient data structures ✓ Precise control of runtime constructs ✓ More predictable performance

Table 3.1 – Comparison of declarative and imperative languages

3.3 Design Space

3.3.1 Imperative vs. Declarative

So far, we have been referring to *high-level* and *low-level* languages without providing a clear definition for these terms. A more precise terminology would have us use *declarative* and *imperative* instead, although these do not always coincide. This dichotomy is not the only criterion to take into account while designing abstraction levels, but it does play a central role, as we will see. In a declarative language, programs are close to specifications of the results we want to compute, while in an imperative language, details about how the result is computed are made explicit. The latter usually involves the use of side-effects like mutation. Table 3.1 summarizes the important differences between these two approaches.

Different mixes of declarative and imperative features in a DSL are amenable to different optimizations, and require different compilation techniques. Programs in high-level languages are smaller, and the search space of semantically equivalent programs is therefore more manageable. This allows us to use search strategies similar to the ones query optimizers use. These result in global optimizations that have more impact on the resulting program than optimizations on low-level programs, since an expression in a high-level program corresponds to many low-level expressions.

Moreover, high-level languages usually target specific domains – in our case, query processing. As a result, they need not be as complete and powerful as general-purpose languages: like SQL, they do not even need to be Turing-complete [222]. This allows compilers to provide more guarantees, and to perform better reasoning, static analysis, and optimization [346]. For example, it is feasible to statically reason about the runtime cost of a SQL query with typically good or acceptable accuracy. This is *not* possible in low-level Turing-complete languages without actually running the program [201].

Since low-level programs are larger, there is a large number of equivalent low-level programs. This makes the use of search techniques [201] impractical, preventing the global optimization of these programs. Instead, optimizing compilers usually perform only local optimizations (typically, so-called “peephole” optimizations, that only consider a small part of the program

at a time).

In the rest of this section, we see how to design our intermediate DSLs to integrate more or less imperative and declarative features, depending on the type of optimizations we want to perform on them. Then, we discuss different possible Intermediate Representations (IRs) to encode programs written in these DSLs.

3.3.2 DSL Design and Optimization

Functional languages are a particularly important class of declarative languages. We use functional features to represent the declarative aspects of our DSLs for several reasons: first, these languages are easy to understand and to reason about, possibly reusing frameworks developed in the PL community [52]; second, functional constructs integrate well with imperative ones [265], supporting our goal of progressively turning programs from a declarative to an imperative form; finally, the hybrid Scala programming language, which we use in our framework, naturally allows such a mix of functional and imperative code.

Although functional programs are executable, they introduce a relatively heavy performance penalty and do not allow enough control on runtime constructs. This prevents the fine tuning of program performance. For example, standard functional data structures are immutable, and do not allow in-place modification of their elements, requiring copies instead. Without aggressive optimizations like deforestation [357], this translates into many unnecessary allocations and copies. Therefore, we need to lower these high-level declarative constructs into specific, optimized imperative representations. Such representations are close to the underlying architecture, providing opportunities for fine-grained performance tuning.

On the other hand, imperative programs that exhibit poor performance are harder to optimize. For example, detecting loop fusion opportunities is much harder in imperative programs than in functional ones. If the loops in an imperative program are not manually fused by the programmer, there is little chance that the compiler will be able to fuse them automatically. The reason is that optimizing imperative programs with side effects is notoriously hard [20], chiefly because the compiler has to reason about aliased mutable memory locations, a problem that has been shown to be intractable in general [281]. This has implications on low-level optimizations.

For example, consider a **`for`** loop written in C that only manipulates local variables. Modern compilers know how to optimize such constructs in near-optimal, almost unbeatable ways. But as soon as one introduces non-trivial function calls inside the loop, the compiler's bets are off and many automatic rewritings become impossible. Consider the following code:

```
for (int i = 0; i < size(str); i++) { str[i] = 'X'; }
```

In general, a compiler must not assume that it is safe to extract the call to `size(str)` out of the loop, because the way it is computed could be influenced by assignments performed inside

the loop body. In fact, for the particular case where `str` is a simple C string, the compiler cannot know that we are not going to override the string termination character while iterating over the string (which would change the result of a subsequent call to `size`).[5] This would prevent the compiler from implementing resetting the characters of a string as an efficient `memset` instruction.

In this context, human expertise becomes important again. Based on domain-specific knowledge, one can make assumptions that low-level C compilers cannot, even after expensive program analyses that try to recover high-level information from the low-level code. Since we progressively lower abstraction one step at a time, we can exploit as many optimization opportunities as possible along the way. Our framework allows the expression of effectful computations, but can still reason about code that is known to be pure, and our transformations can leverage invariants that are known to hold in the intermediate DSLs.

Next, we discuss various intermediate representation choices for each abstraction level.

3.3.3 Intermediate Representation

Compilers usually convert input programs, given as text strings, into an *Intermediate Representation (IR)* which contains all essential information available about the program after parsing[6]. Optimizing compilers use IRs to facilitate the definition and application of optimizations.

The simplest form of an IR is an *Abstract Syntax Tree (AST)*. In database management systems, an AST represents a query in relational algebra or its physical plan. This representation is sufficient for performing algebraic rewrite rules on such algebraic languages without variable bindings. Examples of transformations include pushing down selections or changing the join order in relational algebra. As an example, Stratego [354] uses ASTs for its IR.

However, there are optimizations that require more sophisticated IRs. For example, *Common Subexpression Elimination* (CSE) involves sharing leaves among sub-trees, which calls for a DAG rather than a tree representation, or equivalently a language with variable bindings (cf. [11, 127]).

Furthermore, as the language becomes more complicated (e.g., through the introduction of mutability), programs start requiring *data-flow analysis* [180, 179, 190] to check for the applicability of optimizations. Performing data-flow analysis for every independent optimization that requires it results in more analysis passes than necessary and is difficult to implement, debug, and maintain [340]. Hence, the need to replace plain ASTs with a data structure that stores the result of data-flow analysis, as a better representation on which to apply optimizations.

[5] A special case could be added in the compiler to handle this particular example, but this approach does not scale, as the general problem is undecidable.

[6] Observe that *different* DSLs or abstractions levels may use the *same* IR as their underlying data structure; however, the information (DSL constructs) encoded using these IRs may vary significantly.

There are several IRs proposed in the PL community which simplify data-flow analysis, chief among them 1) SSA [292, 76], 2) CPS [17, 189], and 3) ANF [110]. hese IRs encode data-flow information by converting a given program into a *canonical representation*. For example, in all of these IRs, every subexpression in a program is bound to a local variable, and reassignment to these variables is not allowed (i.e. they are immutable) [52]. Although it has been proven that these IRs are semantically equivalent [110, 16, 188], from a practical point of view there are advantages and disadvantages for using each one. Even in the PL community, there is no consensus on which IR is the best [189, 110, 52]. It is, however, undisputed in the PL community that a simple use of ASTs, the dominant choice in work by the database community, creates the problems mentioned before and is inferior to these three for many uses.

All DSLs in our stack use A-normal form (ANF) [110]. The reasons for using ANF can be summarized as follows: first, ANF simplifies data-flow analysis by allowing a single definition of variables [52], which has a great impact on simplifying optimizations that use data-flow information, such as CSE – indeed, it facilitates the mix of effectful and pure computations, and has been shown to make optimizing transformations and analyses easier to write [25]; second, both ANF and SSA preserve the natural organization of programs (called "direct-style"), making them more understandable than CPS (which uses "continuation-passing style", whereby functions call continuations instead of returning values [110]); third, as ANF is a direct-style representation of λ-calculus [110], there are very well-known frameworks for the analysis and verification of λ-calculus developed in the PL community [217]. Hence, reasoning about compiler optimizations in ANF is simpler [52]; finally, by converting many semantically equivalent programs into a canonical representation, expressing optimizations becomes simpler [195], as there is no more any need to express optimizations for all semantically equivalent cases: it suffices to only express one for the canonical form.

ANF is better illustrated with an example. Consider the following expression which represents a part of an aggregation:

```
agg1 += R_A * R_B
agg2 += R_A * R_B * (1 - R_C)
agg3 += R_D * (1 - R_C)
```

After converting to ANF, all operators should accept either a constant or a local variable. Hence, every arithmetic operation is converted to a version which uses the local variable bound to its arguments. The ANF representation of this expression is as follows:

```
val x1 = R_A * R_B
agg1 += x1
val x2 = 1 - R_C
val x3 = x1 * x2
agg2 += x3
val x4 = R_D * x2
agg3 += x4
```

While converting a subexpression to an immutable variable, one can look up the mappings

between the existing variable bindings and their corresponding subexpressions. If there is already a subexpression with the same operator and the same arguments, then the existing bound variable can be reused. This provides CSE *for free*. In the previous example, the expression `R_A * R_B` is computed once and is used twice for both `agg1` and `agg2`. The same happens for `1 - R_C` in both `agg2` and `agg3`. Observe that this optimization is only one of the advantages of the ANF form.

Finally, we encode additional information (other than data-flow information) about the expressions in the IR. There are cases in which we need some high-level information about the expressions which is not available in the current abstraction level. Such information can be guided through *annotations* from a higher level of abstraction. Note that since ANF assigns a unique symbol to each subexpression, this process is simplified by keeping a hash-table from these unique symbols to their associated annotations.

There are other IRs proposed in the literature, which mainly target optimizations for parallel architectures (e.g. PDG [109]), or are meant to help and combine compiler optimizations (e.g. a *sea of IR nodes* [67] and E-PEG [331]). However, these IRs complicate the debugging process of program compilation, and are seldom used in real-world, mainstream compilers [313]. Although we did not consider using these IRs, it would be an interesting direction for our future work, particularly when targeting parallel architectures.

3.4 DSL Stack

In this section, we present the construction of our DSL stack by progressively refining a naïve two-level stack consisting of query plans and C, respectively the source and target languages commonly used in existing query compilers. We progressively add intermediate abstraction levels to perform new optimizations in a modular way, as described in Section 3.2. Finally, we demonstrate the straightforward addition of a new front-end language which reuses the lower abstraction levels already defined in the DSL stack, thereby benefiting from all transformations that apply to them.

Scala DSLs. Figure 3.3 shows the final DSL stack. On the right, we show which constructs are added and removed by each intermediate DSL. Scala is the implementation language of the framework, and we use a subset of its features to encode intermediate DSLs as well. We refer to the main subset as ScaLite. We write ScaLite[X,Y,...] to denote ScaLite augmented with features X, Y, etc. QPlan and QMonad are Scala DSLs used as two possible front-ends for the DSL stack. ScaLite[Map, List] and ScaLite[List] are intermediate, data-structure-aware DSLs used for specializing the abstract *hash table* and *list* data structures. As we will see, we enforce more restrictions on higher-level DSLs, so that for example, restricted mutability makes ScaLite[Map, List] in fact *less* expressive than ScaLite[List], where mutable Maps can be implemented directly. In ScaLite, all data structure are completely implemented in the language itself, but the memory is assumed to be managed by the garbage collector of a runtime system (e.g.

the JVM). Finally, C.Scala is another Scala DSL that expresses C constructs, and in particular memory manipulation constructs, so a program written in C.Scala is equivalent to a C program, modulo a straightforward syntactic transformation (called *stringification* or *unparsing*).

The advantage of using Scala to host these DSLs (we say they are *embedded* [158] in Scala) is that we benefit from its compilation tool-chain: parsing, type-checking, execution and debugging. Indeed, each DSL is executable as a Scala program, with low performance but improved debugging possibilities. Note that these DSLs could be designed in other programming languages (e.g., *quoted* DSLs [253] in Haskell) or as external DSLs if one is willing to build the compilation tool-chain from scratch.

Example Query. We use one query as a running example for demonstrating transformations. The query is shown in SQL and expressed in each intermediate DSL (with Scala syntax) in Figure 3.2.

3.4.1 Two-Level Stack (QPlan & C)

This is the two-level stack corresponding to existing query compilers, which are template-based.[7] The high-level DSL is an algebraic representation of query operators. A *query optimizer* typically finds the best query plan for a given SQL query, and produces a program in this declarative DSL. The low-level DSL is an architecture-dependent language that can express implementation details useful for tuning performance. Typical choices include C and LLVM IR.

QPlan. The QPlan DSL contains query plan operators typically encountered in various commercial database systems, including semi-, anti- and outer joins. These operators are sufficient for expressing a large class of SQL queries, including the 22 TPC-H [343] queries.

C.Scala. C.Scala is an extension of ScaLite with basic memory management constructs (e.g. `malloc` and `free`) and memory referencing constructs (pointers and pointer arithmetic). We use the GLib data structures to represent dynamic `Array`s and sorted lists (as binary search trees).

Transformations. At this stage, we perform pipelining while producing C.Scala code, either by pulling [128] or pushing [255] data. With this approach, we remove many materialization points, which results in improved data locality and I/O costs. This transformation is discussed further in Section 3.5.1.

[7] A state-of-the-art database system will often manipulate SQL, relational algebra and basic query plans before handing the result to the query compiler, but these are not seen by the query compiler, and are thus not considered in this chapter.

```
SELECT COUNT(*)
FROM R, S
WHERE R.name == "R1"
  AND R.id == S.id
```

(a) The example query in SQL.

```
AggOp(HashJoinOp(
  SelectOp(ScanOp(R),
    r => r.name == "R1"),
  ScanOp(S),
    (r,s) => r.id == s.id
), (rec, count) => count + 1)
```

(b) The example query in QPlan.

```
R.filter(r =>

  r.name == "R1"
).hashJoin(S,
  r => r.sid, s => s.rid
).count
```

(c) The example query in QMonad.

```
val hm = new MultiMap[Int,R]

for(r <- R) {
  if(r.name == "R1") {


    hm.addBinding(r.id, r)

  }
}
var count = 0
for(s <- S) {
  hm.get(s.id) match {
    case Some(rList) =>
      for(r <- rList) {
        if(r.id == s.id)
          count += 1
      }
    case None => ()
} }
return count
```

(d) The example query in ScaLite[Map, List].

```
val MR: Array[Seq[R]] =
 new Array[Seq[R]](BUCKETSZ)
for(r <- R) {
  if(r.name == "R1") {


    MR(r.id) += r

  }
}
var count = 0
for(s <- S) {

  val rList = MR(s.id)
  for(r <- rList) {
    if(r.id == s.id)
      count += 1

  }
}
return count
```

(e) The example query in ScaLite[List]. Note that List is a mutable data structure.

```
val MR: Array[R] =
  new Array[R](BUCKETSZ)
for(r <- R) {
  if(r.name == "R1") {
    if(MR(r.id) == null) {
      MR(r.id) = r
    } else {
      r.next = MR(r.id)
      MR(r.id) = r
} } }
var count = 0
for(s <- S) {

  var r: R = MR(s.id)
  while(r != null) {
    if(r.id == s.id)
      count += 1
    r = r.next
  }
}
return count
```

(f) The example query in ScaLite.

```
val MR: Array[Pointer[R]] =
  malloc[Pointer[R]](BUCKETSZ)
for(r <- R) {

  if(r->name == "R1") {
    if(MR(r->id) == null) MR(r->id) = r
    else {
      r->next = MR(r->id)
      MR(r->id) = r
} } }
var count = 0
for(s <- S) {
  var r: Pointer[R] = MR(s->id)
  while(r != null) {
    if(r->id == s->id) count += 1
    r = r->next
} }
return count
```

(g) The example query in C.Scala.

```
R** MR = (R**)
  malloc(BUCKETSZ * sizeof(R*))
for(int i=0; i < R_REL_SIZE; i++) {
  R* r = R[i];
  if(strcmp(r->name, "R1") == 0) {
    if(MR[r->id] == NULL) MR[r->id] = r;
    else {
      r->next = MR[r->id];
      MR[r->id] = r;
} } }
int count = 0;
for(int i=0; i < S_REL_SIZE; i++) {
  S* s = S[i]; R* r = MR[s->id];
  while(r != NULL) {
    if(r->id == s->id) count += 1;
    r = r->next;
} }
return count;
```

(h) The example query in C.

Figure 3.2 – Representations of a query in different DSLs.

3.4.2 Three-Level Stack (+ ScaLite)

We saw in the introduction that this two-level stack with pipelining is not appropriate for adding a transformation that affects constructs that are also affected by pipelining. This is

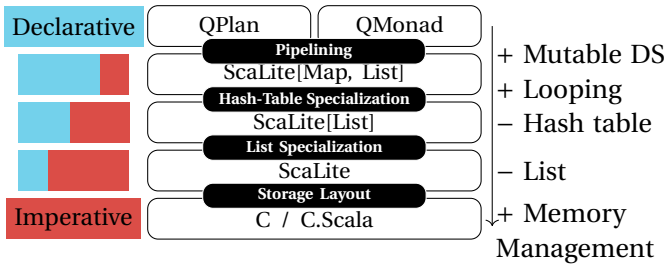


Figure 3.3 – A DSL stack for query compilation

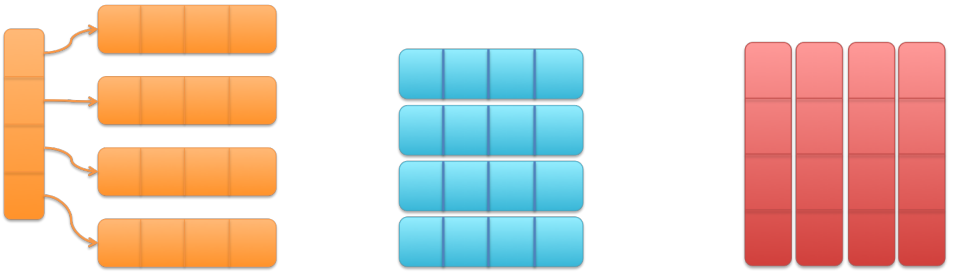
because both transformations interfere by manipulating related constructs at the same time (single compilation stage). Here, we want to introduce memory-management and layout optimizations. To resolve the problem as suggested in Section 3.2, we add a new intermediate DSL that can express pipelining, but does not contain memory management constructs yet.

ScaLite. The core of ScaLite is the simply-typed λ-calculus, which does not have recursion and mainly consists of constructs for function abstraction (a.k.a. λ abstraction) and for function application (invoking a function with an input parameter, possibly another function). ScaLite additionally supports control-flow constructs such as **`if`** statements and bounded loops (loops for which we statically know the maximum number of iterations). This DSL is not a purely functional language, as it also supports variables that are either immutable (as in **`val`** `x = e; f(x)`) or mutable (as in **`var`** `x = e1; f(x); x = e2`). It supports user-defined records and three data structures: fixed-size arrays, dynamic arrays, and sorted lists.

These make ScaLite a powerful enough low-level language satisfying the expressibility principle. The restrictions (bounded loops, no recursion) simplify program analysis.

Transformations. While lowering ScaLite programs to C.Scala, we enhance memory management. For example, we use memory pools to preallocate intermediate records. By using statistical information about the input, a worst-case estimate of the cardinality of elements is used to preallocate a memory pool, which obviates the need to perform a system call when allocating new memory. Another memory-management enhancement is to specialize the memory layout of data structures. For example, depending on the context, we represent an array of records either as: 1) an array of pointers to structs – a *boxed* [370] *layout*; 2) an array of structs [352]; or 3) a struct containing one array for each record field – a *columnar layout* [162, 317], which often has a positive impact on cache locality. These data-layout representations are demonstrated in Figure 3.4.

Figure 3.2g shows our working example in C.Scala. Line 1 explicitly specifies the representation of an array of records as an array of pointers to records. In line 2, we use the `malloc` function to allocate the array. In the rest of the program, we use arrows (`->`) to access the fields of a referenced record, as in C.



(a) Boxed layout (b) Row layout (c) Columnar layout

Figure 3.4 – Different data-layout representations.

3.4.3 Four-Level Stack (+ ScaLite[Map, List])

Now, consider adding a transformation for data-structure specialization, which requires a data-structure-aware DSL, as explained in Section 3.2. Such a DSL has specific constructs for representing the operations of a set of data structures.

Hash tables and lists are two essential data structures for query engines [280]. Hash tables are used for implementing the aggregation and hash join operators, while lists are used for storing intermediate collections of records. There are two kinds of hash tables we are interested in: `HashMap`s associate every key to a single value, and are used for expressing aggregation operators; `MultiMap`s associate every key to a set or list of values, and are used for expressing hash join operators.

These data structures are specialized to efficient implementations, depending on the context in which they are used, which requires a prior analysis phase. This is not possible to do with naïve expansion strategies, like with C++ templates or C macros. It could also be interesting to consider other specialized data structures such as indexed trees to extend the DSL further. This can be useful for other workloads, which we leave as future work.

ScaLite[Map, List]. This DSL is an extension of ScaLite with the `HashMap`, `MultiMap`, and `List` data structures, as well as operations defined on them. For simplifying program analysis, we ensure that for every program in this DSL, the following invariant holds: hash-table data structures are not allowed to contain mutable elements, which means that the records we put into these hash tables should be immutable. This is because if we were allowed to read *and* write fields of an element obtained from a hash table, inferring the access patterns for this hash table would become significantly more complicated. Another way of expressing this invariant is to state that the DSL does not allow *nested mutability*.

Transformations. At this level, we can perform optimizations that use hash tables, such as *string dictionaries* [35]. This optimization maps string operations to integer operations, as it was discussed in Section 2.3.4. Also, hash-table specialization is performed while lowering ScaLite[Map, List] programs. Access patterns are analyzed before this transformation in order

to make informed *materialization* decisions and push some computations to a pre-processing phase, which is explained in more details in Section 3.5.2.

Figure 3.2d shows our working example in ScaLite[Map, List]. Implementing a hash join is done in two phases: the first phase (lines 3-12) iterates over the elements of the first relation and *builds* a hash table which groups these elements based on their join key; in the second phase, the algorithm *probes* the elements of the second relation and iterates over the corresponding elements of the first relation using the constructed hash table.

3.4.4 Five-Level Stack (+ ScaLite[List])

Directly translating hash tables to arrays is not necessarily optimal. We would like to translate them to lists first, so we can reuse the fine-grained, context-dependent lowering already defined that converts lists to arrays. Sometimes, it is better to lower lists to linked lists, whereas in some other cases, it is better to lower them to arrays. In order to do that, we add an abstraction level similar to the one above, but without hash tables.

ScaLite[List]. ScaLite[List] is also built atop ScaLite, but only adds constructs related to Lists. However, to encode `MultiMap`s using arrays of lists, we need to relax the restrictions imposed on ScaLite[Map, List]: if nested mutability was forbidden, there would be no way to express `MultiMap`s in a useful way, because we would not be able to update the set of elements associated with a particular key incrementally (a capability that was hidden behind the `MultiMap` interface of ScaLite[Map, List]).

Transformations. We perform list specialization while lowering from ScaLite[List] to ScaLite. Consider the case in which we lower lists to linked lists. In a typical scenario, lists are used for holding records. In that case, we use *intrusive linked lists*, which store the next pointer of each list node in the records themselves. This removes one level of indirection caused by the separate allocations of the container nodes and the records. On the other hand, since we are working with ScaLite, which only has bounded loops, it is sometimes possible to perform worst-case size analysis and obtain an estimate of the maximum cardinality of some lists. We consequently lower them to native static `Array`s, instead of linked lists. This way, we benefit from the existing array layout optimizations provided for ScaLite down the DSL stack (e.g. columnar layout).

Figure 3.2e shows the working example in ScaLite[List]. Lines 1-2 contain the lowered representation of the `MultiMap` data structure. Line 7 shows the implementation of the `addBinding` method using the `+=` method of List. Line 16 shows how we lower the `get` method of `MultiMap` by accessing a bucket in the lowered array.

3.4.5 Collection Programming Front-end

We refer to *collection programming* as the practice of preferring generic operations defined on collections like lists and associative maps (`filter`, `groupBy`, `sum`, etc.) rather than writing them out as loops. The user base of *collection programming* APIs is growing. These APIs improve the integration of applications with database back-ends by making them more seamless [238, 135, 134]. Thus, it makes sense to consider a DSL with a collection programming API as an alternative front-end for a query compiler.

QMonad. The QMonad DSL is a functional language inspired by Monad Calculus on lists [44, 45, 358], Query and Monoid Comprehensions [136, 345, 107] and other collection programming APIs like Spark *RDD*s [376]. In addition to standard collection operators (such as `map`, `filter`, `fold`, etc.), this DSL contains different join operators including semi-, anti-, and outer joins.[8]

Transformations. As we discussed in Section 3.3.2, declarative and functional languages are not appropriate for performance tuning. It is thus necessary to compile and optimize programs written in QMonad before executing them. Thankfully, by simply lowering those programs to ScaLite[Map, List], we can reuse the transformations provided by the lower level DSLs of our stack *for free*. To produce even faster code, we should perform pipelining to remove materialization points (pipeline breakers). This transformation has a similar effect to what we do for QPlan, by pushing or pulling data. Section 3.5.1 gives more detail about it.

Figure 3.2c shows our working example in QMonad. The `filter` method is a higher order function that corresponds to the selection operator in relational algebra. It takes the selection predicate as a parameter. The first parameter of the `hashJoin` method is the second relation, and the second and third parameters indicate the join keys of the first and second relations, respectively. The `count` method returns the number of elements in a list.

3.4.6 Extensibility

One of the main advantages of our DSL stack design is its extensibility in various dimensions. First, as we just saw, one can use another algebra as the front-end by replacing the front-end DSL (here, QMonad and QPlan). By providing a lowering transformation from the new algebra to one of our intermediate DSLs, the existing infrastructure generates optimized C code for that new front-end.

Second, user-defined functions (UDFs) can be added to input queries. There are two approaches for doing so: 1) expressing them in terms of our low-level DSLs – in this case, we miss high-level optimization opportunities that could apply to them; 2) adding UDFs as constructs in a high-level DSL, and defining lowerings to immediately-lower DSLs in the appropriate

[8] Map and join expressions are expressively redundant with nested fold expressions, but represent an important performance choice, and are hard to reconstruct from folds.

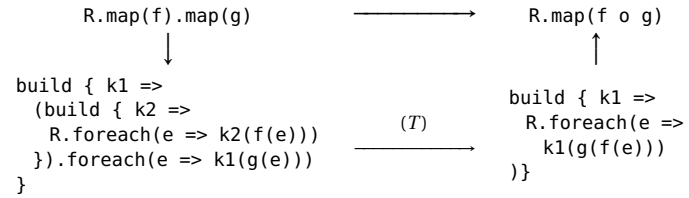


Figure 3.5 – A simple example of loop fusion using short-cut fusion.

phase. This approach works best if the user provides additional information (by using annotations, which was desribed in Section 3.3.3) about the additional language construct (e.g. their side-effects).

Third, we can change our target language without any need to change the higher-level DSLs, and still benefit from the optimizations provided in those higher-level DSLs. The only thing we need to do is to provide a lowering from ScaLite (or a higher-level DSL) to our desired target language, and then *unparse* the generated IR in a similar way as we unparse C.Scala to C. This approach works well as long as the underlying architecture is not changed. The extensibility of our DSL stack in the case of changing the target architecture (e.g. using a multi-core architecture instead of a single-core one) is discussed in Section 3.8.

3.5 Transformations

In this section, we detail three transformations that were previously introduced, and which are expressed using the proposed DSL stack. First, we present the pipelining transformation used to improve data locality. Second, we present data-structure synthesis, which specializes and materializes data structures to improve query execution performances. Finally, we provide more details about the string dictionaries.

3.5.1 Pipelining – From Fusion to Push Query Engines

A query written using QMonad consists of chained invocations of higher-order functions such as `map`, `filter`, `flatMap`, etc. After naively generating low-level code for such queries (e.g. using template expansion), the generated code typically contains: 1) intermediate list constructions and destructions; and 2) loops corresponding to each higher-order function. Creating intermediate lists causes space and time overheads due to unnecessary allocations.

Loop fusion removes these overheads by removing the need to create intermediate lists. This is because in the fused version of the loops, the elements are pipelined from one operation to the next. Moreover, by merging several loops into a single loop, a single traversal is performed, reducing the iteration overhead.

```
class QueryMonad[T] {
  /* These methods are both consumer and producer */
  def map[S](f: T => S): QueryMonad[S] = build { k =>
    for(e <- this) k(f(e))
  }
  def filter(p: T => Boolean): QueryMonad[T] = build { k =>
    for(e <- this) if(p(e)) k(e)
  }
  def hashJoin[S](list2: QueryMonad[S])
    (leftHash: T => Int)
    (rightHash: S => Int): QueryMonad[(T, S)] = build { k =>
    val hm = new MultiMap[Int, T]()
    for(e <- this)
      hm.addBinding(leftHash(e), e)
    for(e2 <- list2) {
      val key = rightHash(e)
      hm.get(key) match {
        case Some(list1) =>
          for(e1 <- list1)
            if(leftHash(e1) == rightHash(e2))
              k((e1, e2))
        case None => ()
      }
    }
  }
  /* This method is only a consumer */
  def count: Int = {
    var result = 0
    for(e <- this)
      result += 1
    result
  }
}
```

Figure 3.6 – Producer-consumer encoding of QMonad operators. Note that in Scala, `for(e <- R)f(e)` is the same as `R.foreach(e => f(e))`.

The literature contains a very well-defined set of algebraic rewrite rules for Monad Calculus [44]. These rules are sufficient to express loop fusion for the Monad Calculus subset of QMonad. However, fusion is also needed outside of that subset. Since we add constructs to the language, we need to add the corresponding loop fusion rewrite rules. For this, it is important to be aware of relevant research that has been conducted in the PL community. In fact, it turns out that QMonad with n constructs needs $O(n^2)$ loop fusion rewrite rules, which is not scalable [121]. Ideally, one should need only $O(n)$ rewrite rules for a DSL with n constructs. Furthermore, there are cases in which the fusion of two operators in QMonad is not expressible in QMonad itself. Figure 3.2c shows a program that exhibits this property: the fusion of `filter` and `hashJoin` cannot be expressed using a single QMonad operator. This transformation needs to be expressed using operations provided by a lower level DSL. Hence, for these cases loop fusion is no longer an optimization but a lowering transformation (refer to Section 3.2).

Deforestation [357] is a well-known technique in the PL community that is used to remove intermediate data structures. This is the approach we use for performing loop fusion. There are several implementations of this technique, among which *short-cut fusion* (known as *cheap deforestation* or *foldr/build fusion*) [121, 122] has been proven to achieve pipelined

query execution [136]. This approach requires defining every operator in the language using two primitive combinators: 1) a `build` combinator that *produces* a list; and 2) a `foldr` combinator that *consumes* a list. This way, the number of rewrite rules needed for QMonad with n constructs become $O(n)$ (the implementations of the n operators using the `build` and `foldr` combinators). Loop fusion is then achieved by eliminating adjacent occurrences of `build` and `foldr`.

We implement a variant of short-cut fusion [175] in which every operator is expressed using the *church-encoding of lists* [275], or *tranducers* [303] . For simplicity, we use the `foreach` operator and side effects, instead of the pure `foldRight` operator. This transformation is implemented as a lowering step from QMonad to ScaLite[Map, List]. Figure 3.6 shows the implementation of a few QMonad operators using `build` and `foreach`. Inlining this high-level implementation leads to pipelining transformation.

Figure 3.5 shows a simple example in which short-cut fusion is used to apply loop fusion (long path at the bottom). Transformation (T) is what allows us to transition from the code at the bottom-left corner to the code at the bottom-right. This example shows that short-cut fusion has the same impact as the corresponding algebraic rewrite rule from Monad Calculus (short path, on top).

Figure 3.2d shows the code resulting from the example in Figure 3.2c, after the pipelining. The `filter` operation is fused with the first loop of the `hashJoin` operation, which is responsible for creating a hash table based on the join key of the first relation. The `count` operation is fused with the second loop of `hashJoin`, which is responsible for iterating over the second relation and probing relevant partitions from the hash table. Our observations show that short-cut fusion has the same effect as the push-engines proposed in [255]. This is not a surprise, since every operator in the latter is modeled after a producer/consumer pattern, which directly corresponds to the foldr/build model described above. We present the connection between loop fusion techniques and pipelined query engines in more detail in Chapter 4.

After pipelining the query engine and lowering it to ScaLite[Map, List], there are new optimization opportunities to be applied on the resulting mutable data structures, as we saw in Section 3.4.3. Next, we discuss how to synthesize specialized data structures from ScaLite[Map, List] programs.

3.5.2 Specialized Data-Structure Synthesis

A pipelined query uses mutable data structures to perform in-place updates, instead of cloning the list every time a single element is updated. The resulting program is no longer in QMonad (or QPlan), but has been lowered to ScaLite[Map, List], and is thus no longer purely functional. This makes optimizations harder to express than in higher level, purely declarative DSLs, mainly because of the presence of side-effects. To resolve this issue, we enforce several restrictions on this DSL, which facilitate program analysis. For example, by not allowing nested

```
val hm = new MultiMap[Int, R]
for(r <- R) {
  if(r.name == "R1")
    hm.addBinding(r.sid, r)
}
var count = 0
for(s <- S) {
  hm.get(s.rid) match {
    case Some(rList) =>
      for(r <- rList) {
        if(r.sid == s.rid)
          count += 1
      }
    case None => ()
  }
}
count
```

(a) The optimized version of the query in ScaLite[Map, List].

```
/*
The iteration over the first
relation is moved to the
next step.
*/
var count = 0
for(s <- S) {


  for(r <- R) {
    if(r.name == "R1")
      if(r.sid == s.rid)
        count += 1
  }

}
count
```

(b) Naïvely removing the `MultiMap` abstraction in ScaLite[List].

```
/* Precomputation */
val MR: Array[List[R]] =
  // Indexed R on sid

/* Actual Query Processing */
var count = 0
for(s <- S) {

  val rList = MR(s.rid)
  for(r <- rList) {
    if(r.name == "R1")
      if(r.sid == s.rid)
        count += 1
  }

}
count
```

(c) The optimized version of the query in ScaLite[List] when `R.sid` is a foreign key. Note that List is a mutable data structure.

```
/* Precomputation */
val MR: Array[R] =
  // Indexed R on sid

/* Actual Query Processing */
var count = 0
for(s <- S) {


  val r = MR(s.rid)
  if (r.name == "R1")
    if(r.sid == s.rid)
      count += 1

}

count
```

(d) The optimized version of the query in ScaLite[List] when `R.sid` is a primary key.

Figure 3.7 – Representations of an example query after applying pipelining and data-structure synthesis.

mutability, we simplify side-effect analysis. This analysis helps identifying data dependencies among statements, after which we can safely reorder them without changing the semantics of the program.

`MultiMaps`, which are used to implement hash joins, can be specialized depending on the way they are used. For example, under circumstances described below, and if there is a one-to-one relationship between a write operation and its corresponding read, we can remove the `MultiMap` altogether. In Figure 3.7a, we iterate over a relation `R` and add its tuples into a `MultiMap`. We then access the relevant partitions of `R` while iterating over a second relation `S`. Instead of these two steps, we would like to directly access the elements of `R` while iterating

over S. For it to be safe, such a transformation requires reordered statements to be free of data dependencies related to the two iterations, outside of read/write dependencies on the elements of the MultiMap we want to elide.

However, naïvely removing the intermediate MultiMap and substituting its read operation with an iteration over R clearly will not improve performance (it is equivalent to a nested loop join). Figure 3.7b shows the result of that naïve transformation. We can see that for each element of S, instead of iterating over the relevant tuples in R, we iterate over the whole relation. In order to correct this, we can materialize R based on the join key sid. First, we make sure R is not an intermediate relation, but an input relation (otherwise, the transformation is not applicable). Then, at query loading time, we materialize an array of lists which is indexed based on sid. As a result, we can now iterate only over the relevant parts of R, as is shown in Figure 3.7c.

Furthermore, in case sid is a primary key, the materialized data structure can be specialized further: since there will only be one tuple associated with each key (by definition of a primary key), there is no need for buckets anymore: a one dimensional array is sufficient, instead of an array of lists. We remove the corresponding bucket iteration in the main loop. Figure 3.7d shows the resulting code, in case the join key is a primary key.

Data-structure synthesis is not limited to MultiMaps. HashMaps, which are used to implement aggregations, can also be specialized. In that case, we synthesize materialized data structures which partition the HashMaps based on their grouping key, automatically inferring the grouping indices.

3.6 Putting it all together – The DBLAB/LB Query Engine

We have implemented the DSLs of our multi-level stack and the associated transformers in DBLAB,[9] a framework for building efficient database systems via high-level programming. Using the components provided by DBLAB we have re-created the LegoBase [199] query engine. We refer to this re-implementation as DBLAB/LB.

First, developers write their queries in DBLAB/LB using one of the front-end languages of DBLAB/LB (QPlan or QMonad). DBLAB/LB uses the Yin-Yang [177] framework to construct the corresponding IR from these queries. As already discussed in Section 3.4, the front-end languages are progressively lowered and optimized until they reach the abstraction level of the C programming language. Finally, the generated C programs are compiled using any traditional C compiler such as CLang or GCC. At this point, our stack has generated a stand-alone executable for the given query, which includes data loading and data processing and whose execution produces the final query results. The overall DSL code base (without the optimizations, whose lines of code are presented in Section 3.7) is around a thousand lines of Scala code.

[9] http://github.com/epfldata/dblab

DBLAB heavily relies upon the functionality provided by SC ("Systems Compiler") [100], a *generic* DSL compiler framework. SC provides a complete tool-chain for easily defining DSLs and the corresponding transformations as well as a number of *general-purpose* optimizations out-of-the-box. Note that applying one of the optimizations mentioned throughout this thesis does not necessarily lead to performance improvements for a given query. In general, finding the combination of optimizations that leads to optimal performance is a very hard problem; for this reason, the SC DSL compiler does not try to automatically infer the optimal combination of optimizations for each incoming query. Instead, SC was designed so that it provides full control over the compilation process to the DSL developers, while hiding the complicated internal implementation details of the compiler itself. Like DBLAB/LB, SC is also written in Scala, and currently consists of around 27K LoC.

In this work, we extend the library of optimizations provided by SC with several *domain-specific* optimizations which we apply on and across the presented DSLs using the interfaces provided by SC. These domain-specific optimizations can be found in previous query compilation approaches, like those found in the HyPer [255] database, the HIQUE [208] query compiler and the LegoBase [199] query engine. More specifically, the DBLAB/LB DSL stack provides support for: 1) A push-based query engine [255], 2) Operator Inlining [255], 3) Specialization of hash-table data structures [199], 4) Control flow optimizations to improve branch prediction and cache locality [255], 5) String Dictionaries [10] [35], 6) Optimization of memory allocations, by converting *malloc* calls to using memory pools instead and, finally, 7) Standard compiler optimizations like Partial Evaluation, Function Inlining, Scalar Replacement, DCE and CSE [199].

3.7 Experimental Results

Our experimental platform consists of a server-type x86 machine equipped with two Intel Xeon E5-2620 v2 CPUs running at 2GHz each, 256GB of DDR3 RAM at 1600Mhz and two commodity hard disks of 2TB storing the experimental datasets. The operating system is Red Hat Enterprise 6.7. For compiling the generated programs throughout our evaluation we use version 2.9 of the CLang compiler with the default optimization flags. Finally, for C data structures we use the GLib library (version 2.42.1).

For our evaluation we use TPC-H [343], a benchmark suite which simulates data-warehousing and decision support; it provides a set of 22 queries which represent actual business analytics operations to a database with sales information. These queries have a high degree of complexity and express most SQL features.

As a reference point for all results presented in this section, we use the LegoBase query engine [199], an in-memory query execution engine written in the high-level programming language Scala. This is in contrast to the traditional wisdom which calls for the use of low-

[10] All performance numbers reported for LegoBase in [199] were obtained by including string dictionaries.

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
LegoBase	168	108	195	283	2220	100	209	462	447	488	105
DBLAB/LB 2	5936	1510	6176	12162	4904	580	4815	20069	27686	5500	797
DBLAB/LB 3	1298	749	4978	10779	3514	289	2714	20069	27686	3411	339
DBLAB/LB 4	177	58	178	141	159	64	109	56	628	433	59
DBLAB/LB 5	177	58	117	141	158	46	109	20	537	433	59
TPC-H Compliant	1298	611	4099	5628	2194	290	2745	6012	19944	3524	80

System	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
LegoBase	281	604	188	134	1326	75	245	371	495	669	191
DBLAB/LB 2	3763	1846	1832	1633	16962	23765	6329	3075	1629	18636	1320
DBLAB/LB 3	3068	1002	1238	955	14956	18471	3055	3075	938	11595	885
DBLAB/LB 4	311	860	39	97	4285	38	198	64	194	385	91
DBLAB/LB 5	120	591	12	27	723	11	196	19	167	385	91
TPC-H Compliant	2567	855	446	1022	4285	4795	4228	2693	427	12070	900

Table 3.2 – Performance results (in milliseconds) for TPC-H (scaling factor 8) for (a) the generated optimized C code of LegoBase [199], and the generated optimized C code of our system using (b) two-level (c) three-level (d) four-level, (e) five-level DSL stack, and (f) TPC-H compliant DSL stack. (b) the generated C code of DBLAB/LB using an increasing number of levels in our DSL stack and, finally, (c) the TPC-H compliant DSL stack.

level languages for DBMS development. To avoid the overheads of a high-level language (e.g. complicated memory management) while maintaining nicely defined abstractions, LegoBase uses *generative programming* [289, 329] and compiles the Scala code to optimized, low-level C code for each SQL query. By programming databases in a high-level style and still being able to get good performance, the time saved can be spent implementing more database features and optimizations. LegoBase already significantly outperforms both a commercial in-memory database and an existing state-of-the-art query compiler.

We present experimental results which demonstrate that by utilizing our multi-level DSL stack, developers can build a query engine that matches or even significantly outperforms the LegoBase system. This is because, by splitting optimizations across different abstraction layers and separating concerns, it becomes easier to detect performance bottlenecks that are evasive otherwise. Note that some optimizations of the SC DSL stack and LegoBase are not compliant with the TPC-H rules. For this reason, we also present results for a TPC-H compliant set of optimizations[11].

Table 3.2 presents experimental results for all 22 queries of the TPC-H benchmark for both our system (while incrementally introducing DSLs and their corresponding optimizations as presented in Section 3.4) and LegoBase [199]. We observe the following three things.

DBLAB/LB achieves a speedup of up to 23× (Q8), with an average performance improvement

[11]To obtain this TPC-H compliant configuration we have disabled: 1) String Dictionaries 2) Data-Structure Partitioning 3) Automatic Index Inference and, finally, 4) Removing unused table attributes.

of 5×. More specifically, for 20 out of 22 queries, DBLAB/LB significantly outperforms the state-of-the-art LegoBase system. This improvement was achieved by introducing optimizations related to removing intermediate materializations that have a significant negative performance impact at query execution. These materialization points (and other performance bottlenecks) were not easy to be detected in the complicated generated code of the original LegoBase system. In addition, we have introduced automatic inference of database indices, based on database statistics, a technique that provided significant performance improvements and was not expressed as a compiler optimization before. However, we also note that in cases where DBLAB/LB is slower than LegoBase (Q1, Q9), this is because the latter was more aggressively optimizing the data structures, e.g. by partitioning data using a composite set of attributes at data loading time. We plan to investigate such optimization opportunities, which are easily expressible with our DSL stack, in the future.

Second, Table 3.2 also presents a more detailed break-down of how performance improves as DSLs and their corresponding optimizations are introduced in DBLAB/LB. In general we observe that as more DSLs are introduced, performance can be significantly improved. More specifically, when moving from a three to a four-level DSL stack we get an additional 56× performance improvement. This is because the introduction of the additional DSL "unlocks" many optimization opportunities that were not expressible before with a three-level stack. Observe also that the introduction of additional DSLs does not always improve performance for all queries. For example, when moving from a two-level to a three-level DSL stack, performance is marginally improved (if at all) for 10 out of 22 queries. Yet, the introduction of an additional DSL significantly improves the performance of Q1. Thus, we see that it is definitely possible to express even query-specific optimizations with the DBLAB/LB DSL stack. More importantly, Table 3.2 clearly illustrates that performance is never negatively affected by the introduction of an additional DSL level, for all 22 TPC-H queries.

Finally, we observe that the TPC-H compliant set of DBLAB/LB optimizations typically leads to a better performance than that of a three-level DSL stack, however it does not outperform the four-level DSL stack. This is because, at the four-level stack we start introducing optimizations that are no-longer TPC-H compliant.

3.8 Outlook: Parallelism

One possible question regarding the extensibility of our approach would be adding parallelism to the query engine. There are many different variants of parallelization for database systems. Here, we focus only on the *intra-operator* (or *partitioned*) parallelism which can be achieved by (a) partitioning the input data of each operator in the operator tree, (b) applying the sequential operator implementations on each partition and, finally, (c) merging the result obtained on each partition [128]. Next, we show how our DSL stack can be enriched with parallelization through demonstrating the modifications needed for the DSLs and the transformations.

First, we present the required modifications for the DSLs in our stack. The parallelization

logic is encoded in the QPlan DSL by adding the split and merge operators [237]. As these two operators are not expressible by middle level DSLs, the expressibility principle is violated. To solve this issue, the intermediate DSLs require an appropriate set of facilities for parallelism. This is achieved by adding threading facilities (i.e. forking and joining threads) to the ScaLite DSL. As ScaLite[List], ScaLite[Map, List], and C.Scala are extensions to ScaLite, these DSLs are also enriched with parallelism *for free*. The framework generates parallel code by unparsing the parallel C.Scala constructs to the corresponding C code (e.g. by using pthreads).

Second, we analyze what modifications are required for the transformations. If the queries are not using parallel physical query plans at all (e.g. there is no use of split and merge operators), then no change is required for the transformations. However, the generated code for a query with split and merge operators should use an appropriate set of parallelization constructs, as we described above. To do so, we add the implementation of these two operators using the threading constructs. This implementation leads to adjust the pipelining transformation rules for these two operators, as was described in Section 3.5.1. The merge operator needs special care based on the class of the aggregation (e.g. `SUM` is distributive and `AVG` is algebraic [129]). Apart from these two operators, there is no other modification needed for the existing query operators. Also, this modification is introduced only once and it is reused for all parallel physical query plans. Furthermore, as the rest of existing transformations are not modified, the generated code for each thread benefits from the optimizations provided for the sequential version of the DSL stack *for free*.

It has been shown that in some cases using shared data structures (which requires a locking mechanism) is better than using private data structures for each thread (which is the approach we demonstrated here) [64]. Also, there is a possibility of using lock-free data structures and work-stealing for scheduling the workload across the worker threads [216]. Finally, although we use hash partitioning, there are a number of other approaches for partitioning the input data (e.g. range partitioning, etc.): the performance of each scheme is dependent on the underlying workload scenario and data characteristics [237]. We plan to investigate these directions by employing program analysis and compilation techniques in the future.

3.9 Conclusions

In this chapter we argue that it is time for a rethinking of how query compilers are designed and implemented. Current solutions are mainly using template expansion to generate low-level code from high-level queries in one single step. Our experience with systems built using this technique indicates that, in practice, it becomes very difficult to maintain and extend such query compilers over time; the complexity of their code-bases increases to unmanageable levels as more and more optimizations are added.

Our suggested approach advocates modularizing a query compiler by defining several abstraction levels. Each abstraction level is responsible for expressing *only* a subset of optimizations, a design decision which creates a separation of concerns between different optimizations. In

addition, we propose *progressively* lowering the high-level query to low-level code through multiple abstraction levels. This allows for a more controlled code-generation approach.

We show that our approach, while introducing multiple abstraction layers, does not in fact negatively impact the performance of the generated code, but instead improves it. This is because it allows for expressing compiler optimizations that are already available in existing query compilers, but also *new* optimizations not available before. More importantly, it does so while actually providing for a great degree of programmer productivity, a property not found in any previous query compilation approach to date.

4 Loop Fusion in Query Engines

Many badly needed goals, like fusion and cancer cure, would be achieved much sooner if we invested more.
– Stephen Hawking

Database query engines use pull-based or push-based approaches to avoid the materialization of data across query operators. In this chapter, we study these two types of query engines in depth and present the limitations and advantages of each engine. Similarly, the programming languages community has developed loop fusion techniques to remove intermediate collections in the context of collection programming. We draw parallels between DB and PL research by demonstrating the connection between pipelined query engines and loop fusion techniques. Based on this connection, we propose a new type of pull-based engine, inspired by a loop fusion technique, which combines the benefits of both approaches. Then we experimentally evaluate the various engines, in the context of query compilation, for the first time in a fair environment, eliminating the biasing impact of ancillary optimizations that have traditionally only been used with one of the approaches. We show that for realistic analytical workloads, there is no considerable advantage for either form of pipelined query engine, as opposed to what recent research suggests. Also, by using microbenchmarks, which demonstrate certain edge cases on which one approach or the other performs better, we show that our proposed engine dominates the existing engines by combining the benefits of both.

4.1 Introduction

Database query engines successfully leverage the compositionality of relational algebra-style query plan languages. Query plans are compositions of operators that, at least conceptually, can be executed in sequence, one after the other. However, actually evaluating queries in this way leads to grossly suboptimal performance. Computing ("materialising") the result of a first operator before passing it to a second operator can be very expensive, particularly if the intermediate result is large and needs to be pushed down the memory hierarchy. The same observation has been made by the programming languages and compilers community and has

led to work on loop fusion and deforestation (the elimination of data structure construction and destruction for intermediate results).

Already relatively early on in the history of relational database systems, a solution to this problem has been proposed in the form of the Volcano Iterator model [128]. In this model, tuples are *pulled* up through a chain of operators that are linked by iterators that advance in lock-step. Intermediate results between operators are not accumulated, but tuples are produced on demand, by request by conceptually "later" operators.

More recently, an operator chaining model has been proposed that shares the advantage of avoiding materialisation of intermediate results but which reverses the control flow; tuples are *pushed* forward from the source relations to the operator producing the final result. Recent papers [255, 199] seem to suggest that this push-model consistently leads to better query processing performance than the pull model, even though no direct, fair comparisons are provided.

One of the main contributions of this work is to debunk this myth. As we show, if compared fairly, push and pull based engines have very similar performance, with individual strengths and weaknesses, and neither is a clear winner. Push engines have in essence only been considered in the context of query *compilation*, conflating the potential advantages of the push paradigm with those of code inlining. To compare them fairly, one has to decouple these aspects.

In this section, we present an in-depth study of the tradeoffs of the push versus the pull paradigm. Choosing among push and pull – or any reasonable alternative – is a fundamental decision which drives many decisions throughout the architecture of a query engine. More specifically, the interface exposed for implementing different query operators is dependent on the type of the query engine [127]. Furthermore, the query processing engine needs to interact with the storage manager to benefit from different *access methods* (such as hash-based and B+ tree indexes) for efficiently accessing data in the underlying storage [145]. Hence, different design choices of query engines result in different design choices for the storage manager component as well. Thus, one must understand the relevant properties and tradeoffs deeply, and should not bet on one's ability to overcome the disadvantages of a choice by a hack later.

Furthermore, we illustrate how the same challenge and tradeoff has been met and addressed by the PL community, and show a number of results that can be carried over from the lessons learned there. Specifically, we study how the PL community's answer to the problem, *stream fusion* [71], can be adapted to the query processing scenario, and show how it combines the advantages of the pull and push approaches. Furthermore, we demonstrate how we can use ideas from the push approach to solve well-known limitations of stream fusion. As a result, we construct a query engine which combines the benefits of both push and pull approaches. In essence, this engine is a pull-based engine on a coarse level of granularity (i.e., on the level of collections), however, on a finer level of granularity (i.e., on the level of tuples), it pushes the individual data tuples.

In summary, this work makes the following contributions:

- We discuss pipelined query engines in Section 4.2. After presenting loop fusion for collection programming in Section 4.3, we show the connection between these two concepts in Section 4.4. Furthermore, we demonstrate the limitations associated with each approach.
- Based on this connection with loop fusion, we propose a new pipelined query engine in Section 4.5 inspired by the stream fusion [71] technique developed for collection programming in the PL community. Also, we discuss implementation concerns and compiler optimizations required for the proposed pipelined query engine in Section 4.6.
- We experimentally evaluate the various query engine architectures in Section 4.7. Using microbenchmarks, we discuss the weaknesses of the existing engines and how the proposed engine circumvents these weaknesses by combining the benefits of both worlds. Then we demonstrate using TPC-H queries that good implementations of these three query engines do not show a considerable advantage for either form of query engine.

Throughout this chapter, we are using the Scala programming language for all code snippets, interfaces and examples. None of the concepts and ideas require specifically this language – other impure functional object-oriented programming languages (or object-oriented with functional features) such as OCaml, F#, C++11, C#, or Java 8 could be used instead.

4.2 Pipelined Query Engines

Database management systems accept a declarative query (e.g., written in SQL). Such a query is passed to a query optimizer to find a fast physical query plan, which then is either interpreted by the query engine or compiled to low-level code (e.g. C code).

A physical query plan is a data flow graph of query operators which perform calculations and data transformations. Each query operator can be connected to one or more input operators (which we refer to as the *source* operators) and one output operator (which we refer to as the *destination* operator).

A sequence of query operators can be *pipelined*, which means that the output of one operator is *streamed* into the next operator. Pipelining a query operator removes the need for *materializing* the intermediate data and reading it back again, which can bring a significant performance gain.

There are two approaches for pipelining. The first approach is demand-driven pipelining in which an operator repeatedly *pulls* the next data tuple from its source operator. The second approach is data-driven pipelining in which an operator *pushes* each data tuple to

its destination operator. Next, we give more details on the pull-based and push-based query engines.

4.2.1 Pull Engine – a.k.a. the Iterator Pattern

The iterator model is the most widely used pipelining technique in query engines. This model was initially proposed in XRM [225]. However, the popularity of this model is due to its adoption in the Volcano system [128], in which this model was enriched with facilities for parallelization.

In a nutshell, in the iterator model, each operator pipelines the data by requesting the next element from its source operator. This way, instead of waiting until the entire intermediate relation is produced, the data is *lazily* generated in each operator. This is achieved by invoking the `next` method of the source operator by the destination operator. The design of pull-based engines directly corresponds to the iterator design pattern in object-oriented programming [355].

Figure 4.1 shows an example query and the control flow of query processing for this query. Each query operator performs the role of a destination operator and *requests* data from its source operator (the predecessor operator along the flow direction of data). In a pull engine, this is achieved by invoking the `next` function of the source operator, and is shown as control flow edges. In addition, each operator serves as source operator and *generates* result data for its destination operator (the successor operator along the flow direction of data). The generated data is the return value of the `next` function, and is represented by the data flow edges in Figure 4.1. Note the opposing directions of control-flow and data-flow edges for the pull engine in Figure 4.1.

From a different point of view, each operator can be considered as a `while` loop in which the `next` function of the source operator is invoked per iteration. The loop is terminated when the `next` function returns a special value (e.g., a `null` value). In other words, whenever this special value is observed, a `break` statement is executed to terminate the loop execution.

There are two main issues with a pull-based query engine. First, the `next` function invocations are implemented as virtual functions – operators with different implementations of `next` have to be chained together. There are many invocations of these functions; each invocation requires looking up a virtual table, which leads to suboptimal instruction locality. Query compilation solves this performance issue by inlining these virtual function calls, which is explained in Section 4.2.3.

Second, in practice, selection operators are problematic. When the `next` method of a selection operator is invoked, the destination operator has to wait until the selection operator returns the next data tuple satisfying its predicate. This makes the control flow of the query engine more complicated by introducing more loops and branches, which is demonstrated in Figure 4.2c. Push engines solve the problem of complicated control flow graphs.

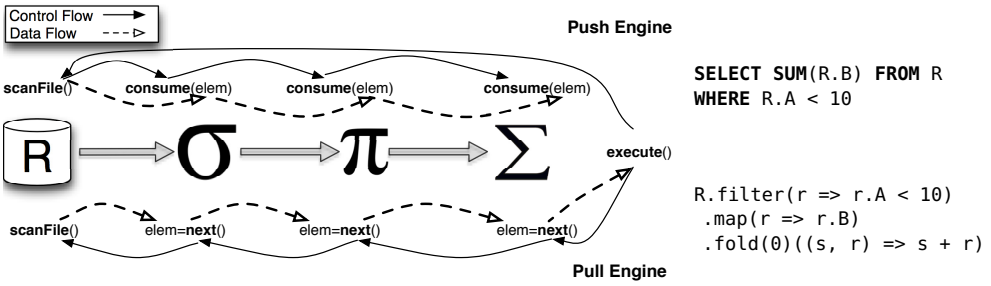


Figure 4.1 – Data flow and control flow for push and pull-based query engine for the provided SQL query.

4.2.2 Push Engine – a.k.a. the Visitor Pattern

Push-based engines are widely used in streaming systems [150]. The Volcano system uses data-driven pipelining (which is a push-based approach) for implementing inter-operator parallelism in query engines. In the context of query compilation, stream processing engines such as StreamBase [335] and Spade [117], as well as HyPer [255] and LegoBase [199, 300] use a push-based query engine approach.

In push-based query engines, the control flow is reversed compared to that of pull-based engines. More concretely, instead of destination operators requesting data from their source operators, data is pushed from the source operators towards the destination operators. This is achieved by the source operator passing the data as an argument to the `consume` method of the destination operator. This results in *eagerly* transferring the data tuple-by-tuple instead of requesting it *lazily* as in pull-engines.

A push engine can be implemented using the *Visitor* design pattern [355] from object-oriented programming. This design pattern allows separating an algorithm from a particular type of data. In the case of query engines, the visitor pattern allows us to separate the query operators (data processing algorithms) from a relation of elements. To do so, each operator should be defined as a visitor class, in which the `consume` method, which is responsible for pushing the elements down the pipeline, has the functionality of the `visit` method, whereas the `produce` method, which is responsible for initializing the chain of operators, has the functionality of the `accept` method of the visitor pattern.

Figure 4.1 shows the query processing workflow for the given example query. Query processing in each operator consists of two main phases. In the first phase, operators prepare themselves for producing their data. This is performed only once in the initialization. In the second phase, they consume the data provided by the source operator and produce data for the destination operator. This is the main processing phase, which consists of invoking the `consume` method of the destination operator and passing the produced data through it. This results in the same direction for both control-flow and data-flow edges, as shown in Figure 4.1.

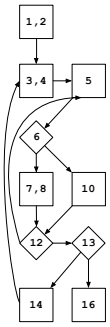
Push engines solve the problem pull engines have with selection operators. A selection

```
1  var sum = 0.0
2  var index = 0
3  while(true) {
4    var rec = null
5    do {
6      if(index < R.length) {
7        rec = R(index)
8        index += 1
9      } else {
10       rec = null
11     }
12   } while(rec != null && !(rec.A < 10))
13   if(rec == null) break
14   else sum += rec.B
15 }
16 return sum
```

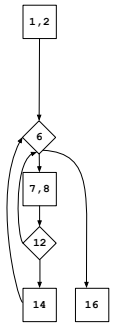
(a) Inlined query in pull engine.

```
var sum = 0.0
var index = 0



while(index < R.length) {
  val rec = R(index)
  index += 1



  if(rec.A < 10)

    sum += rec.B
}
return sum
```

(b) Inlined query in push engine.



(c) The CFG of the inlined query in pull engine.



(d) The CFG of the inlined query in push engine.

Figure 4.2 – Specialized version of the example query in pull and push engines and the corresponding control-flow graphs (CFG).

operator ignores the produced data if it does not satisfy the given predicate by not passing the data to the destination operator. This is in contrast with pull engines in which the destination operator should have waited for the selection operator to serve the request.

However, push engines experience difficulties with *limit* and *merge join* operators. For limit operators, push engines do not allow terminating the iteration by nature. This is because, in push engines, the operators cannot control when the data should no longer be produced by their source operator. This lack of control over when to stop producing more elements causes the production of elements which will never be used.

The merge join operator suffers from a similar problem. There is no way for the merge join operator to guide which one of its two source operators (which are both sorted and there is a 1-to-n relationship between them) should produce the next data item. Hence, it is not possible to pipeline the data coming from both source operators in a merge join. As a result, at least for one of the source operators, the pipeline needs to be broken. Hence, the incoming data coming from one of the source operators can be pipelined (assuming it is correctly sorted, of course), but the input data coming from the other source operator must be materialized.

The mentioned limitation is not specific to operators such as merge joins. A similar situation can arise in the case of more sophisticated analytical tasks where one has to use collection programming APIs (such as Spark RDDs [376]). In collection programming many different methods (for example, element-wise operations on two numeric vectors) are implemented using the `zip` method. The situation is similar for operations which are variants of the merge join operator such as Leapfrog Triejoin [349]. These methods require parallel traversal on two (or more) collections similar to the merge join operator, which cannot be easily pipelined in push-based engines.

Note that these limitations can be resolved by providing special cases for these two operators in the push engine. In the case of limit operator, one can avoid producing unnecessary elements by manually fusing the limit operator with its source operator, which is an ordering operator in most cases. This is because in most cases limit-queries have an order-by clause.[1] For the merge join operator, one can implement a variant of this operator which uses different threads for its source operators and uses synchronization constructs in order to control the production of data by its two inputs, which can be costly. However, such engines can be considered as hybrid engines, and in this chapter, by push engine, we mean a *purely standard* push engine without such augmentations.

4.2.3 Compiled Engines

In general, the runtime cost of a given query is dependent on two factors. The first factor is the time it takes to transfer the data across storage and computing components. The second factor is the time taken for performing the actual computation (i.e., running the instructions of the query). In disk-based DBMSes, the dominating cost is usually the data transfer from/to the secondary storage. Hence, as long as the pipelining algorithm does not break the pipeline, there is no difference between pull engines and push engines. As a result, the practical problem with selections in pull engines (c.f. Section 4.2.1) is obscured by data transfer costs.

With the advent of in-memory DBMSes, the code layout of the instructions becomes an ever more important factor. More specifically, the virtual function calls (or function calls via function to pointers) appearing in the iterator model start becoming a bottleneck in the performance. One solution is block-oriented processing of elements instead of processing elements one-by-one, which hides the cost of virtual calls behind the cost of processing a large number of elements [263, 42]. The alternative approach, which is the main focus of this work, is query compilation. This approach uses code generation and compilation techniques in order to inline virtual functions and further specialize the code to improve cache locality [134, 10, 202, 208, 255, 204, 205, 199, 352, 74, 252, 185, 19, 301, 184]. As a result of that, the code pattern used in each pipelining algorithm really matters. Hence, it is important to investigate the performance of each pipelining algorithm for different workloads.

[1] Even without manually fusing the ordering and limit operators, the cost of sorting dominates the cost of final scan. This reduces the impact of pipelining the limit operator in realistic workloads, as we observe in Section 4.7.2.

Figure 4.2a shows the inlined pull-engine code for the example SQL query given in Figure 4.1. Note that for the selection operator, we need an additional `while` loop. This additional loop creates more branches in the generated code, which makes the control-flow graph (CFG) more complicated. Figure 4.2c demonstrates the CFG of the inlined pull-engine code. Each rectangle in this figure corresponds to a block of statements, whereas diamonds represent conditionals. The edges between these nodes represent the execution flow. The backward edges represent the jumps inside a loop. This complicated CFG makes the code harder to understand and optimize for the optimizing compiler. As a result, during the runtime execution, performance degrades.

Figure 4.2b shows the specialized query for a push engine of the previous example SQL query. The selection operator here is summarized in a single `if` statement. As a result, the CFG for the inlined push-engine code is simpler than the one for the pull engine, as shown in Figure 4.2d. This simpler CFG results in fewer branching machine instructions generated by the underlying optimizing compiler, leading to better run time performance.

Up to now, there is no separation of the concept of pipelining from the associated specializations. For example, HyPer [255] is in essence a push engine which uses compiler optimizations by default, without identifying the individual contributions to performance by these two factors. As another example, LegoBase [199] assumes that a push engine is followed by operator inlining, whereas the pull engine does not use operator inlining [200]. On the other hand, there is no comparison between an inlined pull engine – we suspect Hekaton [92] to be of that class – with a push-based inlined engine in the same environment. Hence, there is no comparison between pull and push engines which is under completely fair experimental conditions, sharing environment and code base to the maximum degree possible. In Section 4.7, we attempt such a fair comparison.

Furthermore, naïvely compiling the pull engine does not lead to good performance. This is because a naïve implementation of the iterator model does not take into account the number of `next` function calls. This can lead to inefficient code, due to the code explosion resulting from inlining too many `next` calls. For example, the naïve implementation of the selection operator invokes the `next` method of its source operator twice, as it is demonstrated below:

```
1 class SelectOp[R] (p: R => Boolean) {
2   def next(): R = {
3     var elem: R = source.next()
4     while(elem != null && !p(elem)) {
5       elem = source.next()
6     }
7     elem
8   }
9 }
```

The first invocation is happening before the loop for the initialization (line 3), and the second invocation is inside the loop (line 5). Inlining can cause an explosion of the code size, which can lead to worse instruction cache behavior. Hence, it is important to take into account these concerns while implementing query engines. For example, our implementation of the

selection operator in a pull-based query engine invokes the `next` method of its source operator only once by changing the shape of the `while` loop (c.f. Figure 4.4a). Section 4.7.2 shows the impact of this *inline-friendly* implementation of pull engines.

4.3 Loop Fusion in Collection Programming

Collection programming APIs are getting more and more popular. Ferry [135, 134] and LINQ [238] use such an API to seemlessly integrate applications with database back-ends. Spark *RDDs* [376] use the same operations as collection programming APIs. Also, functional collection programming abstractions exist in mainstream programming languages such as Scala, Haskell, and recently Java 8. The theoretical foundation of such APIs is based on Monad Calculus and Monoid Comprehensions [44, 45, 358, 136, 345, 107, 368, 49].

Similar to query engines, the declarative nature of collection programming comes with a price. Each collection operation performs a computation on a collection and produces a transformed collection. A chain of these invocations results in creating unnecessary intermediate collections.

Loop fusion or Deforestation [357] removes the intermediate collections in collection programs. This is a nonlocal and brittle transformation which is difficult to apply to impure functional programs (i.e., in languages which include imperative features) and is thus absent from mainstream compilers for such languages. In order to provide a practical implementation, one can restrict the language to a pure functional DSL for which the fusion rules can be applied locally. Here, intermediate collections are removed using local transformations instead of global transformations. This approach is known as *short-cut* deforestation. It is more realistic to integrate this approach into real compilers; short-cut deforestation has been successfully implemented in the context of Haskell [325, 71, 121] and Scala-based DSLs [175, 301].

Next, we present two approaches for short-cut deforestation, *fold fusion* and *unfold fusion*, in the order they were discovered. They employ two kinds of "collection" micro-instructions each, to which a large number of collection operations can be mapped. This allows to implement fusion using very few rewrite rules.

4.3.1 Fold Fusion

In this approach, every collection operation is implemented using two constructs: 1) the `build` method for *producing* a collection, and 2) the `foldr` method for *consuming* a collection. Some collection-transforming methods such as `map` use both of these constructs for consuming the given collection and producing a new collection. However, some methods such as `sum`, which produces an aggregated result from a collection, require only the `foldr` method for consuming the given collection.

We consider an imperative variant of this algorithm, in which the `foldr` method is substituted

by `foreach`. The main difference is that the `foldr` method explicitly handles the state, whereas in the case of `foreach`, the state is handled internally and is not exposed to the interface.

Using Scala syntax, the signature of the `foreach` method on lists is as follows:

```
class List[T] {
  def foreach(f: T => Unit): Unit
}
```

The `foreach` method consumes a collection by iterating over the elements of that collection and applying the given function to each element. The `build` function is the corresponding producer for the `foreach` method. This function produces a collection for which the `foreach` method applies the higher-order function `consumer` to the function `f`. The signature of the `build` function is as follows:

```
def build[T](consumer: (T => Unit) => Unit): List[T]
```

We illustrate the meanings of these two methods by an example. Consider the `map` method of a collection, which transforms a collection by applying a given function to each element. This method is expressed in the following way using the `build` and `foreach` functions:

```
class List[T] {
  def map[S](f: T => S): List[S] = build { k =>
    this.foreach(e => k(f(e)))
  }
}
```

The implementation of several other collection operators using these two methods is given in Figure 4.3b.

After rewriting the collection operations using the `build` and `foreach` constructs, a pipeline of collection operators involves constructing intermediate collections. These intermediate collections can be removed using the following rewrite rule:

Fold-Fusion Rule:

```
build(f1).foreach(f2)  ⇝  f1(f2)
```

For example, there is a loop fusion rule for the `map` function, which fuses two consecutive `map` operations into one. More concretely, the expression `list.map(f).map(g)` is converted into `list.map(f o g)`. Figure 3.5 demonstrates how the fold-fusion technique can derive this conversion by expressing the `map` operator in terms of `foreach` and `build`, following by application of the fold-fusion rule.

One of the key advantages of this approach is that instead of writing fusion rewrite rules for every combination of collection operations, it is sufficient to only express these operations in terms of the `build` and `foreach` methods. This way, instead of writing $O(n^2)$ rewrite rules for n collection operations, it is sufficient to express these operations in terms of `build` and `foreach`, which is only $O(n)$ rewrite rules. Hence, this approach greatly simplifies the maintenance of the underlying compiler transformations [301].

This approach successfully deforests most collection operators very well. However, it is not successful in the case of `zip` and `take` operations. The `zip` method involves iterating over two collections, which cannot be expressed using the `foreach` construct which iterates only over one collection. Hence, this approach can deforest only one of the collections. For the other one, an intermediate collection must be created. Also, for the `take` method, there is no way to stop the iteration of the `foreach` method halfway to finish. Hence, the fold fusion technique does not perform well in these two cases. The next fusion technique solves the problem with these two methods.

4.3.2 Unfold Fusion

This is considered a dual approach to fold fusion. Every collection operation is expressed in terms of the two constructs `unfold` and `destroy`. We use an imperative version of unfold fusion here, which uses the `generate` function instead of `unfold`. The prototype of `generate` and `destroy` are as follows:

```
class List[T] {
  def destroy[S](f: (() => T) => S): S
}
def generate[T](gen: () => T): List[T]
```

The `destroy` method consumes the given list. Each element of this collection is accessible by invoking the `next` function available by the `destroy` method. The `generate` function generates a collection whose elements are specified by the input function passed to this method. In the case of `map` operator, the elements of the result collection are the images of the elements of the input collection under the function `f`.

The `map` method of collections is expressed in the following way using the `generate` and `destroy` methods:

```
class List[T] {
  def map[S](f: T => S): List[S] = this.destroy { n =>
    generate { () =>
      val elem = n()
      if(elem == null) null
      else f(elem)
    }
  }
}
```

The implementation of some other collection operators using these two methods is given in Figure 4.4b.

In order to remove the intermediate collections, the chain of intermediate `generate` and `destroy` can be removed. This fact is shown in the following transformation rule:

Unfold-Fusion Rule:

`generate(f1).destroy(f2)` $\rightsquigarrow$ `f2(f1)`

Operator Category	Query Operator	Collection Operator
Producer	Scan	`fromArray`
Transformer	Selection	`filter`
	Projection	`map`
	OrderBy	`sortBy`
	Limit	`take`
	Join*	`flatMap`*
	Merge Join†	`zip`†
Consumer	Agg‡	`fold`‡

* Nested loop join can be expressed using two nested flatMaps, but there is no equivalent for hash-based joins. Also, flatMaps can express nested collections, whereas in relational query engines every relation is considered to be flat.
† Both merge join and `zip` perform parallel traversal on two collections, even though they are otherwise quite different.
‡ An Agg operator representing a GROUP BY is a transformer, whereas the one which folds into only a single result is a consumer.

Table 4.1 – Correspondence between query operators and collection operators

Pipelined Query Engines	Object-Oriented Design Pattern	Collection Loop Fusion
Pull Engine	Iterator	Unfold fusion [325] Stream fusion [71]
Push Engine	Visitor	Fold fusion [121]

Table 4.2 – Correspondence among pipelined query engines, object-oriented design patterns, and collection programming loop fusion.

Figure 3.5 demonstrates how this rule fuses the previous example, `list.map(f).map(g)` into `list.map(f o g)`. Note that the null checking statements, which are for checking the end of a list, are removed for brevity.

This approach introduces a recursive iteration for the `filter` operation. In practice, such a recursive iteration, which is for finding the next satisfying element, can cause performance issues, even though the deforestation is applied successfully [149]. Also, this approach does not fuse operations on nested collections, which is beyond the scope of this thesis.

4.4 Loop Fusion is Operator Pipelining

By chaining query operators, one can express a given (say, SQL) query. Similarly, a given collection program can be expressed using a pipeline of collection operators. The relationship between relational queries and collection programs has been well studied. In particular, one can establish a precise correspondence between relational query plans and a class of collection programs [266].

```
class ScanOp[R](arr: Array[R]) {
  def init(): Unit = {
    var i = 0
    while(i < arr.length) {
      dest.consume(arr(i))
} } }
class ProjectOp[R, P](f: R => P) {
  def consume(e: R): Unit =
     dest.consume(f(e))
}
class SelectOp[R](p: R => Boolean) {
  def consume(e: R): Unit =
    if(p(e))
       dest.consume(e)
}
class AggOp[R, S](f: (R, S) => S) {
  var result = zero[S]
  def consume(e: R): Unit = {
     result = f(e, result)
  }
  def getResult: S = result
}
class HashJoinOp[R, R2]
  (leftHash: R => Int)
  (rightHash: R2 => Int)
  (cond: (R, R2) => Boolean) {
  val hm = new MultiMap[Int, R]()
  def consumeLeft(e: R): Unit = {
     hm.addBinding(leftHash(e) -> e)
  }
  def consumeRight(e: R2): Unit = {
    hm.get(rightHash(e)) match {
      case Some(list) =>
        for(l <- list) {
          if(cond(l, e)) {
             dest.consume(l.concat(e))
          }
        }
      case None =>
} } }
```

(a) Push-based query engine

```
class QueryMonad[R] {
  def fromArray[R](arr: Array[R]) =
    build { k =>
      var i = 0
      while(i < arr.length) {
        k(arr(i))
    } }
  def map[S](f: R => S) = build { k =>
    for(e <- this)
       k(f(e))
  }
  def filter(p: R => Boolean) = build { k =>
    for(e <- this)
      if(p(e))
        k(e)
  }
  def fold[S](zero: S)(f: (R, S) => S): S = {
    var result = zero
    for(e <- this) {
      result = f(e, result)
    }
    result
  }
  def hashJoin[R2](rightList: QueryMonad[R2])
    (leftHash: R => Int)
    (rightHash: R2 => Int)
    (cond: (R, R2) => Boolean) = build { k =>
    val hm = new MultiMap[Int, R1]()
    for(e <- this) {
       hm.addBinding(leftHash(e) -> e)
    }
    for(e <- rightList) {
      hm.get(rightHash(e)) match {
        case Some(list) =>
          for(l <- list) {
            if(cond(l, e)) {
               k(l.concat(e))
            }
          }
        case None =>
} } } }
```

(b) Fold fusion of collections.

Figure 4.3 – Correspondence between push-based query engines and fold fusion of collections.

Operators can be divided into three categories: 1) The operators responsible for *producing* a collection from a given source (e.g., a file or an array), 2) the operators which *transform* the given collection to another collection, and 3) the *consumer* operators which aggregate the given collection into a single result.

The mapping between query operators and collection operators is summarized in Table 4.1. Most join operators do not have a directly corresponding collection operator, with two exceptions: Nested loop joins can be expressed using nested `flatMap`s and the `zip` collection operator is very similar to the merge join query operator. Both operators need to traverse two input sequences in parallel. For the rest of join operators, we extend collection programming with join operators (e.g. `hashJoin`, `semiHashJoin`, etc.). A similar mapping between the LINQ [238] operators and Haskell lists is shown in Steno [251]. Note that we do not consider nested

```
class ScanOp[R](arr: Array[R]) {
  var i = 0
  def next(): R = {
    if(i < arr.length) {
      val elem = arr(i)
      i += 1
      elem
    } else
      null
} }
class SelectOp[R](p: R => Boolean) {
  def next(): R = {
    var elem: R = null
    do {
       elem = source.next()
    } while (elem != null && !p(elem))
    elem
} }
class ProjectOp[R, P](f: R => P) {
  def next(): P = {
    val elem = source.next()
    if(elem == null) null
    else f(elem)
} }
class AggOp[R, S]
  (f: (R, S) => S) {
  def next(): S = {
    var result = zero[S]
    while(true){
      val elem = source.next()
      if(elem = null)
        break
      else
        result = f(elem, result)
    }
    result
} }
class LimitOp[R](n: Int) {
  var count = 0
  def next(): R = {
    if(count < n) {
      count += 1
      source.next()
    } else {
      null
} } }
```

(a) Pull-based query engine

```
class QueryMonad[R] {
  def fromArray[R](arr: Array[R]) = {
    var i = 0
    generate { () =>
      if(i < arr.length) {
        val elem = arr(i)
        i += 1
        elem
      } else
        null
  } }
  def filter(p: R=>Boolean) = destroy { n =>
    generate { () =>
      var elem: R = null
      do {
        elem = n()
      } while(elem != null && !p(elem))
      elem
  } }
  def map[P](p: R => P) = destroy { n =>
    generate { () =>
      val elem = n()
      if(elem == null) null
      else f(elem)
  } }
  def fold[S](zero: S)
    (f: (R, S) => S): S =
    destroy { n =>
      var result = zero
      while(true){
        val elem = n()
        if(elem = null)
          break
        else
          result = f(elem, result)
      }
      result
    }
  def take(n: Int) = {
    var count = 0
    destroy { n =>
      if(count < limit) {
        count += 1
        n()
      } else {
        null
} } } }
```

(b) Unfold fusion of collections.

Figure 4.4 – Correspondence between pull-based query engines and unfold fusion of collections.

collections here, although straightforward to support in collection programming, in order to emphasize similarity with relational query engines.

Pipelining in query engines is analogous to loop fusion in collection programming. Both concepts remove the intermediate relations and collections, which break the stream pipeline. Also, pipelining in query engines matches well-known design patterns in object-oriented programming [355]. The correspondence among pipelining in query engines, design patterns

in object-oriented languages, and loop fusion in collection programming is summarized in Table 4.2.

Push Engine = Fold Fusion. There is a similarity between the Visitor pattern and fold fusion. On one hand it has been proven that the Visitor design pattern corresponds to the Church-encoding [40] of data types [47]. On the other hand, the `foldr` function on a list corresponds to the Church-encoding of lists in λ-calculus [275, 303]. Hence, both approaches eliminate intermediate results by converting the underlying data structure into its Church-encoding. In the former case, specialization consists of inlining, which results in removing (virtual) function calls. In the latter case, the fold-fusion rule and β-reduction are performed to remove the materialization points and inline the λ expressions. The correspondence between these two approaches is shown in Figure 4.3 (compare (a) vs. (b)). The invocations of the `consume` method of the destination operators in the push engine correspond to the invocations of the `consume` function, which is passed to the `build` operator, in fold fusion.

Pull Engine = Unfold Fusion. In a similar sense, the Iterator pattern is similar to unfold fusion. Although the category-theoretic essence of the iterator model was studied before [119], there is no literature on the direct correspondence between the `unfold` function and the Iterator pattern. However, Figure 4.4 shows how a pull engine is similar to unfold fusion (compare Figure 4.4 (a) vs. (b)), to the best of our knowledge for the first time. Note the correspondence between the invocation of the `next` function of the source operator in pull engines and the invocation of the `next` function which is passed to the `destroy` operator in unfold fusion, which is highlighted in the figure.

4.5 An Improved Pull-Based Engine

In this section, we first present yet another loop-fusion technique for collection programs. Then, we suggest a new pull-based query engine inspired by this fusion technique based on the correspondence between queries and collection programming.

4.5.1 Stream Fusion

In functional languages, loops are expressed as recursive functions. Reasoning about recursive functions is very hard for optimizing compilers. Stream fusion tries to solve this issue by converting all recursive collection operations to non-recursive stream operations. To do so, first all collections are converted to streams using the `stream` method. Then, the corresponding method on the stream is invoked which results in a transformed stream. Finally, the transformed stream is converted back to a collection by invoking the `unstream` method.

The signature of the `unstream` and `stream` methods is as follows:

```
def unstream[T](gen: () => Step[T]): List[T]
class List[T] {
  def stream(): Step[T]
```

```
trait Step[T] {
  def filter(p: T => Boolean): Step[T]
  def map[S](f: T => S): Step[S]
  def fold[S](yld: T => S, skip: () => S, done: () => S): S
}

case class Yield[T](e: T) extends Step[T] {
  def filter(p: T => Boolean) =
    if(p(e)) Yield(e) else Skip
  def map[S](f: T => S) =
    Yield(f(e))
  def fold[S](yld: T => S, skip: () => S, done: () => S): S =
    yld(e)
}

case object Skip extends Step[Nothing] {
  def filter(p: Nothing => Boolean) =
    Skip
  def map[S](f: Nothing => S) =
    Skip
  def fold[S](yld: Nothing => S, skip: () => S, done: () => S): S =
    skip()
}

case object Done extends Step[Nothing] {
  def filter(p: Nothing => Boolean) =
    Done
  def map[S](f: Nothing => S) =
    Done
  def fold[S](yld: Nothing => S, skip: () => S, done: () => S): S =
    done()
}
```

Figure 4.5 – The operations of the Step data type.

```
}
```

For example, the `map` method is expressed in using these two methods as:

```
class List[T] {
  def map[S](f: T => S): List[S] = unstream { () =>
    this.stream().map(f)
  }
}
```

The `stream` method converts the input collection to an intermediate stream, which is specified by the `Step` data type. The function `f` is applied to this intermediate stream using the `map` function of the `Step` data type. Afterwards, the result stream is converted back to a collection by the `unstream` method.

As discussed before, one of the main advantages of the intermediate stream, the `Step` data structure, is that its operations are mainly non-recursive. This simplifies the task of the optimizing compiler to further specialize the program. The implementation of several methods of the `Step` data structure is given in Figure 4.5.

Such transformations do not result in direct performance gain – they may even degrade performance. This is because of the intermediate conversions between streams and collections. However, these intermediate conversions can be removed using the following rewrite rule:

```
class ScanOp[R](arr: Array[R]) {
  var i = 0
  def stream(): Step[R] = {
    if(i < arr.length)
      Yield(arr(i))
    else
      Done
} }
class SelectOp[R](p: R => Boolean) {
  def stream(): Step[R] = {
    source.stream().filter(p)
} }
class ProjectOp[R, P](f: R => P) {
  def stream(): Step[P] = {
    source.stream().map(f)
} }
class AggOp[R, S](f: (R, S) => S) {
  def stream(): Step[S] = {
    var result = zero[S]
    var done = false
    while(!done){
       source.stream().fold(
        e => { result = f(e, result) },
        () => ,
        () => { done = true }
      )
    }
    result
} }
class LimitOp[R](n: Int) {
  var count = 0
  def stream(): Step[R] = {
    if(count < n) {
      source.stream().map(e => {
        count += 1
        e
      })
    } else {
      Done
} } }
```

(a) Stream-fusion query Engine

```
class QueryMonad[R] {
  def fromArray[R](arr: Array[R]) = {
    var i = 0
    unstream { () =>
      if(i < arr.length)
        Yield(arr(i))
      else
        Done
  } }
  def filter(p: R => Boolean) = {
    unstream { () =>
      stream().filter(p)
  } }
  def map[P](f: R => P) = {
    unstream { () =>
      stream().map(f)
  } }
  def fold[S](z: S)(f: (R, S) => S): S = {
    unstream { () =>
      var result = zero[S]
      var done = false
      while(!done){
        stream().fold(
          e => { result = f(e, result) },
          () => ,
          () => { done = true }
        )
      }
      result
  } }
  def take(n: Int) = {
    var count = 0
    unstream { () =>
      if(count < n) {
        stream().map(e => {
          count += 1
          e
        })
      } else {
        Done
} } } }
```

(b) Stream fusion of collections.

Figure 4.6 – Correspondence between stream-fusion query engine and the stream fusion technique.

Stream-Fusion Rule:

```
unstream(() => e).stream()  ⇝  e
```

Figure 3.5 demonstrates how the stream fusion technique transforms `list.map(f).map(g)` into `list.map(f o g)`. Note that for the `Step` data type, the `step.map(f).map(g)` expression is equivalent to `step.map(f o g)`.

The idea behind stream fusion is very similar to unfold fusion. The main difference is the `filter` operator. Stream fusion uses a specific value, called `Skip`, to implement the `filter` operator. This is in contrast with the unfold fusion approach for which the `filter` operator is implemented using an additional nested `while` loop for skipping the unnecessary elements. Hence, stream

fusion solves the practical problem of unfold fusion associated with the `filter` operator.

Next, we define a new pipelined query engine based on the ideas of stream fusion.

4.5.2 Stream-Fusion Engine

The proposed query engine follows the same design as the iterator model. Hence, this engine is also a pull engine. However, instead of invoking the `next` method, this engine invokes the `stream` method, which returns a wrapper object of type `Step`. We refer to our proposed engine as the *stream-fusion engine*.

As we mentioned in Section 4.2.1, one of the main practical problems with a pull engine is the case of the selection operator. In this case, an operator waits until the selection operator returns the next satisfying element. The proposed engine solves this issue by using the `Skip` object which specifies that the current element should be ignored. Hence, selection operators are no longer a blocker for their destination operator.

The correspondence between the stream fusion algorithm and the stream-fusion engine is shown in Figure 4.6. Every query operator provides an appropriate implementation for the `stream` method, which invokes the `stream` method of the source operator to request the next element. Similarly, stream fusion uses the `stream` method to fetch the next element. Then, by invoking the `unstream` method, the generated stream is converted back to a collection.

From a different point of view, a push engine can be expressed using a `while` loop and a construct for skipping to the next iteration (e.g. `continue`). By nature, it is impossible for a push-based engine to finish the iteration before the producer's `while` loop finishes its job. In other words, the generated C code using a push-based engine never uses the `break` construct.

In contrast, a pull engine is generally expressible using a `while` loop and a construct for terminating the execution of the `while` loop (i.e., the `break` construct). This is because of the demand-driven nature of pull engines. However, in a pull-based engine there is no way to skip an iteration. As a result, skipping an iteration should be expressed using a nested `while` loop which results in performance issues (c.f. Section 4.2).

The stream-fusion engine combines the benefits of both engines by providing the following two constructs for early termination of loops and skipping an iteration. First, the `Done` construct denotes the termination of loops, and in essence has the same effect as the `break` construct in an imperative language like C. Second, the `Skip` construct results in skipping to the next iteration, and has an equivalent effect to the `continue` construct in an imperative language like C. Table 4.3 summarizes the differences among the aforementioned query engines.

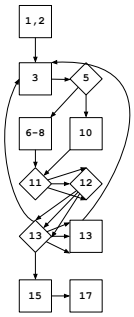
Consider a relation of two elements for which we select its first element and the second element is filtered out. The first call to the `stream` method of the selection operator in the stream-fusion engine produces a `Yield` element, which contains the first element of the relation. The second

```
1  var index = 0
2  var sum = 0.0
3  while(true) {
4    val step1 =
5      if(index < R.length) {
6        val rec = R(index)
7        index += 1
8        Yield(rec)
9      } else
10       Done
11   step1.filter(x => x.A < 10)
12   .map(x => x.B)
13   .fold(x => sum += x,
14         () => ,
15         () => break)
16 }
17 return sum
```

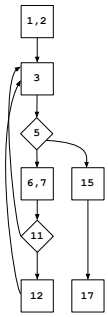
(a) Inlined query in stream-fusion engine without further specializations.

```
var index = 0
var sum = 0.0
while(true) {

  if(index < R.length) {
    val rec = R(index)
    index += 1



    if(rec.A < 10)
      sum += rec.B
  }
  else
    break
}
return sum
```

(b) Inlined query in stream-fusion engine by inlining the visitor model of `Step`.



(c) The CFG of the inlined query in stream-fusion engine without further specializations.



(d) The CFG of the inlined query in stream-fusion engine by inlining the visitor model of `Step`.

Figure 4.7 – Specialized version of the example query in stream-fusion engine and the corresponding control-flow graph (CFG).

invocation of the same method returns a `Skip` element, specifying that this element, which is the second element of the relation, is filtered out and should be ignored. The next invocation of this method results in a `Done` element, denoting that there is no more element to be produced by the selection operator. The `Done` value has the same role as the `null` value in the pull engine.

The specialized version of the example query (which was introduced in Figure 4.1) based on the stream-fusion engine is shown in Figure 4.7a. The code is as compact as the push engine code. However, the control flow graph is similar to (or even more complicated than) the one of a pull engine (c.f. Figure 4.7c). Furthermore, the specialized stream-fusion engine suffers from more performance problems due to the intermediate `Step` objects created. The next section discusses implementation aspects and the optimizations needed for tuning the performance of the stream-fusion engine.

Pipelined Query Engines	Looping Constructs
Push Engine	`while` + `continue`
Pull Engine	`while` + `break`
Stream-Fusion Engine	`while` + `break` + `continue`

Table 4.3 – The supported looping constructs by each pipelined query engine.

4.6 Implementation

In this section, we discuss the implementation of the presented query engines. First, we show the architecture of our query compiler. Then, we discuss how the fusion rules are implemented for each approach. Finally, we show how the problem associated with intermediate objects is resolved for the stream-fusion engine.

4.6.1 Architecture

We have implemented different types of query engines and the associated optimizations in the DBLAB/LB query compiler [301]. This query compiler is a component of DBLAB,[2] a framework for building efficient database systems in the high-level programming language Scala.

The DBLAB/LB query compiler uses the compilation facilities provided by Systems Compiler (SC)[3] in order to implement several intermediate languages (through *language embedding* [158]), the transformations inside and across these languages (using the transformation framework), and finally unparsing the transformed program into Scala or C code (using the pretty printers). Furthermore, SC provides several generic optimizations out-of-the-box, which DBLAB/LB uses during query compilation. These optimizations include Common-Subexpression Elimination (CSE), Dead-Code Elimination (DCE), Partial Evaluation, and Scalar Replacement.

The architecture of DBLAB/LB is shown in Figure 4.8. The input programs can either be expressed using physical (relational algebra-style) query plans in the QPlan language or collection programming using the QMonad language. Depending on the input language, the query compiler performs either pipelining or loop fusion. These transformations result in a low-level Scala program, which does not have the high-level constructs of the QPlan and QMonad languages.[4] In order to transform this Scala program into a C program, the storage layout for records should be specified. DBLAB/LB provides both row and columnar storage layouts for relations [301]. Finally, the DBLAB/LB query compiler uses the C pretty printer provided by SC to generate C code.

[2] http://github.com/epfldata/dblab

[3] http://github.com/epfldata/sc-public

[4] Note that DBLAB/LB defines more intermediate languages in its compilation stack [301] which can be found in Figure 3.3.

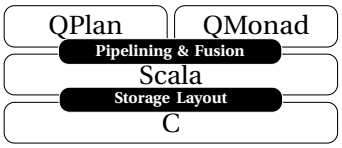


Figure 4.8 – The architecture of the DBLAB/LB query compiler.

We have implemented the collection programming operations and the corresponding loop fusion techniques described earlier in this thesis. Thanks to the equivalence which was shown in Section 4.4 between query engines and collection programming, it is clear how different pipelining techniques can be implemented for query engines. As a result, it is not surprising that the experimental results presented in the next section for different fusion techniques match the results for the corresponding pipelined query engines. Next, we discuss how the fusion rules for different loop fusion algorithms can be expressed in this framework.

4.6.2 Fusion By Inlining

As mentioned in Section 4.4, a fusion rule is expressed as a local transformation rule which is applied as an extension to the host language compiler (which is GHC [174] in the case of the mentioned papers). In this section, we show how these fusion rules are implemented by only using inlining. This was proposed for implementing fold fusion in Scala [175]. Here, we use a similar approach for other fusion techniques.

Figure 4.9a shows the definition of the `build` operator. By inlining the definition of this operator, an object of type `QueryMonad` is created. The `foreach` method of this object applies the higher-order function passed to the `build` method (`f1`) to the input parameter of the `foreach` method (`f2`). By inlining this `foreach` method, we derive the same rule as the fold-fusion rule which was introduced in Section 4.3. This derivation is shown in Figure 4.9a. The constructs and derivation of unfold fusion are shown in Figure 4.9c and Figure 4.9d. Stream fusion follows a similar pattern which is given in Figure 4.10 and Figure 4.11. Figures E.1 and E.2 show the fusion process for the working example using fold and unfold fusion respectively.

Next, we discuss the problematic creation of intermediate objects by the stream-fusion engine, as well as our solution.

4.6.3 Removing Intermediate Results

Although the stream-fusion engine removes intermediate relations, it creates intermediate `Step` objects. There are two problems with these intermediate objects. First, the `Step` data type operations are virtual calls. This causes poor cache locality and degrades performance. Second, normally these intermediate objects lead to heap allocations. This causes higher memory consumption and worse running times. This is why the original stream fusion approach is dependent on optimizations provided by its source language compiler (i.e., the GHC [174]

```
type Cont[T] = (T => Unit) => Unit

class Consumer[T](cont: Cont[T])
  extends QueryMonad[T] {
  def foreach(f: T => Unit): Unit =
    cont(f)
}

def build[T](cont: Cont[T])
  : QueryMonad[T] =
  new Consumer[T](cont)
```

(a) The constructs for fold fusion.

```
build(f1).foreach(f2)

  | (inline build definition)

new Consumer(f1).foreach(f2)

  | (inline foreach definition)

f1(f2)
```

(b) The derivation of the fold-fusion rule.

```
type Gen[T] = () => T

class DestroyGen[T](gen: Gen[T])
  extends QueryMonad[T] {
  def destroy[S](f: Gen[T] => S): S =
    f(gen)
}

def generate[T](gen: Gen[T])
  : QueryMonad[T] =
  new DestroyGen[T](gen)
```

(c) The constructs for unfold fusion.

```
generate(f1).destroy(f2)

  | (inline generate definition)

new DestroyGen(f1).destroy(f2)

  | (inline destroy definition)

f2(f1)
```

(d) The derivation of the unfold-fusion rule.

Figure 4.9 – Constructs and derivation of fold fusion and unfold fusion.

compiler). Implementing an effective version of it for other languages requires supporting similar optimizations supported by the GHC compiler.

The first problem with virtual calls can be solved by rewriting the `Step` operations by enumerating all cases for the `Step` object. This is possible because there are only three possible concrete cases (1. `Yield` 2. `Skip` 3. `Done`) for this data type. To do so, one can use if-statements. In functional languages, the pattern matching feature can be used. Although this approach solves the first problem, still there are heap allocations which are not removed.

The good news is that these heap allocations can be converted to stack allocations. This is because the created objects are not escaping their usage scope. For example, these objects are not copied into an array and not used as an argument to a function. This fact can be verified by the well-known compilation technique of *escape analysis* [63]. Based on that, the heap allocations can be converted to stack allocations.

The compiler optimizations can go further and remove the stack allocations as well. Instead of the stack allocation for creating a `Step` object, the fields necessary to encode this type are converted to local variables. Hence, the `Step` abstraction is completely removed. This optimization is known as *scalar replacement* in compilers.

From a different point of view, removing the intermediate `Step` objects is a similar problem to removing the intermediate relations and collections in query engines and collection programming. Hence, one can borrow similar ideas and apply it for the `Step` objects in a fine-grained

```
type GenStream[T] = () => Step[T]

class Streamer[T](gen: GenStream[T]) extends QueryMonad[T] {
  def stream(): Step[T] = gen()
}

def unstream[T](gen: GenStream[T]): QueryMonad[T] =
  new Streamer[T](gen)
```

Figure 4.10 – The constructs for stream fusion.

```
unstream(() => e).stream()

  | (inline unstream definition)
  ↓

new Streamer(() => e).stream()

  | (inline stream definition)
  ↓

e
```

Figure 4.11 – The derivation of the stream-fusion rule.

granularity. More specifically, we use the church-encoding of the `Step` data type to completely remove its abstraction. A similar idea is used in the strymonas library [197].

To do so, we implement a variant of the `Step` data type using the Visitor pattern. As we discussed in Section 4.4, this is similar to the Church-encoding of data types. This encoding results in *pushing* `Step` objects down the pipeline. Hence, the stream-fusion engine implements a pull engine on a coarse-grained level (i.e. relation level) and pushes the individual elements on a fine-grained level (i.e. tuple level). The Visitor pattern version of the `Step` data type is shown in Figure 4.12.

The result of applying this enhancement to our working example is shown in Figure 4.7b. By comparing this code to the code produced by a push engine, we see a clear similarity. First, there are no more additional virtual calls associated with the `Step` operators. Second, there is no more materialization of the intermediate `Step` objects. Finally, similar to push engines, the produced code does not contain any additional nested `while` loop for selection. This leads to a simpler control flow graph, which is shown in Figure 4.7d.

As an alternative implementation, one can implement the `Step` data type as a *sum* type, a type with different distinct cases in which an object can be one and only one of those cases. Hence, the implementation of the `Step` methods can use the pattern matching feature of the Scala programming language. However, it has been proven that the Visitor pattern is a way to encode the sum types in object-oriented languages [47]. On the other hand, pattern matching in Scala is a way to express the Visitor pattern [99]. Hence, from a conceptual point of view there is no difference between these implementations [152].

```
trait StepVisitor[T] {
  def yld(e: T): Unit
  def skip(): Unit
  def done(): Unit
}

trait Step[T] { self =>
  def __match(v: StepVisitor[T]): Unit
  def filter(p: T => Boolean): Step[T] =
    new Step[T] {
      def __match(v: StepVisitor[T]): Unit =
        self.__match(new StepVisitor[T] {
          def yld(e: T): Unit =
            if (p(e)) v.yld(e) else v.skip()
          def skip(): Unit = v.skip()
          def done(): Unit = v.done()
        })
    }
  def map[S](f: T => S): Step[S] =
    new Step[S] {
      def __match(v: StepVisitor[S]): Unit =
        self.__match(new StepVisitor[T] {
          def yld(e: T): Unit = v.yld(f(e))
          def skip(): Unit = v.skip()
          def done(): Unit = v.done()
        })
    }
}

case class Yield[T](e: T) extends Step[T] {
  def __match(v: StepVisitor[T]): Unit = v.yld(e)
}
case object Skip extends Step[Nothing] {
  def __match(v: StepVisitor[Nothing]): Unit = v.skip()
}
case object Done extends Step[Nothing] {
  def __match(v: StepVisitor[Nothing]): Unit = v.done()
}
```

Figure 4.12 – Step data type implemented using the Visitor pattern.

4.7 Experimental Results

We use a server-type x86 machine equipped with two Intel Xeon E5-2620 v2 CPUs running at 2GHz each, 256GB of DDR3 RAM at 1600Mhz and two commodity HDDs of 2TB. The operating system is Red Hat Enterprise 6.7.

Our query compiler uses the same set of transformations for different pipelining techniques to allow for a fair comparison. These transformations consist of *dead code elimination* (DCE), *common subexpression elimination* (CSE) or *global value numbering* (GVN), and *partial evaluation* (inlining and constant propagation). These transformations are provided out-of-the-box by DBLAB [301], which we use as our testbed. Also, the scalar replacement transformation is always applied unless otherwise specified. We do not use any data-structure specialization transformations or inverted indices for these experiments. Finally, all experiments use DBLAB's in-memory row-store representation.

For compiling the generated programs throughout our evaluation we use version 3.9.1 of the

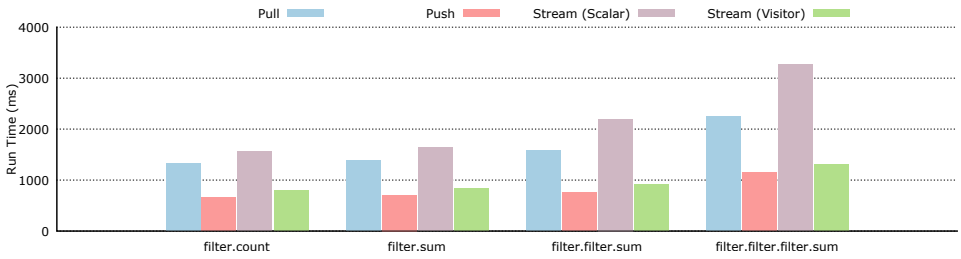


Figure 4.13 – Single-pipeline queries compiled without any optimization flags specified for CLang.

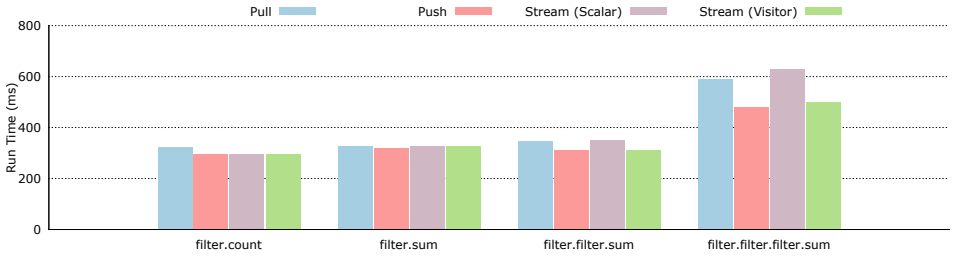


Figure 4.14 – Single-pipeline queries compiled with the `-O3` optimization flag for CLang.

CLang compiler. We use the most aggressive optimization strategy provided by the CLang compiler (the `-O3` optimization flag)[5]. Finally, for C data structures we use the GLib library (version 2.42.1).

Our evaluation consists of two parts. First, by using micro benchmarks, we clearly demonstrate the differences between different query engines. Then, for more complex queries, we use the TPC-H [343] benchmark to demonstrate how different query engines behave in more complicated scenarios.

4.7.1 Micro Benchmarks

The micro benchmarks belong to three categories, (1) queries consisting of only selection and aggregation without group by attributes leading to a single result, (2) queries consisting of a limit operator, which return a list of results, and (3) queries with selection and different join operators, such as hash join, merge join, and hash semi-join, which are followed by an aggregation operator resulting to a single result. All these queries use generated TPC-H databases at scaling factor 8, unless otherwise specified. The corresponding SQL queries for all these micro benchmarks are shown in Table D.1.

Aggregated Single Pipeline. Next, we measure the performance obtained by each engine

[5]We observed similar performance results with the `-O1` optimization flag. The `-O1` optimization flag provides all the transformation passes used in HyPer [255] except global value numbering (GVN). This transformation is not needed in our case, as it is already provided by DBLAB [301].

for queries with a single pipeline, which aggregate into a single result. Figure 4.13 shows the performance of different engines when the generated C code is compiled without any optimization flags. The push engine is behaving $2X$ better than the pull engine in most cases. The visitor-based stream-fusion engine hides this limitation of the pull engine, and has a similar performance to push engines. However, the stream-fusion engines that use scalar replacement perform worse than pull engines.

The difference is more obvious whenever there are chains of selection operations. A similar effect was shown in HyPer [255] in the case of using up to four consecutive selection operations. Again the visitor-based stream-fusion engine is resolving this practical limitation of pull engines. From a practical point of view, as the query optimizer is merging all conjunctive predicates into a single selection operator, the case in which a chain of several selection operators are followed by each other never happens in practice.

The difference among all types of engines can be removed by using more aggressive optimizations of the underlying optimizing compiler. Figure 4.14 shows that using the `-O3` optimization flag of CLang, the performance of all types of engines is similar. This is mainly thanks to the CFG simplification performed by CLang. However, queries with more complicated selection predicates (e.g. user-defined functions or external functions such as `strcmp`) make the reasoning hard for the underlying optimizing compiler. Hence, CFG simplification cannot be applied, and push-based engine and stream-fusion engine have a superior performance in comparison with a pull-based engine. The impact of the optimizations provided by an underlying optimizing compiler is discussed in more details in Section F.

Single Pipeline with Limit. Next, we examine the results for single pipeline queries which have a limit operator at the end of the pipeline. In all three queries, the push engine is performing worse than the pull-based engine and the stream-fusion engine. This is because the standard push engine cannot perform early loop-termination when using the limit query operator (c.f. Section 4.2.2).

To better illustrate the mentioned behavior, Figure 4.15 shows the generated code for the `take.sum` query for pull and push-based query engines. The pull engines do not require traversing all the elements and can stop immediately after reaching the limit operator (c.f. line 18 of Figure 4.15a). However, the push engine should wait until all elements are produced to be able to finish the execution (c.f. Figure 4.15b). A more detailed explanation on a similar query is given in Section G.1. A similar behavior has been observed for pull-based and push-based fusion techniques for Java 8 streaming API in [34].

Single Join. Finally, we investigate the performance of different join operations, which is demonstrated in Figure 4.16. In the case of hash join and left-semi hash-join operators, there is no obvious difference among the engines. However, in the case of merge join, there is a great advantage for pull engines in comparison with the push engine. This is mainly because the push engine cannot pipeline both inputs of a merge join. Hence, it is forced to break the

```
var sum = 0.0
var index = 0
var count = 0
while(true) {
  if(count < 1000) {
    var rec = null
    if(index < R.length) {
      rec = R(index)
      index += 1
    } else {
      rec = null
    }
    if(rec == null) break
    else sum += rec.B
    count += 1
  } else {
    break
} }
return sum
```

(a) Inlined query in pull engine.

```
var sum = 0.0
var index = 0
var count = 0


while(index < R.length) {
  val rec = R(index)
  index += 1


  if(count < 1000) {
     sum += rec.B
     count += 1
  }

}
return sum
```

(b) Inlined query in push engine.

Figure 4.15 – Compiled version of the `take.sum` query in pull and push engines.

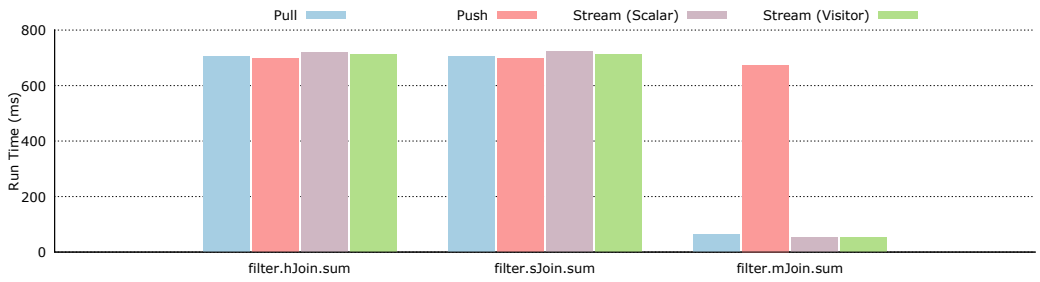


Figure 4.16 – Single-join queries using hash join (hJoin), left-semi hash join (sJoin), and merge join (mJoin) operators.

pipeline in one of the inputs (c.f. Section 4.2.2)[6]. A more detailed investigation of the merge join operator is given in Section G.2.

4.7.2 Macro Benchmarks

In this section, we investigate scenarios which are happening more often in practice. To do so, we use the larger and more complicated analytical queries defined in the TPC-H benchmark. First, we investigate the difference between an inlined and an uninlined version of a pull-based query engine on 12 TPC-H queries. Then, we show the impact of fine-grained optimizations as well as a inline-friendly way of implementing pull engines on one of the TPC-H queries. Finally, we demonstrate the performance difference among different types of query engines for

[6]The stream-fusion engine should have a special case for handling merge joins followed by filter operations. By skipping the elements in the main loop of merging, many CPU cycles are wasted for retrieving the next satisfying element. However, accessing them by using a similar approach to the Iterator model (keep iterating until the next satisfying element is found in a tight loop) gives a better performance.

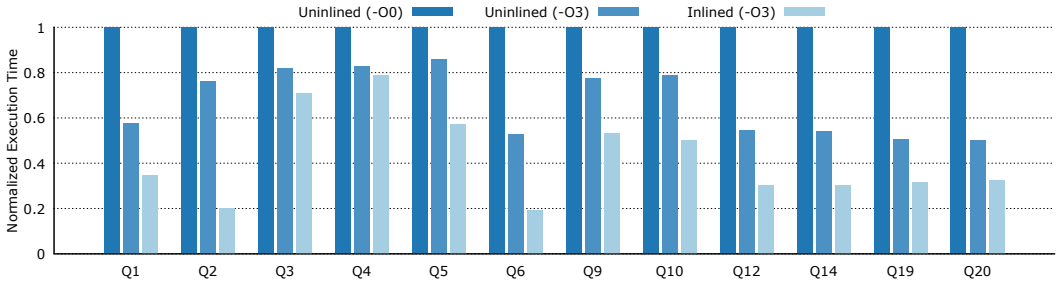


Figure 4.17 – The impact of inlining and low-level optimizations of CLang on a pull-based engine for TPC-H queries.

Type of Engine	Run Time (ms)
Pull (Interpreted)	3486
Pull (Naïve)	2405
Pull (Inline-Friendly)	2165
Stream (Scalar Replacement for `Step` objects)	2447
Stream (Visitor model for `Step` objects)	2217
Stream (No removal of `Step` objects)	6886

Table 4.4 – The performance comparison of several variants of different engines on TPC-H query 19.

12 TPC-H queries. The remaining 10 TPC-H queries require features which are not supported by all our query engines. All these experiments use 8 GBs of TPC-H generated data.

The Impact of Inlining on Pull Engine. As it was explained in Section 4.2.3, we expect inlined (compiled) query engines to perform better than their corresponding uninlined (interpreted) version. Figure 4.17 demonstrates the normalized execution time for 12 TPC-H queries for interpreted and compiled pull-based query engines. The compiled query engine inlines the `next` function invocations of a pull-based query engine, whereas the interpreted query engine invokes the (virtual) functions during run time. Performing aggressive compilation of the interpreted query using the `-O3` flag of the CLang compiler, improves the performance of the interpreted query. However, the best performance is achieved by generating C code using query compilation, and then compiling the generated code using the `-O3` flag of the CLang compiler. On average, inlining the pull-based engine gives 67% improvement. In particular, for TPC-H query 2, we observe a 4 times speedup. This considerable performance improvement is the result of the removal of intermediate object allocations which is achieved after inlining the operators of the query by DBLAB/LB. One exception is TPC-H query 4 which we see a negligible performance improvement after inlining. This query uses a semi hash join operator for implementing the functionality required for the EXISTS clause. The cost required for building the intermediate hash table (which is implemented using the GLib library) dominates the cost of (virtual) function calls. Hence, we do not see a significant improvement by inlining

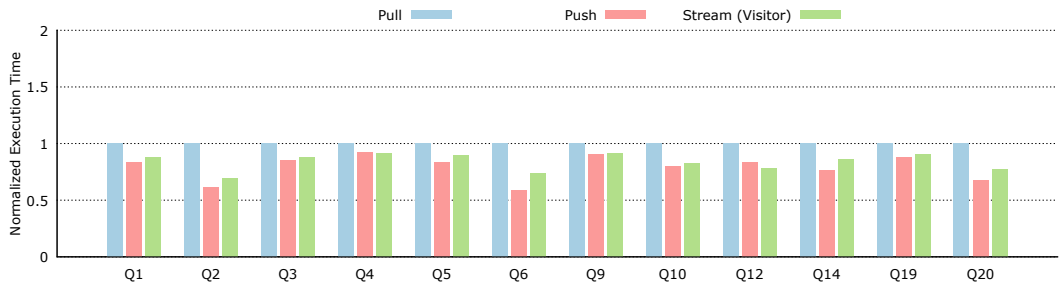


Figure 4.18 – Performance of different compiled query engines for TPC-H queries, when using the `-O0` flag with the CLang compiler.

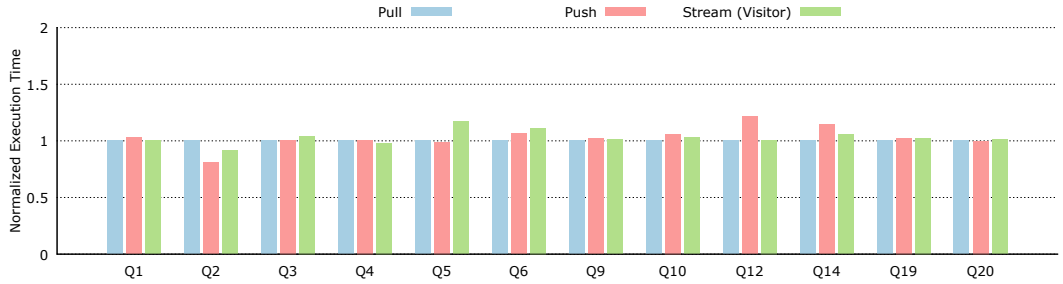


Figure 4.19 – Performance of different compiled query engines for TPC-H queries, when using the `-O3` flag with the CLang compiler.

those function calls. The absolute execution times for all these queries can be found in Table 4.5. Note that, the performance difference with LegoBase [199, 300] and DBLAB/LB [301] is due to the lack of additional optimizations provided by these systems such as data-structure specialization.

Inline-Friendly Pull Engine Implementation. A naïve implementation of the selection operator in a pull-based query engine, invokes the `next` method of its source operator twice. This can exponentially grow the code size in the case of a chain of selection operators. This case is not frequent in practice, since the selection operator is mainly used right after the scan operator. However, in the case of TPC-H query 19 the selection operator is used after a join[7]. Table 4.4 shows that the inline-friendly implementation of the selection operator in pull engines, improves performance by 15%. One of the main reasons is that the inline-friendly implementation generates around 40% less query processing code in comparison with the naïve implementation for query processing in these two queries. This improves instruction cache locality, as a larger part of the code can fit into the instruction cache.

[7] An alternative implementation is to fuse the selections happening after joins in the join operator itself. The experiments performed in [296] are based on this assumption for join operators. This means that the join operator is not a pure join operator, but a super operator containing a join operator followed by a selection operator. For the purposes of this thesis we do not consider this case.

Removing Intermediate Object Allocations. Table 4.4 shows the impact of intermediate object allocations on performance. Overall, removing heap allocations of intermediate `Step` objects improves the performance of a stream-fusion engine up to three times. More specifically, the visitor model for `Step` objects improves performance by 50% in comparison with the case in which Scalar Replacement is used for removing intermediate heap allocations (c.f. Section 4.6.3). Furthermore, our experiments show that removing intermediate `Step` objects (either by visitor model or Scalar Replacement) decreases the memory consumption from 14 GBs to 11 GBs for TPC-H query 19.

Different Engines on Analytical Queries. Figures 4.18 shows the performance of several TPC-H queries using different engines, when they are not using any optimizations provided by CLang. Overall, this figure shows that the difference between engines is not in terms of "orders of magnitude"; in most cases, improvements are minor. This is because the comparison is performed in a fair scenario in which specialization is performed on all engines, in contrast with previous work in which operator inlining was not applied to pull engines [199].

Based on this figure we make the following observations. First, in all cases, the push-based engine is outperforming both pull-based engines. This is justified by the simplified control flow produced by push-based engines. Second, the visitor-based stream-fusion engine has a similar performance to a push-based engine, thanks to the simplified control flow offered by its visitor-based tuple-level implementation. Finally, even for queries with limit and merge join the pull-based engine is not performing better than the push-based engine. For queries with limit, as the limit operator is followed by an ordering operator, the cost of sorting outweighs the cost of the final scan. As a result, pipelining the limit operator does not have a considerable impact. For TPC-H query 12, which has a merge join operator, the performance penalty caused by the complicated control flow graph of pull-based engine hides the improvement of pipelining this join operator. However, the stream-fusion engine has a better performance than both pull- and push-based engines, thanks to pipelining the merge join operator, while keeping the control flow simple.

Now, we answer the following question: to which extent the underlying optimizing compiler can hide the limitations of each engine? Figures 4.19 shows the performance of several TPC-H queries using different engines, when the CLang compiler is used with the `-O3` flag. Overall, this figure shows that for most queries all types of engines have a similar performance. This means that the underlying optimizing compiler (CLang) successfully optimizes the code generated by pull-based engines, to the extent that the generated machine code behaves similarly to the generated machine code for push-based engines.

However, there are still some cases for which CLang cannot compensate the limitation of a particular engine. Query 12 falls into this category because of its use of the merge join operator. This query has an average 70% speed up for a pull engine in comparison with a push engine. It is important to note that in this query, the query plan that uses a merge join is almost two times faster than the one that uses hash join. This is because both input relations are already

Query Engine	Q1	Q2	Q3	Q4	Q5	Q6	Q9	Q10	Q12	Q14	Q19	Q20
Pull Uninlined -O0	10249	3872	9022	14574	6925	1748	19918	5155	3039	4205	6882	2674
Pull Uninlined -O3	5911	2949	7371	12042	5947	919	15420	4055	1652	2271	3486	1338
Pull -O0	8344	1907	8810	14834	6528	1470	16655	4979	2788	2749	6853	2378
Push -O0	6982	1166	7521	13690	5460	868	15045	3994	2326	2097	6022	1605
Stream (Visitor) -O0	7287	1314	7730	13560	5827	1077	15278	4087	2188	2358	6168	1833
Pull -O3	3540	769	6387	11474	3970	338	10612	2588	921	1263	2165	869
Push -O3	3652	622	6429	11557	3927	359	10809	2742	1121	1447	2218	868
Stream (Visitor) -O3	3552	704	6652	11260	4640	376	10703	2659	928	1334	2217	881

Table 4.5 – Execution times (in milliseconds) of different compiled query engines for TPC-H queries.

sorted on the join key. Hence, the merge join implementation can perform the join on the fly, as opposed to the hash join implementation which needs to construct an intermediate hash table while joining the two input relations.

The stream-fusion engine always uses the Visitor pattern throughout this experiment. Interestingly, it is performing as well as push engines and significantly better than pull engines, whenever one does not rely on an underlying optimizing compiler to simplify the control flow. Furthermore, in the cases where push engines require to break the pipeline (the limit and merge join operators) the stream-fusion engine is as efficient as pull engines.

In this section, we have shown the experimental results for different design choices for building complied query engines. Although in realistic workloads there is no considerable advantage for either form of query engine, in certain edge cases each of the pull and push-based engines have their own advantages. We have seen how the stream-fusion engine combines the individual advantages of both approaches in such edge cases, while avoiding their weaknesses. This makes the stream-fusion engine a good alternative choice for building compiled query engines.

4.8 Discussion: Column Stores and Vectorization

Column Stores. For analytical workloads, there is merit in column-store databases [162, 317]. The PL community is using a similar representation [274], known as structs of arrays. Furthermore, database systems can leverage the compression opportunities provided by the column-stores which can improve performance and space consumption [35, 2, 384].

As it was explained in Section 4.6.1, DBLAB/LB supports both row layout and columnar layout representations. The columnar layout representation is achieved by translating the array of records coming from a row layout representation, each one containing N fields, into N arrays, where each array corresponds to the values of a particular column. Figure 4.20 shows the generated code of our running example for push and pull-based engines. Lines 12 and 14 of this figure show how the accesses to the fields A and B of a record of the relation R are

```
var sum = 0.0
var index = 0
while(true) {
  var recNull = true
  do {
    if(index < R_length) {
      recNull = false
      index += 1
    } else {
      recNull = true
    }
  } while(!recNull && !(R_A(index) < 10))
  if(recEmpty) break
  else sum += R_B(index)
}
return sum
```

(a) Inlined query in a column-store pull engine.

```
var sum = 0.0
var index = 0


while(index < R_length) {

  index += 1


  if(R_A(index) < 10)

    sum += R_B(index)
}
return sum
```

(b) Inlined query in a column-store push engine.

Figure 4.20 – Specialized version of the example query in column-store pull and push engines.

transformed into the accesses to the corresponding column arrays `R_A` and `R_B` in the columnar layout representation.[8]

Although the experiments shown in this chapter use row layout, we have found similar results when comparing different engines using columnar layout representation. More specifically, Figure 4.21 shows the performance of the single-pipeline aggregated queries for different types of compiled columnar and row layout query engines. The performance of column-store engines is better than the performance of the row-store counterparts. Furthermore, the relative speedup of different engines over a pull-based engine using the columnar layout representation is similar to the speedup of the corresponding engines over a pull-based engine for the row layout representation. The only considerable difference is the first query, which counts the number of the filtered elements. This query gives a better performance for column-store push and stream-fusion engines, thanks to the automatic vectorization performed by CLang. However, the generated code of a column-store pull-based engine cannot be automatically vectorized by the compiler due to the existence of data-dependent exit conditions [267].

Supporting hash-based join operators for column-stores requires careful consideration of where to construct the tuples (a.k.a. the materialization strategies [4]), and even implementing other join operators (such as JIVE join [221]), which we leave for future work.

Vectorization. Using SIMD operations for implementing query operators has been extensively investigated by the DB community [276, 60, 383]. MonetDB [42] implemented a vectorized query engine by performing block-wise data processing instead of tuple-wise processing,

[8] Note that in most real-world database systems a column-store consists of other abstractions such as pages. However, for the purposes of this thesis, we have presented a simplified form which only uses column arrays. Furthermore, as the column arrays have a primitive type (i.e. cannot have a null value), the check for termination in a pull-based engine (i.e. checking the equality to null) is handled through the intermediate boolean variable `recNull`.

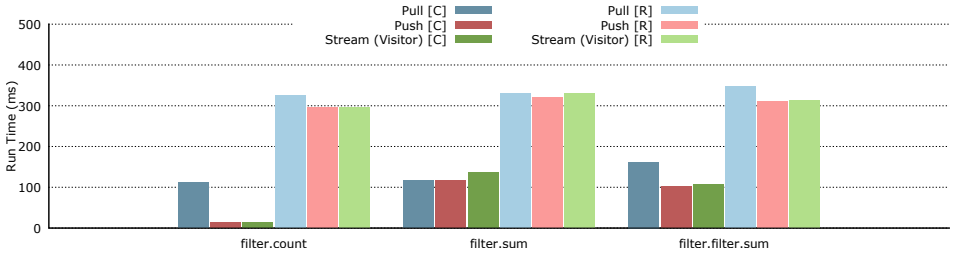


Figure 4.21 – Performance of different compiled query engines with columnar layout and row layout representations (denoted by [C] and [R], respectively), when using the `-O3` flag with the CLang compiler.

through transferring a block of elements in the iterator model instead of a single element. Generalized stream fusion [229] followed a similar idea and showed how, by exploring vectorization opportunities, a high-level functional language can beat handwritten C code for collection programs. We can follow a similar idea to perform vectorization for the stream-fusion engine. On the other hand, push engines can also benefit from vectorization by pushing a block of elements and processing them using SIMD operations as explained in [255].

4.9 Conclusions

If one effects a fair comparison of push and pull-based query processing – particularly if one attempts to inline and optimize code in both approaches as much as possible – neither approach clearly outperforms the other. We have discussed the reasons for this, and indeed, when considered closely how each approach fundamentally works, it should seem rather surprising if either approach dominated the other performance-wise.

We have also drawn close connections to three fundamental approaches to loop fusion in programming languages – fold, unfold, and stream fusion. As it turns out, there is a close analogy between pull engines and unfold fusion on one hand and push engines and fold fusion on the other.

Finally, we have applied the lessons learned about the weaknesses of either approach and propose a new approach to building query engines which draws its inspiration from stream fusion and combines the individual advantages of pull and push engines, avoiding their weaknesses.

5 Efficient Memory Management

A programming language is low level when its programs require attention to the irrelevant.
– Alan Perlis

In the previous chapters of this thesis, we have seen how to efficiently compile analytical applications expressed in languages based on relational algebra such as SQL. In the rest of this thesis, we turn our attention to more sophisticated analytics expressed in languages inspired by linear algebra. More specifically, in this chapter we show how to compile high-level functional array-processing programs, drawn from image processing and machine learning, into C code that runs as fast as hand-written C. The key idea is to transform the program to *destination-passing style*, which in turn enables a highly-efficient stack-like memory allocation discipline.

5.1 Introduction

Applications in computer vision, robotics, and machine learning [344, 332] may need to run in memory-constrained environments with strict latency requirements, and have high turnover of small-to-medium-sized arrays. For these applications the overhead of most general-purpose memory management, for example malloc/free, or of a garbage collector, is unacceptable, so programmers often implement custom memory management directly in C.

In this chapter we propose a technique that automates a common custom memory-management technique, which we call *destination-passing style* [213, 242] (DPS), as used in efficient C and Fortran libraries such as BLAS. We allow the programmer to code in a high-level functional style, while guaranteeing efficient stack allocation of all intermediate arrays. Fusion techniques for such languages are absolutely essential to eliminate intermediate arrays, and are well established. But fusion leaves behind an irreducible core of intermediate arrays that *must* exist to accommodate multiple or random-access consumers.

The key idea behind DPS is that every function is given the storage in which to store its result. The caller of the function is responsible for allocating the destination storage, and deallocating it as soon as it is no longer needed. This incurs a burden at the call site of computing the size of the callee result, but we will show how a surprisingly rich input language can nevertheless allow these computations to be done statically, or in negligible time. Our contributions are:

- We propose a new destination-passing style intermediate representation that captures a stack-like memory management discipline and ensures there are no leaks (Section 5.3). This is a good compiler intermediate language because we can perform transformations on it and reason about how much memory a program will take. It also allows efficient C code generation with bump-allocation. Although it is folklore to compile functions in this style when the result size is known, we have not seen DPS used as an actual compiler intermediate language, despite the fact that DPS has been used for other purposes (cf. Section 4.8).

- DPS requires to know at the call site how much memory a function will need. We design a carefully-restricted higher-order functional language, $\widetilde{\text{F}}$ (Section 5.2) which is a subset of F#, and a *compositional* shape translation (Section 5.3.3) that guarantees to compute the result size of any $\widetilde{\text{F}}$ expression, either statically or at runtime, with no allocation, and a run-time cost independent of the data or its size (Section 5.3.6). Other languages with similar properties [166] expose shape concerns intrusively at the language level, while $\widetilde{\text{F}}$ programs are just F#.

- We present the implementation of of the technique (Section 5.4) and evaluate the runtime and memory performance of both micro-benchmarks and real-life computer vision and machine-learning workloads written in our high-level language and compiled to C via DPS (as shown in Section 5.5). We show that our approach gives performance comparable to, and sometimes better than, idiomatic C++.

5.2 $\widetilde{\text{F}}$

$\widetilde{\text{F}}$ (we pronounce it F smooth) is a subset of F#, an ML-like functional programming language (the syntax in this thesis is slightly different from F# for presentation reasons). It is designed to be *expressive enough* to make it easy to write array-processing workloads, while simultaneously being *restricted enough* to allow it to be compiled to code that is as efficient as hand-written C, with very simple and efficient memory management. We are willing to sacrifice some expressiveness to achieve higher performance. As presented here, $\widetilde{\text{F}}$ strictly imposes its language restrictions, rejecting programs for which shape inference is not efficient. Of course it would also be possible to emit compilation warnings for inefficient constructs, and defer shape calculation to runtime, and also to add heap allocation using F#'s explicit "new".

e	::=	e e | `fun x -> e` | x	– Application, Abstraction, and Variable Access
	|	n | i | N	– Scalar, Index, and Cardinality Value
	|	c	– Constants (see below)
	|	`let` x = e `in` e	– (Non-Recursive) Let Binding
	|	`if` e `then` e `else` e	– Conditional
T	::=	M	– (Non-Functional) Expression Type
	|	T ⇒ T	– Function Types
M	::=	Num	– Numeric Type
	|	`Array<M>`	– Vector, Matrix, ... Type
	|	`Bool`	– Boolean Type
Num	::=	`Double` | `Index` | `Card`	– Scalar, Index, and Cardinality Type

Typing Rules:

$$\frac{e_1\ e_2 : T_2}{e_1 : T_1 \Rightarrow T_2 \quad e_2 : T_1}\ \text{(T-App)} \qquad \frac{\Gamma \vdash \texttt{fun x -> e} : T_1 \Rightarrow T_2}{\Gamma \cup x : T_1 \vdash e : T_2}\ \text{(T-Abs)} \qquad \frac{\Gamma \vdash x : T}{x : T \in \Gamma}\ \text{(T-Var)}$$

$$\frac{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2}{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}\ \text{(T-Let)} \qquad \frac{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : M}{e_1 : \texttt{Bool} \quad e_2 : M \quad e_3 : M}\ \text{(T-If)}$$

Scalar Function Constants:

`+` | `-` | `*` | `/` | `**`	:	Num, Num ⇒ Num	`>` | `<` | `==` | `<>`	:	Num ⇒ Num ⇒ `Bool`
`sin` | `cos` | `tan` |			`&&` | `||`	:	`Bool` ⇒ `Bool` ⇒ `Bool`
`log` | `exp`	:	Num ⇒ Num	`!`	:	`Bool` ⇒ `Bool`

Vector Function Constants:

`build` : `Card` ⇒ (`Index` ⇒ M) ⇒ `Array<M>`		`get`	: `Array<M>` ⇒ `Index` ⇒ M
`ifold` : (M ⇒ `Index` ⇒ M) ⇒ M ⇒ `Card` ⇒ M		`length`	: `Array<M>` ⇒ `Card`

Syntactic Sugar:

$e_0[e_1]$	=	`get` e_0 e_1	`Vector`	=	`Array<Double>`
e_1 *bop* e_2	=	*bop* e_1 e_2	`Matrix`	=	`Array<Array<Double>>`

Figure 5.1 – The syntax, type system, and function constants of the core $\widetilde{\text{F}}$.

5.2.1 Syntax and Types of $\widetilde{\text{F}}$

In addition to the usual λ-calculus constructs (abstraction, application, and variable access), $\widetilde{\text{F}}$ supports let binding and conditionals. The syntax, type system, and several built-in functions are shown in Figure 5.1. Note that Figure 5.1 shows an abstract syntax and parentheses can be used as necessary. Also, $\overline{\text{x}}$ and $\overline{\text{e}}$ denote one or more variables and expressions, respectively, which are separated by spaces, whereas, $\overline{\text{T}}$ represents one or more types which are separated by commas.

In support of array programming, the language has several built-in functions defined: `build` for producing arrays; `ifold` for iteration for a particular number of times (from `0` to `n-1`) while maintaining a state across iterations; `length` to get the size of an array; and `get` to index an array.

Although $\widetilde{\text{F}}$ is a higher-order functional language, it is carefully restricted in order to make it efficiently compilable:

Matlab	R	NumPy	$\widetilde{\text{M}}$
A * B	A %*% B	A.dot(B)	matrixMult A B
A + B	A + B	A + B	matrixAdd A B
A'	t(A)	A.T	matrixTranspose A
ones(n, m)	matrix(1, n, m)	ones((n, m))	matrixOnes n m
zeros(n, m)	matrix(0, n, m)	zeros((n, m))	matrixZeros n m
eye(n)	diag(n)	eye(n)	matrixEye n

Table 5.1 – Equivalent operations in Matlab, R, NumPy, and $\widetilde{\text{M}}$.

- $\widetilde{\text{F}}$ does not support arbitrary recursion, hence is not Turing Complete. Instead one can use `build` and `ifold` for producing and iterating over arrays.
- The type system is monomorphic. The only polymorphic functions are the built-in functions of the language, such as `build` and `ifold`, which are best thought of as language constructs rather than first-class functions.
- An array, of type `Array<M>`, is one-dimensional but can be nested. If arrays are nested they are expected to be rectangular, which is enforced by defining the specific `Card` type for dimension of arrays, which is used as the type of the first parameter of the `build` function. This assumption simplifes our shape inference algorithm as can be seen in Section 5.3.3.
- We assume that no partial application is allowed as an expression in this language. Additionally, an abstraction cannot return a function value.

Next, we show how a Linear Algebra domain-specific language (DSL) can be defined on top of $\widetilde{\text{F}}$.

5.2.2 $\widetilde{\text{M}}$

$\widetilde{\text{M}}$ is a functional Linear Algebra DSL, mainly inspired by MATLAB and R, programming languages which are heavily used by data analysts. By providing high-level vector and matrix operations, $\widetilde{\text{M}}$ frees the users from low-level details and enables them to focus on the algorithmic aspects of the problem in hand.

$\widetilde{\text{M}}$ is an *embedded DSL* (EDSL) [158] in $\widetilde{\text{F}}$; it is defined as a library on top of $\widetilde{\text{F}}$. Figure 5.2 demonstrates a subset of $\widetilde{\text{M}}$ operations which are defined as functions in $\widetilde{\text{F}}$. This DSL is expressive enough for constructing vectors and matrices, elementwise-operations, accessing a slice of elements, reduction-based operations (computing the sum of vector elements), matrix transpose, and matrix multiplication. More specifically, there are vector mapping operations (vectorMap and vectorMap2) which *build* vectors using the size of the input vectors. The i^{th} element (using a zero-based indexing system) of the output vector is the result of the application of the given function to the i^{th} element of the input vectors. Using the vector

mapping operations, one can define vector addition, vector element-wise multiplication, and vector-scalar multiplication. Then, there are several vector operations which consume a given vector by *folding* over its elements. For example, vectorSum computes the sum of the elements of the given vector, which is used by the vectorDot and vectorNorm operations. Similarly, several matrix operations are defined using these vector operations. More specifically, matrix-matrix multiplication is defined in terms of vector dot product and matrix transpose. Supporting more sophisticated operations such as matrix determinant and matrix decomposition is beyond the scope of the current work, and we leave it for the future.

$\widetilde{\text{M}}$ is inspired by MATLAB and R, the programming languages heavily used by data scientists. As a result, there is a mapping among the constructs of $\widetilde{\text{M}}$ and these matrix-based languages. Hence, it is easily possible to translate a program written in one of these languages to $\widetilde{\text{M}}$. Table 5.1 demonstrates the mapping among a subset of the constructs of MATLAB, R, NumPy and $\widetilde{\text{M}}$.

5.2.3 Fusion

Fusion is essential for array programs, without it they cannot be efficient. However fusion is also extremely well studied [357, 121, 325, 71], and we simply take it for granted in this work. Let us work through one example which illustrates how fusion can be applied to an $\widetilde{\text{F}}$ program.

Consider this function, which returns the norm of the vector resulting from the addition of its two input vectors.

```
f = fun vec1 vec2 ->
  vectorNorm (vectorAdd vec1 vec2)
```

Executing this program, as is, involves constructing two vectors in total: one intermediate vector which is the result of adding the two vectors `vec1` and `vec2`, and another intermediate vector which is used in the implementation of vectorNorm (vectorNorm invokes vectorDot, which invokes vectorEMul in order to perform the element-wise multiplication between two vectors). After using the rules presented in Figure 5.3, the fused function is as follows:

```
f = fun vec1 vec2 ->
  sqrt (ifold (fun sum idx ->
    let tmp = vec1[idx]+vec2[idx] in
      sum + tmp * tmp
  ) 0 (length vec1))
```

This is better because it does not construct the intermediate vectors. Instead, the elements of the intermediate vectors are consumed as they are produced.

```
let vectorRange = fun n ->
  build n (fun i -> i)
let vectorFill = fun n e ->
  build n (fun i -> e)
let vectorHot = fun n i ->
  build n (fun j -> if i = j then 1 else 0)
let vectorMap = fun v f ->
  build (length v) (fun i -> f v[i])
let vectorMap2 = fun v1 v2 f ->
  build (length v1) (fun i -> f v1[i] v2[i])
let vectorAdd = fun v1 v2 ->
  vectorMap2 v1 v2 (+)
let vectorEMul = fun v1 v2 ->
  vectorMap2 v1 v2 (×)
let vectorSMul = fun v s ->
  vectorMap v (fun a -> a × s)
let vectorSum = fun v ->
  ifold (fun s i -> s + v[i]) 0 (length v)
let vectorDot = fun v1 v2 ->
  vectorSum (vectorEMul v1 v2)
let vectorNorm = fun v ->
  sqrt (vectorDot v v)
let vectorSlice = fun v s e ->
  build (e − s + 1) (fun i -> v[i + s])
let vectorToMatrix = fun v ->
  build 1 (fun i -> v)
let vectorOutProd = fun v1 v2 ->
  let m1 = vectorToMatrix v1
  let m2 = vectorToMatrix v2
  let m2T = matrixTranspose m2
  matrixMul m1 m2T
```

```
let matrixRows = fun m -> length m
let matrixCols = fun m -> length (m[0])
let matrixZeros = fun r c ->
  build r (fun i -> vectorFill c 0)
let matrixOnes = fun r c ->
  build r (fun i -> vectorFill c 1)
let matrixEye = fun n ->
  build n (fun i -> vectorHot n i)
let matrixHot = fun n m r c ->
  build n (fun i ->
    build m (fun j ->
      if (i = r && j = c) then 1 else 0
  ) )
let matrixMap = fun m f ->
  build (length m) (fun i -> f m[i])
let matrixMap2 = fun m1 m2 f ->
  build (length m1) (fun i -> f m1[i] m2[i])
let matrixAdd = fun m1 m2 ->
  matrixMap2 m1 m2 vectorAdd
let matrixTranspose = fun m ->
  build (matrixCols m) (fun i ->
    build (matrixRows m) (fun j ->
      m[j][i]
  ) )
let matrixMul = fun m1 m2 ->
  let m2T = matrixTranspose m2
  build (matrixRows m1) (fun i ->
    build (matrixCols m2) (fun j ->
      vectorDot (m1[i]) (m2T[j])
  ) )
```

Figure 5.2 – A subset of $\widetilde{\text{M}}$ constructs defined in $\widetilde{\text{F}}$.

However, our focus is on efficient allocation and de-allocation of the arrays that fusion cannot remove. For example: the array might be passed to a foreign library function; or it might be passed to a library function that is too big to inline; or it might be consumed by multiple consumers, or by a consumer with a random (non-sequential) access pattern. In these cases there are good reasons to build an intermediate array, but we want to allocate, fill, use, and de-allocate it extremely efficiently. In particular, we do not want to rely on a garbage collector.

5.3 Destination-Passing Style

Thus motivated, we define a new intermediate language, DPS-$\widetilde{\text{F}}$, in which memory allocation and deallocation is explicit. DPS-$\widetilde{\text{F}}$ uses *destination-passing style*: every array-returning

$$
\begin{array}{lcl}
(\texttt{build}\ e_0\ e_1)[e_2] & \rightsquigarrow & e_1\ e_2 \\
\texttt{length}\ (\texttt{build}\ e_0\ e_1) & \rightsquigarrow & e_0
\end{array}
$$

Figure 5.3 – Fusion rules of $\widetilde{\text{F}}$.

$$
\begin{array}{rcll}
t & ::= & t\ \overline{t} \mid \texttt{fun}\ \overline{x}\ \texttt{->}\ t \mid n \mid i \mid x \mid c \mid \texttt{let}\ x = t\ \texttt{in}\ t & \\
 & \mid & P & \text{– Shape Value} \\
 & \mid & r & \text{– Reference Access} \\
 & \mid & \bullet & \text{– Empty Memory Location} \\
 & \mid & \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t & \text{– Conditional} \\
 & \mid & \texttt{alloc}\ t\ (\texttt{fun}\ r\ \texttt{->}\ t) & \text{– Memory Allocation} \\
P & ::= & \circ & \text{– Zero Cardinality} \\
 & \mid & N & \text{– Cardinality Value} \\
 & \mid & (N, P) & \text{– Vector Shape Value} \\
c & ::= & \textit{[See Figure 5.5]} & \\
D & ::= & M \mid \overline{D} \Rightarrow M \mid \texttt{Bool} & \\
 & \mid & Shp & \text{– Shape Type} \\
 & \mid & \texttt{Ref} & \text{– Machine Address} \\
M & ::= & Num \mid \texttt{Array<M>} & \\
Num & ::= & \texttt{Double} \mid \texttt{Index} & \\
Shp & ::= & \texttt{Card} & \text{– Cardinality Type} \\
 & \mid & (\texttt{Card} * Shp) & \text{– Vector Shape Type}
\end{array}
$$

Figure 5.4 – The core DPS-$\widetilde{\text{F}}$ syntax.

function receives as its first parameter a pointer to memory in which to store the result array. No function allocates the storage needed for its result; instead the responsibility of allocating and deallocating the output storage of a function is given to the caller of that function. Similarly, all the storage allocated inside a function can be deallocated as soon as the function returns its result.

Destination passing style is a standard programming idiom in C. For example, the C standard library procedures that return a string (e.g. `strcpy`) expect the caller to provide storage for the result. This gives the programmer full control over memory management for string values. Other languages have exploited destination-passing style during compilation [146, 147].

5.3.1 The DPS-$\widetilde{\text{F}}$ Language

The syntax of DPS-$\widetilde{\text{F}}$ is shown in Figure 5.4, while its type system is in Figure 5.5. The main additional construct in this language is the one for allocating a particular amount of storage space `alloc` t1 (`fun r ->` t2). In this construct t1 is an expression that evaluates to the size (in bytes) that is required for storing the result of evaluating t2. This storage is available in the lexical scope of the lambda parameter, *and is deallocated outside this scope.*

The previous example can be written in the following way in DPS-$\widetilde{\text{F}}$:

Typing Rules:

$$\text{(T-Alloc)}\ \frac{\texttt{alloc}\ t_0\ (\texttt{fun}\ r\ \texttt{->}\ t_1)\colon M}{\Gamma \vdash t_0 : \texttt{Card} \quad \Gamma, r : \texttt{Ref} \vdash t_1 : M}$$

Vector Function Constants:

```
build  : Ref, Card, (Ref, Index ⇒ M ),
         Card, (Card ⇒ Shp )
           ⇒ Array<M>
ifold  : Ref, (Ref, M, Index ⇒ M ), M,
         Card, (Shp, Card ⇒ Shp ),
           Shp, Card ⇒ M
get    : Ref, Array<M>, Index,
         Shp, Card ⇒ M
length : Ref, Array<M>, Shp ⇒ Card
copy   : Ref, Array<M> ⇒ Array<M>
```

Scalar Function Constants:

DPS version of $\widetilde{F}$ Scalar Constants (Figure 5.1)

```
stgOff  :  Ref, Shp ⇒ Ref
vecShp  :  Card, Shp ⇒ (Card * Shp)
fst     :  (Card * Shp) ⇒ Card
snd     :  (Card * Shp) ⇒ Shp
bytes   :  Shp ⇒ Card
```

Syntactic Sugar:

$t_0.[t_1]\{r\} = \texttt{get}\ r\ t_0\ t_1$ $\quad \texttt{length}\ t = \texttt{length} \bullet t$

$(t_0, t_1) = \texttt{vecShp}\ t_0\ t_1$

$e_1\ \mathit{bop}\ e_2 = \mathit{bop} \bullet e_1\ e_2$

Figure 5.5 – The type system and built-in constants of DPS-$\widetilde{F}$

```
f = fun r1 vec1 vec2 -> alloc (vecBytes vec1) (fun r2 ->
      vectorNorm_dps • (vectorAdd_dps r2 vec1 vec2) )
```

Each lambda abstraction typically takes an additional parameter which specifies the storage space that is used for its result. Furthermore, every application should be applied to an additional parameter which specifies the memory location of the return value in the case of an array-returning function. However, a scalar-returning function is applied to a dummy empty memory location, specified by •. In this example, the memory location r_1 can be ignored, whereas the number of bytes allocated for the memory location r_2 is specified by the expression (vecBytes vec1) which computes the number of bytes of the array vec1.

5.3.2 Translation from $\widetilde{F}$ to DPS-$\widetilde{F}$

We now turn present the translation from $\widetilde{F}$ to DPS-$\widetilde{F}$. Before translating $\widetilde{F}$ expressions to their DPS form, the expressions should be transformed into a normal form similar to ANF [110]. In this representation, each subexpression of an application is either a constant value or a variable. This greatly simplifies the translation rules, specially the (D-App) rule.[1] The representation of our working example in ANF is as follows:

```
f = fun vec1 vec2 ->
  let tmp = vectorAdd vec1 vec2 in
  vectorNorm tmp
```

[1] In a true ANF, *every* subexpression is a constant value or a variable, whereas in our case, we only care about the subexpressions of an application. Hence, our representation is *almost* ANF.

$$\mathscr{D}[\![\mathrm{e}]\!]\mathrm{r} = \mathrm{t}$$

(D-App)	$\mathscr{D}[\![\mathrm{e}_0\ \mathrm{x}_1\ \ldots\ \mathrm{x}_k]\!]\mathrm{r}$	$= (\mathscr{D}[\![\mathrm{e}_0]\!]\bullet)\ \mathrm{r}\ \mathrm{x}_1\ \ldots\ \mathrm{x}_k\ \mathrm{x}_1{}^{shp}\ \ldots\ \mathrm{x}_k{}^{shp}$
(D-Abs)	$\mathscr{D}[\![\texttt{fun}\ \mathrm{x}_1\ \ldots\ \mathrm{x}_k\ \texttt{->}\ \mathrm{e}_1]\!]\bullet$	$= \texttt{fun}\ \mathrm{r}_2\ \mathrm{x}_1\ \ldots\ \mathrm{x}_k\ \mathrm{x}_1{}^{shp}\ \ldots\ \mathrm{x}_k{}^{shp}\ \texttt{->}\ \mathscr{D}[\![\mathrm{e}_1]\!]\mathrm{r}_2$
(D-VarScalar)	$\mathscr{D}[\![\mathrm{x}]\!]\bullet$	$= \mathrm{x}$
(D-VarVector)	$\mathscr{D}[\![\mathrm{x}]\!]\mathrm{r}$	$= \texttt{copy}\ \mathrm{r}\ \mathrm{x}$
(D-Let)	$\mathscr{D}[\![\texttt{let}\ \mathrm{x} = \mathrm{e}_1\ \texttt{in}\ \mathrm{e}_2]\!]\mathrm{r}$	$= \texttt{let}\ \mathrm{x}^{shp} = \mathscr{S}[\![\mathrm{e}_1]\!]\ \texttt{in}$ $\texttt{alloc}\ (\texttt{bytes}\ \mathrm{x}^{shp})\ (\texttt{fun}\ \mathrm{r}_2\ \texttt{->}$ $\texttt{let}\ \mathrm{x} = \mathscr{D}[\![\mathrm{e}_1]\!]\mathrm{r}_2\ \texttt{in}\ \mathscr{D}[\![\mathrm{e}_2]\!]\mathrm{r})$
(D-If)	$\mathscr{D}[\![\texttt{if}\ \mathrm{e}_1\ \texttt{then}\ \mathrm{e}_2\ \texttt{else}\ \mathrm{e}_3]\!]\mathrm{r}$	$= \texttt{if}\ \mathscr{D}[\![\mathrm{e}_1]\!]\bullet\ \texttt{then}\ \mathscr{D}[\![\mathrm{e}_2]\!]\mathrm{r}\ \texttt{else}\ \mathscr{D}[\![\mathrm{e}_3]\!]\mathrm{r}$

$$\mathscr{D}_{\mathscr{T}}[\![\mathrm{T}]\!] = \mathrm{D}$$

(DT-Fun)	$\mathscr{D}_{\mathscr{T}}[\![\mathrm{T}_1, \ldots, \mathrm{T}_k \Rightarrow \mathrm{M}\,]\!]$	$= \texttt{Ref}, \mathscr{D}_{\mathscr{T}}[\![\mathrm{T}_1]\!], \ldots, \mathscr{D}_{\mathscr{T}}[\![\mathrm{T}_k]\!],$ $\mathscr{S}_{\mathscr{T}}[\![\mathrm{T}_1]\!], \ldots, \mathscr{S}_{\mathscr{T}}[\![\mathrm{T}_k]\!] \Rightarrow \mathscr{D}_{\mathscr{T}}[\![\mathrm{M}]\!]$
(DT-Mat)	$\mathscr{D}_{\mathscr{T}}[\![\mathrm{M}]\!]$	$= \mathrm{M}$
(DT-Bool)	$\mathscr{D}_{\mathscr{T}}[\![\texttt{Bool}]\!]$	$= \texttt{Bool}$
(DT-Card)	$\mathscr{D}_{\mathscr{T}}[\![\texttt{Card}]\!]$	$= \texttt{Card}$

Figure 5.6 – Translation from $\widetilde{\mathrm{F}}$ to DPS-$\widetilde{\mathrm{F}}$

Figure 5.6 shows the translation from $\widetilde{\mathrm{F}}$ to DPS-$\widetilde{\mathrm{F}}$, where $\mathscr{D}[\![\mathrm{e}]\!]\mathrm{r}$ is the translation of a $\widetilde{\mathrm{F}}$ expression e into a DPS-$\widetilde{\mathrm{F}}$ expression that stores e's value in memory r. Rule (D-Let) is a good place to start. It uses `alloc` to allocate enough space for the value of e_1, the right hand side of the let — but how much space is that? We use an auxiliary translation $\mathscr{S}[\![\mathrm{e}_1]\!]$ to translate e_1 to an expression that computes e_1's *shape* rather than its *value*. The shape of an array expression specifies the cardinality of each dimension. We will discuss why we need shape (what goes wrong with just using bytes) and the shape translation in Section 5.3.3. This shape is bound to x^{shp}, and used in the argument to `alloc`. The freshly-allocated storage r_2 is used as the destination for translating the right hand side e_1, while the original destination r is used as the destination for the body e_2.

In general, every variable x in $\widetilde{\mathrm{F}}$ becomes a *pair* of variables x (for x's value) and x^{shp} (for x's shape) in DPS-$\widetilde{\mathrm{F}}$. You can see this same phenomenon in rules (D-App) and (D-Abs), which deal with lambdas and application: we turn each lambda-bound argument x into *two* arguments x and x^{shp}.

Finally, in rule (D-App) the context destination memory r is passed on to the function being called, as its additional first argument; and in (D-Abs) each lambda gets an additional argument, which is used as the destination when translating the body of the lambda. Figure 5.6 also gives a translation of an $\widetilde{\mathrm{F}}$ type T to the corresponding DPS-$\widetilde{\mathrm{F}}$ type D.

For variables there are two cases. In rule (D-VarScalar) a scalar variable is translated to itself, while in rule (D-VarVector) we must copy the array into the designated result storage using the `copy` function. The `copy` function copies the array elements as well as the header information

(the second argument) into the given storage (the first argument).

5.3.3 Shape Translation

As we have seen, rule (D-Let) relies on the *shape translation* of the right hand side. This translation is given in Figure 5.7. If e has type T, then $\mathscr{S}[\![e]\!]$ is an expression of type $\mathscr{S}_{\mathscr{T}}[\![T]\!]$ that gives the shape of e. This expression can always be evaluated without allocation.

A *shape* is an expression of type Shp (Figure 5.4), whose values are given by P in that figure. There are three cases to consider. First, a scalar value has shape $\circ$ (rules (S-ExpNum), (S-ExpBool)). Second, when e is an array, $\mathscr{S}[\![e]\!]$ gives the shape of the array as a nested tuple, such as $(3,(4,\circ))$ for a 3-vector of 4-vectors. So the "shape" of an array specifies the cardinality of each dimension. Finally, when e is a function, $\mathscr{S}[\![e]\!]$ is a function that takes the shapes of its arguments and returns the shape of its result. You can see this directly in rule (S-App): to compute the shape of (the result of) a call, apply the shape-translation of the function to the shapes of the arguments. This is possible because $\widetilde{\text{F}}$ programs do not allow the programmer to write a function whose result size depends on the contents of its input array.

What is the shape-translation of a function f? Remembering that every in-scope variable f has become a pair of variables—one for the value and one for the shape—we can simply use the latter, f^{shp}, as we see in rule (S-Var).

For arrays, could the shape be simply the number of bytes required for the array, rather than a nested tuple? No. Consider the following function, which returns the first row of its argument matrix:

```
firstRow = fun m: Matrix -> m[0]
```

The shape translation of firstRow, namely firstRow^{shp}, is given the shape of m, and must produce the shape of m's first row. It cannot do that given only the number of bytes in m; it must know how many rows and columns it has. But by defining shapes as a nested tuple, it becomes easy: see rule (S-Get).

The shape of the result of the iteration construct (`ifold`) requires the shape of the state expression to remain the same across iterations, which is by checking the beta equivalence of the initial shape and the shape of each iteration. Otherwise the compiler produces an error, as shown in rule (S-Ifold).

The other rules are straightforward. *The key point is that by translating every in-scope variable, including functions, into a pair of variables, we can give a* compositional *account of shape translation, even in a higher order language.*

$$\mathcal{S}[\![\mathrm{e}]\!] = \mathrm{s}$$

(S-App)	$\mathcal{S}[\![\mathrm{e}_0\ \mathrm{e}_1\ ...\ \mathrm{e}_k\ ]\!]$	$= \mathcal{S}[\![\mathrm{e}_0]\!]\ \mathcal{S}[\![\mathrm{e}_1]\!]\ ...\ \mathcal{S}[\![\mathrm{e}_k]\!]$
(S-Abs)	$\mathcal{S}[\![\texttt{fun}\ x_1\ ...,\ x_k\ \texttt{->}\ \mathrm{e}\ ]\!]$	$= \texttt{fun}\ {x_1}^{shp}, ..., {x_k}^{shp}\ \texttt{->}\ \mathcal{S}[\![\mathrm{e}]\!]$
(S-Var)	$\mathcal{S}[\![\mathrm{x}]\!]$	$= \mathrm{x}^{shp}$
(S-Let)	$\mathcal{S}[\![\texttt{let}\ \mathrm{x} = \mathrm{e}_1\ \texttt{in}\ \mathrm{e}_2]\!]$	$= \texttt{let}\ \mathrm{x}^{shp} = \mathcal{S}[\![\mathrm{e}_1]\!]\ \texttt{in}\ \mathcal{S}[\![\mathrm{e}_2]\!]$
(S-If)	$\mathcal{S}[\![\texttt{if}\ \mathrm{e}_1\ \texttt{then}\ \mathrm{e}_2\ \texttt{else}\ \mathrm{e}_3]\!]$	$= \begin{cases} \mathcal{S}[\![\mathrm{e}_2]\!] & \mathcal{S}[\![\mathrm{e}_2]\!] \cong \mathcal{S}[\![\mathrm{e}_3]\!] \\ \text{Comp. Error!} & \mathcal{S}[\![\mathrm{e}_2]\!] \ncong \mathcal{S}[\![\mathrm{e}_3]\!] \end{cases}$
(S-ExpNum)	$\mathrm{e}\text{: Num} \vdash \mathcal{S}[\![\mathrm{e}]\!]$	$= \circ$
(S-ExpBool)	$\mathrm{e}\text{: }\texttt{Bool} \vdash \mathcal{S}[\![\mathrm{e}]\!]$	$= \circ$
(S-ValCard)	$\mathcal{S}[\![\mathrm{N}]\!]$	$= \mathrm{N}$
(S-AddCard)	$\mathcal{S}[\![\mathrm{e}_0 + \mathrm{e}_1]\!]$	$= \mathcal{S}[\![\mathrm{e}_0]\!] + \mathcal{S}[\![\mathrm{e}_1]\!]$
(S-MulCard)	$\mathcal{S}[\![\mathrm{e}_0 * \mathrm{e}_1]\!]$	$= \mathcal{S}[\![\mathrm{e}_0]\!] * \mathcal{S}[\![\mathrm{e}_1]\!]$
(S-Build)	$\mathcal{S}[\![\texttt{build}\ \mathrm{e}_0\ \mathrm{e}_1]\!]$	$= (\mathcal{S}[\![\mathrm{e}_0]\!], (\mathcal{S}[\![\mathrm{e}_1]\!] \circ))$
(S-Get)	$\mathcal{S}[\![\mathrm{e}_0[\mathrm{e}_1]]\!]$	$= \texttt{snd}\ \mathcal{S}[\![\mathrm{e}_0]\!]$
(S-Length)	$\mathcal{S}[\![\texttt{length}\ \mathrm{e}_0]\!]$	$= \texttt{fst}\ \mathcal{S}[\![\mathrm{e}_0]\!]$
(S-Ifold)	$\mathcal{S}[\![\ \texttt{ifold}\ \mathrm{e}_1\ \mathrm{e}_2\ \mathrm{e}_3\ ]\!]$	$= \begin{cases} \mathcal{S}[\![\mathrm{e}_2]\!] & \mathcal{S}[\![\mathrm{e}_1\ \mathrm{e}_2\ 0]\!] \cong \mathcal{S}[\![\mathrm{e}_2]\!] \\ \text{Comp. Error!} & \text{otherwise} \end{cases}$

$$\mathcal{S}_{\mathcal{T}}[\![\mathrm{T}]\!] = \mathrm{S}$$

(ST-Fun)	$\mathcal{S}_{\mathcal{T}}[\![\mathrm{T}_1, \mathrm{T}_2, ..., \mathrm{T}_k \Rightarrow \mathrm{M}\ ]\!]$	$= \mathcal{S}_{\mathcal{T}}[\![\mathrm{T}_1]\!], \mathcal{S}_{\mathcal{T}}[\![\mathrm{T}_2]\!], ..., \mathcal{S}_{\mathcal{T}}[\![\mathrm{T}_k]\!] \Rightarrow \mathcal{S}_{\mathcal{T}}[\![\mathrm{M}]\!]$
(ST-Num)	$\mathcal{S}_{\mathcal{T}}[\![\mathrm{Num}]\!]$	$= \texttt{Card}$
(ST-Bool)	$\mathcal{S}_{\mathcal{T}}[\![\texttt{Bool}]\!]$	$= \texttt{Card}$
(ST-Card)	$\mathcal{S}_{\mathcal{T}}[\![\texttt{Card}]\!]$	$= \texttt{Card}$
(ST-Vector)	$\mathcal{S}_{\mathcal{T}}[\![\texttt{Array<M>}]\!]$	$= (\texttt{Card} * \mathcal{S}_{\mathcal{T}}[\![\mathrm{M}]\!])$

Figure 5.7 – Shape Translation of $\widetilde{\mathrm{F}}$

5.3.4 An Example

Using this translation, the running example at the beginning of Section 5.3.2 is translated as follows:

```
f = fun r0 vec1 vec2 vec1^shp vec2^shp ->
  let tmp^shp = vectorAdd^shp vec1^shp vec2^shp in
  alloc (bytes tmp^shp) (fun r1 ->
    let tmp =
      vectorAdd r1 vec1 vec2 vec1^shp vec2^shp in
    vectorNorm r0 tmp tmp^shp
  )
```

The shape translations of some $\widetilde{\mathrm{M}}$ constructs from Figure 5.2 are as follows:

```
let vectorRange^shp = fun n^shp -> (n^shp, (fun i^shp -> ∘) ∘)
let vectorMap2^shp = fun v1^shp v2^shp f^shp ->
  (fst v1^shp, (fun i^shp -> ∘) ∘)
let vectorAdd^shp = fun v1^shp v2^shp ->
  vectorMap2^shp v1^shp v2^shp (fun a^shp b^shp -> ∘)
let vectorNorm^shp = fun v^shp -> ∘
```

5.3.5 Simplification

As is apparent from the examples in the previous section, code generated by the translation has many optimisation opportunities. This optimisation, or simplification, is applied in three stages: 1) $\widetilde{\mathrm{F}}$ expressions, 2) translated Shape-$\widetilde{\mathrm{F}}$ expressions, and 3) translated DPS-$\widetilde{\mathrm{F}}$ expressions. In the first stage, $\widetilde{\mathrm{F}}$ expressions are simplified to exploit fusion opportunities that remove intermediate arrays entirely. Furthermore, other compiler transformations such as constant folding, dead-code elimination, and common-subexpression elimination are also applied at this stage.

In the second stage, the Shape-$\widetilde{\mathrm{F}}$ expressions are simplified. The simplification process for these expressions mainly involves partial evaluation. By inlining all shape functions, and performing β-reduction and constant folding, shapes can often be computed at compile time, or at least can be greatly simplified. For example, the shape translations presented in Section 5.3.3 after performing simplification are as follows:

```
let vectorRange^shp = fun n^shp -> (n^shp, ∘)
let vectorMap2^shp = fun v1^shp v2^shp f^shp -> v1^shp
let vectorAdd^shp = fun v1^shp v2^shp -> v1^shp
let vectorNorm^shp = fun v^shp -> ∘
```

Empty Allocation:
`alloc` ∘ (`fun` r `->` t_1) $\rightsquigarrow$ $t_1[r \mapsto \bullet]$

Allocation Merging:
`alloc` t_1 (`fun` r_1 `->`
  `alloc` t_2 (`fun` r_2 `->`
    t_3)) $\rightsquigarrow$ `alloc` (t_1 + t_2) (`fun` r_1 `->`
      `let` r_2 = `stgOff` r_1 t_1 `in`
      t_3)

Dead Allocation:
`alloc` t_1 (`fun` r `->` t_2) $\rightsquigarrow$ t_2 if $r \notin FV(t_2)$

Allocation Hoisting:
`fun` x `->` `alloc` t_1 (`fun` r `->` t_2) $\rightsquigarrow$ `alloc` t_1 (`fun` r `->` `fun` x `->` t_2) if $x \notin FV(t_1)$

Cardinality Simplification:
`bytes` ∘ $\rightsquigarrow$ ∘
`bytes` (∘,∘) $\rightsquigarrow$ ∘
`bytes` (N,∘) $\rightsquigarrow$ `NUM_BYTES` ∗ N + `HDR_BYTES`
`bytes` (N,s) $\rightsquigarrow$ (`bytes` s) ∗ N + `HDR_BYTES`

Figure 5.8 – Simplification rules of DPS-$\widetilde{\text{F}}$

The final stage involves both partially evaluating the shape expressions in DPS-$\widetilde{\text{F}}$ and simplifying the storage accesses in the DPS-$\widetilde{\text{F}}$ expressions. Figure 5.8 demonstrates simplification rules for storage accesses. The first two rules remove empty allocations and merge consecutive allocations, respectively. The third rule removes a dead allocation, i.e. an allocation for which its storage is never used. The fourth rule hoists an allocation outside an abstraction whenever possible. The benefit of this rule is amplified more in the case that the storage is allocated inside a loop (`build` or `ifold`). Note that none of these transformation rules are available in $\widetilde{\text{F}}$, due to the lack of explicit storage facilities.

After applying the presented simplification process, our working example is translated to the following program:

```
f = fun r0 vec1 vec2 vec1^shp vec2^shp ->
  alloc (bytes vec1^shp) (fun r1 ->
    let tmp = vectorAdd r1 vec1 vec2
            vec1^shp vec2^shp in
    vectorNorm r0 tmp vec1^shp
  )
```

In this program, there is no shape computation at runtime.

5.3.6 Properties of Shape Translation

The target language of shape translation is a subset of DPS-$\widetilde{\text{F}}$ called Shape-$\widetilde{\text{F}}$. The syntax of the subset is given in Figure 5.9. It includes nested pairs, of statically-known depth, to represent shapes, but it does not include vectors. That provides an important property for Shape-$\widetilde{\text{F}}$ as

$$
\begin{array}{rcl}
s & ::= & s\ \overline{s} \mid \texttt{fun}\ \overline{x}\ \texttt{->}\ s \mid x \mid P \mid c \mid \texttt{let}\ x = s\ \texttt{in}\ s \\
P & ::= & \circ \mid N \mid (N, P) \\
c & ::= & \texttt{vecShp} \mid \texttt{fst} \mid \texttt{snd} \mid + \mid * \\
S & ::= & \overline{S} \Rightarrow \text{Shp} \mid \text{Shp} \\
\text{Shp} & ::= & \texttt{Card} \mid (\texttt{Card} * \text{Shp})
\end{array}
$$

Figure 5.9 – Shape-$\widetilde{\text{F}}$ syntax, which is a subset of the syntax of DPS-$\widetilde{\text{F}}$ presented in Figure 5.4.

follows:

Theorem 1 *All expressions resulting from shape translation, do not require any heap memory allocation.*

Proof. All the non-shape expressions have either scalar or function type. As shown in Figure 5.7 all scalar type expressions are translated into zero cardinality ($\circ$), which can be stack-allocated. On the other hand, the function type expressions can also be stack allocated. This is because functions are not allowed to return functions. Hence, the captured environment in a closure does not escape its scope. Hence, the closure environment can be stack allocated. Finally, the last case consists of expressions which are the result of shape translation for vector expressions. As we know the number of dimensions of the original vector expressions, the translated expressions are tuples with a known-depth, which can be easily allocated on stack.

Next, we show the properties of our translation algorithm. First, let us investigate the impact of shape translation on $\widetilde{\text{F}}$ types. For array types, we need to represent the shape in terms of the shape of each element of the array, and the cardinality of the array. We encode this information as a tuple. For scalar type and cardinality type expressions, the shape is a cardinality expression. This is captured in the following theorem:

Theorem 2 *If the expression e in $\widetilde{F}$ has the type* T*, then $\mathscr{S}[\![e]\!]$ has type $\mathscr{S}_{\mathscr{T}}[\![\text{T}]\!]$.*

Proof. Can be proved by induction on the translation rules from $\widetilde{\text{F}}$ to Shape-$\widetilde{\text{F}}$.

In order to have a simpler shape translation algorithm as well as better guarantees about the expressions resulting from shape translation, two important restrictions are applied on $\widetilde{\text{F}}$ programs.

1. The accumulating function used in the `ifold` operator should preserve the shape of the initial value. Otherwise, converting the result shape into a closed-form polynomial expression requires solving a recurrence relation.
2. The shape of both branches of a conditional should be the same.

These two restrictions simplify the shape translation as is shown in Figure 5.7.

Theorem 3 *All expressions resulting from shape translation require linear computation time with respect to the size of terms in the original $\widetilde{F}$ program.*

Proof. This can be proved in two steps. First, translating a $\widetilde{F}$ expression into its shape expression, leads to an expression with smaller size. This can be proved by induction on translation rules. Second, the run time of a shape expression is linear in terms of its size. An important case is the `ifold` construct, which by applying the mentioned restrictions, we ensured their shape can be computed without any need for recursion.

Finally, we believe that our translation is correct based on our successful implementation. However, we leave a formal semantics definition and the proof of correctness of the transformation as future work.

5.3.7 Discussion

One possible question is whether the DPS technique can go beyond the $\widetilde{F}$ language. In other words, is it possible to support programs which require an arbitrary recursion, such as filtering an array, changing the size while recursing, or computing a Fibonacci-size array?

The answer is yes; instead of producing compilation errors (cf. Figure 5.7), the compiler produces warnings and postpones the shape computation until the run time. However, this can cause a massive run time overhead, as it is no longer possible to benefit from the performance guarantees mentioned in Section 5.3.6. More specifically, the shape computation could be as time consuming as the original array expressions [154], which can cause massive computation and space overheads. As an example, the computation complexity of a Fibonacci-size array will be $O(2.7^n)$ instead of $O(1.6^n)$ (the former is the closed form of $f(n) = 2f(n-1) + 2f(n-2)$, while the latter is the closed form of $f(n) = f(n-1) + f(n-2)$).

5.4 Implementation

5.4.1 $\widetilde{F}$ Language

We implemented $\widetilde{F}$ as a subset of F#. Hence $\widetilde{F}$ programs are normal F# programs. Furthermore, the built-in constants (presented in Figure 5.1) are defined as a library in F# and all $\widetilde{M}$ constructs (presented in Figure 5.2) are implemented as library functions using these built-in constants. If a given expression is in the subset supported by $\widetilde{F}$, the compiler accepts it.

For implementing the transformations presented in the previous sections, instead of modifying the F# compiler, we use F# quotations [327]. Note that there is no need for the user to use F# quotations in order to implement a $\widetilde{F}$ program. The F# quotations are only used by the compiler developer in order to implement transformation passes.

Although $\widetilde{F}$ expressions are F# expressions, it is not possible to express memory management

constructs used by DPS-$\widetilde{\text{F}}$ expressions using the F# runtime. Hence, after translating $\widetilde{\text{F}}$ expressions to DPS-$\widetilde{\text{F}}$, we compile down the result program into a programming language which provides memory management facilities, such as C. The generated C code can either be used as kernels by other C programs, or invoked in F# as a native function using inter-operatorability facilities provided by Common Language Runtime (CLR).

Next, we discuss why we choose C and how the C code generation works.

5.4.2 C Code Generation

There are many programming languages which provide manual memory management. Among them we are interested in the ones which give us full control on the runtime environment, while still being easy to debug. Hence, low-level imperative languages such as C and C++ are better candidates than LLVM mainly because of debugging purposes.

One of the main advantages of DPS-$\widetilde{\text{F}}$ is that we can generate idiomatic C from it. More specifically, the generated C code is similar to a handwritten C program as we can manage the memory in a stack fashion. The translation from DPS-$\widetilde{\text{F}}$ programs into C code is quite straightforward.

As our DPS encoded programs are using the memory in a stack fashion, the memory could be managed more efficiently. More specifically, we first allocate a specific amount of buffer in the beginning. Then, instead of using the standard `malloc` function, we bump-allocate from our already allocated buffer. Hence, in most cases allocating memory is only a pointer arithmetic operation to advance the pointer to the last allocated element of the buffer. In the cases that the user needs more than the amount which is allocated in the buffer, we need to double the size of the buffer. Furthermore, memory deallocation is also very efficient in this scheme. Instead of invoking the `free` function, we need to only decrement the pointer to the last allocated storage.

We compile lambdas by performing closure conversion. As functions in DPS-$\widetilde{\text{F}}$ do not return functions, the environment captured by a closure can be stack allocated.

As mentioned in Section 5.2, polymorphism is not allowed except for some built-in constructs in the language (e.g. `build` and `ifold`). Hence, all the usages of these constructs are monomorphic, and the C code generator knows exactly which code to generate for them. Furthermore, the C code generator does not need to perform the closure conversion for the lambdas passed to the built-in constructs. Instead, it can generate an efficient for-loop in place. As an example, the generated C code for a running sum function of $\widetilde{\text{F}}$ is as follows:

```
double vector_sum(vector v) {
  double sum = 0;
  for (index idx = 0; idx < v->length; idx++) {
    sum = sum + v->elements[idx];
  }
  return sum;
}
```

Finally, for the `alloc` construct in DPS-$\widetilde{\text{F}}$, the generated C code consists of three parts. First, a memory allocation statement is generated which allocates the given amount of storage. Second, the corresponding body of code which uses the allocated storage is generated. Finally, a memory deallocation statement is generated which frees the allocated storage. The generated C code for our working example is as follows:

```
double f(storage r0, vector vec1, vector vec2,
         vec_shape vec1_shp, vec_shape vec2_shp) {
 storage r1 = malloc(vector_bytes(vec1_shp));
 vector tmp = vector_add_dps(r1, vec1, vec2, vec1_shp, vec2_shp);
 double result = vector_norm_dps(r0,tmp,vec1_shp);
 free(r1);
 return result;
}
```

We use our own implementation of `malloc` and `free` for bump allocation.

5.5 Experimental Results

For the experimental evaluation, we use an iMac machine equipped with an Intel Core i5 CPU running at 2.7GHz, 32GB of DDR3 RAM at 1333Mhz. The operating system is OS X 10.10.5. We use Mono 4.6.1 as the runtime system for F# programs and CLang 700.1.81 for compiling the C++ code and generated C.[2]

Throughout this section, we compare the performance and memory consumption of the following alternatives:

- **F#:** Using the array operations (e.g. map) provided in the standard library of F# to implement vector operations.
- **CL: Leaky C code**, which is the generated C code from $\widetilde{\text{F}}$, using malloc to allocate vectors, never calling free.
- **CG: C code using Boehm GC**, which is the generated C code from $\widetilde{\text{F}}$, using `GC_malloc` of Boehm GC to allocate vectors.
- **CLF: CL + Fused Loops**, performs deforestation and loop fusion before CL.
- **D: DPS C code using system-provided malloc/free**, translates $\widetilde{\text{F}}$ programs into DPS-$\widetilde{\text{F}}$ before generating C code. Hence, the generated C code frees all allocated vectors. In this variant, the malloc and free functions are used for memory management.
- **DF: D + Fused Loops**, which is similar to the previous one, but performs deforestation before translating to DPS-$\widetilde{\text{F}}$.
- **DFB: DF + Buffer Optimizations**, which performs the buffer optimizations described in Section 5.3.5 (such as allocation hoisting and merging) on DPS-$\widetilde{\text{F}}$ expressions.

[2] All code and outputs are available at `http://github.com/awf/Coconut`.

- **DFBS: DFB using stack allocator**, same as DFB, but using bump allocation for memory management, as previously discussed in Section 5.4.2. This is the best C code we generate from $\widetilde{\mathrm{F}}$.
- **C++: Idiomatic C++**, which uses an handwritten C++ vector library, depending on C++14 move construction and copy elision for performance, with explict programmer indication of fixed-size (known at compile time) vectors, permitting stack allocation.
- **E++: Eigen C++**, which uses the Eigen [137] library which is implemented using C++ expression templates to effect loop fusion and copy elision. Also uses explicit sizing for fixed-size vectors.

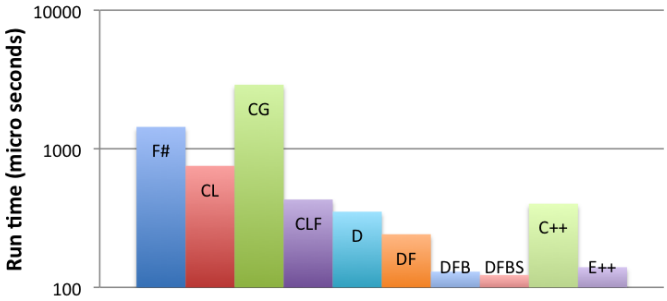
First, we investigate the behavior of several variants of generated C code for two micro benchmarks. More specifically we see how DPS improves both run-time performance and memory consumption (by measuring the maximum resident set size) in comparison with an F# version. The behavior of the generated DPS code is very similar to manually handwritten C++ code and the Eigen library.

Then, we demonstrate the benefit of using DPS for some real-life computer vision and machine learning workloads motivated in [312]. Based on the results for these workloads, we argue that using DPS is a great choice for generating C code for numerical workloads, such as computer vision algorithms, running on embedded devices with a limited amount of memory available.

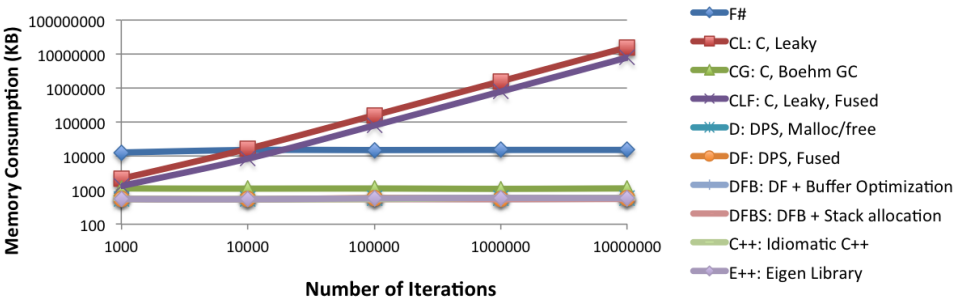
5.5.1 Micro Benchmarks

Figure 5.10 shows the experimental results for adding three vectors, and Figure 5.11 shows the experimental results for cross product of two vectors.

add3 : vectorAdd(vectorAdd(vec1, vec2), vec3)
in which all the vectors contain 100 elements. This program is run one million times in a loop, and timing results are shown in Figure 5.10a. In order to highlight the performance differences, the figure uses a logarithmic scale on its Y-axis. Based on these results we make the following observations. First, we see that all C and C++ programs are outperforming the F# program, except the one which uses the Boehm GC. This shows the overhead of garbage collection in the F# runtime environment and Boehm GC. Second, loop fusion has a positive impact on performance. This is because this program involves creating an intermediate vector (the one resulting from addition of vec1 and vec2). Third, the generated DPS C code which uses buffer optimizations (DFB) is faster than the one without this optimization (DF). This is mainly because the result vector is allocated only once for DFB whereas it is allocated once per iteration in DF. Finally, there is no clear advantage for C++ versions. This is mainly due to the fact that the vectors have sizes not known at compile time, hence the elements are not stack allocated. The Eigen version partially compensates this limitation by using vectorized operations, making the performance comparable to our best generated DPS C code.



(a) Run time performance comparison by running for one million times.

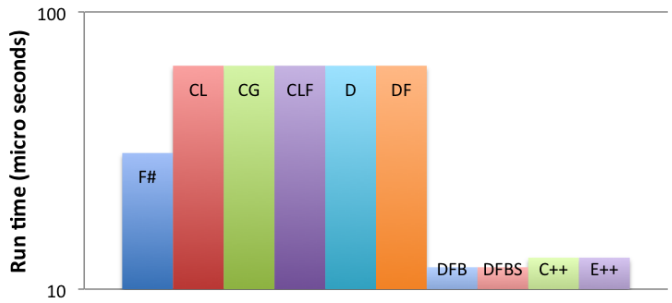


(b) Memory consumption comparison by varying the number of iterations. All the invisible lines are hidden under the bottom line.

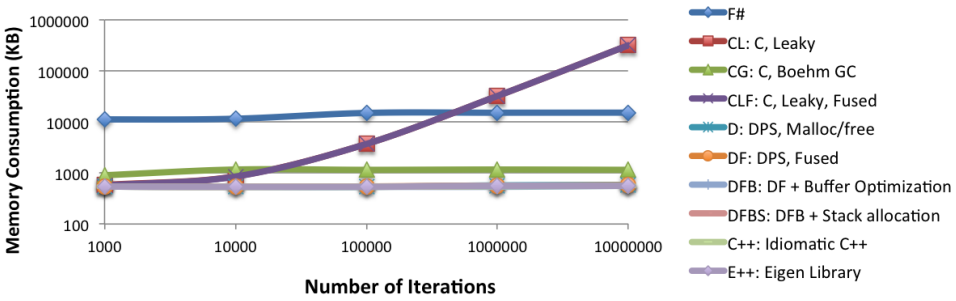
Figure 5.10 – Experimental results for adding three vectors of 100 elements.

The peak memory consumption of this program for different approaches is shown in Figure 5.10b. This measurement is performed by running this program by varying number of iterations. Both axes use logarithmic scales to better demonstrate the memory consumption difference. As expected, F# uses almost the same amount of memory over the time, due to GC. However, the runtime system sets the initial amount to 15MB by default. Also unsurprisingly, leaky C uses memory linear in the number of iterations, albeit from a lower base. The fused version of leaky C (CLF) decreases the consumed memory by a constant factor. Finally, DPS C, and C++ use a constant amount of space which is one order of magnitude less than the one used by the F# program, and half the amount used by the generated C code using Boehm GC.

cross : vectorCross(vec1, vec2)
This micro-benchmark is one million runs in which the two vectors contain 3 elements. Timing results are in Figure 5.11a. We see that the F# program is faster than the generated leaky C code, perhaps because garbage collection is invoked less frequently than in *add3*. Overall, in both cases, the performance of F# program and generated leaky C code is very similar. In this example, loop fusion does not have any impact on performance, as the program contains only one operator. As in the previous benchmark, all variants of generated DPS C code have a similar performance and outperform the generated leaky C code and the one using Boehm



(a) Run time performance comparison by running for one million times.



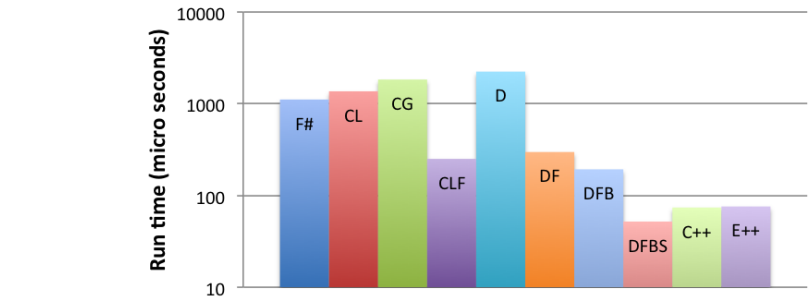
(b) Memory consumption comparison by varying the number of iterations. All the invisible lines are hidden under the bottom line.

Figure 5.11 – Experimental results for cross product of two vectors of three elements.

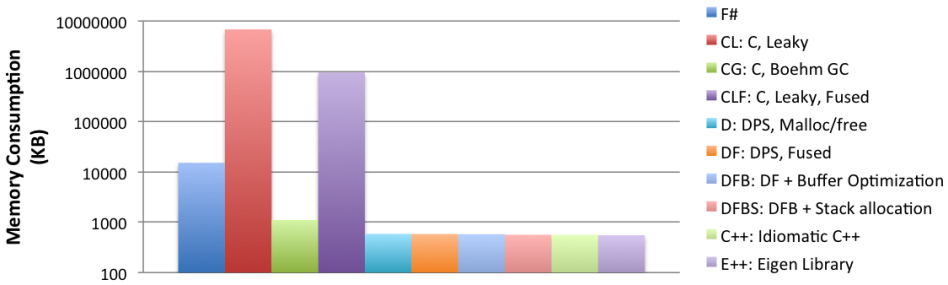
GC, for the same reasons. Finally, both handwritten and Eigen C++ programs have a similar performance to our generated C programs. For the case of this program, both C++ libraries provide fixed-sized vectors, which results in stack allocating the elements of the two vectors. This has a positive impact on performance. Furthermore, as there is no SIMD version of the cross operator, we do not observe a visible advantage for Eigen.

Finally, we discuss the memory consumption experiments of the second program, which is shown in Figure 5.11b. This experiment leads to the same observation as the one for the first program. However, as the second program does not involve creating any intermediate vector, loop fusion does not improve the peak memory consumption.

The presented micro benchmarks show that our DPS generated C code improves both performance and memory consumption by an order of magnitude in comparison with an equivalent F# program. Also, the generated DPS C code promptly deallocates memory which makes the peak memory consumption constant over the time, as opposed to a linear increase of memory consumption of the generated leaky C code. In addition, by using bump allocators the generated DPS C code can improve performance as well. Finally, we see that the generated DPS C code behaves very similarly to both handwritten and Eigen C++ programs.



(a) Run time performance comparison



(b) Peak memory consumption comparison

Figure 5.12 – Experimental results for Bundle Adjustment

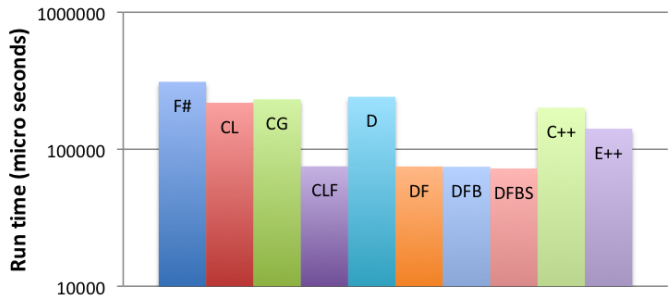
5.5.2 Computer Vision and Machine Learning Workloads

In this section, we investigate the performance and memory consumption of real-life workloads.

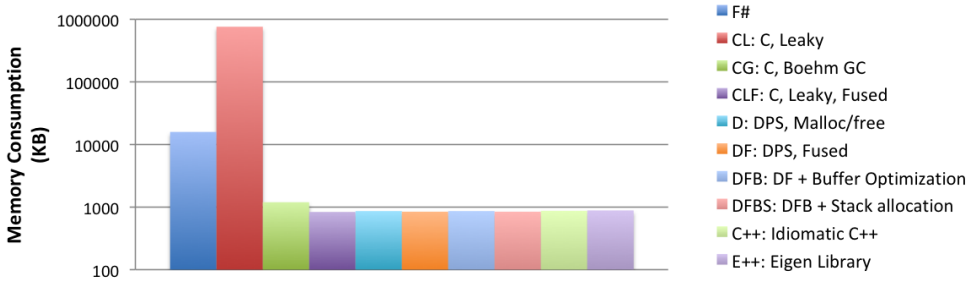
Bundle Adjustment [344] is a computer vision problem which has many applications. In this problem, the goal is to optimize several parameters in order to have an accurate estimate of the projection of a 3D point by a camera. This is achieved by minimizing an objective function representing the reprojection error. This objective function is passed to a nonlinear minimizer as a function handle, and is typically called many times during the minimization.

One of the core parts of this objective function is the *project* function which is responsible for finding the projected coordinates of a 3D point by a camera, including a model of the radial distortion of the lens. The $\widetilde{\text{F}}$ implementation of this method is shown in Figure 5.15.

Figure 5.12a shows the runtime of different approaches after running *project* ten million times. First, the F# program performs similarly to the leaky generated C code and the C code using Boehm GC. Second, loop fusion improves speed fivefold. Third, the generated DPS C code is slower than the generated leaky C code, mainly due to costs associated with intermediate deallocations. However, this overhead is reduced by using bump allocation and performing loop fusion and buffer optimizations. Finally, we observe that the best version of our generated



(a) Run time performance comparison



(b) Peak memory consumption comparison

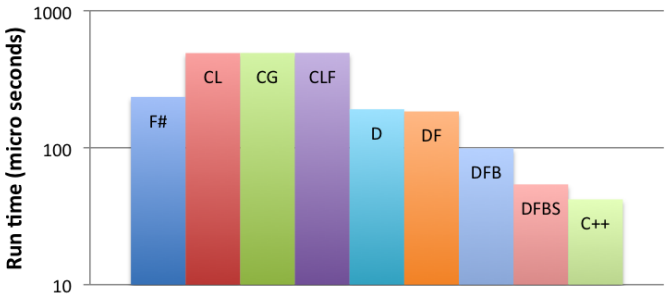
Figure 5.13 – Experimental results for GMM

DPS C code marginally outperforms both C++ versions.

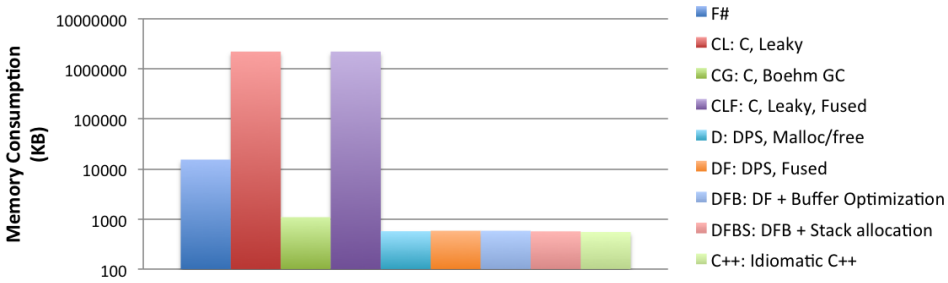
The peak memory consumption of different approaches for Bundle Adjustment is shown in Figure 5.12b. First, the F# program uses three orders of magnitude less memory in comparison with the generated leaky C code, which remains linear in the number of calls. This improvement is four orders of magnitude in the case of the generated C code using Boehm GC. Second, loop fusion improves the memory consumption of the leaky C code by an order of magnitude, due to removing several intermediate vectors. Finally, all generated DPS C variants as well as C++ versions consume the same amount of memory. The peak memory consumption of is an order of magnitude better than the F# baseline.

The Gaussian Mixture Model is a workhorse machine learning tool, used for computer vision applications such as image background modelling and image denoising, as well as semi-supervised learning.

In GMM, loop fusion can successfully remove all intermediate vectors. Hence, there is no difference between CL and CLF, or between DS and DSF, in terms of both performance and peak memory consumption as can be observed in Figure 5.13a and Figure 5.13b. Both C++ libraries behave two to three times worse than our fused and DPS generated code, due to the lack of support for fusion needed for GMM.



(a) Run time performance comparison



(b) Peak memory consumption comparison

Figure 5.14 – Experimental results for Hand Tracking

Due to the cost for performing memory allocation (and deallocation for DPS) at each iteration, the F# program, the leaky C code, and the generated DPS C code exhibit a worse performance than the fused and stack allocated versions. Furthermore, as the leaky C code does not deallocate the intermediate vectors, the consumed memory is increasing.

Hand tracking is a computer vision/computer graphics workload [332] that includes matrix-matrix multiplies, and numerous combinations of fixed- and variable-sized vectors and matrices. Figure 5.14a shows performance results of running one of the main functions of hand-tracking for 1 million times. As in the *cross* micro-benchmark we see no advantage for loop fusion, because in this function the intermediate vectors have multiple consumers. As above, generating DPS C code improves runtime performance, which is improved even more by using bump allocation and performing loop fusion and buffer optimizations. However, in this case the idiomatic C++ version outperforms the generated DPS C code. Figure 5.14b shows that DPS generated programs consume an order of magnitude less memory than the F# baseline, equal to the C++ versions.

5.6 Outlook and Conclusions

In this chapter we presented a new destination-passing style intermediate representation that enables a highly-efficient stack-like memory allocation discipline for memory management.

```
let radialDistort = fun (radical: Vector) (proj: Vector) ->
  let rsq = vectorNorm proj
  let L = 1.0 + radical.[0] * rsq + radical.[1] * rsq * rsq
  vectorSMul proj L
let rodriguesRotate = fun (rotation: Vector) (x: Vector) ->
  let sqtheta = vectorNorm rotation
  if sqtheta != 0. then
    let theta = sqrt sqtheta
    let thetaInv = 1.0 / theta
    let w = vectorSMul rotation thetaInv
    let wCrossX = vectorCross w x
    let tmp = (vectorDot w x) * (1.0 - (cos theta))
    let v1 = vectorSMul x (cos theta)
    let v2 = vectorSMul wCrossX (sin theta)
    vectorAdd (vectorAdd v1 v2) (vectorSMul w tmp)
  else
    vectorAdd x (vectorCross rotation x)
let project = fun (cam: Vector) (x: Vector) ->
  let Xcam = rodriguesRotate (vectorSlice cam 0 2) (
    vectorSub x (vectorSlice cam 3 5) )
  let distorted = radialDistort (vectorSlice cam 9 10) (
      vectorSMul (vectorSlice Xcam 0 1) (1.0/Xcam.[2]) )
  vectorAdd (vectorSlice cam 7 8) (
    vectorSMul distorted cam.[6] )
```

Figure 5.15 – Bundle Adjustment functions in $\widetilde{\text{F}}$.

Also, we presented a carefully-restricted higher-order functional language, called $\widetilde{\text{F}}$, which is guaranteed to be compiled into efficient C code. We plan to relax some of the restrictions that we currently have for the $\widetilde{\text{F}}$ language, and investigate the subset of guarantees that are still valid.

On top of $\widetilde{\text{F}}$, we presented $\widetilde{\text{M}}$, a linear algebra language for expressing advanced data analytics tasks in a high-level of abstraction. We plan to provide different front-ends to convert the programs written in MATLAB, R, and NumPy to $\widetilde{\text{M}}$ programs (cf. Figure 5.1 for a mapping among these languages). As $\widetilde{\text{M}}$ does not support all the features provided by these languages, the provided front-ends should check whether the given program is in the subset that can be compiled to $\widetilde{\text{M}}$ or not.

Finally, we experimentally validated that the run time and memory performance of micro benchmarks and real-life computer vision and machine-learning workloads written in $\widetilde{\text{F}}$, are as good as the ones optimized by hand in C. We plan to implement more data anlaytics tasks in $\widetilde{\text{F}}$ and experimentally evaluate their performance in comparison with the corresponding implementations in other programming langauges. One possible system to compare against is the ML Kit [338] compiler, which implements the region-based memory management for

the Standard ML language.

6 Efficient Differentiable Programming

> *... in the summer of 1958 John McCarthy decided to investigate differentiation as an interesting symbolic computation problem, which was difficult to express in the primitive programming languages of the day. This investigation led him to see the importance of functional arguments and recursive functions in the field of symbolic computation.*
> – Norvig [259, p248].

In this chapter, we present a system for the automatic differentiation of the higher-order functional array-processing language presented in Chapter 5. This core functional language simultaneously supports both source-to-source automatic differentiation and global optimizations such as loop transformations. Thanks to this feature, we demonstrate how for some real-world machine learning and computer vision benchmarks, the system outperforms the state-of-the-art automatic differentiation tools.

6.1 Introduction

Functional programming (FP) and automatic differentiation (AD) have been natural partners for sixty years, and major functional languages all have elegant automatic differentiation packages [98, 27, 183]. With the increasing importance of numerical engineering disciplines such as machine learning, speech processing, and computer vision, there has never been a greater need for systems which mitigate the tedious and error-prone process of manual coding of derivatives. However the popular packages (TensorFlow, CNTK) all implement clunky (E)DSLs in procedural languages such as Python and C++. Why? One reason is that the FP packages are slower than their imperative counterparts, by many orders of magnitude [312], because modern applications depend heavily on array processing, with vectors, matrices, and tensors as the canonical datatypes. In contrast, AD for FP has generally handled only scalar workloads efficiently [183].

Our key contribution in this chapter is to take a recently introduced F# subset designed for

efficient compilation of array-processing workloads (cf. Section 5.2), and to augment it with vector AD primitives, yielding a functional AD tool that is competitive with the best C/C++ and Fortran tools on many benchmarks, and considerably faster on others.

6.1.1 The problem we address

Automatic differentiation is one of the main techniques for automating the process of computing derivatives. This technique systematically applies the chain rule, and evaluates the derivatives for the primitive arithmetic operations (such as addition, multiplication, etc.). One of the main advantages of automatic differentiation over its main competitive technique, *symbolic differentiation*, is the constant-time overhead of the differentiated program with respect to the original code. Symbolic differentiation can lead to code explosion if one is not careful about sharing, and requires a closed-form representation of the programs [27].

There are two approaches for implementing AD. Forward-mode AD computes the derivative part (tangent part) alongside the original computation while making a forward pass over the program. Reverse-mode AD makes a forward pass to compute the original part of the program, followed by a backward pass for computing the derivative part (adjoint part). We present these two techniques through an example.

Example. Consider the function $f(x_1, x_2) = ln(x_1) + sin(x_2)$, for which we would like to compute the partial derivatives with respect to x_1 at point $x_1 = 1$ and $x_2 = 3$. First let us name each intermediate expression with a variable v_i:

$$
\begin{aligned}
f(x_1, x_2) = \quad & \texttt{let } v_1 = ln(x_1) \\
& \texttt{let } v_2 = sin(x_2) \\
& \texttt{let } y = v_1 + v_2 \\
& y
\end{aligned}
$$

This function is computed as follows:

$$
\begin{array}{rcccc}
v_1 & = & ln(1) & = & 0 \\
v_2 & = & sin(3) & = & 0.1411 \\
y & = & 0 + 0.1411 & = & 0.1411
\end{array}
$$

To compute the derivative of this function using the forward-mode AD, we associate the derivative $\overrightarrow{v_i} = \frac{\partial v_i}{\partial x_1} + \frac{\partial v_i}{\partial x_2}$ to each variable v_i. As we are computing the partial derivative of f with respect to x_1, we have $\overrightarrow{x_1} = 1$ and $\overrightarrow{x_2} = 0$. By applying the chain rule, the evaluation trace for the derivative of this function is as follows:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m \qquad \mathbf{J} = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix}$$

Reverse Mode

Forward Mode

$d\widetilde{F}$

Figure 6.1 – The Jacobian Matrix of a function. Forward-mode AD computes a column of this matrix, whereas the reverse-mode AD computes a row of this matrix. $d\widetilde{F}$ computes the full Jacobian matrix using a vectorized variant of the forward-mode AD.

$$\begin{array}{lllllllll}
\vec{v_1} & = & \vec{x_1} \times \frac{\partial ln(x_1)}{\partial x_1} & = & \frac{\vec{x_1}}{x_1} & = & \frac{1}{1} & = & 1 \\
\vec{v_2} & = & \vec{x_2} \times \frac{\partial sin(x_2)}{\partial x_2} & = & \vec{x_2} \times cos(x_2) & = & 0 \times cos(3) & = & 0 \\
\vec{y} & = & \vec{v_1} \times \frac{\partial (v_1+v_2)}{\partial v_1} + \vec{v_2} \times \frac{\partial (v_1+v_2)}{\partial v_2} & = & \vec{v_1} + \vec{v_2} & = & 1+0 & = & 1
\end{array}$$

To compute the derivative of this function using the reverse-mode AD, we associate the *adjoin* term $\overleftarrow{v_i} = \frac{\partial y}{\partial v_i}$ to each variable v_i. As a result, if we are interested in computing the partial derivative of function f with respect to x_1, we have to compute the value of $\overleftarrow{x_1}$. To do so, we have to apply the chain rule in the reverse order, leading to the following execution trace:

$$\begin{array}{lllllll}
\overleftarrow{y} & = & \frac{\partial y}{\partial y} & = & 1 & & \\
\overleftarrow{v_1} & = & \overleftarrow{y} \times \frac{\partial y}{\partial v_1} & = & 1 \times 1 & = & 1 \\
\overleftarrow{v_2} & = & \overleftarrow{y} \times \frac{\partial y}{\partial v_2} & = & 1 \times 1 & = & 1 \\
\overleftarrow{x_2} & = & \overleftarrow{v_2} \times \frac{\partial v_2}{\partial x_2} & = & 1 \times cos(3) & = & \text{-0.9899} \\
\overleftarrow{x_1} & = & \overleftarrow{v_1} \times \frac{\partial v_1}{\partial x_1} & = & 1 \times \frac{1}{1} & = & 1
\end{array}$$

△

Forward and reverse mode compute a column and a row, respectively, of the full Jacobian matrix $\mathbf{J}$ at each invocation. [1] More precisely, for a function with an input vector of size m and an output vector of size n, the forward mode approach computes a column vector of size n, and the reverse mode computes a row vector of size m (see Figure 6.1).

From a different point of view, for a given function f with an input vector parameter a, forward-mode AD produces the function df, where

[1] $\mathbf{J}|_a$ is a matrix consisting of partial derivatives of the output elements of function f with respect to the elements of the input vector at point a.

$$df\ a\ b = \mathbf{J}|_a\ .\ b$$

In the case of passing a one-hot vector as b, where only the i^{th} element is one, the forward-mode AD computes the i^{th} column of the full Jacobian matrix. Similarly, for the same function, the reverse-mode AD produces the function bf, where

$$bf\ a\ c = (\mathbf{J}|_a)^T\ .\ c$$

This expression computes the j^{th} row of the full Jacobian matrix, if c is a one-hot vector with a single one at the j^{th} position and zeros elsewhere.

For a class of optimization problems, such as various computer vision problems using the Levenberg-Marquardt algorithm [234, 220, 248], one is required to compute the *full* Jacobian matrix. In such cases, neither of the two techniques perform efficiently,

To compute the full Jacobian matrix, both forward and reverse-mode techniques must iterate either over the columns or the rows of the Jacobian matrix, respectively. Given that both approaches have a constant overhead over the original computation, the forward mode technique is more efficient for computing the full Jacobian matrix when $n \gg m$, whereas the reverse mode AD is more efficient when $m \gg n$, an uneasy choice. Moreover:

- By carefully examining the body of the loops needed for computing the full Jacobian matrix, one can observe that many computations are loop-invariant and are unnecessarily performed multiple times. Thus, there is a lost opportunity for *loop-invariant code motion* for hoisting such expressions outside the loop, thus improving the performance (cf. the Bundle Adjustment experiment in Section 6.6).

- Furthermore, while the result of automatic differentiation is known to only have by a constant factor more arithmetic operations than the original program, the constant can be significant; this overhead can have a dramatic impact on the run-time performance in practice. More specifically, in applications involving the manipulation of vectors, many intermediate vectors are allocated that can be removed. The optimization for eliminating such intermediate vectors is known as *deforestation* [357, 121, 325, 71] or loop fusion in the functional programming community. This optimization opens the door for many other optimizations such as turning loops iterating over sparse vectors with a single non-zero element into a single statement (cf. Example 4 in Section 6.4).

6.1.2 Our contributions

In this chapter, we present a novel automatic differentiation technique based on forward mode, which combines the benefits of both forward and reverse mode in many cases, and which,

even for cases that require computing the full Jacobian matrix, outperforms both techniques. The key idea behind our technique is that we use a vector-aware programming language, in which the loops required for constructing the full Jacobian matrix are exposed to the compiler. Thus, the compiler can employ global optimization techniques such as loop-invariant code motion and loop fusion for simplifying the differentiated programs.

Example 1. Assume that we have a matrix M and two vectors u and v (which are represented as row matrices and are independent of M). Based on matrix calculus one can prove that $\frac{\partial\left(uMv^T\right)}{\partial M} = u^T v$. However, computing the differentiated version of this function using forward-mode AD tools requires multiple iterations over the differentiated program for every element in the matrix M. By using the reverse-mode AD, one can invoke the differentiated function only once, and the adjoin parts of the input matrix M will be filled in. We show in Section 6.4 that $\mathrm{d}\widetilde{\mathrm{F}}$ derives the gradient of this expression with respect to M, resulting in an expression equivalent to $u^T v$. This removes the need for multiple iterations over the differentiated program for each element of matrix M, in contrast to the existing AD tools based on the forward-mode AD technique.

The contributions of this chapter are summarized as follows:

- We present $\widetilde{\mathrm{F}}$, a higher-order functional array-processing language in Section 5.2. This language can be efficiently compiled into low-level C code with efficient memory management. Then, we present $\widetilde{\mathrm{M}}$, a linear algebra DSL inspired by MATLAB, embedded [158] in this language in Section 5.2.2.
- Then, we show the differentiation programming capabilities provided by $\widetilde{\mathrm{F}}$. First, Section 6.3.1 shows the high-level API exposed in $\widetilde{\mathrm{F}}$ for performing various matrix derivatives such as scalar derivatives, gradients, and Jacobians. Then, we show transformation rules for performing source-to-source automatic differentiation of $\widetilde{\mathrm{F}}$ expressions in Section 6.3.2.
- Afterwards, we show how $\mathrm{d}\widetilde{\mathrm{F}}$ produces efficient differentiated programs by introducing several global optimizations such as loop-invariant code motion, loop fusion, and partial evaluation, as well as generating C code with efficient stack-discipline memory management in Section 6.4.
- Finally, using several micro benchmarks and several functions used in machine learning and computer vision workloads, we show how $\mathrm{d}\widetilde{\mathrm{F}}$ outperforms the state-of-the-art AD techniques in Section 6.6.

6.2 Overview

In this section, we start with an overview of the compilation process in $\mathrm{d}\widetilde{\mathrm{F}}$, which is shown in Figure 6.2. This figure demonstrates the position of $\mathrm{d}\widetilde{\mathrm{F}}$ with respect to existing AD tools. $\mathrm{d}\widetilde{\mathrm{F}}$

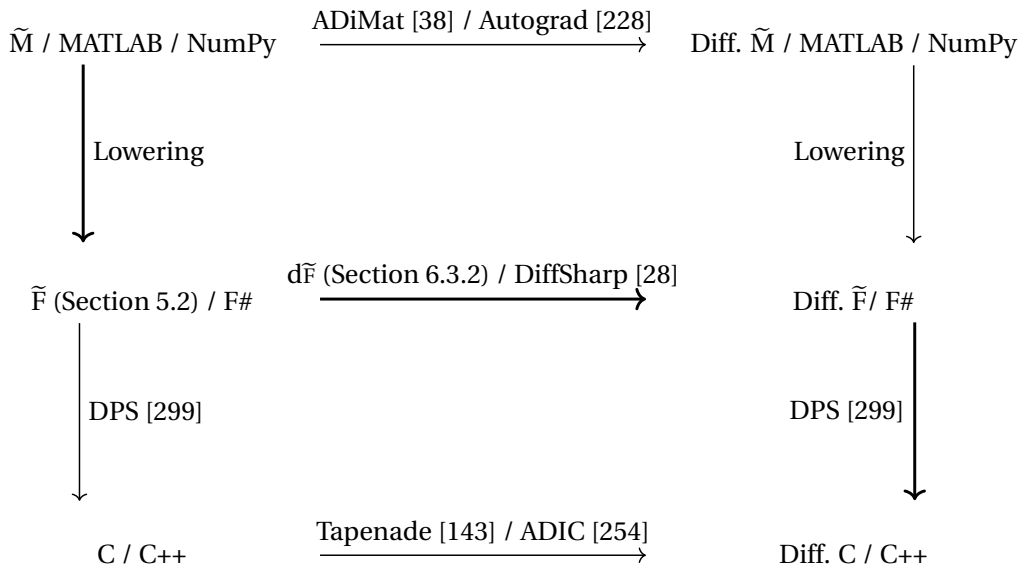


Figure 6.2 – Compilation process in dF̃ and other AD systems. The solid arrows correspond to the pipeline used in dF̃.

e	::=	*[See Figure 5.1]*	M	::=	*[See Figure 5.1]*	
T	::=	*[See Figure 5.1]*		|	$M_1 \times M_2$	– Pair Type

Pair Function Constants:

`pair` : $M_1 \Rightarrow M_2 \Rightarrow M_1 \times M_2$ `fst` : $M_1 \times M_2 \Rightarrow M_1$ `snd` : $M_1 \times M_2 \Rightarrow M_2$

Syntactic Sugar:

(e_0, e_1) = `pair` e_0 e_1 `VectorD = Array<Double × Double>`

`DoubleD = Double × Double` `MatrixD = Array<Array<Double × Double>>`

Figure 6.3 – The syntax, types, and function constants of the extended F̃ language used in dF̃.

starts from a program written in a high-level linear algebra DSL, called M̃ (Section 5.2.2). This program is lowered into its implementation in a higher-order functional language with array support, called F̃ (Section 5.2). If a part of the program requires computing differentiation (which are specified by using high-level differentiation API exposed by dF̃, as mentioned in Section 6.3.1) dF̃ uses AD transformation rules (Section 6.3.2) for transforming the involved expressions into their differentiated form. Finally, after applying several simplifications such as loop fusion, partial evaluation, data layout transformation, etc. (Section 6.4) the differentiated program is transformed into low-level C code. The generated C code uses efficient stack-discipline memory management by using the destination-passing style (DPS) technique [299].

Figure 6.3 shows the abstract syntax , types, and several built-in functions of the extended F̃ used in dF̃. The key additional constructs in the extended version of F̃ corresponds to pair construction and projections.

One of the key features of $\widetilde{\text{F}}$ is its support for both source-to-source automatic differentiation and global optimizations such as loop-invariant code motion and loop fusion in the same time. The transformations required for automatic differentiation are presented in Section 6.3.2, and the ones for optimization and simplification are shown in Section 6.4.

As we have seen in Section 5.2.2, we have embedded $\widetilde{\text{M}}$, a functional Linear Algebra DSL, in $\widetilde{\text{F}}$. Next, we see how our working example can be expressed in $\widetilde{\text{M}}$.

Example 1 (Continued). The matrix expression uMv^T is expressed as the following function in $\widetilde{\text{M}}$:

```
let f = fun u M v ->
  let um = vectorToMatrix u
  let vt = matrixTranspose (vectorToMatrix v)
  let m = matrixMult um (matrixMult M vt)
  m[0][0]
```

The last expression is for accessing the single scalar element of a 1×1 matrix.

△

6.3 Differentiation

In this section, we show the differentiation process in $\text{d}\widetilde{\text{F}}$. First, we start by the high-level API exposed by $\text{d}\widetilde{\text{F}}$ to the end users. Then, we show how $\text{d}\widetilde{\text{F}}$ uses automatic differentiation behind the scenes for computing derivatives. Finally, we present the optimizations offered by $\text{d}\widetilde{\text{F}}$, and we demonstrate how $\text{d}\widetilde{\text{F}}$ can use these optimizations to deduce several matrix calculus identities.

6.3.1 High-Level API

For computing the derivative of an arbitrary function, $\text{d}\widetilde{\text{F}}$ provides the `deriv` construct. This construct can be better thought of as a macro, which is expanded during compilation time. The expanded expression includes the expression of the original computation, which is given as the first argument (and can be an arbitrary scalar, vector, or matrix expression), and the derivative of this expression with respect to the variable given as the second argument, referred to as the *independent variable*. Note that one can easily compute the derivative of an expression with respect to a list of free variables by multiple invocation of the `deriv` construct.

Algorithm 1 shows a pseudo-code implementation of the `deriv` construct. First, `deriv` constructs a lambda function which has the free variables of the given expression as its input parameters (cf. line 6). This function is given as input to source-to-source automatic differentiation for computing the derivative (cf. line 8). The differentiated function is applied to

the dual number encoding of all the free variables (cf. lines 5-8). If the free variable is different than the input variable with respect to which we are differentiating (i.e., the independent variable), the derivative part is a zero scalar, vector, or matrix (cf. lines 26-33). Otherwise, the derivative part is a one-hot encoding scalar, vector, or matrix (cf. lines 35-42).

If the independent variable has a scalar type, `deriv` returns the applied function (cf. lines 9-11). However, if the independent variable has a vector type, `deriv` constructs a vector with the same number of elements as the independent variable. For computing the ri^{th} element of the result vector, the corresponding input vector is a one-hot encoding with a single one at the ri^{th} position (cf. lines 12 and 39). The situation is similar for an independent variable with a matrix type; the corresponding one-hot encoding matrix has a single one at the ri^{th} row and ci^{th} column (cf. lines 14 and 41). Note that the two variables ri and ci are treated specially and are distinguished variables.

Example 2. Let us assume that we would like to compute the derivative of a program computing the cosine function with respect to its input:

```
cos(a)
```

The derivative of this program at point a is represented as follows:

```
snd (deriv (cos a) a)
```

This expression is transformed into the following expression after expanding the `deriv` macro:

```
snd ((𝒟⟦fun a -> cos(a)⟧) (a, 1))
```

△

Furthermore, d$\widetilde{\text{F}}$ provides three additional differentiation constructs, inspired by AD tools such as DiffSharp [28]: 1) `diff` computes the derivative a function, from a real number to a real number, with respect to its input, 2) `grad` computes the gradient of a function, from a vector of real numbers to a real number, with respect to its input vector, and 3) `jacob` computes the Jacobian matrix of a vector-valued function, a function from a vector of real numbers to a vector of real numbers, with respect to its input vector. Figure 6.4 demonstrates how these high-level differentiation constructs are defined in terms of the source-to-source AD transformation construct $\mathscr{D}$.

Example 2 (Continued). For the previous example, if we would like to use the `diff` construct, first we have to define the following function:

```
g = fun x -> cos(x)
```

The derivative of this function at point a is represented as follows:

Algorithm 1 A pseudo-code implementation of the `deriv` construct.

```
1: // Returns an expression including both the original and the derivative computation.
2: function DERIV(e, x)
3:     args ← ∅
4:     f ← e
5:     for all v ← FREEVARS(e) do
6:         f ← fun v -> f
7:         args ← args ∪ DUAL(v, if(v = x) then ONEHOT(v) else ZERO(v))
8:     df ← (𝒟⟦f⟧) args
9:     if TYPE(x) = Double then
10:        return df
11:    else if TYPE(x) = Vector then
12:        return build (length x) (fun ri -> df)
13:    else if TYPE(x) = Matrix then
14:        return build (matrixRows x) (fun ri -> build (matrixCols x) (fun ci -> df))
15: end function
16: // Returns the dual number encoding of the two input expressions.
17: function DUAL(e1, e2)
18:    if TYPE(e1) = Double then
19:        return (e1, e2)
20:    else if TYPE(e1) = Vector then
21:        return vectorZip e1 e2
22:    else if TYPE(e1) = Matrix then
23:        return matrixZip e1 e2
24: end function
25: // Returns a zero scalar, vector, or matrix expression based on the type of input.
26: function ZERO(e)
27:    if TYPE(e) = Double then
28:        return 0
29:    else if TYPE(e) = Vector then
30:        return vectorZeros (length e)
31:    else if TYPE(e) = Matrix then
32:        return matrixZeros (matrixRows e) (matrixCols e)
33: end function
34: // Returns a one-hot encoding scalar, vector, or matrix expression.
35: function ONEHOT(e)
36:    if TYPE(e) = Double then
37:        return 1
38:    else if TYPE(e) = Vector then
39:        return vectorHot (length e) ri
40:    else if TYPE(e) = Matrix then
41:        return matrixHot (matrixRows e) (matrixCols e) ri ci
42: end function
```

Oper.	Type	Definition
`diff`	(`Double⇒Double`) ⇒ `Double⇒DoubleD`	`fun f x ->` $\mathscr{D}[\![f]\!]$ (x, 1)

`grad`	(`Vector⇒Double`) ⇒`Vector⇒VectorD`	`fun f v ->`
		`build (length v) (fun i ->`
		$\mathscr{D}[\![f]\!]$ (vectorZip v (vectorHot (`length` v) i))
		)

`jacob`	(`Vector⇒Vector`) ⇒`Vector⇒MatrixD`	

Figure 6.4 – High-Level Differentiation API for $\widetilde{\mathrm{F}}$.

Input Type \ Output Type	Scalar	Vector	Matrix
Scalar	`diff`	`vdiff`	`mdiff`
Vector	`grad`	`jacob`	–
Matrix	`mgrad`	–	–

Table 6.1 – Different types of matrix derivatives.

`snd` ((`diff` g) a)

which is expanded to the following program:

`snd` ($\mathscr{D}[\![g]\!]$ (a, 1))

△

Table 6.1 summarizes different matrix derivatives, and how they can be computed using our high-level API. Note that the definition of `vdiff` and `mdiff` is similar to `diff`, and the definition of `mgrad` is similar to `grad` and `jacob` (cf. Figure 6.4). Note that the `deriv` construct subsumes all these operators.

One key advantage of defining different matrix derivatives in terms of automatic differentiation is that one no longer needs to define the matrix calculus derivative rules for all different combinations shown in Table 6.1. Instead these rules can be deduced automatically from the automatic differentiation rules defined for scalar values. Moreover, even the algebraic identities for matrix derivative can be deduced by using the simplification rules presented in Section 6.4.

Next, we present the source code transformation required for applying automatic differentiation rules.

6.3.2 Source-to-Source Automatic Differentiation

$\mathrm{d}\widetilde{\mathrm{F}}$ relies on source-to-source translation for implementing forward-mode automatic differentiation. Each expression is converted into an expression containing both the original

computation, together with the derivative computation, a.k.a. the dual number technique. The scalar expressions are transformed into a pair of values, the original computation and the derivative computation. The vector expressions are transformed into vectors containing tuple expressions, instead of scalar expressions. The situation is similar for higher-rank tensors such as matrices.

The rules for automatic differentiation are demonstrated in Figure 6.5. $\mathscr{D}[\![\text{e}]\!]$ specifies the AD translation for expression e. A variable y is translated as $\vec{\text{y}}$, emphasizing that the translated variable keeps the derivative part as well (D-Abs, D-Var, and D-Let). $\mathscr{V}[\![\text{e}]\!]$ is a shorthand for extracting the original computation from the translated term $\mathscr{D}[\![\text{e}]\!]$, while $\mathscr{T}[\![\text{e}]\!]$ is a shorthand for accessing the derivative part.

Constructing an array is differentiated as an array with the same size, however, the way that each element of the array is constructed is differentiated (D-Build). Differentiating an iteration results in an iteration with the same number of iterations, and with the initial state and the next state function both differentiated (D-IFold). The differentiation of the length and indexing an array, is the same as the length and indexing the differentiated array, respectively (D-Length and D-Get).

Differentiating a pair of elements results in the pair of differentiated elements (D-Pair). Similarly, differentiating the projection of a pair, is the projection of the differentiated pair (D-Fst, D-Snd). For other scalar-valued functions, the differentiation rules are similar to the corresponding rules in mathematics.

Example 2 (Continued). In the previous example, based on the automatic differentiation rules, the differentiated program would be as follows:

$\vec{\text{g}}$ = `fun` $\vec{\text{x}}$ `->` `-snd` ($\vec{\text{x}}$) * sin(`fst` ($\vec{\text{x}}$))

Based on the definition of the `diff` construct, we have to use the AD version of the function (i.e., g) and assign 1 to the derivative part of the input. So the value of cos' for the input a is computed as follows:

`snd` ((`diff` g) a) $\leadsto$ `snd` ($\mathscr{D}[\![\text{g}]\!]$ (a, 1)) $\leadsto$ `snd` ($\vec{\text{g}}$ (a, 1)) $\leadsto$

`-snd` ((a, 1)) * sin(`fst` ((a, 1))) $\leadsto$ -1 * sin(a) $\leadsto$ -sin(a)

△

Similarly, we can compute the partial derivatives of a given function, by setting the desired derivative part to one, and the rest of derivatives to zero. This process is illustrated in the next example.

Example 3. Assume that we would like to compute the partial derivative of the expression a ∗

b with respect to a, which is represented as follows in $\widetilde{\text{F}}$:

`snd` (`deriv` (a ∗ b) a)

This expression is expanded as follows:

`snd` ($\mathscr{D}$⟦`fun a b -> a ∗ b`⟧ (a, 1) (b, 0))

Note that the derivative part of the second input is set to 0. Similar to the previous example, the result is as follows:

`snd` ((`fun` $\vec{\text{a}}$ $\vec{\text{b}}$ `->` (`fst` ($\vec{\text{a}}$)∗`fst` ($\vec{\text{b}}$), `fst` ($\vec{\text{a}}$)∗`snd` ($\vec{\text{b}}$) + `snd` ($\vec{\text{a}}$)∗`fst` ($\vec{\text{b}}$))) (a, 1) (b, 0))

which is evaluated as follows:

`snd` ((a ∗ b, 1 ∗ b + a ∗ 0)) $\rightsquigarrow$ 1 ∗ b + a ∗ 0 $\rightsquigarrow$ b

△

It is important to note that $\text{d}\widetilde{\text{F}}$ performs many of the evaluation steps shown for the previous examples during compilation time, i.e., performs partial evaluation.

6.3.3 Perturbation Confusion and Nested Differentiation

In several problems such as computing the Hessian matrix, one requires to compute the differentiation of a differentiated program. In such cases, one should be careful on dealing with tangent parts. We demonstrate this problem in the next example.

Example. Consider the following expression:

$$\frac{\partial(x\frac{\partial x+y}{\partial y})}{\partial x}$$

This expression should be evaluated to 1 at every point. However, an AD tool can mistakenly evaluate this expression to 2. This is because of confusing the tangent part (perturbation) of the free variable x with the tangent of the variable y, while computing the inner derivative. This is known as the *perturbation confusion* problem in the AD literature.

We show how $\text{d}\widetilde{\text{F}}$ avoids this problem by using the `deriv` macro. This expression is implemented as follows in the $\widetilde{\text{F}}$ language:

$$
\begin{array}{lrcl}
\text{(D-App)} & \mathscr{D}[\![\texttt{e}_0\ \texttt{e}_1]\!] & = & (\mathscr{D}[\![\texttt{e}_0]\!])\ (\mathscr{D}[\![\texttt{e}_1]\!]) \\
\text{(D-Abs)} & \mathscr{D}[\![\texttt{fun x -> e}]\!] & = & \texttt{fun}\ \vec{\texttt{x}}\ \texttt{->}\ \mathscr{D}[\![\texttt{e}]\!] \\
\text{(D-Var)} & \mathscr{D}[\![\texttt{y}]\!] & = & \vec{\texttt{y}} \\
\text{(D-Let)} & \mathscr{D}[\![\texttt{let x = e}_1\ \texttt{in}\ \texttt{e}_2]\!] & = & \texttt{let}\ \vec{\texttt{x}} = \mathscr{D}[\![\texttt{e}_1]\!]\ \texttt{in} \\
 & & & \quad \mathscr{D}[\![\texttt{e}_2]\!] \\
\text{(D-If)} & \mathscr{D}[\![\texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3]\!] & = & \texttt{if}\ (\texttt{fst}\ \mathscr{D}[\![\texttt{e}_1]\!])\ \texttt{then}\ \mathscr{D}[\![\texttt{e}_2]\!]\ \texttt{else}\ \mathscr{D}[\![\texttt{e}_3]\!] \\
 & & & \\
\text{(D-Build)} & \mathscr{D}[\![\texttt{build}\ \texttt{e}_0\ \texttt{e}_1]\!] & = & \texttt{build}\ (\texttt{fst}\ \mathscr{D}[\![\texttt{e}_0]\!])\ (\texttt{fun i ->}\ (\mathscr{D}[\![\texttt{e}_1]\!])\ (\texttt{i}, 0)) \\
\text{(D-IFold)} & \mathscr{D}[\![\texttt{ifold}\ \texttt{e}_0\ \texttt{e}_1\ \texttt{e}_2]\!] & = & \texttt{ifold}\ (\texttt{fun x i ->} \\
 & & & \quad (\mathscr{D}[\![\texttt{e}_0]\!])\ \texttt{x}\ (\texttt{i}, 0))\ \mathscr{D}[\![\texttt{e}_1]\!]\ (\texttt{fst}\ \mathscr{D}[\![\texttt{e}_2]\!]) \\
\text{(D-Get)} & \mathscr{D}[\![\texttt{e}_0[\texttt{e}_1]]\!] & = & (\mathscr{D}[\![\texttt{e}_0]\!])[\texttt{fst}\ \mathscr{D}[\![\texttt{e}_1]\!]] \\
\text{(D-Length)} & \mathscr{D}[\![\texttt{length}\ \texttt{e}_0]\!] & = & (\texttt{length}\ \mathscr{D}[\![\texttt{e}_0]\!], 0) \\
 & & & \\
\text{(D-Pair)} & \mathscr{D}[\![(\texttt{e}_0, \texttt{e}_1)]\!] & = & (\mathscr{D}[\![\texttt{e}_0]\!], \mathscr{D}[\![\texttt{e}_1]\!]) \\
\text{(D-Fst)} & \mathscr{D}[\![\texttt{fst}\ \texttt{e}_0]\!] & = & \texttt{fst}\ (\mathscr{D}[\![\texttt{e}_0]\!]) \\
\text{(D-Snd)} & \mathscr{D}[\![\texttt{snd}\ \texttt{e}_0]\!] & = & \texttt{snd}\ (\mathscr{D}[\![\texttt{e}_0]\!]) \\
 & & & \\
\text{(D-NumV)} & \texttt{e: Num} \vdash \mathscr{V}[\![\texttt{e}]\!] & = & \texttt{fst}\ \mathscr{D}[\![\texttt{e}]\!] \\
\text{(D-NumT)} & \texttt{e: Num} \vdash \mathscr{T}[\![\texttt{e}]\!] & = & \texttt{snd}\ \mathscr{D}[\![\texttt{e}]\!] \\
 & & & \\
\text{(D-Neg)} & \mathscr{D}[\![\texttt{-e}_1]\!] & = & (\ \texttt{-}\mathscr{V}[\![\texttt{e}_1]\!]\ ,\ \texttt{-}\mathscr{T}[\![\texttt{e}_1]\!]\ ) \\
\text{(D-Add)} & \mathscr{D}[\![\texttt{e}_1 + \texttt{e}_2]\!] & = & (\ \mathscr{V}[\![\texttt{e}_1]\!]+\mathscr{V}[\![\texttt{e}_2]\!]\ ,\ \mathscr{T}[\![\texttt{e}_1]\!]+\mathscr{T}[\![\texttt{e}_2]\!]\ ) \\
\text{(D-Mult)} & \mathscr{D}[\![\texttt{e}_1 * \texttt{e}_2]\!] & = & (\ \mathscr{V}[\![\texttt{e}_1]\!]*\mathscr{V}[\![\texttt{e}_2]\!]\ ,\ \mathscr{T}[\![\texttt{e}_1]\!]*\mathscr{V}[\![\texttt{e}_2]\!] + \mathscr{V}[\![\texttt{e}_1]\!]*\mathscr{T}[\![\texttt{e}_2]\!]\ ) \\
\text{(D-Div)} & \mathscr{D}[\![\texttt{e}_1 / \texttt{e}_2]\!] & = & (\ \mathscr{V}[\![\texttt{e}_1]\!]/\mathscr{V}[\![\texttt{e}_2]\!]\ , \\
 & & & \quad (\mathscr{T}[\![\texttt{e}_1]\!]*\mathscr{V}[\![\texttt{e}_2]\!] - \mathscr{V}[\![\texttt{e}_1]\!]*\mathscr{T}[\![\texttt{e}_2]\!])\ /\ (\mathscr{V}[\![\texttt{e}_2]\!]**2)\ ) \\
\text{(D-Pow)} & \mathscr{D}[\![\texttt{e}_1 ** \texttt{e}_2]\!] & = & (\ \mathscr{V}[\![\texttt{e}_1]\!]**\mathscr{V}[\![\texttt{e}_2]\!]\ ,\ (\mathscr{V}[\![\texttt{e}_2]\!] * \mathscr{T}[\![\texttt{e}_1]\!] / \mathscr{V}[\![\texttt{e}_1]\!] + \\
 & & & \quad \log(\mathscr{V}[\![\texttt{e}_1]\!])*\mathscr{T}[\![\texttt{e}_2]\!])\ *\ (\mathscr{V}[\![\texttt{e}_1]\!]**\mathscr{V}[\![\texttt{e}_2]\!])\ ) \\
\text{(D-Sin)} & \mathscr{D}[\![\sin(\texttt{e}_1)]\!] & = & (\ \sin(\mathscr{V}[\![\texttt{e}_1]\!])\ ,\ \mathscr{T}[\![\texttt{e}_1]\!] * \cos(\mathscr{V}[\![\texttt{e}_1]\!])\ ) \\
\text{(D-Cos)} & \mathscr{D}[\![\cos(\texttt{e}_1)]\!] & = & (\ \cos(\mathscr{V}[\![\texttt{e}_1]\!])\ ,\ \texttt{-}\mathscr{T}[\![\texttt{e}_1]\!] * \sin(\mathscr{V}[\![\texttt{e}_1]\!])\ ) \\
\text{(D-Tan)} & \mathscr{D}[\![\tan(\texttt{e}_1)]\!] & = & (\ \tan(\mathscr{V}[\![\texttt{e}_1]\!])\ ,\ \mathscr{T}[\![\texttt{e}_1]\!]\ /\ (\cos(\mathscr{V}[\![\texttt{e}_1]\!])\ **\ 2)\ ) \\
\text{(D-Log)} & \mathscr{D}[\![\log(\texttt{e}_1)]\!] & = & (\ \log(\mathscr{V}[\![\texttt{e}_1]\!])\ ,\ \mathscr{T}[\![\texttt{e}_1]\!]\ /\ \mathscr{V}[\![\texttt{e}_1]\!]\ ) \\
\text{(D-Exp)} & \mathscr{D}[\![\exp(\texttt{e}_1)]\!] & = & (\ \exp(\mathscr{V}[\![\texttt{e}_1]\!])\ ,\ \mathscr{T}[\![\texttt{e}_1]\!]\ *\ \exp(\mathscr{V}[\![\texttt{e}_1]\!])\ ) \\
 & & & \\
\text{(DT-Fun)} & \mathscr{D}_\texttt{T}[\![\texttt{T}_1 \Rightarrow \texttt{T}_2\ ]\!] & = & \mathscr{D}_\texttt{T}[\![\texttt{T}_1]\!] \Rightarrow \mathscr{D}_\texttt{T}[\![\texttt{T}_2]\!] \\
\text{(DT-Exp)} & \mathscr{D}_\texttt{T}[\![\texttt{Num}]\!] & = & \texttt{Num} \times \texttt{Num} \\
\text{(DT-Arr)} & \mathscr{D}_\texttt{T}[\![\texttt{Array<M>}]\!] & = & \texttt{Array<}\mathscr{D}_\texttt{T}[\![\texttt{M}]\!]\texttt{>} \\
\text{(DT-Pair)} & \mathscr{D}_\texttt{T}[\![\texttt{M}_1 \times \texttt{M}_2]\!] & = & \mathscr{D}_\texttt{T}[\![\texttt{M}_1]\!] \times \mathscr{D}_\texttt{T}[\![\texttt{M}_2]\!]
\end{array}
$$

Figure 6.5 – Automatic Differentiation Rules for $\widetilde{\text{F}}$ Expressions.

```
fun x y ->
  snd (
    deriv (x * (snd (
      deriv (x + y) y
    ))) x
  )
```

After expanding the inner `deriv` macro, the following expression is derived:

```
fun x y ->
  snd (
    deriv (x * (snd (
      (fun x̄ ȳ -> (fst (x̄) + fst (ȳ), snd (x̄) + snd (ȳ))) (x, 0) (y, 1)
    ))) x
  )
```

After partially evaluating the inner expression we have:

```
fun x y ->
  snd (
    deriv x x
  )
```

Expanding this `deriv` macro results in the following expression:

```
fun x y ->
  snd (
    (fun x̄ -> x̄) (x, 1)
  )
```

This expression equivalent to the following expression after partial evaluation:

```
fun x y ->
  1
```

△

Correctly handling the perturbation confusion problem is an important feature, enabling $d\widetilde{F}$ to efficiently handle nested differentiation constructs such as computing the Hessian matrix. We plan to investigate the support for the Hessian matrix for the future.

Next, we give more details on the optimizations and simplifications offered by $d\widetilde{F}$.

6.4 Efficient Differentiation

In this section, we show how $d\widetilde{F}$ achieves efficient differentiable programming. First, we show several transformation rules applicable on $\widetilde{F}$ expressions. Then, we show how we generate C code from $\widetilde{F}$ expressions for a more efficient memory management.

e + 0 = 0 + e	⇝	e
e ∗ 1 = 1 ∗ e	⇝	e
e ∗ 0 = 0 ∗ e	⇝	0
e + -e = e - e	⇝	0
$e_0 * e_1 + e_0 * e_2$	⇝	$e_0 * (e_1 + e_2)$

(a) Ring-Structure Rules

(`fun` x `->` e_0) e_1	⇝	$e_0[x \mapsto e_1]$
let x = e_0 in e_1	⇝	$e_1[x \mapsto e_0]$
let x = let y = e_0 in e_1 in e_2	⇝	let y = e_0 in let x = e_1 in e_2
f(let x = e_0 in e_1)	⇝	let x = e_0 in f(e_1)

(b) λ-Calculus Rules

(`build` e_0 e_1)[e_2]	⇝	e_1 e_2
`length` (`build` e_0 e_1)	⇝	e_0

(c) Fusion Rules

`fst` (e_0, e_1)	⇝	e_0
`snd` (e_0, e_1)	⇝	e_1

(d) Tuple Partial Evaluation Rules

`ifold` f z 0	⇝	z
`ifold` (`fun` a i `->` a) z n	⇝	z
`ifold` f z n	⇝	`ifold` (`fun` a i `->` f a (i+1)) (f z 0) (n - 1)
`ifold` (`fun` a i `->` `if` (i = j) `then` f a i `else` a) z n	⇝	f z j

(e) Iteration Rules

`if` true `then` e_1 `else` e_2	⇝	e_1
`if` false `then` e_1 `else` e_2	⇝	e_2
`if` e_0 `then` e_1 `else` e_1	⇝	e_1
f (`if` e_0 `then` e_1 `else` e_2)	⇝	`if` e_0 `then` f (e_1) `else` f (e_2)
`if` e_0 `then` e_1 `else` e_2	⇝	`if` e_0 `then` $e_1[e_0 \mapsto \text{true}]$ `else` $e_2[e_0 \mapsto \text{false}]$

(f) Conditional Rules

`ifold` (`fun` a i `->` (f (`fst` a) i, g (`snd` a) i)) (z1, z2) n ⇝ (`ifold` f z1 n, `ifold` g z2 n)

(g) Loop Fission

Figure 6.6 – Optimizations for $\widetilde{\mathrm{F}}$.

6.4.1 Transformation Rules

There are various algebraic identities that one can define for $\widetilde{\mathrm{F}}$. Based on these identities, differentiated programs can be heavily optimized. Figure 6.6 shows a set of optimizations defined for $\widetilde{\mathrm{F}}$.

There are various optimizations defined for scalar operations based on the ring structure of addition and multiplication, which are shown in Figure 6.6a. Note that other ring-based algebraic identities, such as associativity and commutativity, do not appear directly in the list of rules that $\mathrm{d}\widetilde{\mathrm{F}}$ applies. This is because they do not necessarily improve the performance, unless they are combined with other rewrite rules.

As $\widetilde{\mathrm{F}}$ is based on λ-calculus, all partial evaluation rules for this calculus come for free. Furthermore, the optimizations defined in the literature for let-binding can also be used. Finally, partial evaluation rules for conditionals are also available. Figure 6.6b shows this set of rules.

As the vector constructs of $\widetilde{\mathrm{F}}$ are based on pull arrays, one can use the pull-array fusion rules for

removing unnecessary intermediate vectors and matrices. The two fusion rules for pull-arrays are shown in Figure 6.6c.

In addition, many intermediate tuples resulting from the dual number technique of AD can be removed by using partial evaluation. Figure 6.6d shows the partial evaluation rules for removing the intermediate tuples which are followed by a projection.

Partially evaluating the tuples across the boundary of a loop requires a sophisticated analysis of the body of the loop. To simplify this task, we perform loop fission for the loops that return a tuple of values. This is possible only when different elements of the tuple are computed independently in different iterations of the loop. Figure 6.6g shows how loop fission turns an iteration creating a pair of elements into a pair of two iterations constructing independently the elements of that pair. After performing this optimization, if we are interested only in a particular element of the result tuple, other loops corresponding to irrelevant elements are removed by partial evaluation.

Based on these rewrite rules, $\mathrm{d}\widetilde{\mathrm{F}}$ derives well-known matrix calculus rules, without requiring to add a rewrite rule in the level of matrices (i.e., $\widetilde{\mathrm{M}}$). However, as we will see, the order in which these rewrite rules should be applied can become tricky and for the moment are defined manually in $\mathrm{d}\widetilde{\mathrm{F}}$. We leave an automatic way of inferring a good sequence of rewrite rules for the future work.

The next example, shows how $\mathrm{d}\widetilde{\mathrm{F}}$ can derive a well-known matrix identity by using a sequence of transformation rules defined in this section.

Example 4. Based on matrix calculus derivative rules, it is known that $\frac{\partial v_1 \cdot v_2}{\partial v_1} = v_2$, where $\cdot$ is the vector dot product operator. We would like to show how $\mathrm{d}\widetilde{\mathrm{F}}$ can deduce the same algebraic identity. The differentiation of dot product of two vectors is represented as follows:

```
fun v1 v2 ->
  vectorMap (deriv (vectorDot v1 v2) v1) snd
```

This expression is expanded as follows:

```
fun v1 v2 ->
  vectorMap (
    build (length v1) (fun i ->
      𝒟⟦fun v1 v2 -> vectorDot v1 v2⟧
        (vectorZip v1 (vectorHot (length v1) i))
        (vectorZip v2 (vectorZeros (length v2))))
  ) snd
```

After inlining the definition of vectorMap (cf. Figure 5.2) and applying the fusion rule (cf. Figure 6.6c), the following program is produced:

```
fun v1 v2 ->
  build (length v1) (fun i ->
    snd (𝒟⟦fun v1 v2 -> vectorDot v1 v2⟧
          (vectorZip v1 (vectorHot (length v1) i))
          (vectorZip v2 (vectorZeros (length v2)))))
```

After inlining the definition of vectorDot, vectorZip, vectorHot, and vectorZeros, and again applying the fusion rule, we have:

```
fun v1 v2 ->
  build (length v1) (fun i ->
    snd (𝒟⟦fun v1 v2 -> ifold (fun s j -> s+v1[j]*v2[j]) 0 (length v1)⟧
          (build (length v1) (fun j -> (v1[j], if(i=j) then 1 else 0)))
          (build (length v2) (fun j -> (v2[j], 0)))))
```

After applying AD transformation rules (cf. Figure 6.5), and partial evaluation rules (cf. Figure 6.6) the following program is derived:

```
fun v1 v2 ->
  build (length v1) (fun i ->
    snd (fun v1⃗ v2⃗ -> ifold (fun s j ->
            ( (fst s) + (fst v1⃗[j]) * (fst v2⃗[j]) ,
              (snd s) + (fst v1⃗[j]) * (snd v2⃗[j]) + (snd v1⃗[j]) * (fst v2⃗[j]) )
            ) (0, 0) (length v1⃗))
          (build (length v1) (fun j -> (v1[j], if(i=j) then 1 else 0)))
          (build (length v2) (fun j -> (v2[j], 0)))))
```

After further applying β-reduction (cf. Figure 6.6b), tuple partial evaluation (cf. Figure 6.6b), and loop fusion the following program is generated:

```
fun v1 v2 ->
  build (length v1) (fun i ->
    snd (ifold (fun s j ->
            ( (fst s) + v1[j] * v2[j] ,
              (snd s) + v1[j] * 0 + (if (i=j) then 1 else 0) * v2[j] )
            ) (0, 0) (length v1))
```

Now we apply loop fission (cf. Figure 6.6g), conditional rules (cf. Figure 6.6f), and several other simplification rules:

```
fun v1 v2 ->
  build (length v1) (fun i ->
    snd (
      ifold (fun s j -> s + v1[j] * v2[j]) 0 (length v1) ,
      ifold (fun s j -> if (i=j) then s + v2[j] else s) 0 (length v1)
    ) )
```

Note that applying the loop fission rule, does not necessarily improve the performance; it is only after performing tuple partial evaluation rules that the iteration responsible for the original computation is removed and the performance is improved. Thus, the strategy for applying rewrite rules can become tricky, and for this example, we manually specify the sequence of transformations that should be applied. After applying the partial evaluation rule, the following program is derived:

```
fun v1 v2 ->
  build (length v1) (fun i ->
    (ifold (fun s j ->
      if(i = j) then
        (s + v2[j])
      else
        s) 0 (length v1)))
```

By using the optimization that turns single access iterations into a single statement (cf. Figure 6.6e), $\mathrm{d}\widetilde{\mathrm{F}}$ produces the following program:

```
fun v1 v2 ->
  build (length v1) (fun i -> v2[i])
```

This program is equivalent to $v2$ if the size of the two input vectors are the same (i.e., `length` $v1$ = `length` $v2$). Otherwise, the input program is ill-formed.

△

Next, we show the power of $\mathrm{d}\widetilde{\mathrm{F}}$ in deriving a matrix calculus identity for gradient of matrices.

Example 5. By using the same set of optimizations, $\mathrm{d}\widetilde{\mathrm{F}}$ can deduce the identity $\frac{\partial tr\left(M\right)}{\partial M} = I$. First, we start from the representation of this gradient in $\widetilde{\mathrm{F}}$:

```
fun m ->
  build (length m) (fun i ->
    build (length m[0]) (fun j ->
      snd (𝒟⟦matrixTrace⟧ (matrixZip m (matrixHot (length m) (length m[0]) i j)))))
```

After applying the AD transformations and the optimizations presented in this section, the following program is produced:

```
fun m ->
  build (length m) (fun i ->
    build (length m[0]) (fun j ->
      if (j = i) then 1 else 0))
```

If the rows and columns of the input matrix are equal, this program represents the identity matrix with the same dimensions as the input matrix.

△

Similarly, $\widetilde{\mathrm{dF}}$ automatically discovers the following identity if A is independent of M: $\frac{\partial tr\left(MA\right)}{\partial M} = A^T$. Now we return to the example shown in the beginning of this chapter.

Example 1 (Continued). If we have a matrix M and two vectors u and v (which are represented as row matrices and are independent of M), using matrix calculus one can prove that $\frac{\partial\left(uMv^T\right)}{\partial M} = u^T v$. First, we start by a partially inlined representation of this program in $\widetilde{\mathrm{F}}$:

```
let f = fun u M v ->
  let m =
    matrixMult
      (build 1 (fun i -> u))
      (matrixMult M
        (matrixTranspose (build 1 (fun i -> v))))
  m[0][0]
fun u M v ->
  (build (length M) (fun i ->
    (build (length M[0]) (fun j ->
      (snd (𝒟⟦f⟧
        (vectorZip v (vectorZeros (length v)))
        (matrixZip M (matrixHot (length M) (length M[0]) i j))
        (vectorZip v (vectorZeros (length v))))))))))
```

Note that the function f is returning the only scalar element of the 1-by-1 matrix uMv^T. After performing loop fusion, loop fission and partial evaluation the following program is derived:

```
fun u M v ->
  build (length M) (fun i ->
    build (length M[0]) (fun j ->
      u[i] * v[j]))
```

This program is equivalent to $u^T v$ if the input program is well formed, i.e., the number of rows and columns of M are the same as the length of u and v, respectively.

△

6.4.2 Code Generation

After applying the optimizations mentioned in the previous section, one can further improve the efficiency by generating programs in a low-level language with manual memory management. This way, the overhead of garbage collection can be removed. Furthermore, by using stack-discipline memory management techniques such as Destination-Passing Style (DPS) [299], one can benefit from efficient bump memory allocation instead of using the expensive `malloc` and `free` calls.

Example 1 (Continued). The generated C code for the optimized differentiated program is as follows:

```
matrix uMv_d(storage s, vector u, matrix M, vector v) {
  matrix res = (matrix)s;
  for(int r = 0; r < M->rows; r++) {
    for(int c = 0; c < M->cols; c++) {
      res->elems[r][c] = u->elems[r] * v->elems[c];
    }
  }
  return res;
}
```

The parameter `s` is the storage area allocated for storing the result matrix.

△

Up to now, we have only seen the cases where only the derivative part of the program was of interest. If we are interested in the original part of the program as well (e.g., the intermediate vectors cannot be fused), we need to store both the original and derivative parts. In such cases, the differentiated vectors, which are represented as arrays of tuples, can be transformed into a more efficient data layout. The well-known array of structs (AoS) to struct of arrays (SoA) transformation represents differentiated vectors as a tuple of two numeric arrays. Further partial evaluation can remove the unnecessary decoupled numeric arrays.

6.5 Implementation

In this section, we discuss the implementation of $\mathrm{d}\widetilde{\mathrm{F}}$. Figure 6.7 gives an overview of the architecture of $\mathrm{d}\widetilde{\mathrm{F}}$. Next, we give more details on the compilation process.

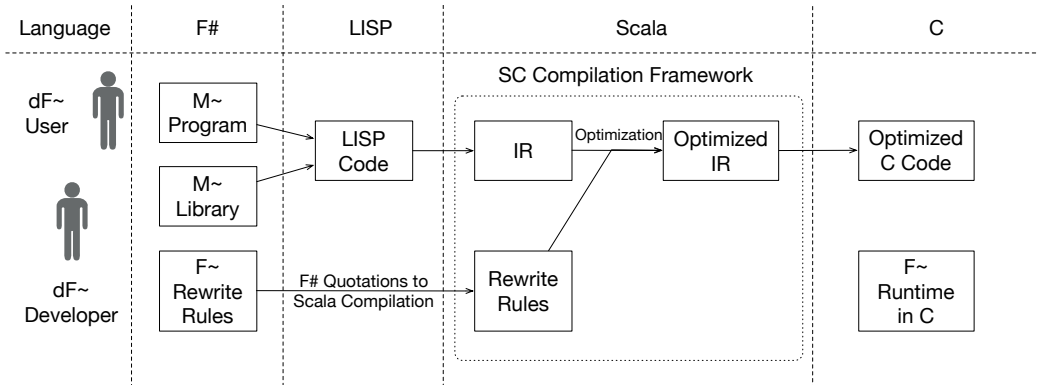


Figure 6.7 – The architecture of d$\widetilde{\text{F}}$.

6.5.1 Compilation Process

As we discussed in Section 5.4, $\widetilde{\text{F}}$ is a subset of F#. As $\widetilde{\text{M}}$ is also embedded inside $\widetilde{\text{F}}$, the programs written in both languages are valid F# programs. We provide input $\widetilde{\text{M}}$ and $\widetilde{\text{F}}$ programs together with the $\widetilde{\text{M}}$ library in F#. These programs can be executed, as they are, as normal F# programs. However, running as normal an F# program causes missing the benefits gained by using the transformations offered by d$\widetilde{\text{F}}$.

In order to benefit from the compilation facilities provided by d$\widetilde{\text{F}}$, the F# programs are converted to LISP [314] code. One clear advantage of using LISP as the intermediate exchanging language is its functional nature and its simplicity for parsing.

The generated LISP programs are passed to SC, an extensible compilation framework implemented in Scala (cf. Chapter 8). SC parses the LISP programs and constructs SC IR nodes. Then, it applies the transformation rules in the order manually specified by the developer, and produces optimized IR nodes. Finally, SC generates C code from the optimized IR.

6.5.2 Rewrite Rules

Many rewrite rules do not need static analysis over code for checking whether they can be applied to a given expression. Such rewrite rules can be implemented as F# quotations. Figure 6.8 shows the implementation of the ring-based rewrite rules (which were presented in Figure 6.6a) using F# quotations.

These rewrite rules are converted into Scala code that uses the API provided by SC. The compiler framework presented in Section 5.4, which was implemented in F#, is extended with a Scala code generator that converts the rewrite rules in F# quotations into Scala pattern matching code. As an example, the following F# quotation:

```
<@ %a + 0 <==> %a @>
```

```
<@        %a + 0          <==>          %a          @>
<@        0 + %a          <==>          %a          @>
<@        %a * 1          <==>          %a          @>
<@        1 * %a          <==>          %a          @>
<@        %a * 0          <==>          0           @>
<@        0 * %a          <==>          0           @>
<@      %a + (-%b)        <==>       %a - %b        @>
<@        %a - %a         <==>          0           @>
<@  %a * %b + %a * %c     <==>   %a * (%b + %c)     @>
```

Figure 6.8 – Ring-structure rules implemented using F# quotations.

is converted to the following Scala code:

```
exp match {
  case Add(a, Const(0)) => Some(a)
  case _ => None
}
```

The rewrite rules that need program analysis for checking their applicability, are not expressible as F# quotations. For example, loop fission rules (cf. Figure 6.6g) require analysing the code for checking the usage pattern of tuples inside the loop. In such cases, the given expression needs to be inspected for checking the applicability of the rewrite rule. These rewrite rules are implemented directly in Scala using the SC rewriting API.

Finally, SC also has the power for visualization of the application of rewrite rules in a web-based frontend. This is thanks to the Scala.js [93] compiler, which compiles the SC compilation framework code base into JavaScript code. Thus, it is possible to execute the SC compiler in a web browser, and visualize the compilation process.

6.6 Experimental Results

In this section, we show how $\mathrm{d}\widetilde{\mathrm{F}}$ performs in practice. We show the performance of the differentiated code for two real-world machine learning and computer vision applications.

Experimental Setup. We have performed the experiments using an iMac machine equipped with an Intel Core i5 CPU running at 2.7GHz, 32GB of DDR3 RAM at 1333Mhz. The operating system is OS X 10.13.1. We use CLang 900.0.39.2 for compiling the generated C code, and Python 2.7.12 for running the Python code.

6.6.1 Micro Benchmarks

The micro benchmarks used for our experiments consist of the following vector expressions: 1) gradient of dot product of two vectors with respect to the first vector (which is a Jacobian

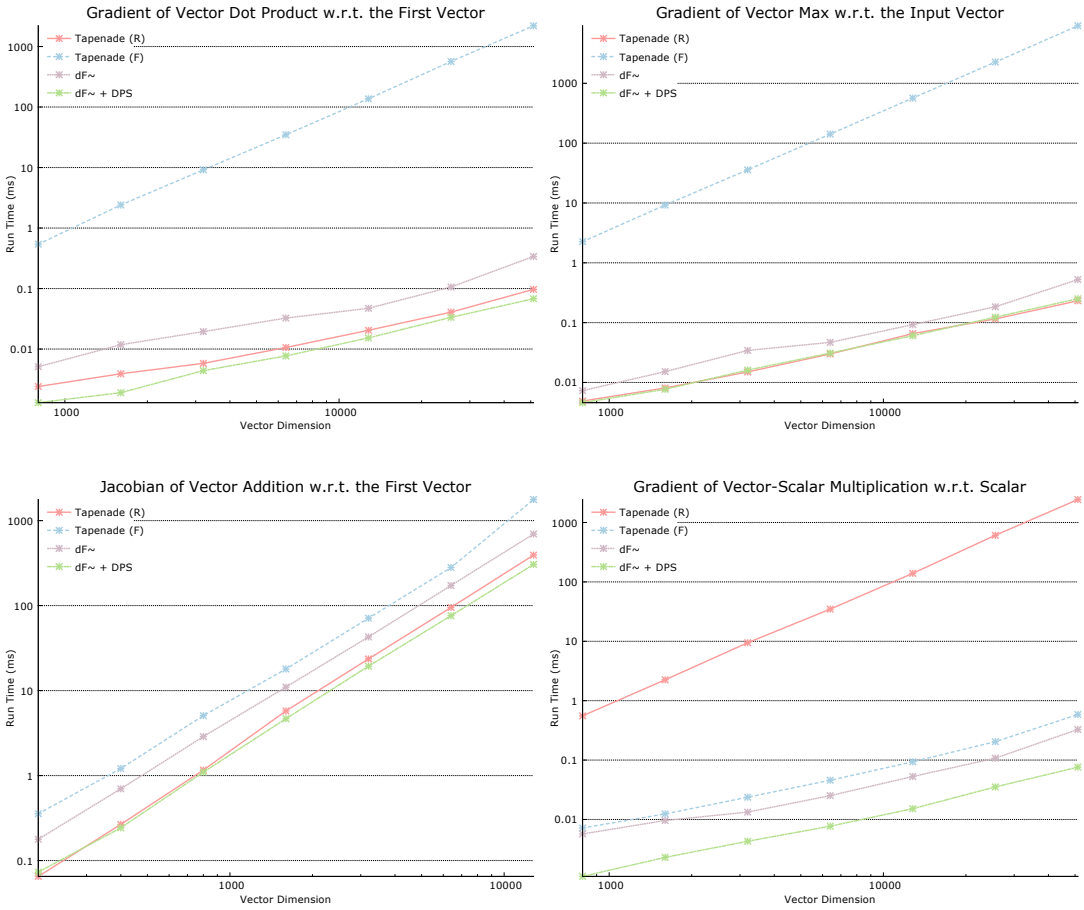


Figure 6.9 – Performance results for Micro Benchmarks.

matrix with a single row), 2) gradient of the maximum value of a vector with respect to the input vector (which is a Jacobian matrix with a single row), 3) gradient of addition of two vectors with respect to the first vector (which is a Jacobian matrix), and 4) gradient of the multiplication of a vector with a scalar value with respect to the scalar value (which is a Jacobian matrix with a single column).

Figure 6.9 shows the performance results for the mentioned micro benchmarks for $\widetilde{\mathrm{dF}}$ and both forward and reverse-mode of Tapenade. In all cases, $\widetilde{\mathrm{dF}}$ outperforms or performs as good as both forward and reverse-mode of Tapenade. The performance is improved further when the generated C code uses Destination-Passing Style (DPS) [299] for stack-discipline memory management.

As in the first two cases the Jacobian matrix is a row vector, reverse-mode AD computes the whole Jacobian matrix in a single backward pass. However, forward-mode AD needs to iterate over each column to compute the corresponding derivative value. For the case of the addition of two vectors, as the Jacobian matrix is a square matrix, reverse-mode AD and

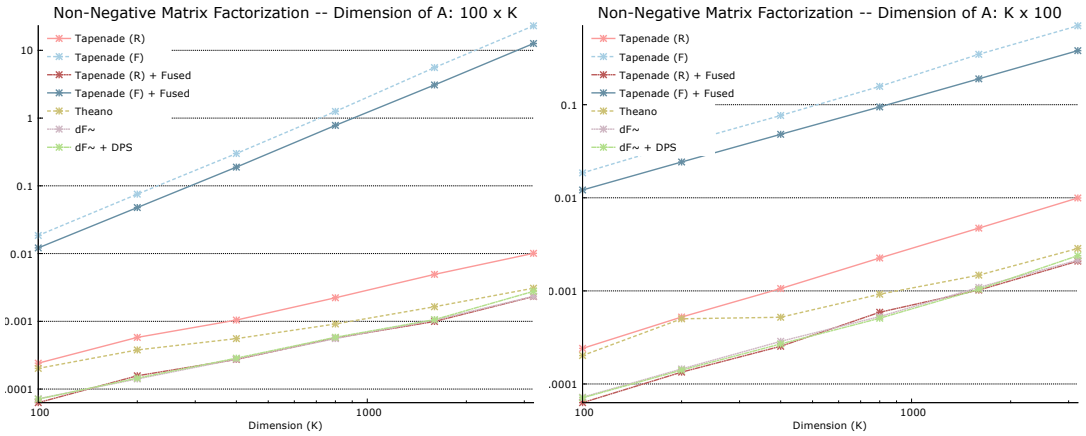


Figure 6.10 – Performance results for NNMF.

forward-mode AD show comparable performance. Finally, for the last case, as the Jacobian matrix is a column vector, the forward mode AD computes the whole Jacobian matrix in a single forward pass. However, the reverse mode AD requires traversing over each row to compute the corresponding partial derivative values.

6.6.2 Computer Vision and Machine Learning Workloads

Non-Negative Matrix Factorization (NNMF) is a useful tool which has many applications in various fields ranging from document clustering, recommendation systems, signal processing, to computer vision. For instance, in [223], the authors study the NNMF of Web dyadic data represented as the matrix A. Dyadic data contains rich information about the interactions between the two participating sets. It is useful for a broad range of practical applications including Web search, Internet monetization, and social media content [223]. For example the (query, clicked URL) data is used in query clustering [169], query suggestions [22] and improving search relevance [9]. Matrix factorization is a commonly used approach to understanding the latent structure of the observed matrix for various applications [33, 311]. The authors present a probabilistic NNMF framework for a variety of Web dyadic data that conforms to different probabilistic distributions. For instance, an Exponential distribution is used to model Web lifetime dyadic data, e.g., user dwell time, and similarly the Poisson distribution is used to model count dyadic data, e.g., click counts.

The iterative algorithm to find W and H depends on the form of the assumed underlying distribution. In particular the update formula for gradient descent are derived by computing the gradient of the negative log of the likelihood function. For example, the negative log of the exponential distribution is represented as follows:

$$\mathscr{D}\big(A||\widetilde{A}\big) = \Sigma_{(i,j)}\left(\log\big(\widetilde{A}_{i,j}\big) + \frac{A_{i,j}}{\widetilde{A}_{i,j}}\right), \qquad \widetilde{A} = WH$$

The update formulas are derived manually, and for each new distribution it is the responsibility of the user to undertake the error prone and laborious task of deriving, optimizing, and implementing the update rules. $\widetilde{\text{dF}}$ automatically derives the gradient of the negative log of the likelihood function for the exponential distribution. After performing optimizations, $\widetilde{\text{dF}}$ produces an expression which is equivalent to the following update formula, which is manually derived by hand in [223]:

$$\frac{\partial \mathscr{D}}{\partial H} = W^T\left(\frac{1}{WH} - \frac{A}{\big(WH\big)^2}\right)$$

Figure 6.10 shows the performance results of executing the derived update rule on Tapenade, Theano, and $\widetilde{\text{dF}}$. For all the experiments, we consider factorizing the matrix A into two vectors W and H (represented as u and v^T, respectively). To have a fair comparison between Tapenade and $\widetilde{\text{dF}}$, we have provided both the fused and unfused versions of the likelihood function. We observe a 2x speed up for the forward mode, and a 5x speed up for the reverse mode, when comparing the fused version with the unfused version. Comparing the fused version of Tapenade and $\widetilde{\text{dF}}$, we observe that the reverse-mode AD of Tapenade behaves similarly to $\widetilde{\text{dF}}$. This shows that $\widetilde{\text{dF}}$ successfully generates efficient code for this case, which is an ideal case for the reverse-mode AD (the loss function is a scalar valued function, which should compute the gradient with respect to all elements of the input vector). Finally, as the dimension of the vectors increases, Theano converges to the same performance as $\widetilde{\text{dF}}$ and reverse-mode AD of Tapenade. This is thanks to the fact that the overhead of invoking C functions from Python becomes negligible as the size of the vector increases.

The Gaussian Mixture Model (GMM) is a statistical method used for various machine learning tasks such as unsupervised and semi-supervised learning, as well as computer vision applications such as image background modelling and image denoising.

Here we focus on computing the gradient of one of function used in GMM: the Log-Sum-Exp (LSE) of a vector is useful in various machine learning algorithms such as GMM [257, 157]. Intuitively, if the multiplication operation in the linear domain is transformed into addition in the log domain, the addition operation is transformed into LSE in the log domain. This expression is computed as follows.

$$LSE\big(x_1,...,x_n\big) = x_{max} + log\big(\Sigma_{i=1}^{n}(e^{x_i - x_{max}})\big)$$

Figure 6.11 shows the performance results for the gradient of this function with respect to its input vector. Applying fusion improves the performance of the differentiated programs by

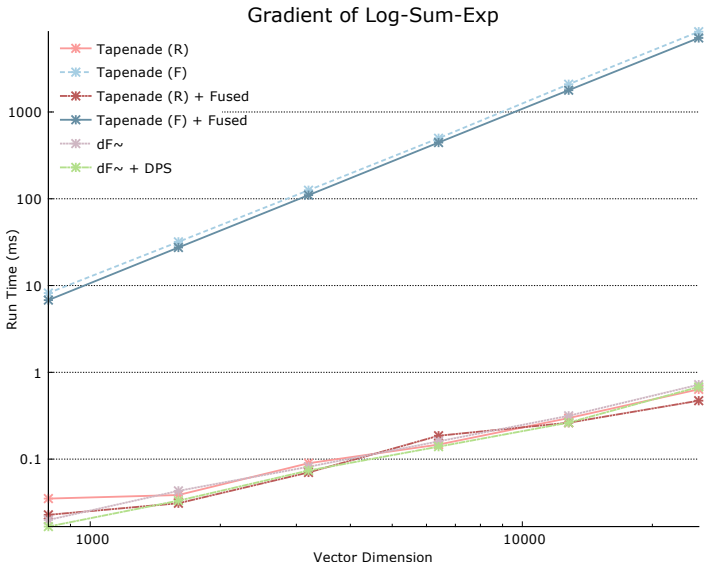


Figure 6.11 – Performance results for log-sum-exp used in GMM.

25%. Comparing the fused versions of the programs, $d\widetilde{F}$ outperforms the forward-mode AD of Tapenade from 2 to 4 orders of magnitude. This gap increases quadratically with the size of the input vector. However, $d\widetilde{F}$ shows a similar performance to the fused reverse-mode AD of Tapenade.

Bundle Adjustment [344, 8, 375] is a computer vision problem, where the goal is to optimize several parameters in order to have an accurate estimate of the projection of a 3D point by a camera. This is achieved by minimizing an objective function representing the reprojection error.

For the experiments, we compute the Jacobian matrix of the Project function in Bundle Adjustment. For a 3D point $X \in \mathbb{R}^3$ and a camera with rotation parameter $r \in \mathbb{R}^3$, center position $C \in \mathbb{R}^3$, focal index $f \in \mathbb{R}$, principal point $x_0 \in \mathbb{R}^2$, and radical distortion $k \in \mathbb{R}^2$, the Project function is computes the projected point as follows:

$$
\begin{aligned}
\text{project}(r, C, f, x_0, k, X) &= \text{distort}(k, \text{p2e}(\text{rodrigues}(r, X - C)))f + x_0 \\
\text{distort}(k, x) &= x(1 + k_1||x||^2 + k_2||x||^4) \\
\text{p2e}(X) &= X_{1..2} / X_3 \\
\text{rodrigues}(r, X) &= X cos\theta + (v \times X) sin\theta + v(v^T X)(1 - cos\theta), \theta = ||r||, v = \frac{r}{||r||}
\end{aligned}
$$

Consider having N 3D points and one particular camera parameter (an input vector of size $3N+11$), we are interested in computing a Jacobian matrix with $3N+11$ rows and $2N$ columns. Figure 6.12 shows the performance results for computing the mentioned Jacobian matrix. As

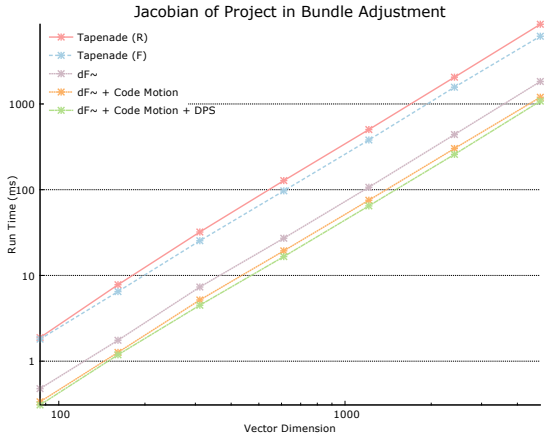


Figure 6.12 – Performance results for Project in Bundle Adjustment.

it can be seen $d\widetilde{F}$ outperforms both forward and reverse mode of Tapenade. This is mainly thanks to the loop transformations, such as loop-invariant code motion, happening in $d\widetilde{F}$.

6.7 Outlook and Conclusions

In this chapter we have demonstrated how to efficiently compute the derivate of a program. The key idea behind our system is exposing all the constructs used in differentiated programs to the underlying compiler. As a result, the compiler can apply various loop transformations such as loop-invariant code motion and loop fusion for optimizing differentiated programs. We have shown how $d\widetilde{F}$ outperforms the existing AD tools on micro benchmarks and real-world machine learning and computer vision applications.

We plan to extend $d\widetilde{F}$ with the reverse-mode AD by employing a similar technique to the one proposed by [273]. In addition, as we have seen in our examples, the strategy for applying rewrite rules can become tricky in some cases; there are some rewrite rules (e.g., loop fission) that do not necessarily improve the performance, unless they are combined with other transformation rules. We plan to investigate the use of search strategies for automated rewriting (e.g., using Monte-Carlo tree search [84]).

7 Efficient Incremental Analytics

The man who moves a mountain begins by carrying away small stones.
– Confucius

This chapter targets the Incremental View Maintenance (IVM) of sophisticated analytics (such as statistical models, machine learning programs, and graph algorithms) expressed as linear algebra programs. We present Lago, a unified framework for linear algebra that automatically synthesizes efficient incremental trigger programs, thereby freeing the user from erroneous manual derivations, performance tuning, and low-level implementation details. The key technique underlying our framework is abstract interpretation, which is used to infer various properties of analytical programs. These properties give the reasoning power required for the automatic synthesis of efficient incremental triggers. We evaluate the effectiveness of our framework on a wide range of applications from regression models to graph computations.

7.1 Introduction

Analytics has seen a paradigm shift from aggregate query processing, e.g., SQL, to more sophisticated analytics where data practitioners, engineers, and scientists utilize advanced statistical data models to gain insights into the collected data. These analytical tasks include machine learning, statistical analyses, scientific computation, and graph computations. The need is increasing for optimized execution workflows that support such analytic workloads. Currently, a wide range of tools and environments for expressing and optimizing such workloads have evolved. These include systems specialized for machine learning tasks such as MLlib [239] and SystemML [118]; platforms dedicated for graph processing such as GraphChi [209], GraphLab [226], Pregel [230] and PowerGraph [126]; low-level autotuned kernels such as Spiral [277, 310] for linear transformations; and Riot [381] for out-of-core statistical analysis.

Recently, data collection has been experiencing a steep increase in volume and velocity. Not only is the data big, but it is changing rapidly as well. More than ever, the demand for

applications and tools that react promptly to dynamic data has been increasing. A broad range of modern applications, including clickstream analysis, algorithmic trading, network monitoring, and recommender systems, compute realtime analytics over rapidly evolving datasets. However, the existing tools lack support for dynamic datasets. High data velocity forces application developers to manually build ad-hoc solutions in order to deliver high performance, responsiveness, and interactivity. Most datasets evolve through changes that are small relative to the overall dataset size. For example, the activity of a single customer, like her purchase history or review ratings, represents only a tiny portion of the overall collected data corpus. Recomputing data analytics on every (moderate) dataset change is far from efficient.

An alternative approach, Incremental View Maintenance (IVM), combines pre-computations with incoming Δ changes to provide a computationally cheap method for updating the final result. IVM [39, 205, 138] of relational calculus is well known in the Databases literature and provides orders of magnitude better performance than traditional approaches for SQL queries.

However, there are two main limitations with the existing IVM approaches for advanced analytics. First, despite some efforts for supporting IVM of recursive matrix equations [258], these techniques are not applicable for a wide range of applications such as many machine learning algorithms, including regression, recommender systems, and matrix factorizations, which are representable as matrix operations [118]. Moreover, recent research [193, 48, 350] suggests that modelling graph analytics using matrix operations results in better parallelization efficiency, e.g., coarse grained parallelism, and higher productivity, e.g., using simpler abstractions.

Second, to the best of our knowledge, the process of generating such incremental triggers is not completely automated. Naïvely incrementalizing analytics can lead to programs with the same computational complexity, if not worse [258]. To make this process automatic while achieving good performance, one needs to decide on the right program optimization choices, which requires advanced forms of reasoning over analytical programs.

This thesis presents techniques and tools that enable automatic synthesis of incremental linear algebra programs. We now present the structure of this thesis while outlining our main contributions:

• **Lago: Automatic Synthesis of IVM:** Lago (Section 7.3) is a unified modular compiler framework that enables end-to-end automatic IVM of a broad class of linear algebra programs. Lago automatically synthesizes incremental trigger programs of analytical computations, thus freeing the user from erroneous manual derivations, low-level implementation details, and performance tuning. Lago benefits from certain mathematical tricks for capturing Δ changes in a compressed factored form [258] (Section 7.2) that enables asymptotic performance improvements for IVM. Furthermore, Lago defines a DSL (Section 7.3.2) for supporting a wide range of applications and domains of different semiring configurations, e.g., graph applications. In addition, we present a set of domain-specific transformation rules that allows for delta derivation, simplification, and cost-based optimization of matrix algebra programs

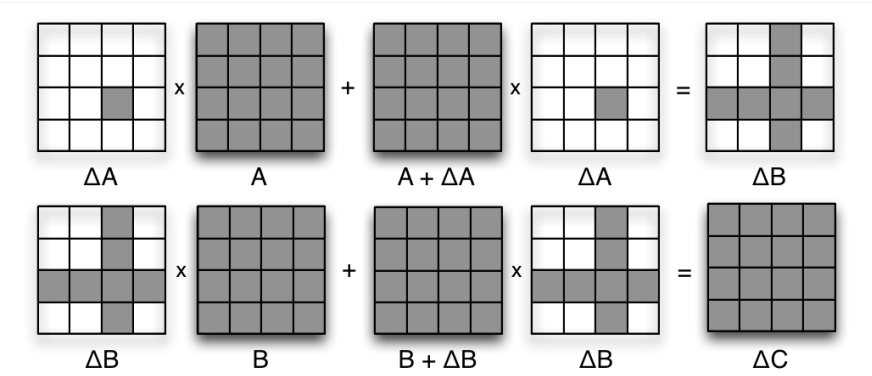


Figure 7.1 – A single data-entry change ΔA in the input A can result in whole matrix perturbations of subsequent Δ expressions. ΔB has changes in a row and a column, whereas ΔC has changes all over all the entries.

(Section 7.3.3).

• **Abstract Interpretation:** Lago leverages matrix-expression properties, called *abstract domains*, including data type, dimensions, cost, matrix structures, etc, for synthesizing efficient incremental analytics. This is achieved by a well-known technique from the programming language community, called *abstract interpretation* (Section 7.4). This idea is inspired by the workload-specific information (e.g., selectivity and cardinality) used in the query optimizers of DBMSes. Examples in this thesis demonstrate how dimensions are used to guide cost-based optimization (Section 7.4.1); how symmetry enables further transformations; and how data and semiring types permit specialization opportunities during code generation (Section 7.4.2).

• **Use cases & Evaluation:** In Section 7.6, we evaluate the IVM of several practical use case examples including computing linear regression models, gradient descent, and all-pairs graph reachability and shortest path computations. The evaluation results demonstrate orders of magnitude better performance of synthesized trigger programs relative to simple re-evaluation.

7.2 Incremental Computation Δ

Most datasets evolve through changes that are small relative to the overall dataset size. For example, a social network graph evolves through connections that are relatively small in comparison to the entire graph size. Recomputing data analytics on every slight dataset change is far from efficient. Incremental View Maintenance [39, 205, 138] (IVM) studies the incremental maintenance of relational queries. IVM trades off storage in favour of cheaper computations. The main idea is to confine the re-evaluation to the changes affected by the incremental updates only. Then, they are used to update materialized views of the precomputed results. Within the Databases literature, several approaches [39, 205, 138] have been proposed to

```
def pow(A)={
   B:= A.A;
   C:= B.B;
   D:= C.C;

   return D;
}
```

(a) Example program that computes the 8^{th} power of input matrix A.

```
def powDelta(ΔA)={
   ΔB:=(A+ΔA).(A+ΔA)-A.A;
   ΔC:=(B+ΔB).(B+ΔB)-B.B;
   ΔD:=(C+ΔC).(C+ΔC)-C.C;
   A+=ΔA;B+=ΔB;C+=ΔC;D+=ΔD;
   return D;
}
```

(b) Trigger program that computes Δ expressions for each statement and finally updates the corresponding materialized views.

```
def powDelta(ΔA)={
   ΔB:= A.ΔA + ΔA.A + ΔA.ΔA;

   ΔC:= B.ΔB + ΔB.B + ΔB.ΔB;

   ΔD:= C.ΔC + ΔC.C + ΔC.ΔC;

   A+=ΔA;B+=ΔB;
   C+=ΔC;D+=ΔD;
   return D;
}
```

(c) An optimized version of the trigger program after applying algebraic simplification.

```
def powDelta(u_A,v_A)={
   U_B:=[u_A (A.u_A + u_A.(v_A^T.u_A))];
   V_B:=[(v_A^T.A) ; v_A^T];
   U_C:=[U_B (B.U_B + U_B.(V_B.U_B))];
   V_C:=[(v_B.B) ; V_B];
   U_D:=[U_C (C.U_C + U_C.(V_C.U_C))];
   V_D:=[(V_C.C) ; V_C];
   A+=u_A.v_A;B+=U_B.V_B;
   C+=U_C.V_C;D+=U_D.V_D;
   return D;
}
```

(d) Final optimized trigger program that represents Δ expressions in a factored form.

Figure 7.2 – The process of delta derivation for an example program that computes the 8^{th} power of input matrix A.

achieve this. Most notably, DBToaster [205] achieves orders of magnitude better performance on SQL queries in comparison to traditional re-evaluation. However, these approaches are not applicable to matrix programs. To demonstrate this, consider the simple example of computing matrix powers. Matrix powers play an important role in many different domains including evaluating the stochastic matrix of a Markov chain after k steps, solving systems of linear differential equations using matrix exponentials, answering graph reachability queries after k hops. Fig. 7.2a demonstrates an example of computing the 8^{th} power of the input matrix A. The program requires computing 3 costly $\mathcal{O}(n^3)$ matrix-matrix multiplications to evaluate the result. Now, consider a trigger program that updates the final result given a single entry change ΔA to the input matrix A. For explanatory reasons, Fig. 7.2b gives a simplistic representation of such a trigger program where it computes the delta expression for each of the intermediate variables B, C, and D, respectively. Then finally, these materialized views are updated with the corresponding delta expressions, e.g., B+=ΔB. Furthermore, when the expressions are expanded algebraically utilizing the associative and distributive laws of matrix

addition over multiplication, one could deduce the more simplified expressions as illustrated in Fig. 7.2c.

On relatively small changes, one could imagine that by confining the computation to the deltas, we could achieve better performance in comparison to re-evaluation. Unfortunately, this is not the case. As depicted in Fig. 7.1, consider a single entry change ΔA in A. As the figure illustrates, dark cells correspond to entry changes where as white cells correspond to the neutral value, i.e., no change. We can easily compute ΔB in $\mathcal{O}(n^2)$ time, as there is only one single entry in ΔA. After the multiplication, the resulting ΔB matrix has entry changes on a single row and a single column. Similarly, computing ΔC can be done in $\mathcal{O}(n^2)$ time, as one only needs to multiply the two vectors from ΔB with full matrices. However, this is not the case anymore when computing ΔD. ΔC has changes all over its matrix entries. When it is used in the subsequent statement to compute ΔD, full fledged $\mathcal{O}(n^3)$ matrix multiplications are required. This renders incremental computation useless in comparison to naive re-evaluation. The above example shows that linear algebra programs are, in general, sensitive to input changes. Even a single entry change in the input can cause an avalanche effect of perturbations, quickly escalating to full matrix perturbations, even after executing only two statements.

7.2.1 The Delta (Δ) Representation

Until now, we have stored the results of Δ expressions into full matrices. However, one can realize that this representation is highly redundant and that Δs are usually characterized by having low ranks. Capturing this information is important, as it enables representing the Δ expressions in a packed factored form which compacts storage and greatly reduces the computation cost of its evaluation. The matrix rank is defined as follows:

Definition 7.2.1 *A matrix $\mathcal{M}$ of dimensions $(n \times n)$ is said to have rank-k if the maximum number of linearly independent rows or columns in the matrix is k. $\mathcal{M}$ is called a low-rank matrix if $k \ll n$.*

For example, a single entry change can be represented as a rank-1 update. In fact, a rank-1 update can represent updates of a single row/column or even several rows/columns that are linearly dependent to each other. A rank-1 update is represented in a compressed compact form as an outer product of two vectors $\Delta = u v^{\mathrm{T}}$ rather than a full matrix. To demonstrate this, consider a matrix A with dimensions 3×3 and a single entry change ΔA that adds the value c at index $[2,2]$ of matrix A. This change can be represented in the factored form as follows:

$$\Delta A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{c} \end{bmatrix} = u.v^{\mathrm{T}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & \mathrm{c} \end{bmatrix}$$

Similarly a row change or a column change at $[2,\_]$ or $[\_,2]$ can be represented as follows

respectively:

$$\Delta A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ c_0 & c_1 & c_2 \end{bmatrix} = u.v^{\mathrm{T}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} c_0 & c_1 & c_2 \end{bmatrix}$$

$$\Delta A = \begin{bmatrix} 0 & 0 & c_0 \\ 0 & 0 & c_1 \\ 0 & 0 & c_2 \end{bmatrix} = u.v^{\mathrm{T}} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

In general, rank-k matrices can represent more general update patterns as they can be represented as a sum of k rank-1 matrices.

Let us illustrate the benefits of this factored form in the previous example. Consider a rank-1 update $\Delta A = u_{\mathrm{A}} v_{\mathrm{A}}^{\mathrm{T}}$, where u_{A} and v_{A} are column vectors. One can compute $\Delta B = u_{\mathrm{A}} (v_{\mathrm{A}}^{\mathrm{T}} A) + (A\, u_{\mathrm{A}})\, v_{\mathrm{A}}^{\mathrm{T}} + (u_{\mathrm{A}}\, v_{\mathrm{A}}^{\mathrm{T}}\, u_{\mathrm{A}})\, v_{\mathrm{A}}^{\mathrm{T}}$ as a sum of three outer products. The evaluation order enforced by these parentheses results in matrix-vector and vector-vector multiplications only. Thus, the evaluation of ΔB requires $\mathcal{O}(n^2)$ operations only. Moreover, rather than representing the delta expressions as a sum of outer products, we represent them in a more compact vectorized form for performance, storage, and presentation reasons. Generally, a sum of k outer products is equivalent to a single product of two matrices with dimensions $(n \times k)$ and $(k \times n)$, which are obtained by horizontally/vertically stacking the corresponding vectors together as follows:

$$u_1.v_1^{\mathrm{T}} + u_2.v_2^{\mathrm{T}} + u_3.v_3^{\mathrm{T}} = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} v_1^{\mathrm{T}} \\ v_2^{\mathrm{T}} \\ v_3^{\mathrm{T}} \end{bmatrix} = U\,V$$

where U and V are block matrices with dimensions $(n \times 3)$ and $(3 \times n)$ respectively. Following the same structure, we can represent ΔB in the factored form $U_B V_B$ as derived in Fig. 7.2d:

$$\begin{array}{rcl} \Delta B & = & u_{\mathrm{A}}.(v_{\mathrm{A}}^{\mathrm{T}}.A) + (A.u_{\mathrm{A}} + u_{\mathrm{A}}.(v_{\mathrm{A}}^{\mathrm{T}}.u_{\mathrm{A}})).v_{\mathrm{A}}^{\mathrm{T}} \Rightarrow \\ U_B & = & [u_{\mathrm{A}} \quad (A.u_A + u_{\mathrm{A}}.(v_{\mathrm{A}}^{\mathrm{T}}.u_{\mathrm{A}}))] \\ V_B & = & [(v_{\mathrm{A}}^{\mathrm{T}}.A) \;;\; v_{\mathrm{A}}^{\mathrm{T}}] \end{array}$$

This factored representation is used further down the program to derive the Δ expressions for each of C and D as depicted in Fig. 7.2d.

In summary and without loss of generality, we capitalize on the low-rank structure of delta matrices by representing a delta expression $\Delta_{n \times n}$ of rank k as a product of two matrices with dimensions $(n \times k)$ and $(k \times n)$, where $k \ll n$. This allows for efficient evaluation of subsequent

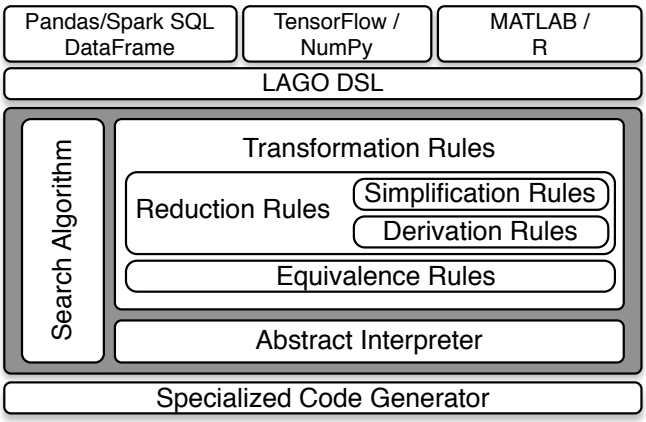


Figure 7.3 – The architecture of the Lago framework.

delta expressions without performing expensive $\mathcal{O}(n^3)$ operations;[1] instead, only $\mathcal{O}(kn^2)$ operations are computed. The benefit of incremental processing diminishes as k approaches n.

7.3 The LAGO Framework

In the previous section, we introduced the concept of incremental computation for matrix algebra and the ability to derive efficient trigger programs by representing updates in a factored form through exploitation of their low rank structure. In this section, we discuss how to automatically derive those trigger programs. One could assume a manual approach in dealing with this problem, however the developer has to put effort into deriving the incremental program, then optimizing it to ensure low cost computation, then finally writing down the code for the trigger program. This is a long and tedious process that includes *a*) delta derivation which is error prone, *b*) optimization which requires simplification, cost-based rewrites, and delicate ordering of operations, and *c*) writing the final trigger program code which requires careful consideration, e.g., evaluating the delta expressions using the precomputed results before updating the views. We propose offloading all of these responsibilities to Lago, a compiler framework dedicated for synthesizing trigger code for various underlying processing substrates, thereby freeing the user from erroneous manual derivations, optimization, and low-level implementation details.

[1] More precisely, the complexity of matrix multiplication is $\mathcal{O}(n^\gamma)$ where $2 \leq \gamma \leq 3$. For all practical reasons, the complexity of matrix multiplication implementations, e.g., using BLAS [363] has cubic cost $\mathcal{O}(n^3)$. Other algorithms, such as Coppersmith-Winograd and its successors, suggest better exponents of $2.37+\epsilon$; however, these algorithms are only applicable for astronomically large matrices. Our incremental techniques remain relevant as long as matrix multiplication stays asymptotically worse than quadratic time. Note that the asymptotic lower bound is $\Omega(n^2)$ operations because it needs to process at least all $2n^2$ entries.

7.3.1 Architecture Overview

In this section, we present the architecture of the Lago framework. Then we describe its underlying components in detail. The main tasks of the Lago framework are as follows: *1*) accepting an input matrix program; *2*) deriving the incremental Δ expressions for the statements; performing cost-based optimizations to optimize the derived expressions; and finally *3*) generating the output trigger program for the underlying architecture using the derived Δ expressions. Fig. 7.3 gives an overview of the Lago architecture.

1. First, Section 7.3.2 presents the domain-specific language (DSL) used to describe matrix programs in the framework. It includes a restricted set of domain-specific operations specialized for matrix algebra that is independent of the application domain. This DSL can be thought of as an intermediate language (similar to the Weld IR [264]), which can be the target of various frontends such as Pandas and Spark Dataframe, Tensorflow, NumPy, MATLAB, and R. Table 7.1 shows the mapping among several operations in MATLAB, R, NumPy, and Lago DSL.

2. To derive the incremental program, Lago needs a set of reduction rules that symbolically derive the incremental expressions from the input program and those that simplify the derived expressions. Afterwards, equivalence rules are applied whenever optimization is required. These are called transformation rules (Section 7.3.3). A search module is required to navigate the search space of functionally equivalent programs created by the transformation rules. Different search algorithms can be utilized for different workloads. Similar to DBMSes, various flavours of search algorithms can be employed, such as brute force, Dynamic Programming, or randomized algorithms [163]. However, the discussion about search strategies is orthogonal to this thesis. To evaluate the cost of candidate programs, one needs to estimate their running costs. Similar to how DBMSes estimate query-plan costs using cardinality and selectivity information, Lago leverages abstract domains (Section 7.4) for matrix programs. The abstract domains encapsulate various properties associated to matrix expressions, such as dimensions, structure, symmetry, rank, etc. Moreover, Lago uses abstract interpretation to automatically derive these abstract domains for candidate programs. This information helps in estimating program costs and in leveraging specialized back-end implementations.

3. Finally, Lago generates trigger code for the underlying processing substrate using code generation. Code generation is extensible, in the sense that one can easily use various code generators for different environments, e.g., Octave, R, Spark, SystemML, etc. The main task in this phase is to generate code for the materialized precomputed results and updating them with their corresponding optimized Δ expressions.

Next, we discuss each of these components in detail, and we discuss how they interact with each other.

$$
\begin{array}{rcll}
\mathfrak{m} & ::= & \mathfrak{m} \cdot \mathfrak{m} & \text{– Matrix-Matrix Multiplication} \\
 & | & \mathfrak{m} + \mathfrak{m} & \text{– Matrix Addition} \\
 & | & \mathfrak{m}^{\top} & \text{– Matrix Transpose} \\
 & | & \mathfrak{m}^{-1} & \text{– Matrix Inverse} \\
 & | & \mathfrak{m} \rightrightarrows \mathfrak{m} & \text{– Matrix Concatenation} \\
 & | & \texttt{vect}[\mathfrak{s}](\mathfrak{s}) & \text{– Row Matrix Construction} \\
 & | & \texttt{diag}[\mathfrak{s}][\mathfrak{s}] & \text{– Diagonal Matrix Construction} \\
 & | & \texttt{let}\ \mathrm{x} = \mathfrak{m}_1\ \texttt{in}\ \mathfrak{m}_2 & \text{– Let binding} \\
 & | & \texttt{iterate}[\mathfrak{s}](\mathfrak{m})(\mathrm{x} \Rightarrow \mathfrak{m}) & \text{– Matrix Iteration} \\
 & | & \mathrm{x} & \text{– Matrix Identifier} \\
\mathfrak{s} & ::= & \mathfrak{s}\ bop\ \mathfrak{s} & \text{– Scalar Binary Operation} \\
 & | & \texttt{cols}(\mathfrak{m}) & \text{– Number of Columns of a Matrix}
\end{array}
$$

Figure 7.4 – The core Lago DSL divided into two main classes, i.e., matrix and scalar operations.

7.3.2 Lago DSL

Many sophisticated data analytics programs, including machine learning, graph algorithms, and statistical programs, express computation using matrices and vectors using a high-level abstraction. Lago exposes a domain-specific language (DSL) that expresses such program formulations. The DSL is formulated using standard matrix manipulation primitives, including elementwise operators[2].

Lago adopts a *functional* approach in the language design. This choice is motivated by the design of languages like relational algebra and Monad algebra [44, 45] in the DB community. Functional programming is a popular paradigm that treats computation as the evaluation of mathematical functions while avoiding state mutation. Since the domain in hand is also mathematical, this paradigm fits well and the inherited benefits are manifold. Most problems that commonly arise in imperative languages from mutable state and side effects are eliminated. This plays a critical role in performing optimizations. In particular, transformations and their compositions preserve semantics. This eliminates worries about the program correctness. Moreover, given the mathematical nature of the DSL, transformation rules and abstract interpretation rules are much easier defined, as later discussed in Section 7.3.3 and Section 7.4.1. Alternatively, other declarative languages proposed in the literature, such as SystemML [118], are imperative and support generic control flow and mutable state. Mutable state enables better runtime performance yet makes reasoning and optimization much harder.

Core Language

Another important design choice that drives the language design is to keep the core language succinct enough while maintaining expressiveness that supports a wider range of linear algebra operations through composition. This keeps the language simple, which in turn keeps all

[2] Note that the addition operator can be parameterized with other binary operators to support other elementwise operators, such as elementwise multiplication.

MATLAB	R	NumPy	Lago
A * B	A %*% B	A.dot(B)	$A \cdot B$
A + B	A + B	A + B	A + B
A'	t(A)	A.T	$A^\top$
$[A, B]$	cbind(A, B)	hstack((A,B))	$A \rightrightarrows B$
$[A; B]$	rbind(A, B)	vstack((A,B))	$A \downdownarrows B$
ones(n, m)	matrix(1, n, m)	ones((n, m))	`ones`[n, m]
zeros(n, m)	matrix(0, n, m)	zeros((n, m))	`zeros`[n, m]
eye(n)	diag(n)	eye(n)	`identity`[n]

Table 7.1 – Equivalent operations in MATLAB, R, NumPy, and Lago respectively. Fig. 7.5 defines the extended operations.

the transformation and inference rules at a simple maintainable level. Fig. 7.4 presents Lago's core language grammar which includes matrix multiplication, matrix addition, transpose, matrix inverse, horizontal concatenation (stacking), diagonal matrix construction, let binding, and declaring matrix identifiers, respectively. `vect`$[\mathfrak{s}_1](\mathfrak{s}_2)$ constructs a $1 \times \mathfrak{s}_1$ matrix with the constant value $\mathfrak{s}_2$. `diag`$[\mathfrak{s}_1][\mathfrak{s}_2]$ creates an $\mathfrak{s}_1 \times \mathfrak{s}_1$ matrix with diagonal elements of value $\mathfrak{s}_2$. The rest of operations are self-explanatory. Additionally, we define binary operations on scalars and computing the columns of a matrix.

These constructs are sufficient for expressing non-iterative matrix operations. However, supporting iterative computation is a challenging undertaking. For example, the declarative language presented in SystemML [118] uses imperative constructs such as while loops. Alternatively, to support iterative computation without using imperative loops or mutable states we present the `iterate`$[\mathfrak{s}](\mathfrak{m})(x \Rightarrow \mathfrak{m})$ construct which allows us to perform step-by-step computations. The construct is defined by specifying the number of iterations $\mathfrak{s}$, the matrix $\mathfrak{m}$ value for the initial step, i.e., neutral value, as well as a function, $x \Rightarrow \mathfrak{m}$, which computes the current step given the accumulator computed from the previous steps. Notice how the `iterate` operator relates to the functional fold operator. It is important to note that the `iterate` construct represents syntactic sugar for recursive functions. An initial program with the `iterate` operator is first expanded into simpler core language constructs using the simplification rules described in Section 7.3.3.

Extensions

Various matrix manipulation operations can be defined as syntactic sugar over the core language. This means that there is no need to extend the core language with further redundant operations, which in turn complicates the language, the transformations, the reasoning power, the search space and eventually the modularity of the framework. Instead, one can define these operations in terms of compositions of the core language constructs. Fig. 7.5 demonstrates the expressiveness of the core Lago DSL and Table 7.1 illustrates some examples of equivalent operations in Matlab, R, NumPy, and Lago using compositions of the small set of core language

$$
\begin{array}{ll}
\texttt{fill}[\mathrm{r},\mathrm{c}](\mathfrak{s}) & \mathfrak{m}_1 \Downarrow\!\Downarrow \mathfrak{m}_2 \\
\quad := \texttt{vect}[\mathrm{r}](1)^\top \cdot \texttt{vect}[\mathrm{c}](\mathfrak{s}) & \quad := (\mathfrak{m}_1{}^\top \rightrightarrows \mathfrak{m}_2{}^\top)^\top \\
\texttt{rows}(\mathfrak{m}) & \texttt{identity}[\mathrm{n}] \\
\quad := \texttt{cols}(\mathfrak{m}^\top) & \quad := \texttt{diag}[\mathrm{n}][1] \\
\mathrm{k} \cdot \mathfrak{m} & \texttt{ones}[\mathrm{r},\mathrm{c}] \\
\quad := \texttt{diag}[\texttt{rows}(\mathfrak{m})][\mathrm{k}] \cdot \mathfrak{m} & \quad := \texttt{fill}[\mathrm{r},\mathrm{c}](1) \\
\mathfrak{m}_1 - \mathfrak{m}_2 & \texttt{zeros}[\mathrm{r},\mathrm{c}] \\
\quad := \mathfrak{m}_1 + \text{-}1 \cdot \mathfrak{m}_2 & \quad := \texttt{fill}[\mathrm{r},\mathrm{c}](0)
\end{array}
$$

Figure 7.5 – Syntactic sugar: Examples of additional operations defined using compositions of the Lago DSL.

operations.

Semiring Configurations. Different domains and applications can be built on top of matrix algebra using various semiring configurations. One domain example that makes use of matrix algebra and semirings is graph computation. To explain this further, let's define a semiring first:

Definition 7.3.1 *Given a set $\mathscr{S}$ and two binary operations $\oplus$, $\otimes$ called addition and multiplication respectively, a semiring $\langle \mathscr{S}, \oplus, \otimes \rangle$ is an algebraic structure, such that $\langle \mathscr{S}, \oplus \rangle$ is a commutative monoid with the identity element 0, $\langle \mathscr{S}, \otimes \rangle$ is a monoid with the identity element $\mathbb{1}$, left and right multiplication $\otimes$ distributes over addition $\oplus$, and multiplication by 0 yields back 0.*

Semirings & Graphs. Graphs are among the most important abstract data structures in computer science. The algorithms that operate on them are critical to a wide variety of applications including bioinformatics, computer networks, and social media. The vast growth in graph data has forced a shift to parallel computing for executing graph algorithms. Implementing parallel graph algorithms while achieving good parallel performance is a difficult task as it requires fine grained synchronization [48]. Recently, there has been an interest in addressing this challenge by exploiting the duality between the canonical representation of graphs as abstract collections of vertices and edges and a sparse adjacency matrix representation [193, 48, 350]. Furthermore, there is a duality between the fundamental operations on graphs, Breadth First Search (BFS), and the fundamental operation of matrices, matrix multiplication.

The benefits of representing graph algorithms as matrices are manifold [193, 48, 350]. Firstly, this linear algebraic approach is widely accessible to scientists and engineers who may not be formally trained in computer science. Secondly, higher performance can be achieved, as parallelizing graph algorithms can leverage decades worth of research on parallelizing matrix operations, coarse grained parallelism, and optimization with regards to the memory hierarchy. Finally, it leverages productivity and ease of implementation.

The common primitive operations used are the numerical addition and multiplication which define a semiring $\langle \mathscr{S}, \oplus, \otimes \rangle$ where $\mathscr{S} \in \{\mathbb{R}\}$, $\oplus = +$, $\otimes = \times$. Many graph problems can be articulated as matrix algebra programs under different semiring semantics. For instance, computing all-pairs graph reachability or shortest path after k hops can be expressed as program $\mathscr{P}$ depicted in Fig. 7.6. The semiring configuration defines the semantics of the program. For example, program $\mathscr{P}$ with the Boolean semiring $\langle \mathbb{B}, \vee, \wedge \rangle$ configuration expresses the k-hop reachability program. Similarly, with the tropical semiring $\langle \mathbb{R}, \min, + \rangle$ [48] configuration, program $\mathscr{P}$ expresses the k-hop shortest path program.

IVM & Semirings. The high-level matrix operations provided by Lago DSL directly allows us to represent graph programs. Moreover, all the primitives previously presented to support the incremental view maintenance of matrix expressions naturally follow under different semiring definitions. For example, as graphs evolve with time, one can model new connections in the graph as the ΔG expression added to G, representing the original adjacency matrix[3].

Deriving the Δ expressions and trigger programs is only concerned with the abstract representation of matrices and their transformations, and is independent of the semiring definition. However, this information is useful later on during the code generation phase for specialization, as demonstrated in Section 7.6. The semiring information can be expressed using the core language except for matrix inverse. We generalize the Lago core language by replacing matrix addition and multiplication with two meta-operators $\oplus_{\mathscr{R}}$ and $\otimes_{\mathscr{R}}$ parameterized by a semiring $\mathscr{R}$ instance. For example, $\oplus_{\mathscr{N}}$ and $\otimes_{\mathscr{N}}$ are concrete instances of the meta-operators with the arithmetic semiring $\mathscr{N}$ parameter.

7.3.3 Transformation Rules

Definition 7.3.2 *The $\Delta_x(\mathfrak{m})$ operator symbolically derives the delta Δ of expression $\mathfrak{m}$ with respect to variable (or matrix) x. Derivation of the delta expressions and their optimizations are achieved by recursively applying delta transformation rules on the expression $\mathfrak{m}$ until all Δ operators are omitted.*

There are two types of transformation rules: First, reduction rules which are used to derive the Δ operators and to perform simplifications on the derived expressions. Second, to further perform cost-based optimizations, Lago relies on a set of equivalence rules that create a space of functionally equivalent programs which are passed to the search algorithm in order to find a program with optimal cost. Transformation rules are responsible for constructing the search space of programs. It is very important that transformation rules preserve program semantics.

For illustration purposes, consider a 2-hop instance of the Graph program in Fig. 7.6 yielding

[3]Note that Δ changes represented as additions are naturally defined within the semiring, however, deletions depend on the availability of an additive inverse. For example under the boolean semiring, the additive inverse does not exist and thus we cannot model deletions.

```
iterate[k](id)(acc=> G · acc + id)
```

Figure 7.6 – Program $\mathscr{P}$ represents all-pairs graph reachability or shortest path after k-hops depending on the underlying semiring configuration.

$\Delta_x(\mathtt{m}_1 + \mathtt{m}_2) \rightarrow \Delta_x(\mathtt{m}_1) + \Delta_x(\mathtt{m}_2)$ DELTA-ADD
$\Delta_x(\mathtt{m}_1 \cdot \mathtt{m}_2) \rightarrow \Delta_x(\mathtt{m}_1) \cdot \mathtt{m}_2 + \mathtt{m}_1 \cdot \Delta_x(\mathtt{m}_2) \ + \ \Delta_x(\mathtt{m}_1) \cdot \Delta_x(\mathtt{m}_2)$ DELTA-MULT
$\Delta_x\left(\mathtt{m}^\top\right) \rightarrow (\Delta_x(\mathtt{m}))^\top$ DELTA-TRANS
$\Delta_x\left(\mathtt{m}^{-1}\right) \rightarrow (\mathtt{m} + \Delta_x(\mathtt{m}))^{-1} - \mathtt{m}^{-1}$ DELTA-INV
$\Delta_x(\mathrm{y}) \rightarrow \delta_y \quad \text{if } \mathrm{x} = \mathrm{y}$ DELTA-VAR-EQ
$\Delta_x(\mathrm{y}) \rightarrow \mathtt{zeros}[\mathtt{rows}(\mathrm{y}), \mathtt{cols}(\mathrm{y})] \quad \text{if } \mathrm{x} \neq \mathrm{y}$ DELTA-VAR-NEQ
$\Delta_x(\mathtt{m}_1 \rightrightarrows \mathtt{m}_2) \rightarrow \Delta_x(\mathtt{m}_1) \ \rightrightarrows \ \Delta_x(\mathtt{m}_2)$ DELTA-STACK
$\Delta_x(\mathtt{let}\ \mathrm{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2) \rightarrow \mathtt{let}\ \delta_x(\mathrm{y}) = \Delta_y(\mathtt{m}_1)\ \mathtt{in}\ \Delta_x(\mathtt{m}_2)$ DELTA-LET
$\Delta_x(\mathtt{vect}[\mathrm{c}](\mathfrak{s})) \rightarrow \mathtt{zeros}[1, \mathrm{c}]$ DELTA-VECT
$\Delta_x(\mathtt{diag}[\mathrm{c}][\mathfrak{s}]) \rightarrow \mathtt{zeros}[\mathrm{c}, \mathrm{c}]$ DELTA-DIAG

Figure 7.7 – Δ derivation rules for the core Lago DSL. The `iterate` construct is first unfolded using the simplification rules in Figure 7.8 before applying Δ rules.

the following expression[4]:

```
G.G + G.id + id
```

We will continue using this simple running example throughout the following subsections.

Reduction Rules

These are rules in the form of lhs → rhs, where it always reduces a matched expression from the left-hand-side to the right-hand-side. There are two classes of reduction rules, in particular, derivation and simplification rules which are explained next.

Derivation Rules. This class of reduction rules are used to derive the delta expressions. Δ operators are expanded and evaluated recursively. Using the distributive and associative properties of common matrix operations, we demonstrate the set of delta derivation rules for the core language as depicted in Fig. 7.7. The derivation of these rules are based on matrix identities. DELTA-ADD distributes the Δ operator across the summands. DELTA-MULT is directly derived from the distributivity of matrix multiplication over addition. DELTA-TRANS pushes the Δ into the expression before evaluating the transpose. DELTA-INV depicts the actual definition of Δ computation, which does not provide any computational savings at first glance, however it enables the Woodbury formula optimization that admits efficient evaluation. DELTA-VAR-EQ simply maps the Δ_x of a matrix y to a variable instance (called the delta variable) if $x = y$,

[4]Notice that we omit the **id** in G.G.**id**. This is only meant to simplify the following flow and avoid redundant discussions as with the subexpression G.**id**.

i.e., the matrix being changed x is identical to expression y. Similarly for DELTA-VAR-NEQ, if $x \neq y$, i.e., the matrix being changed x is different than the expression y, then the delta expression for y is `zeros`. DELTA-STACK distributes the Δ across the stacked matrices. DELTA-LET simply instantiates a delta variable instance and pushes the Δ across the expressions. Finally, DELTA-VECT and DELTA-DIAG reduce the Δ of the constant matrices to `zeros`.

These rules are applied recursively until all deltas of expressions are evaluated, i.e., no more matching derivation rules exist. To illustrate this, given our running example, consider that graph G is evolving with Δ_G changes and that we would like to evaluate the following expression:

$$\Delta_G(\texttt{G.G + G.id + id})$$

We notice, that the Δ operator is applied over an entire expression that can be reduced by the derivation rules. First, after applying the DELTA-ADD rule, the expression becomes:

$$\Delta_G(\texttt{G.G}) + \Delta_G(\texttt{G.id}) + \Delta_G(\texttt{id})$$

Furthermore, applying the DELTA-MULT rule yields:

$$\begin{aligned}&\Delta_G(\texttt{G}).\texttt{G} + \texttt{G}.\Delta_G(\texttt{G}) + \Delta_G(\texttt{G}).\Delta_G(\texttt{G}) + \Delta_G(\texttt{G}).(\texttt{id})\\ &\quad + \texttt{G}.\Delta_G(\texttt{id}) + \Delta_G(\texttt{G}).\Delta_G(\texttt{id}) + \Delta_G(\texttt{id})\end{aligned}$$

Moreover, Δ_G operators on expressions without any G bindings are further reduced to `zeros` using the DELTA-VAR-NEQ rule. Also, using the DELTA-VAR-EQ rule, all the Δ_G(G) expressions are reduced to delta variable instances δ_G. This yields the expression:

$$\begin{aligned}&\delta_G.\texttt{G} + \texttt{G}.\delta_G + \delta_G.\delta_G + \delta_G.\textbf{zeros}\\ &+ \texttt{G}.\textbf{zeros} + \delta_G.\textbf{zeros} + \textbf{zeros}\end{aligned}$$

Simplification Rules. The second class of reduction rules represents expression simplification. Symbolic computation is commonly accompanied by simplification. The derived expression is usually unnecessarily large and contains redundant computations. The expression is generally amenable to simplification. This is a major step in performing symbolic computations in computer algebra systems (CAS). For example, in CASs, right after deriving symbolic differentials, they usually perform simplification with the goal of minimizing the expression size. The same artifact happens while deriving Δ expressions, however, the goal is to avoid unnecessary redundant operations that will most probably result in higher cost. Fig. 7.8 demonstrates a subset of simplification rules used within Lago. These kinds of transformation rules are important when the expression tree is undergoing derivation or major transformations by Lago and requires simplification along the way. For instance, in the previous running example

$$\frac{(\mathrm{m}^\top)^\top \rightarrow \mathrm{m}}{} \quad \frac{\texttt{rows}(\mathrm{m}) \rightarrow \mathrm{r}}{\mathrm{m} : \mathscr{D}_{(r,c)}} \quad \frac{\texttt{iterate}[0](\mathrm{m})(\mathrm{x} \Rightarrow \mathrm{f}(\mathrm{x})) \rightarrow \mathrm{m}}{}$$

$$\frac{\texttt{iterate}[\mathrm{n}](\mathrm{m}_0)(\mathrm{x} \Rightarrow \mathrm{f}(\mathrm{x})) \rightarrow \texttt{iterate}[\mathrm{n\text{-}1}](\mathrm{f}(\mathrm{m}_0))(\mathrm{x} \Rightarrow \mathrm{f}(\mathrm{x}))}{n > 0}$$

$$\frac{\texttt{let } x_1 = x_2 \texttt{ in } \mathrm{m}_1 \rightarrow \mathrm{m}_1\,[x_1 := x_2]}{} \quad \frac{\mathrm{m} \cdot \texttt{identity}[\mathrm{c}] \rightarrow \mathrm{m}}{\mathrm{m} : \mathscr{D}_{(r,c)}}$$

$$\frac{\mathrm{m} \cdot \texttt{zeros}[\mathrm{k},\ \mathrm{c}] \rightarrow \texttt{zeros}[\mathrm{r},\ \mathrm{c}]}{\mathrm{m} : \mathscr{D}_{(r,k)}} \quad \frac{\mathrm{m} + \texttt{zeros}[\mathrm{r},\ \mathrm{c}] \rightarrow \mathrm{m}}{}$$

Figure 7.8 – A subset of simplification rules

$$\frac{\mathrm{m}_1 + \mathrm{m}_2 \leftrightarrow \mathrm{m}_2 + \mathrm{m}_1}{} \quad \frac{(\mathrm{m}_1 + \mathrm{m}_2) + \mathrm{m}_3 \leftrightarrow \mathrm{m}_1 + (\mathrm{m}_2 + \mathrm{m}_3)}{}$$

$$\frac{(\mathrm{m}_1 \cdot \mathrm{m}_2) \cdot \mathrm{m}_3 \leftrightarrow \mathrm{m}_1 \cdot (\mathrm{m}_2 \cdot \mathrm{m}_3)}{} \quad \frac{(\mathrm{m}_1 \cdot \mathrm{m}_2)^\top \leftrightarrow \mathrm{m}_2{}^\top \cdot \mathrm{m}_1{}^\top}{}$$

$$\frac{(\mathrm{m}_1 + \mathrm{m}_2)^\top \leftrightarrow \mathrm{m}_1{}^\top + \mathrm{m}_2{}^\top}{} \quad \frac{(\mathrm{m}_1 \rightrightarrows \mathrm{m}_2)^\top \leftrightarrow \mathrm{m}_1{}^\top \downdownarrows \mathrm{m}_2{}^\top}{}$$

$$\frac{\mathrm{m}_1 \cdot (\mathrm{m}_2 + \mathrm{m}_3) \leftrightarrow \mathrm{m}_1 \cdot \mathrm{m}_2 + \mathrm{m}_1 \cdot \mathrm{m}_3}{}$$

$$\frac{\texttt{let } x = \mathrm{m}_1 \texttt{ in } \mathrm{m}_2 \leftrightarrow \mathrm{m}_2\,[x_1 := \mathrm{m}_1]}{}$$

Figure 7.9 – A subset of equivalence rules

there are many `zeros` matrices that have been created throughout the derivation process. After applying several simplification rules as demonstrated in Fig. 7.8, our running example is simplified to the following expression:

$$\delta_G\texttt{.G + G.}\delta_G\texttt{ + }\delta_G\texttt{.}\delta_G$$

Equivalence Rules

These are rules that define equivalent expressions lhs ↔ rhs. In particular, it is not clear beforehand which form of the expression (lhs or rhs) will result in an optimized program down the road. However, their presence is important not only because of their probable performance improvement, but also their possible impact on permitting other rewrite rules later. This effect is known as *enabling transformations* in compilers. Fig. 7.9 presents a subset of equivalence rules used within Lago. Common subexpression elimination (`CSE`) and forward substitution (`FS`) are among these rules. In essence, these rules are the reverse of each other, hence it is not clear which one should be applied. General purpose compilers adopt `CSE` as a best-effort heuristic to enhance performance. That is, they apply them whenever possible as an enabling compiler optimization. Moreover, other domain-specific frameworks such as SystemML adopt these optimizations as static optimization opportunities, i.e., heuristics.

In contrast, we argue that decisions about these optimizations should be taken under the light of cost-based optimization. In particular, algebraic and domain structure information often enable optimizations that override these general compiler heuristics. To illustrate this, consider our running example: δ_G`.G + G.`δ_G `+` δ_G`.`δ_G, where the matrix G has dimensions $n \times n$. Now, suppose that δ_G is a simple single entry change which can represented as an outer-product $u.v^{\mathrm{T}}$, i.e., $(n \times 1) \times (1 \times n)$. Using simple heuristics a compiler can directly detect that the expression $\delta_G = u.v^{\mathrm{T}}$ occurs several times within the program, hence by applying `CSE`, one can compute $u.v^{\mathrm{T}}$ once and then reuse it later on for further computation. In particular, the derived program becomes:

$$\texttt{let D := } u.v^T \texttt{ in D.G + G.D + D.D}$$

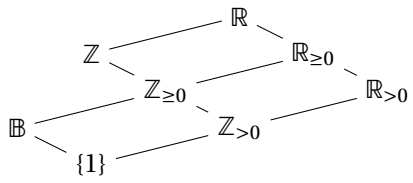
Although `CSE` saved us from re-computing uv^{T}, i.e., saving $\mathcal{O}(n^2)$ operations, it results in more costly computations further on, in particular, the $\mathcal{O}(n^3)$ matrix multiplications `G.D`, `D.G`, and `D.D`. On the other hand, given that $u.v^{\mathrm{T}}$ is an outer product of two vectors, using cost-based optimization, it is much cheaper, i.e., $\mathcal{O}(n^2)$ overall, to avoid `CSE` and keep the computations inlined as follows:

$$u.(v^T.\texttt{G}) + (\texttt{G}.u).v^T + u.(v^T.u).v^T$$

This pattern occurs frequently in the derivation of incremental programs.Equivalence rewrite rules construct programs which should be included in the search space. This is because it is not possible to decide locally if a rewrite rule will produce a better program or not. Even if it locally generates a better program, it might disable further transformations along the way. In other words, in order to not fall into a local optimum, one should traverse the search space of equivalent programs and rely on the search algorithm along with cost estimation to decide globally which program to pick.

To find the best program, we can use an exhaustive search algorithm, but given that the search space is very large, coming up with an effective search algorithm is challenging. Lago uses breadth-first search (BFS) as its main search algorithm. However, in order to prune the large search space, only the equivalence rules are participating in constructing the search space. The reduction rules are always deterministically applied, and the choice of applying them is *not* left to the search algorithm. This means that these rules do not increase the size of the search space. It would be interesting to see how randomized search algorithms [163] behave in comparison to the suggested pruned BFS algorithm.

Some transformation rules require specific conditions to check for their applicability in order to preserve semantics. These are known as *conditional rewrite rules* in the literature [182]. Apart from the structure of the program, these rules can use abstract domains to check their applicability. This way, the framework can make sure that the transformation rules are *sound*, meaning that they do not change the program semantics. For example, $\mathtt{m}^\top$ is equivalent to $\mathtt{m}$



$$\frac{\mathtt{m}_1 \circ \mathtt{m}_2 : \mathscr{T}_1 \vee \mathscr{T}_2}{\mathtt{m}_1 : \mathscr{T}_1 \;\; \mathtt{m}_2 : \mathscr{T}_2 \;\; \circ \in \{.,+,\rightrightarrows\}} \qquad \frac{\mathtt{m}_1{}^{\diamond} : \mathscr{T}_1}{\mathtt{m}_1 : \mathscr{T}_1 \;\; \diamond \in \{\top, -1\}}$$

$$\frac{\mathtt{let}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2 : \mathscr{T}_2}{\mathtt{m}_1 : \mathscr{T}_1 \;\; \mathtt{x} : \mathscr{T}_1 \vdash \mathtt{m}_2 : \mathscr{T}_2} \qquad \frac{\mathtt{iterate}[\mathtt{p}](\mathtt{m}_1)(\mathtt{x} \Rightarrow \mathtt{m}_2) : \mathscr{T}_2}{\mathtt{m}_1 : \mathscr{T}_1 \;\; \mathtt{x} : \mathscr{T}_1 \vdash \mathtt{m}_2 : \mathscr{T}_2}$$

Figure 7.10 – The lattice of data types and the typing rules for a subset of Lago DSL.

in the case that the matrix $\mathtt{m}$ is symmetric. Next, we discuss how such information about the programs can be inferred.

7.4 Abstract Interpretation

DBMSes extensively use workload-specific information such as selectivity and cardinality in order to estimate query costs during query optimization. We observe that such an idea of *abstract domains* used by query optimizers can be generalized and used for other domains. For example, in the domain of matrix algebra, symmetry, dimensions, and structure of matrices are workload-specific information that permits further enhancements. To that end, Lago permits encoding information about matrix and expression properties. The data type is the most obvious example of such information, e.g., a binary matrix or a matrix of real numbers. Matrix dimensions are another, which are, in essence, very similar to relation cardinalities. The sparsity and the rank of a matrix are similar to the notion of selectivity in databases. Finally, the cost estimate itself is another abstract domain that can be used during cost-based optimization.

Abstract interpretation [70] is a methodology for acquiring information about the semantics of a program (e.g. control-flow, data-flow) in a well-defined way, without fully executing the program itself. Abstract interpretation coarsens the operational semantics of the language by transforming the actual program values (i.e., concrete domain) into more abstract values (of an abstract domain). As an example, instead of computing the actual value of the result of adding two matrices, abstract interpretation says that the elements of the result matrix are *some* real numbers, i.e., the data type of matrix elements is $\mathbb{R}$.

There are many benefits to abstract interpretation: 1) First, to verify program correctness. For example, in the case of matrix multiplication, dimensions are used to check if the number of columns of the first matrix is equal to the number of rows of the second matrix. 2) Secondly, to guide the optimizing compiler to reason about optimization opportunities by evaluating cost

estimates. 3) Thirdly, to enable conditional rewrites. 4) Finally, to enable further specialization opportunities during the code generation phase.

Lago is extensible; in a sense, one can introduce more abstract domains that capture the workload-specific information available. These abstract domains are not limited to the information about the input data provided by the input program. Lago uses abstract interpretation to propagate these abstract domains throughout the whole program whenever possible. This is achieved by defining inference rules for various abstract domains as described next.

7.4.1 Abstract Domains

The user provides information about the input matrices by specifying their associated abstract domain. In order to leverage this information, abstract interpretation propagates the abstract domains throughout the program. This way, the optimizing compiler can utilize the provided information for the whole program. Abstract interpretation uses a lattice-based model and inference rules in order to infer the abstract domain of an expression based on its input data through a bottom-up derivation approach. The inference rules should be specified for all the constructs of the core Lago DSL. The inference rules for the extension constructs can be derived from the core constructs used in their implementation (cf. Fig. 7.5).

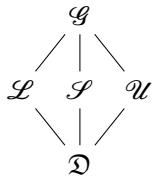
We now enumerate several useful abstract domains.

Data Type. Types can be represented as an abstract domain [69]. Every matrix expression is associated with a data type for its underlying elements. The matrix elements can have the following data types: boolean ($\mathbb{B}$), integer ($\mathbb{Z}$), and real ($\mathbb{R}$). Moreover, the numerical types can be further constrained to positive ($\mathbb{Z}_{>0}$, $\mathbb{R}_{>0}$) and non-negative ($\mathbb{Z}_{\geq 0}$, $\mathbb{R}_{\geq 0}$) data types, which are useful for reasoning on applications such as non-negative matrix factorization.

These data types form a lattice, which is shown in Figure 7.10. This lattice is useful for specifying the result data type of a matrix-expression. Operations on matrices with different element data types can lead to a matrix with the least upper bound data type. For example, the multiplication of a binary matrix with a positive real matrix, leads to a non-negative real matrix, denoted as follows: $\mathbb{B} \vee \mathbb{R}_{>0} = \mathbb{R}_{\geq 0}$, where $\vee$ is the *join* operator of the mentioned lattice, responsible for computing the least upper bound type.

Matrix Structure. Matrices can have different structures such as lower/upper triangle, symmetric, sparse/dense, etc. These structures can be expressed as abstract domains, and are useful for using certain transformation rules and specializations (e.g., storing only half of the matrix when the matrix is symmetric).

Matrix structures also form a lattice, which is demonstrated in Figure 7.11. $\mathscr{G}$ represents a matrix without a particular structure, whereas $\mathscr{L}$, $\mathscr{U}$, $\mathscr{S}$, and $\mathfrak{D}$ are representing lower/upper triangle, symmetric, and diagonal matrices. These structures can be further extended to be more fine grained for encoding the structure of each block of a matrix [310]. Additionally, one



$$\frac{\mathfrak{m}_1 \,.\, \mathfrak{m}_2 : \sigma_1 \vee \sigma_2}{\mathfrak{m}_1 : \sigma_1 \quad \mathfrak{m}_2 : \sigma_2 \quad \sigma_1, \sigma_2 \in \{\mathfrak{D}, \mathscr{L}, \mathscr{U}, \mathscr{G}\}}$$

$$\frac{\mathfrak{m}_1 + \mathfrak{m}_2 : \sigma_1 \vee \sigma_2}{\mathfrak{m}_1 : \sigma_1 \quad \mathfrak{m}_2 : \sigma_2 \quad \sigma_1, \sigma_2 \in \{\mathfrak{D}, \mathscr{L}, \mathscr{S}, \mathscr{U}, \mathscr{G}\}}$$

$$\frac{\mathfrak{m}^\top : \mathscr{L}}{\mathfrak{m} : \mathscr{U}} \qquad \frac{\mathfrak{m}^\top : \mathscr{U}}{\mathfrak{m} : \mathscr{L}} \qquad \frac{\mathfrak{m}^\top : \sigma}{\mathfrak{m} : \sigma \quad \sigma \in \{\mathfrak{D}, \mathscr{S}\}}$$

$$\frac{\mathfrak{m}.\, \mathfrak{m}^\top : \mathscr{S}}{\mathfrak{m} : \sigma} \qquad \underline{\texttt{diag[c][s]} : \mathfrak{D}}$$

Figure 7.11 – The lattice of several abstract domains for matrix structure and a subset of inference rules.

$$\frac{\mathfrak{m}_1 \cdot \mathfrak{m}_2 : \mathscr{D}_{(n,p)}, \mathscr{C}_{(c_1+c_2+n\cdot m\cdot p)}}{\mathfrak{m}_1 : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_1)} \quad \mathfrak{m}_2 : \mathscr{D}_{(m,p)}, \mathscr{C}_{(c_2)}} \qquad \frac{\mathfrak{m}_1 + \mathfrak{m}_2 : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_1+c_2+n\cdot m)}}{\mathfrak{m}_1 : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_1)} \quad \mathfrak{m}_2 : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_2)}}$$

$$\frac{\mathfrak{m}^\top : \mathscr{D}_{(m,n)}, \mathscr{C}_{(c+n\cdot m)}}{\mathfrak{m} : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c)}} \qquad \frac{\mathfrak{m}^{-1} : \mathscr{D}_{(n,n)}, \mathscr{C}_{(c+n^3)}}{\mathfrak{m} : \mathscr{D}_{(n,n)}, \mathscr{C}_{(c)}} \qquad \frac{\mathfrak{m}_1 \rightrightarrows \mathfrak{m}_2 : \mathscr{D}_{(n,m+p)}, \mathscr{C}_{(c_1+c_2+n(m+p))}}{\mathfrak{m}_1 : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_1)} \quad \mathfrak{m}_2 : \mathscr{D}_{(n,p)}, \mathscr{C}_{(c_2)}}$$

$$\frac{\texttt{let}\ \mathrm{x} = \mathfrak{m}_1\ \texttt{in}\ \mathfrak{m}_2 : \mathscr{D}_{(p,k)}, \mathscr{C}_{(c_1+c_2)}}{\mathfrak{m}_1 : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_1)} \quad \mathrm{x} : \mathscr{D}_{(n,m)} \vdash \mathfrak{m}_2 : \mathscr{D}_{(p,k)}, \mathscr{C}_{(c_2)}} \qquad \underline{\texttt{diag[c][s]} : \mathscr{D}_{(c,c)}, \mathscr{C}_{(c^2)}}$$

$$\underline{\texttt{vect[c](s)} : \mathscr{D}_{(1,c)}, \mathscr{C}_{(c)}} \qquad \frac{\texttt{iterate[p]}(\mathfrak{m}_1)(\mathrm{x} \Rightarrow \mathfrak{m}_2) : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_1+p\cdot c_2)}}{\mathfrak{m}_1 : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_1)} \quad \mathrm{x} : \mathscr{D}_{(n,m)} \vdash \mathfrak{m}_2 : \mathscr{D}_{(n,m)}, \mathscr{C}_{(c_2)}}$$

Figure 7.12 – Inferring dimensions and cost of matrices.

can encode information about density by keeping track of the number of zero elements and the dimension of a matrix.

Dimensions and Cost Model. One of the most essential abstract domains is cost estimation. The cost estimate is used to guide the search space in choosing efficient derived programs. This is in contrast to most systems doing empirical search, that evaluate the run time cost of programs by executing them directly on the machine [373]. Although this sacrifices precision, it had three advantages. First, it can estimate the runtime cost of programs abstractly independent from the underlying processing substrate. Second, as there is no need for running the program, it can explore more programs in a given fixed amount of time. Third, it can extend the cost model without changing the underlying architecture of that runs the Lago framework. Currently, the cost estimate is modelled as a function of the number of arithmetic operations that need evaluation. Similar to cost estimation used in the query optimizer of database systems which requires cardinality information of relations, the cost estimation in Lago also requires knowledge about the dimensions of matrices. This means, before starting the inference process for the cost estimation abstract domain, the dimension inference should be performed. The inference rules for dimensions and estimation are given in Fig. 7.12, where $\mathscr{D}_{(n,m)}$ represents the dimension of a matrix with n rows and m rows, and $\mathscr{C}_{(f)}$ corresponds to f floating point operations.

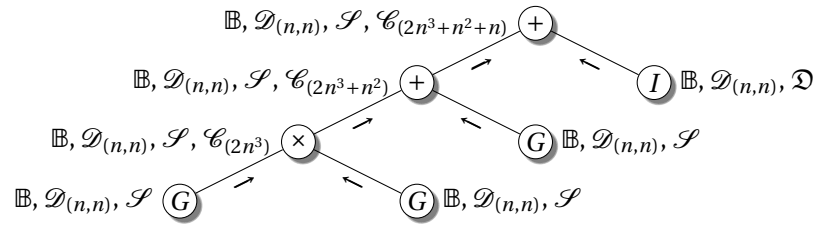


Figure 7.13 – Abstract interpretation propagates abstract domains in a bottom-up manner.

Returning back to our running example, the search algorithm uses the estimated cost and favors forward substitution of δ_G over CSE given the following parenthesization order:

$$u.(v^T.\mathtt{G}) + (\mathtt{G}.u).v^T + u.(v^T.u).v^T$$

Cost estimation is extensible as well. For instance, in addition to dimensions, one can introduce and define additional abstract domains that can introduce specialized solutions and, therefore, better cost estimation. For example, based on the structure of matrices, e.g., upper/lower triangular, one can use specialized matrix multiplication algorithms, e.g., SSYMM in BLAS, that only computes the required entries, thereby saving space and computation cost. This can be reflected in the cost model by inferring the structure of the matrix, before performing cost inference. By further reflecting this information as new inference rules in the cost estimation, one could further specialize the precision of the cost model estimation.

Example. Consider our original reachability example. If the input graph G is undirected, then it is symmetric and requires to be stored in a binary adjacency matrix. As depicted in Fig. 7.13, starting from these properties of matrix G, the abstract domain propagates upwards to infer the abstract domain of the intermediate and final results using the inference rules described before. This information is useful for specialization purposes, which is described next.

7.4.2 Specialized Code Generation

As explained in the previous section, in the graph reachability example the input G is an undirected binary graph. This abstract domain can be encoded into the input data and Lago propagates this information and tries to infer properties for the subsequent and intermediate statements. Abstract interpretation leverages specialization opportunities at the code generation phase. For instance, in the graph reachability program example, the following specialization opportunities are possible.

Bit Vectors. The domain values are either zeros or ones only. Accordingly, rather than representing the adjacency matrix entries using the more generic single or double-precision types, one can utilize compressed bit vectors to pack every eight cells into a single byte. As we will demonstrate later, this compact storage, allows for large matrix constructions, and avoids expensive communication costs for data shuffling in a distributed setting.

Boolean Algebra. The semiring operations, i.e., Boolean conjunction and disjunction, enable Boolean algebra optimizations and specialization opportunities. For instance, the dot product of two vectors can be translated as computing the bitwise-`AND` of the two bit vectors followed by evaluating if the result is bigger than zero. This leverages vectorized operations. Furthermore, one can benefit from short-circuiting by early terminating the computation of the dot product after reaching the first bit one, rather than passing over the entire bit vectors. Matrix multiplication $G \times G$ can be specialized along the same lines. Alternatively, in a general purpose environment, i.e. R or Matlab, this expression is evaluated using the following expression $(G \times G) > 0$. That is, it computes the matrix multiplication numerically first and then a logical indexing is applied over the result matrix to bring it back to the binary domain.

Symmetry. Matrix symmetry enables many specializations ranging from compact representation, e.g., lower triangular, to call specialized matrix operations that exploit symmetry. In this evaluation, we focus on a specific specialization that leverages the matrix layout. Since the bit vector matrices can be represented as rows or columns of bit vectors, there are two layout configurations, i.e., row-major layout and column-major layout. The operations on the matrices define the ideal layout representation of these matrices. For example, consider $G \times G$, ideally matrix G should have both row-major and column-major layouts to support direct use of the bitwise operations; otherwise transformations to the layout should be applied which incurs an overhead. However, if G is symmetric then $G^T = G$, which means that matrix G represents both logical layouts, independently from its underlying physical representation. This eliminates the need for transforming G's layouts and therefore its associated costs.

Lago provides an extensible code generator which leverages the inferred abstract domains in order to generate specialized efficient code for different targets. For distributed analytical workloads, we implement a Spark code generator, for the subset of Lago DSL required for the experiments in this thesis, together with a runtime using Spark RDDs that allow for mutable operations on block matrices that call efficient BLAS routines locally.[5]

7.5 Implementation

In Section 7.3, we have discussed the building blocks that constitute the Lago framework. In this section, we discuss how all these parts are put together to generate incremental trigger programs. An incremental program consists of an initialization phase that precomputes and materializes the initial value of the intermediate and result expressions, and a trigger function that computes a set of Δ expressions that update their corresponding views.[6]

[5] All experiments on matrix RDDs have a predefined block distribution of 10×10 blocks. Efficient partitioning of matrices is orthogonal to the discussion of this thesis and can be handled by other systems like SystemML [118]. For example, Lago can generate SystemML matrix programs, i.e., compositions of matrix operations using the SystemML DSL, which is then handled and optimized by SystemML.

[6] For presentation clarity purposes, in the rest of this thesis we omit the "$\cdot$" symbol whenever multiplication is understood within context.

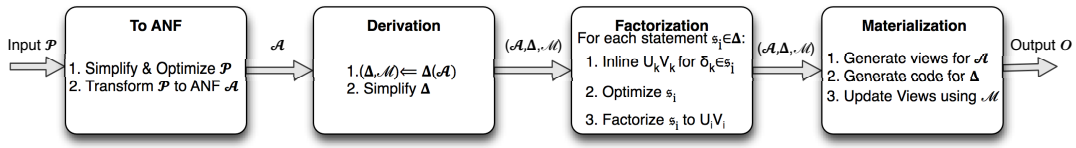


Figure 7.14 – Lago IVM phases.

$\mathscr{P}$:

```
let x0 = G + id
let x1 = G x0 + id
    x1
```

(a) Simplified program.

$\mathscr{A}$:

```
let x0 = G + id
let x1 = G x0
let x2 = x1 + id
    x2
```

(b) ANF version of the program.

Δ:

```
let δ0 = δG
let δ1 = δG x0 + G δ0 + δG δ0
let δ2 = δ1
    δ2
```

$\mathcal{M}$:

$$[G \rightarrow \delta_G, x_0 \rightarrow \delta_0, \\ x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2]$$

(c) After the Derivation phase.

Δ:

```
let U0 = u
let V0 = v^T
let U1 = u ⇉ G U0
let V1 = v^T (x0 + U0 V0) ⇊ V0
let U2 = U1
let V2 = V1
    U2 V2
```

(d) After the Factorization phase.

```
// Global Views:
M_G := G
M_0 := G + id
M_1 := G M_0
M_2 := M_1 + id
// P Inc(u,v):
/* Generate Δ */
// Update Views:
M_G += u v^T ; M_0 += U_0 V_0 ;
M_1 += U_1 V_1 ; M_2 += U_2 V_2 ;
```

(e) After Materialization phase.

Figure 7.15 – Going through the IVM phases of program $\mathscr{P}$ from Fig. 7.6.

To achieve these goals, an input program accepted by Lago undergoes several phases. As depicted in Figure 7.14, the input program passes through four stages, namely, ANF, DERIVATION, FACTORIZATION and MATERIALIZATION. Next, we explain each phase while illustrating it using our running example.

1. ANF. As a preprocessing step, an input Lago program $\mathscr{P}$ is first simplified and then optimized using cost-based optimization to find an appropriate ordering of operations. After that, it is converted to the administrative normal form (ANF) $\mathscr{A}$ [110]. In our context, the ANF is defined as a simple representation of the program that assigns a unique variable to each

subexpression while also ensuring that each variable is assigned before it is used. The ANF is extensively used in optimizing compilers due to its simplistic canonical representation that facilitates reasoning and optimization [110]. To explain this, consider our reachability program $\mathscr{P}$ as depicted in Figure 7.6. First, the program is simplified using the simplification rules in Figure 7.8 where the `iterate` construct is unfolded and multiplications with the identity matrix are omitted, yielding the simple program in Fig 7.15a. Afterwards, in Figure 7.15b, the simplified program is transformed into its ANF $\mathscr{A}$, where each subexpression containing one operation is assigned to a unique variable.

2. Derivation. In this phase, the delta derivation rules are recursively applied over the $\mathscr{A}$ program reducing it to Δ. During derivation, a map $\mathscr{M}$ is created that maps each intermediate result variable x_i in $\mathscr{A}$ to its corresponding delta δ_i variable. Moreover, simplification rules are applied whenever possible. This ensures that each statement $\mathbf{s}_i \in \Delta$ represents a sum of matrix products, i.e., $\Sigma_i \mathfrak{m}_i$ where $\mathfrak{m}_i$ is an expression of matrix products. Figure 7.15c presents the final delta program derived from $\Delta_G(\mathscr{A})$ and the corresponding map $\mathscr{M}$.

3. Factorization. The main goal of this phase is to represent each δ_i variable in a compact factored form $U_i V_i$. This is achieved by recursively propagating and forward substituting each δ_k variable with its corresponding factored form $U_k V_k$ within each statement that calls δ_k. Note that forward substitution begins with the initial substitution of δ_G with uv^T. Given that Δ is in ANF form, then it is ensured that δ_k and therefore, $U_k V_k$ is defined before being used. Then, each statement $\mathbf{s}_i$ is optimized using the equivalence rules along with cost-based optimizations. In particular, the search algorithm explores the search space created by applying valid equivalence rules on the statement $\mathbf{s}_i$. Then, it chooses the program with minimum inferred cost as explained in Section 7.4.1. Notice that search is confined within the scope of a single statement $\mathbf{s}_i$. This guided approach avoids searching a vast search space that includes all the statements of the whole program.

This ensures that each statement $\mathbf{s}_i \in \Delta$ is a sum of matrix products containing $U_k V_k$, i.e., $\Sigma_j \mathfrak{p}_j \mathfrak{q}_j$ where $\mathfrak{p}_j$ and $\mathfrak{q}_j$ are expressions of matrix products and j represents the index of minimum dimension within the overall matrix products. Finally, each statement is factorized to $U_i V_i$ using the FACTORIZATION rule in Figure 7.9, such that $U_i = \mathfrak{p}_0 \rightrightarrows \mathfrak{p}_1 \cdots \rightrightarrows \mathfrak{p}_n$ and similarly $V_i = \mathfrak{q}_0 \downdownarrows \mathfrak{q}_1 \cdots \downdownarrows \mathfrak{q}_n$. Figure 7.15d presents the factorized Δ program with its corresponding updated map $\mathscr{M}$.

4. Materialization. Finally, in this phase Lago generates the incremental program. First, it generates global materialized views for each of the variables defined in $\mathscr{A}$. Then, it generates the trigger program Δ derived from the previous stage. Finally, it updates the global views with their corresponding δ_i expression derived in $\mathscr{M}$. Figure 7.15e illustrates the final incremental program for P given incremental changes to G, i.e., $\delta_G := uv^T$.

	Iter. (#)	Compilation Time (s)	Rules (#)	Equiv. Rules (%)	Programs Revisited (%)
OLS	-	36	10869640	7%	90.98%
BGDS	4	0.567	4526	7%	75.56%
BGDS	16	0.944	28826	4%	81.08%
BGDS	128	15.139	1328854	0.8%	87.76%
BGDS	256	128.599	3841206	0.5%	87.35%

Table 7.2 – Report on compilation metrics.

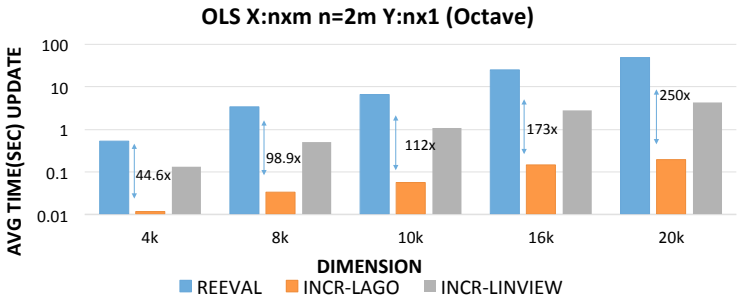
7.6 Evaluation

In the previous sections, we have presented a concrete framework for expressing, deriving, and optimizing incremental view maintenance of matrix algebra programs. In this section, we demonstrate the performance of the derived incremental programs in comparison to re-evaluation. We illustrate three case studies that build upon matrix algebra: computing linear regression, matrix powers, and evaluating graph reachability and shortest path after k hops. Moreover, we evaluate the opportunity benefits of specialization leveraged by inferred abstract domains. We show how Lago pushes the burden and complications of IVM derivation, optimization, and low-level specialization down to the compiler framework, while generating trigger programs that achieve orders of magnitude better performance.

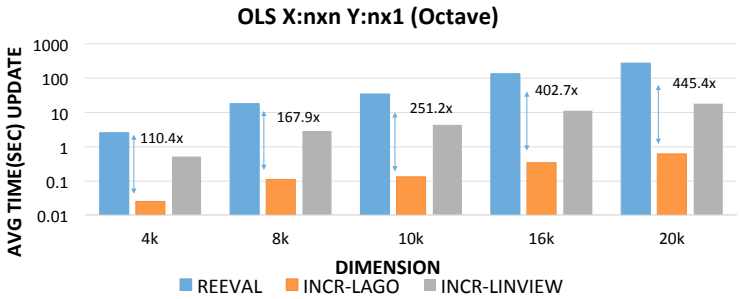
Experimental Environment. The experiments are conducted under two different configurations: *a*) **Local**: For moderate size experiments, we use a multiprocessing workstation environment with a 2.66GHz Intel Xeon with 2×6 cores, each with 2 hardware threads, 64GB of DDR3 RAM, and Mac OS X El Capitan 10.11.5. Dense BLAS operations are supported through the underlying Mac VecLib framework. *b*) **Distributed**: For large-scale experiments, we use a cluster of 100 server instances connected via a full-duplex 10GbE network. Each instance is equipped with an Intel Xeon E5-2630L @ 2.40GHz server with 2×6 cores, each with 2 hardware threads, 15MB of cache, 128GB of DDR3 RAM, and Ubuntu 14.04.2 LTS. We rely on the ATLAS library to support multithreaded BLAS operations. We use GNU Octave 4.2.0, Spark 1.6.1, YARN 2.7.1, and Scala 2.10.4. For all IVM experiments, unless stated otherwise, we simulate a stream of rank-one updates to evaluate the performance.

7.6.1 Incremental Linear Regression

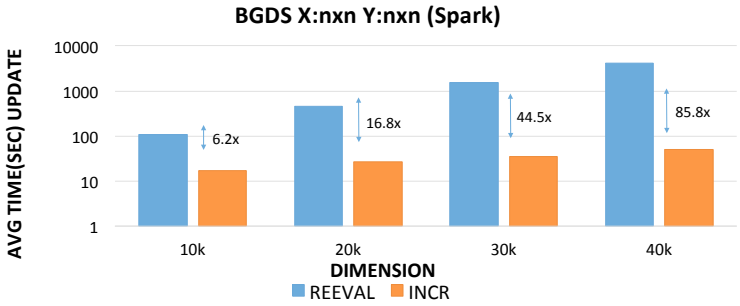
Linear regression is an approach for modelling the relationship between the dependent variables $Y_{m \times j}$ and independent variables $X_{m \times n}$. It is extensively used in fitting predictive models to an observed dataset of X and Y values. The goal is to estimate, given the input, the unknown parameters. The Ordinary Least Squares (OLS) method solves this problem by finding a statistical estimate of the parameter β^* best satisfying $Y = X\beta$. The program, written as a linear algebra program, is $\beta^* := (X^T X)^{-1} X^T Y$. The evaluation of the previous closed form equation requires expensive $\mathcal{O}(n^3)$ matrix-matrix computations for computing matrix



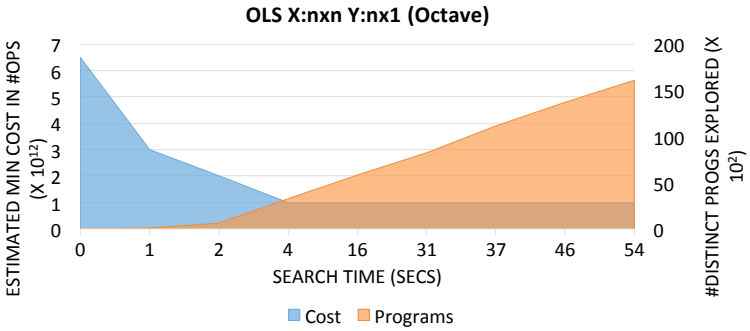
(a) Performance results for Ordinary Least Squares on a single machine.



(b) Performance results for Ordinary Least Squares on a single machine.



(c) Performance results for Batched Gradient Descent on a cluster.



(d) Cost-based optimization search results.

Figure 7.16 – Performance evaluation of Incremental Linear Regression.

multiplications and inverses. It is far from efficient to re-evaluate the expression over and over again as the input datasets evolve. Input datasets naturally evolve by either growing (for example as more observations are accumulated to X and Y), or by changing (as more accurate estimates arrive). Alternatively, Lago derives materialized views of precomputed intermediate results and their corresponding delta expressions to generate trigger programs.

In this set of experiments, we evaluate the performance of the common machine learning task of building a linear regression model given the independent and dependent variables X and Y respectively. In particular, we experiment on two programs: *a*) **OLS**: the Ordinary Least Squares method to evaluate the statistical estimate β^* via the matrix expression $\beta^* := (X^{\mathrm{T}} X)^{-1} X^{\mathrm{T}} Y$, and *b*) **BGDS**: the Iterative Batch Gradient Descent method (BGDS) which, similar to OLS, evaluates the statistical estimate Θ that is computed via the recurrence relation $\Theta_{i+1} := \Theta_i - X^T(X\Theta_i - Y)$. Given ΔX changes, Lago derives the incremental version of each program and generates the corresponding trigger code. Furthermore, to demonstrate portability, we generate Octave code (**Local**) for OLS and Spark code (**distributed**) for BGDS. For comparison, we compare the re-evaluation of the original programs REEVAL against the derived trigger code INCR.

OLS Evaluation. We conduct a set of experiments to evaluate the statistical estimator β^*. The predictor matrix X has dimension $(n \times m)$ and the response matrix Y is of dimension $(n \times 1)$. Given a continuous stream of ΔX updates on X, Figure 7.16a and Figure 7.16b compare the average execution time per ΔX update of REEVAL, INCR-LAGO, and INCR-LINVIEW [258] for different values of n (x-axis). In particular, we experiment on two settings: 1) when X is a tall skinny matrix with dimensions $n \times m$, where $n = 2m$ (Figure 7.16a), and 2) when X is a square matrix with dimensions $n \times n$ (Figure 7.16b), i.e., $m = n$. The graphs illustrate how INCR-LAGO and INCR-LINVIEW outperform REEVAL in computing the β^* estimate. Notice the asymptotic difference between the two approaches; the performance gap between REEVAL and INCR-LAGO widens as the matrix size increases, i.e., 44.6x — 250x and 110.4x — 445.4x, respectively. The cost of REEVAL is dominated by costly $\mathcal{O}(n^3)$ matrix operations including matrix inversion and multiplication, whereas INCR-LAGO and INCR-LINVIEW avoid matrix inversions and evaluate cheaper $\mathcal{O}(n^2)$ matrix multiplications. Finally, INCR-LAGO is on average 20x better than INCR-LINVIEW thanks to the cost-based optimizations provided by Lago.

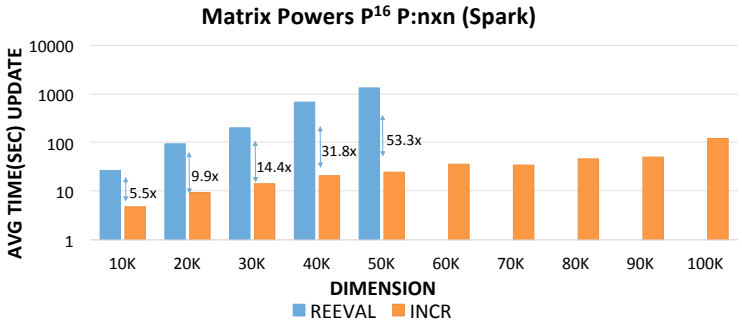
BGDS Evaluation. We also experiment on the batch gradient descent method to compute the estimate Θ for the linear regression problem. This method is usually used, instead of OLS, as a fast approximate or when the expression $X^T X$ is singular, i.e., non-invertible. For experimental purposes, we set X and Y with dimensions $(n \times n)$ and we fix the number of iterations to 16 assuming that the result converges to an appropriate solution after this number of steps. Figure 7.16c demonstrates the average computation time per incremental update ΔX for each of REEVAL and INCR. The results demonstrate 6.2x-85.8x performance speedups as the dimension size n increases. Distributed matrix multiplications require partitioning the matrices appropriately to evaluate the final result. This poses large communication overhead due to repartitioning. On the other hand, if one of the matrices undergoing multiplication is

fairly small, e.g., vector, it is broadcasted to all partitions instead of repartitioning the bigger matrix. Given that the delta expressions in INCR are materialized in factored forms, their multiplications are much cheaper. Therefore, INCR avoids both costly matrix multiplication operations and expensive communication overheads.

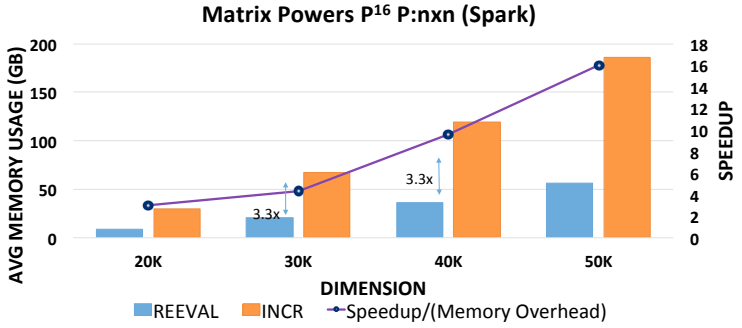
Search Space Evaluation. We also evaluate the explored search space using the traditional breadth-first-search (BFS) for both OLS and BGDS. OLS represents a small program and BGDS represents a big size program, i.e., defined by the number of iterations. We experiment on two different search configurations. For the OLS program with $n = 10000$ we run a complete BFS search on the whole derived delta program, whereas for BGDS, cost-based optimization complies with the phased approach described in Section 7.5 which confines the search on each statement independently from the other statements. Figure 7.16d illustrates two dimensions against elapsed search time. First, the number of distinct programs explored and secondly, the minimum inferred cost of the explored programs. The cost here depicts the sum of costs for both the original and the trigger program. Notice how the minimum cost decays fast during the early stages of the search as more programs are being explored.

The search begins with an initial program requiring 2 matrix multiplications, 1 matrix-vector multiplication, 1 matrix inverse, and 2 matrix transposes. That is a sum of $3n^3$ and $3n^2$ operations. Moreover, the initial trigger program, which is achieved by naïvely replacing each X with X+ΔX, requires $3n^3$ and $6n^2$ operations. All in all, this requires $6n^3 + 9n^2$ operations which when substituted with $n = 10000$ gives around 6 billion operations as depicted in the figure at time 0. At time 1, the search algorithm is able to transform all expensive operations with run time cost of $\mathcal{O}(n)^3$ in the trigger program to cheaper $\mathcal{O}(n)^2$ ones. Then the total cost is reduced to that of the original program (pre-computations) which accounts for 3 billion operations as depicted in Figure 7.16d. Interestingly, the search algorithm finds a simple equivalent program as follows $\beta := X^{-1}Y$. Although the program is numerically unstable in comparison to computing the pseudo-inverse $(X^TX)^{-1}$, it is analytically equivalent to the original program and it is much cheaper to evaluate as it only requires computing one matrix inverse (n^3). The program is found at time 4 secs in Figure 7.16d.

The search reaches a point where it introduces negligible savings. The search algorithm finds the minimal cost at second 36, after it has explored a search space created from applying 10,105,018 simplification rules and 764,622 equivalence rules. To avoid re-visiting the same programs within the search space, we maintain a cache that saves the hash-codes of the canonical representation of visited programs, i.e. canonical representation of the IR tree. This saves a lot from doing redundant work. For instance after 36 seconds the cache reports 90.983% hits and 9.016% misses. This suggests that a large number of the generated candidate programs have been explored before, which also means that many of the different orderings of the transformation rules yield the same programs.



(a) Scalability results.



(b) Scalability and additional storage results.

Figure 7.17 – Evaluation results for search-space and scalability metrics.

Zipf factor	5.0	4.0	3.0	2.0	1.0	0.0
Octave (10K)	6.3	6.8	7.5	10.9	68.4	236.5
Spark (30K)	28.1	41.5	67.3	186.1	508.9	1678.8

Table 7.3 – The average Octave and Spark view refresh times in seconds for INCR of P^{16} and a batch of 1,000 updates. The row update frequency is drawn from a Zipf distribution.

7.6.2 Incremental Matrix Powers

Matrix powers play an important role in many different domains including evaluating the stochastic matrix of a Markov chain after k steps and solving systems of linear differential equations using matrix exponentials. They also lay the foundation for more advanced analytics like batch gradient descent and furthermore, computing graph analytics. Consider the same running example as in section 7.2 that computes the 4^{th} power of an input matrix A. The original program can be represented using a simple `iterate` construct, similar to Fig 7.6. Once again, the evaluation of the program requires expensive $\mathcal{O}(n^3)$ matrix-matrix computations. Re-evaluation of the entire program on any delta change $\delta_A := uv^{\mathrm{T}}$ is a costly process. On the other hand, Lago derives the delta of these expressions on each incremental change.

In this section we evaluate several dimensions of IVM in comparison to re-evaluation, in particular, scalability, memory consumption of materialized view and performance of batch updates. For these experiments we evaluate on the matrix powers problem on dense matrices, in particular we compute the matrix power P^{16} on Spark. Figure 7.17a shows how incremental evaluation outperforms evaluation as previous results. Moreover, INCR can scale to larger dimensions, i.e. n= 100k, whereas REEVAL cannot go beyond 50k, as the sizes of shuffled data for matrix multiplication increase resulting in large communication overheads and unmaintainable states at each machine.

On the other hand, IVM requires additional storage for maintaining materialized views of intermediate results. Figure 7.17b demonstrates the average memory usage of INCR in comparison to REEVAL. INCR consistently uses 3.3x more storage no matter the dimension n, that is because the program maintains 4 intermediate results, in particular P^2, P^4, P^8, and the result P^{16}. To compare the performance speedup gains in comparison to the costs of extra storage, the figure also demonstrates the ratio between (speedup gain)/(storage-cost). The results show how the gains ratio keep on increasing with the dimensions size, i.e., 3x — 16x. This is consistent with the asymptotic increase in the computational gain versus the constant increases in storage costs.

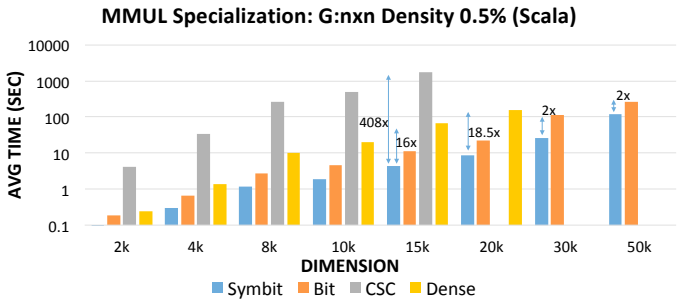
In the final set of experiments we explore the efficiency of IVM under batches of updates. We simulate a use case where some regions (rows) of the matrix P are updated more frequently than others. The frequency of updates is set by a Zipf distribution that is controlled by the Zipf exponent factor. When the factor value increases, it simulates a more skewed distribution, on the other hand, if it decreases it converges more towards a uniform distribution of changes. Table 7.3 reports on the performance results of IVM of P^{16} under a batch of 1000 tuples. As the Zipf factor tends to a uniform distribution, the overall rank of the updates increases and thus IVM looses its benefit in comparison to re-evaluation. To put the results in context, the cost of re-evaluation is 99.1 seconds and 203.4 for Octave and Spark respectively.

7.6.3 Graph Analytics

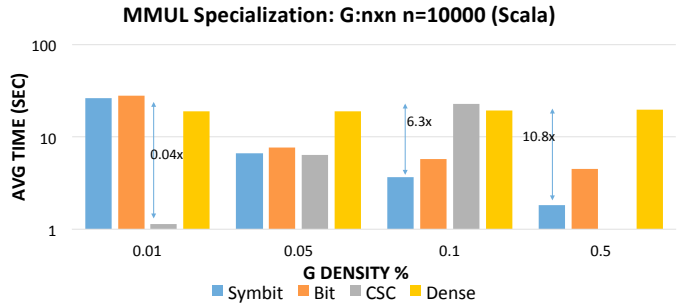
Many graph computations can be formulated as matrix operations. In this section, we experiment on the all pairs k-hop reachability/shortest path problem for the undirected graph G. The program, in Fig. 7.6, is the same as the running example used through out this thesis. As mentioned earlier, different semiring configurations for this program result in different programs. In particular, a Boolean semiring $\langle \mathbb{B}, \vee, \wedge \rangle$ defines the reachability problem, whereas the tropical semiring $\langle \mathbb{R}, \min, + \rangle$ defines the shortest path problem.

Specialization using Abstract Domains

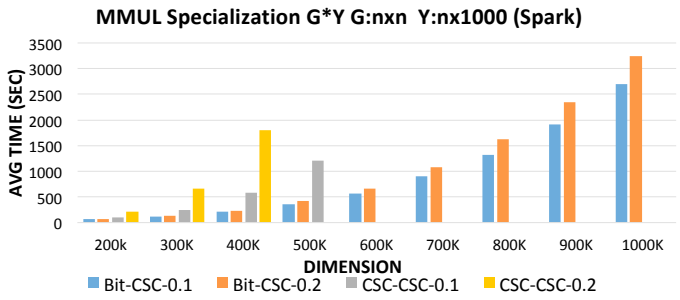
We have shown in Section 7.4.2 how Lago leverages various abstract domains to specialize graph computations. Next, we evaluate the impact of such specializations.



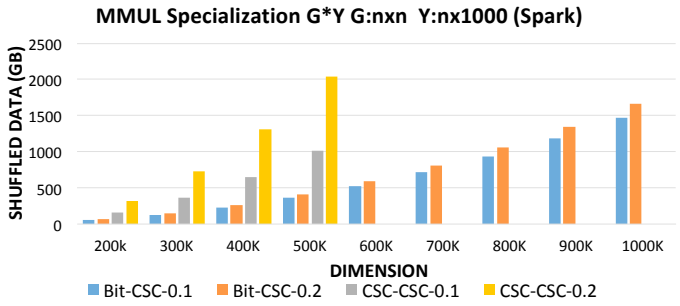
(a) Performance results of matrix-matrix (MM) multiplication by varying the dimension.



(b) Performance results of MM multiplication by varying the density.

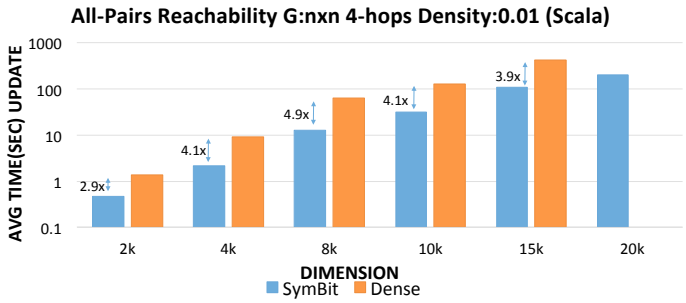


(c) Performance results of MM multiplication by varying the dimension on a cluster.

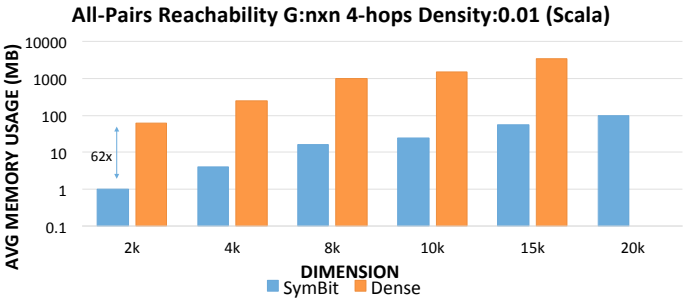


(d) Shuffled data of MM multiplication by varying the dimension on a cluster.

Figure 7.18 – Performance evaluation of specialization opportunities enabled by abstract interpretation.



(a) Run time of all-pairs reachability by varying the dimension.



(b) Memory consumption of all-pairs reachability by varying the dimension.

Figure 7.19 – Performance evaluation of the all-pairs reachability problem.

Evaluation. Graph analysis in this domain relies on matrix algebra operations and most notably on matrix multiplication which is commonly used in graph clustering, betweenness centrality, graph contraction, subgraph extraction, cycle detection, and quantum chemistry [48, 324, 94]. To that end, we focus our attention on the microbenchmark of evaluating the performance of specialized sparse matrix multiplications in two different settings:

Local. We compare between four different specialized implementations in Scala: a) `SymBit` represents the implementation of all the previous specializations. b) `Bit` is similar to `SymBit`, but excludes the symmetry specialization. c) `CSC` represents the implementation using the conventional Compressed Column Storage format [294]. This format is mainly used for sparse matrices and maintains matrix values along with their indexes in a compact form. d) `Dense` represents the multithreaded general purpose implementation that calls native dense BLAS routines for double precision operations. Notice that the following evaluation results are for a single thread except for `Dense` that leverages native multithreading capabilities.

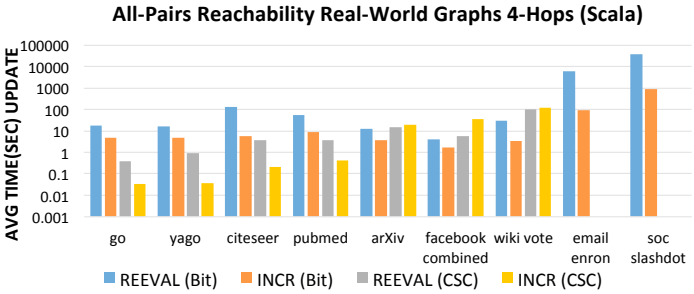
For the first set of experiments, we evaluate the potential of the aforementioned specializations. To that end, we focus on the micro-benchmarks of a single matrix-matrix multiplication $G \times G$. The input binary matrix is randomly generated with density 50%. Although real-world social graphs are really sparse, we set this density configuration because the reachability program results in denser (intermediate) results after a few iterations (hops). Fig. 7.18a reports the average execution time for each implementation with varying dimension size n. First, let us

compare the general purpose implementations `CSC` and `Dense`. Notice how `CSC` performs poorly in comparison to `Dense` as n grows and how it begins to fail after 20k. This is because of the high matrix density, which makes `CSC` inefficient for storage, i.e., saving index information, and for computation, i.e., no cache locality. Moreover, `CSC` computes on one core whereas `Dense` leverages all the available cores. On the other hand, the bit vector implementations `Bit` and `Symbit` exhibit scalable performance. They can scale to larger sizes n while maintaining a very compact storage representation up to n = 100k. Moreover, they benefit from short-circuiting given the density of the matrices. This saves from passing over all the entries within the whole matrix and achieves more than two order of magnitudes better performance than `CSC` and one order of magnitude better than `Dense` with one core only. Moreover, `SymBit` exploits the symmetric property and has 2x better performance than `Bit`.

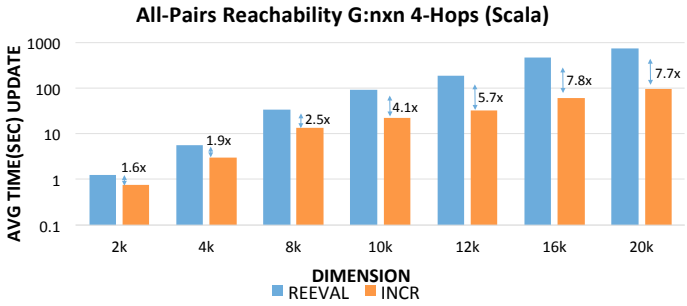
To explore the effect of Graph density on the previous implementations, we fix the dimensions size n = 10k and vary the density parameter. Fig. 7.18b illustrates the results. At the density level 0.01, `CSC` beats all the others due to the sparsity of the input which makes this format and implementation the most suitable. `SymBit` and `Bit` cannot leverage short-circuiting at this stage due to sparsity. However, as the density increases `SymBit` and `Bit` outperform the others. Notice how the performance of `Dense` does not change, as it is agnostic to the underlying structure of the input matrices.

Putting it all together, we experiment on the overall reachability program on randomly generated scale-free graphs G with density 0.01. Fig. 7.19a and Fig. 7.19b present the execution time and space utilization respectively. `SymBit` outperforms `Dense` by up to 3x-5x in performance and up to 62x in space savings. The reduction in space is consistent with the fact that bit vectors allow compacting 64 item into 8 bytes rather than a double value that represents a single item in 8 bytes.

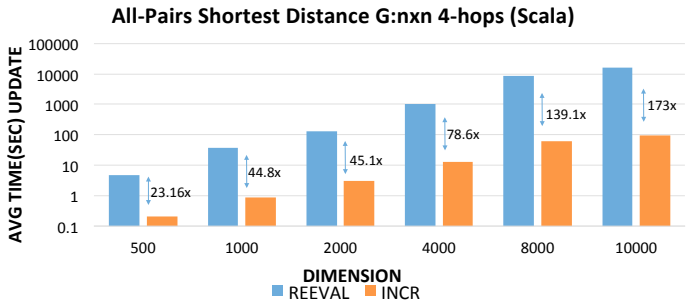
Distributed. In this experiment, we evaluate the large scale matrix multiplication $G \times Y$ under the numerical semiring, where G is a binary graph and Y is a matrix with arbitrary values. This operation is common in graph algorithms such as vertex clustering [120]. We compare between two implementations in Spark: *a*) `Bit-CSC` and *b*) `CSC-CSC`. `Bit-CSC` represents the first matrix in bit vector format and the other matrix in CSC. We experiment on graphs with two density settings 0.1 and 0.2 with variable dimension size n. Fig. 7.18c and Fig. 7.18d demonstrate how the specialized code `Bit-CSC` outperforms `CSC-CSC` as n grows. The performance gains are pronounced in the communication savings of shuffling compressed bit vectors rather than larger unnecessary general-purpose data-structures. Since communication dominates cost in a distributed environment, these savings result in better resource utilization and performance. Notice how `Bit-CSC` can scale to large graphs. In summary, Lago leverages useful abstract domains that open up opportunities for optimization at the code generation phase. As we have demonstrated, these optimizations can range from data-structure to computation specializations.



(a) Performance results on real-world graphs using `Bit` and `CSC` for representing matrices.



(b) Performance results on a synthetic graph with the density value of 0.01 using `Symbit`.



(c) Performance results on a synthetic graph with the density value of 0.01 using `Dense`.

Figure 7.20 – Performance evaluation of incremental graph programs on real-world and synthetic datasets.

Incremental Graph Analytics

By combining semiring configurations with IVM capability as described in Section 7.6.1, one can directly derive incremental graph analytics. Notice that the delta rules operate at the abstract level of matrix algebra operations. This permits reasoning about the derivation of incremental programs at a high level without delving into the details of the underlying operations used within the matrix operators, i.e., semiring.

Evaluation. We first evaluate the performance of the incremental version of 4-hops all-pairs reachability problem on real-world graphs obtained from [170] and the SNAP dataset [219] (cf. Table 7.4). Fig. 7.20a shows that in most cases the incremental variants outperform

Dataset	Vertices	Edges	Density (%)
arXiv	6000	66707	0.185
citeseer	10720	44258	0.038
go	6793	13361	0.028
pubmed	9000	40028	0.049
yago	6642	42392	0.096
facebook combined	4039	88234	0.540
wiki vote	7115	103689	0.204
email enron	36692	367662	0.027
soc slashdot	77360	905468	0.015

Table 7.4 – Properties of real-world graphs taken from [170] and SNAP [219].

re-evaluation. Also, as the matrices get more sparse, the CSC representations show a better performance, to the extent that REEVAL (CSC) outperforms INCR (BIT). However, as the density increases we observe that for the CSC, even the incremental version becomes worse than re-evaluation. This is because of the additional non-sparse intermediate matrix-vector multiplications. Finally, for larger graphs the CSC representation settings do not finish the processing after two days of computing, due to dense results.

Then, we evaluate the performance on synthetic graphs with the density value of 0.01. The update performance of the previously described 4-hops all-pairs reachability problem is compared against its incremental version as generated by Lago in Fig. 7.20b. Similarly, Fig. 7.20c compares the update performance for the 4-hops all-pairs shortest path problem. Lago is able to derive the incremental program of the matrix program independent from the underlying semiring semantics. Again, we can observe performance gains for IVM in comparison to re-evaluation, i.e., 1.6x — 7.7x in the case of reachability and 23.16x — 173x in the case of shortest paths. Notice how the performance gains in the boolean semiring are not as big as the other experiments. This is because of short-circuiting (cf. Section 7.4.2) which introduces large performance gains that are comparable to the gains of IVM. Also, notice how the shortest path program is much more slower than that of reachability, although they only differ in the semiring definition. The reason is that the Tropical semiring requires evaluating the **min** operator $\mathcal{O}(n^3)$ times. There is no specialized machine instruction for this operator as opposed to addition/multiplication in the numerical semiring and vectorwise and/or in the boolean semiring. Therefore, the **min** operator is expanded to many other machine instructions which is even much more costly in a loop, i.e., matrix multiplication.

7.7 Discussion

One of the key limitations of Lago is that elementwise operations (e.g., Hadamard product) cannot propagate factorized expressions down the program. For instance, the expression $X . * u.v^t$, cannot exploit the low rank structure and be further factorized into UV. That said, Lago

provides no performance guarantee for the incremental programs utilising such elementwise operators. However, Lago can provide efficient triggers for programs involving elementwise multiplication such as triangle counting for graphs [68]. Efficient factored representations for elementwise operations requires further research which we leave for the future.

One interesting future direction for Lago is supporting statistical queries. In the query $B = f(A)$, an update ΔA results in the update $\Delta B = f(A + \Delta A) - f(A)$. Similarly, an input error δ results in the output error $\epsilon = f(A + \delta) - f(A)$. One can use the Lago IVM infrastructure for efficiently computing the output error based on the input error. Based on the computed error value, Lago can decide to update the value immediately or wait for more updates to batch them together. In addition, one can use Lago for incrementally updating the influence of different inputs, which has been explored for *sensitivity analysis* in the context of probabilistic databases [181]. However, the applicability of the proposed technique to the Lago DSL needs further research.

Another extension is using other semiring configurations for supporting even further application domains. As an example, by supporting the set semiring (set union $\cup$ and set intersection $\cap$) one can express certain kinds of data-flow analysis problems (a technique for program analysis) in terms of matrix expressions [94]. Hence, it would be interesting to see how Lago can be used to incrementally compute the data-flow analysis of large programs.

8 Compiler-Compilation for Embedded DSLs

Everything that happens once can never happen again. But everything that happens twice will surely happen a third time.

– Paulo Coelho, The Alchemist

In this chapter, we present a framework to generate compilers for embedded domain-specific languages (EDSLs). This framework provides facilities to automatically generate the boilerplate code required for building DSL compilers on top of extensible optimizing compilers. We evaluate the practicality of our framework by demonstrating several use-cases successfully built with it.

8.1 Introduction

Domain-specific languages (DSLs) have gained enormous success in providing productivity and performance simultaneously. The former is achieved through their concise syntax, while the latter is achieved by using specialization and compilation techniques. These two significantly improve DSL users' programming experience.

Building DSL compilers is a time-consuming and tedious task requiring much boilerplate code related to non-creative aspects of building a compiler, such as the definition of intermediate representation (IR) nodes and repetitive transformations [43]. There are many extensible optimizing compilers to help DSL developers by providing the required infrastructure for building compiler-based DSLs. However, the existing optimizing compilation frameworks suffer from a steep learning curve, which hinders their adoption by DSL developers who lack compiler expertise. In addition, if the API of the underlying extensible optimizing compiler changes, the DSL developer would need to globally refactor the code base of the DSL compiler.

The key contribution of this work is to use a generative approach to help DSL developers with the process of building a DSL compiler. Instead of asking the DSL developer to provide the boilerplate code snippets required for building a DSL compiler, we present a framework which

automatically generates them.

More specifically, we present Alchemy, a *language workbench* [113, 103] for generating compilers for *embedded DSLs* (EDSLs) [158] in the Scala programming language. DSL developers define a DSL as a normal library in Scala. This plain Scala implementation can be used for debugging purposes without worrying about the performance aspects (handled separately by the DSL compiler).

Alchemy provides a customizable set of annotations for encoding the domain knowledge in the optimizing compilation frameworks. A DSL developer annotates the DSL library, from which Alchemy generates a DSL compiler that is built on top of an extensible optimizing compiler. As opposed to the existing compiler-compilers and language workbenches, Alchemy does not need a new *meta-language* for defining a DSL; instead, Alchemy uses the reflection capabilities of Scala to treat the plain Scala code of the DSL library as the language specification.

A compiler expert can customize the behavior of the predefined set of annotations based on the features provided by a particular optimizing compiler. Furthermore, the compiler expert can extend the set of annotations with additional ones for encoding various domain knowledge in an optimizing compiler.

This chapter is organized as follows. In Section 8.2 we review the background material and related work. Then, in Section 8.3 we give a high-level overview of the Alchemy framework. In Section 8.4 we present the process of generating a DSL compiler in more detail. Section 8.5 presents several use cases built with the Alchemy framework. Finally, Section 8.6 concludes.

8.2 Background & Related Work

In this section, we present the background and related work to better understand the design decisions behind Alchemy.

8.2.1 Compiler-Compiler

A compiler-compiler (or a meta compiler) is a program that generates a compiler from the specification of a programming language. This specification is usually expressed in a declarative language, called a *meta-language*.

Yacc [171] is a compiler-compiler for generating parsers specified using a declarative language. There are numerous systems for defining new languages, referred to as language workbenches [113, 103], such as Stratego/Spoofax [186], SugarJ [101], Sugar* [102], KHEPERA [105], and MPS [168].

8.2.2 Domain-Specific Languages

DSLs are programming languages tailored for a specific domain. There are many successful examples of systems using DSLs in various domains such as SQL in database management, Spiral [277] for generating digital signal processing kernels, and Halide [278] for image processing. The software development process can also be improved by using DSLs, referred to as language-oriented programming [361]. Cadelion [224] is a language workbench developed for language-oriented programming.

There are two kinds of DSLs: 1) external DSLs which have a stand-alone compiler, and 2) embedded DSLs [158] (EDSLs) which are embedded in another generic-purpose programming language, called a *host language*.

Various EDSLs have been successfully implemented in different host languages, such as Haskell [21, 253, 158] or Scala [288, 215, 261, 295]. The main advantage of EDSLs is reusing the existing infrastructure of the host language, such as the parser, the type checker, and the IDEs among others.

There are two ways to define an EDSL. The first approach is by defining it as a plain library in the host language, referred to as *shallowly* embedding it in the host language. A shallow EDSL is reusing both the frontend and backend components of the host language compiler. However, the opportunities for domain-specific optimizations are left unexploited. In other words, the library-based implementation of the EDSL in the host language is served an *interpreter*.

The second approach is *deeply* embedding the DSL in the host language. A deep EDSL is only using the frontend of the host language, and requires the DSL developer to implement a backend for the EDSL. This way, the DSL developer can leverage domain-specific opportunities for optimizations and can leverage different target backends through code generation.

8.2.3 Extensible Optimizing Compilers

There are many extensible optimizing compilers which provide facilities for defining optimizations and code generation for new languages. Such frameworks can significantly simplify the development of the backend component of the compiler for a new programming language.

Stratego/Spoofax [186] uses strategy-based term-rewrite systems for defining domain-specific optimizations for DSLs. Stratego uses an approach similar to quasi-quotation [353] to hide the expression terms from the user. For the same purpose, Alchemy uses annotations for specifying a subset of optimizations specified by the compiler expert. One can use quasi-quotes [297, 268] for implementing domain-specific optimizations in concrete syntax (rather than abstract syntax) similar to Stratego.

8.2.4 What is Alchemy?

Alchemy is a compiler-compiler, designed for EDSLs that use Scala as their host language. Alchemy uses the Scala language itself as its meta-language; it takes an annotated library as the implementation of a shallow EDSL and produces the required boilerplate code for defining a backend for this EDSL using a particular extensible optimizing compiler. In other words, Alchemy converts an interpreter for a language (a shallow EDSL) to a compiler (a deep EDSL).

Truffle [159] provides a DSL for defining self-optimizing AST interpreters, using the Graal [369] optimizing compiler as the backend. This system mainly focuses on providing just-in-time compilation for dynamically typed languages such as JavaScript and R, by annotating AST nodes. In contrast, Alchemy uses annotation on the library itself and generates the AST nodes based on strategy defined by the compiler expert.

Forge [322] is an embedded DSL in Scala for specifying other DSLs. Forge is used by the Delite [215] and LMS [288, 290] compilation frameworks. This approach requires DSL developers to learn a new specification language before implementing DSLs. In contrast, Alchemy developers write a DSL specification using *plain* Scala code. Then, domain-specific knowledge is encoded using simple Alchemy annotations.

Yin-Yang [177] uses Scala macros for automatically converting shallow EDSLs to the corresponding deep EDSLs. Thus, it completely removes the need for providing the definition of a deep DSL library from the DSL developer. However, contrary to our work, the compiler-compilation of Yin-Yang is specific to the LMS [288] compilation framework. Also, Yin-Yang does not generate any code related to optimizations of the DSL library. We have identified the task of automatically generating the optimizations to be not only a crucial requirement for DSL developers but also one that is significantly more complicated than the one handled by Yin-Yang.

8.3 Overview

Figure 8.1 shows the overall design of the Alchemy framework. Alchemy is implemented as a compiler plugin for the Scala programming language.[1] After parsing and type checking the library-based implementation of an EDSL, Alchemy uses the type-checked Scala AST to generate an appropriate DSL compiler. The generated DSL compiler follows the API provided by an extensible optimizing compiler to implement transformations and code generation needed for that DSL.

There are two different types of users for Alchemy. The first type is a DSL developer, who is the end-user of the Alchemy framework for defining a new DSL together with a set of domain-specific optimizations specified by a set of annotations. A DSL developer is a domain expert,

[1] We decided to implement Alchemy as a compiler plugin rather than using the macro system of Scala, due to the restrictions imposed by def macros and macro annotations.

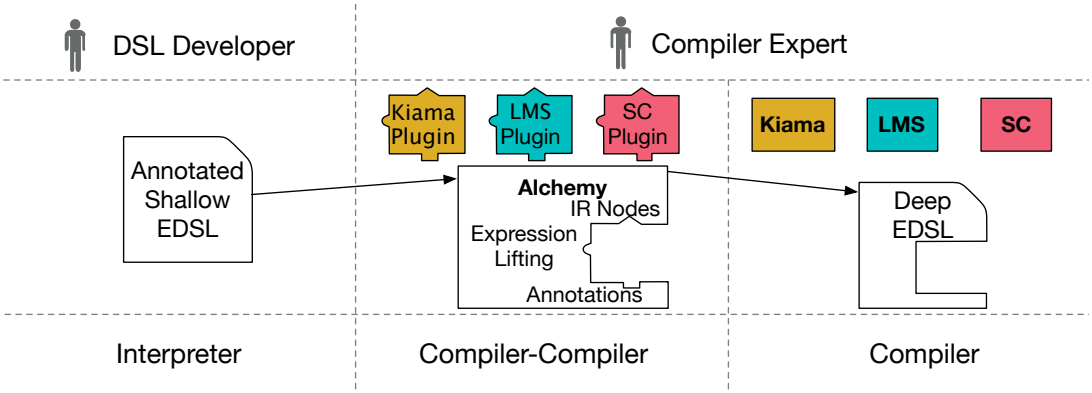


Figure 8.1 – Overall design of Alchemy.

without too much expertise in compilers.

The second type of users is a compiler expert, who is not necessarily knowledgeable in various domains; instead, she is an expert in building optimizing compilers. In particular, a compiler expert has detailed knowledge about the internals of a specific extensible optimizing compiler. Thus, she can use the API provided by the Alchemy framework to specify how the definition of an annotated Scala library is converted into the boilerplate code required for a DSL compiler built on top of an extensible optimizing compiler. Furthermore, she can extend the set of existing annotations provided by Alchemy, for encoding the domain knowledge to be used by an optimizing compiler.

8.4 Compiler-Compilation

In this section, we give more details on the process of generating a DSL compiler. First, we present the annotations defined by the Alchemy framework. Then, we show the process of gathering the DSL information from an annotated library. Afterwards, through an example we give more details on the process of generating a DSL compiler based on the gathered DSL information. Then, we show how Alchemy uses the implementation body of the annotated library for building DSL compilers. Finally, we show the process of generating EDSL compilers using a well-known embedding technique through our running example.

8.4.1 Alchemy Annotations

Deep Types. The DSL developers use the `@deep` annotation for specifying the types for which they are interested in generating a corresponding deep embedding. In other words, this annotation should be used for the types that are actually participating in the definition of a DSL, rather than helper classes which are used for debugging, profiling, and logging purposes.

```
case class ShallowDSL(types: List[ShallowType])
case class ShallowType(tpe: Type,
  methods: List[ShallowMethod]) {
  def annotations: List[Annotation]
  def reflectType: Option[Type]
}
case class ShallowMethod(sym: MethodSymbol,
  body: Option[Tree]) {
  def annotations: List[Annotation]
  def paramAnnots: List[(Int, Annotation)]
}

trait AlchemyCompiler {
  type DSLContext
  def liftType(t: Type)(implicit ctx: DSLContext): Type
  def liftExp(e: Tree)(implicit ctx: DSLContext): Tree
  def compileDSL(dsl: ShallowDSL)
      (implicit ctx: DSLContext): Tree
  def compileType(t: ShallowType)
      (implicit ctx: DSLContext): Tree
  def compileMethod(m: ShallowMethod)
      (implicit ctx: DSLContext): Tree
}
```

Figure 8.2 – The API of Alchemy for compiler experts.

Reflected Types. The `@reflect` annotation is used for annotating the classes the source code of which the DSL developers have no access to. This annotation is used in Alchemy for a) annotating the methods of the Scala core libraries, such as HashMap, ArrayBuffer, etc. which are frequently used, as well as for b) providing alternative implementations for the DSL library and the Scala core library.

User-Defined Annotations. Alchemy allows compiler experts to define their custom annotations, together with the behavior of the target DSL compiler for the annotated method. A compiler expert extends the API exposed by Alchemy to implement the desired behavior (cf. Figure 8.2).

8.4.2 Gathering DSL Information

The Alchemy framework inspects the Scala AST of the given annotated library after the type checking phase of the Scala compiler. Based on the typed Scala AST, Alchemy produces the information about the shallow version of the EDSL by building `ShallowMethod`, `ShallowType`, and `ShallowDSL` objects, corresponding to the DSL methods, DSL types, and the whole DSL, respectively.

A `ShallowMethod` instance has the symbol of the DSL method (the `sym` parameter) and the AST of its body, if available. Also, this instance returns the list of annotations that the DSL developer has used for the method (`annotations`) and its parameters (`paramAnnots`).

A `ShallowType` instance contains the information of the DSL type (the `tpe` parameter) and the list of its methods. In addition, this instance has the list of annotations used for the type

```
@deep
class Complex(val re: Double, val im: Double)
object Complex {
  def add(c1: Complex, c2: Complex): Complex =
    new Complex(c1.re + c2.re, c1.im + c2.im)
  def sub(c1: Complex, c2: Complex): Complex =
    new Complex(c1.re - c2.re, c1.im - c2.im)
  def zero(): Complex =
    new Complex(0, 0)
}
```

Figure 8.3 – The annotated complex DSL implementation.

```
// Predefined by a compiler expert
trait Exp
case class DoubleConstant(v: Double) extends Exp
// Automatically generated by Alchemy
case class ComplexNew(re: Exp, im: Exp) extends Exp
case class ComplexAdd(c1: Exp, c2: Exp) extends Exp
case class ComplexSub(c1: Exp, c2: Exp) extends Exp
case class ComplexZero() extends Exp
```

Figure 8.4 – The generated IR nodes for the Complex DSL.

(`annotations`) and the type it reflects (`reflectType`) in the case where it is annotated with `@reflect`.

Finally, a `ShallowDSL` instance has the information of all DSL types that are annotated with `@deep`. Next, we show how this information is used to build a compiler for a simple DSL.

8.4.3 Generating an EDSL Compiler

Let us consider a DSL for working with complex numbers as our running example. For this DSL, we generate a DSL compiler using a simple form of expression terms as the intermediate representation, which is used by compilation frameworks such as Kiama [307].

Figure 8.3 shows the implementation of this EDSL as an annotated Scala library. This implementation can be used as a normal Scala library to benefit from all the tool-chains provided for Scala such as debugging tools and IDEs.

Figure 8.4 shows the definition of IR nodes generated by Alchemy. The IR nodes are algebraic data types (ADTs), each one specifying a different construct of the Complex DSL. For each method of the `Complex` companion object, Alchemy generates a case class with a default naming scheme in which the name of the object is followed by the name of the method. For example, the method `add` of the `Complex` object is converted to the `ComplexAdd` case class. As another example, the constructor of the `Complex` class is converted to the `ComplexNew` case class. Each case class has the same number of arguments as the corresponding shallow method.

The methods of a class are converted in a similar manner. The key difference is that the generated case class has an additional argument corresponding to `this` object. For example, the method `+` of the `Complex` class is converted to a case class with two parameters, where

```
@deep
class Complex(val re: Double, val im: Double) {
  @name("ComplexAdd")
  def +(c2: Complex): Complex =
    new Complex(this.re + c2.re, this.im + c2.im)
  @name("ComplexSub")
  def -(c2: Complex): Complex =
    new Complex(this.re - c2.re, this.im - c2.im)
}
object Complex {
  def zero(): Complex =
    new Complex(0, 0)
}
```

Figure 8.5 – The second version of the annotated Complex DSL implementation.

the first parameter corresponds to `this` object of the `Complex` class, and the second parameter corresponds to the input parameter of the `+` method.

As explained before, Alchemy allows a compiler expert to define user-defined annotations. In Figure 8.5, the `@name` annotation is used for overriding the default naming scheme provided by Alchemy. For example, the `+` method is converted to the `ComplexAdd` case class.

8.4.4 Lifting the Implementation

As Figure 8.2 shows, Alchemy also provides two methods for lifting the expression and the type of the implementation body of DSL library methods. These two methods are useful for defining syntactic sugar constructs for a DSL (i.e., the DSL constructs that do not have an actual node in the compiler, instead they are defined in terms of other constructs of the DSL). An example of such a construct can be found in Section 8.4.5.

In addition, by providing several reflected versions (cf. Section 8.4.1) for a particular type, each one with a different implementation, Alchemy can generate several transformations for those DSL constructs. This removes the need to implement a DSL IR transformer, which manipulates the IR defined in the underlying optimizing compiler.

To specify the way that expressions should be transformed, compiler experts can implement a Scala AST to Scala AST transformation (cf. the `liftExp` method in Figure 8.2). Note that implementing Scala AST to Scala AST transformations from scratch can be a tedious and time-consuming task. Alternatively, if the target optimizing compiler uses the tagless final [50] or polymorphic embedding [153] approaches, one can use frameworks such as Yin-Yang [177], which are already providing the translation required for these approaches. Next, we show a DSL compiler generated based on the polymorphic embedding approach.

8.4.5 Generating a Polymorphic EDSL Compiler

Let us consider the third version of the Complex DSL, shown in Figure 8.7. This version has an additional construct for negating a complex number, specified by the `unary_-` method.

```
// Shallow expression
new Complex(2, 3) - Complex.zero()
// Lifted expression
ComplexSub(
  ComplexNew(
    DoubleConstant(2), DoubleConstant(3)
  ), ComplexZero()
)
```

Figure 8.6 – An example expression and its lifted version in Complex DSL.

```
@deep
class Complex(val re: Double, val im: Double) {
  @name("ComplexAdd")
  def +(c2: Complex): Complex =
    new Complex(this.re + c2.re, this.im + c2.im)
  @name("ComplexNeg")
  def unary_-(): Complex =
    new Complex(-this.re, -this.im)
  @sugar
  def -(c2: Complex): Complex =
    this + (-c2)
}
object Complex {
  def zero(): Complex =
    new Complex(0, 0)
}
```

Figure 8.7 – The third version of the annotated Complex DSL implementation.

Subtracting two complex numbers is a syntactic sugar (annotated with `@sugar`) for adding the first complex number with the negation of the second complex number.

Polymorphic embedding [153] (or tagless final [50]), is an approach for implementing EDSLs where every DSL construct is converted into a function (rather than an ADT) and the interpretation of these functions are left abstract. Thus, it is possible to provide such abstract interpretations with different instances, such as actual evaluation, compilation, and partial evaluation [153, 50].

Figure 8.8 shows the polymorphic embedding interface generated by Alchemy for the third version of the Complex DSL. The type member `Rep[T]` is an abstract type representation for different interpretations of Complex DSL programs.

Figure 8.9 shows the generated deep embedding interface for the polymorphic embedding of the Complex DSL. Instead of using ADTs for defining IR nodes, this time we use generalized algebraic data types (GADTs). The invocation of each DSL construct method results in the creation of the corresponding node. As the subtraction of two complex numbers is a syntactic sugar, no corresponding IR node is created for it. Instead, the `complexSub` method results in the invocation of the `complexAdd` and `complexNeg` methods, which is generated using the `liftExp` method of Alchemy.

Figure 8.10 shows the lifted expression of the example of Figure 8.6. In this case, instead of converting expressions to their ADT definition, Alchemy converts them to their corresponding

```
trait ComplexOps {
  type Rep[T]
  def doubleConst(d: Double): Rep[Double]
  def complexAdd(self: Rep[Complex],
      c2: Rep[Complex]): Rep[Complex]
  def complexNeg(self: Rep[Complex]): Rep[Complex]
  def complexSub(self: Rep[Complex],
      c2: Rep[Complex]): Rep[Complex]
  def complexZero(): Rep[Complex]
  def complexNew(re: Rep[Double],
      im: Rep[Double]): Rep[Complex]
}
```

Figure 8.8 – The generated polymorphic embedding interface for the Complex DSL.

DSL method definition in polymorphic embedding. In addition, this figure shows the generated IR nodes for this program, in which the subtraction construct is desugared into the addition and negation nodes. Note that the negation of zero and addition with zero can be further simplified by providing yet another optimized interface implementation in polymorphic embedding. Examples of such simplifications are given later in Section 8.5.4.

Up to now, we have used simple expression terms for the definition of IR nodes. Alchemy can easily generate other types of IR nodes such as A-Normal Form [110], where the children of a node are either constant values or variable accesses. This means that all non-trivial sub-expressions are let-bound, which helps in applying optimizations such as common-subexpression elimination (CSE) and dead-code elimination (CSE). Such normalized types of IR nodes are used in various optimizing compilers such as Graal [369], LMS [288], Squid [268], and SC. We will see more detailed examples of SC in the next section.

8.5 SC (The Systems Compiler)

When specializing data analytics systems code, an optimizing compiler effectively needs to specialize *high-level* systems code which will naturally employ a hierarchy of components and libraries from relatively high to very low level of abstraction. To scale to such complex code bases, an optimizing compiler must guarantee two properties, not offered by existing compiler frameworks for generative programming.

First, existing optimizing compilers expose a large number of low-level compiler internals such as nodes of an intermediate representation (IR), dependency information encoded in IR nodes, and code generation templates to their users. This necessary interaction with low-level semantics when coding optimizations, but more importantly the introduction of the IR as an additional level of abstraction, significantly increases the difficulty of debugging, as developers cannot easily track the relationship between the source code, its optimizated form – expressed using IR constructs – and the final, generated code [177, 322].

Second, to achieve maximum efficiency, developers must have tight control over the compiler's phases – admitting custom optimization phases and phase orderings. This is necessary as code

```
// Predefined by a compiler expert
trait Exp[T]
case class DoubleConstant(d: Double) extends Exp[Double]
// Automatically generated by Alchemy
case class ComplexNew(re: Exp[Double],
                im: Exp[Double]) extends Exp[Complex]
case class ComplexAdd(self: Exp[Complex],
                c2: Exp[Complex]) extends Exp[Complex]
case class ComplexNeg(self:Exp[Complex])extends Exp[Complex]
case class ComplexZero() extends Exp[Complex]

trait ComplexExp extends ComplexOps {
 type Rep[T] = Exp[T]
 def doubleConst(d: Double): Rep[Double] =
  DoubleConstant(d)
 def complexAdd(self: Rep[Complex],
     c2: Rep[Complex]): Rep[Complex] =
  ComplexAdd(self, c2)
 def complexNeg(self: Rep[Complex]): Rep[Complex] =
  ComplexNeg(self)
 def complexSub(self: Rep[Complex],
     c2: Rep[Complex]): Rep[Complex] =
  complexAdd(self, complexNeg(c2))
 def complexZero(): Rep[Complex] =
  ComplexZero()
 def complexNew(re: Rep[Double],
     im: Rep[Double]): Rep[Complex] =
  ComplexNew(re, im)
}
```

Figure 8.9 – The generated IR node definitions and deep embedding interface for the Complex DSL.

transformers with different optimization objectives may have to be combined in arbitrary orderings, depending on architectural, data, or query characteristics. However, existing generative programming frameworks do not offer much control over the compilation process. This absence of control effectively forces developers to provision for *all* possible optimization orderings. This pollutes the code base of individual optimizations, making some of them dependent on other, possibly semantically independent, optimizations. In general, the code complexity grows exponentially with the number of supported transformations.[2]

In this section, we present a new compilation framework, called the Systems Compiler (SC). SC's native language is Scala, and the system as well as the compiler extensions are all written in *plain* Scala,[3] admitting its advanced software composition features to allow for well-designed systems and compiler extensions. SC gives full control over the compilation process including the stack of compiler phases without exporting compiler internals such as intermediate representations, thanks to Alchemy.

[2] As an example, consider the case of a compiler that is to support only two optimizations: 1) data-layout optimizations (i.e. converting a row layout to a column or PAX-like layout [12]) and 2) data-structure specialization (i.e. adapting the definition of a data structure to the particular context in which it is used). This means that if the second optimization handles three different types of specialization, one has to provision for $2 \times 3 = 6$ cases to handle all possible combinations of these optimizations.

[3] The principles described in this section can be applied to a wider range of host-languages and are not limited to Scala.

```
// lifted expression in polymorphic embedding
complexSub(
  complexNew(
    doubleConst(2), doubleConst(3)
  ), complexZero()
)
// generated IR nodes
ComplexAdd(
  ComplexNew(
    DoubleConstant(2), DoubleConstant(3)
  ), ComplexNeg(
    ComplexZero()
  )
)
```

Figure 8.10 – Polymorphic embedding version of the example in Figure 8.6, and the generated IR nodes.

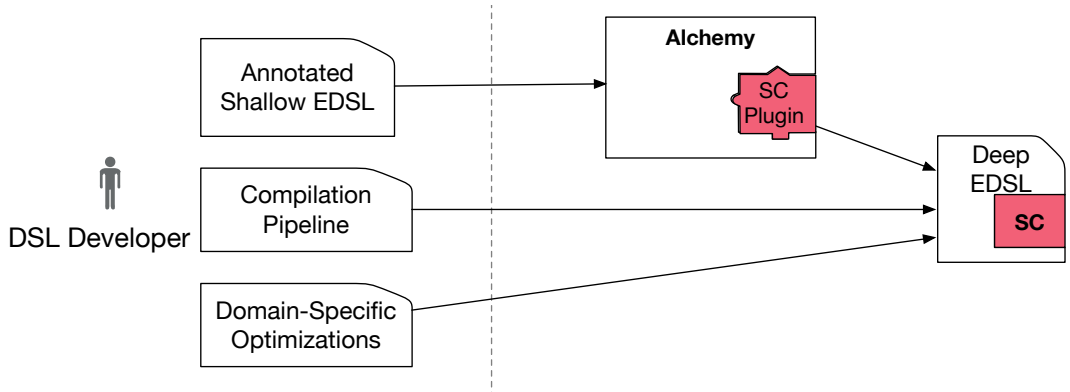


Figure 8.11 – Overall design of SC used with Alchemy.

8.5.1 Overall Design

SC (the Systems Compiler) is a compilation framework for building compilation-based systems in the Scala programming language. Different system component libraries can be considered as different DSLs, for which system developers extend SC to build DSL compilers. To hide the internal implementation details of the compiler, Alchemy provides an abstraction layer between the system component libraries and the SC optimizing compiler itself. Figure 8.11 shows the overall design of Alchemy and SC, which operates as follows.

The system developer (who is actually a DSL developer) uses the SC plugin of Alchemy to create a DSL compiler. Many systems optimizations are automatically converted by Alchemy to functions that manipulate the IR of the compiler. The system developer uses a set of *annotations* provided by the compiler expert of the SC framework, to specify the IR transformations. To provide more advanced domain-specific optimizations that cannot be encoded by annotations, as well as compilation phases, the system developer uses the transformation API provided by SC.

SC converts the systems code to a graph-like intermediate representation (IR). As SC fol-

lows the polymorphic embedding approach [153] for deeply embedding DSLs, SC uses Yin-Yang [177][4] which applies several transformations (e.g., language virtualization [51]) in order to convert the plain Scala code into the corresponding IR.

We have used SC to build two different compilation-based query engines: a) an analytical query processing engine [300, 301], and b) a transactional query processing engine [83].

From the perspective of the abstraction level of a program, the transformations are classified into two categories. First, *optimizing* transformations transform a program into another program on the *same* level of abstraction. Second, *lowering* transformations convert a program into one on a *lower* abstraction level. SC provides a set of built-in transformations out-of-the-box. These mainly consist of generic compiler optimizations such as common-subexpression elimination (CSE), dead-code elimination (DCE), partial evaluation (PE), etc.

The last phase in the SC compiler is code generation where the compiler generates the code based on the desired target language. Observe that since each lowering transformation brings the program closer to the final target code, this provides the excellent property that code generation (e.g., C code generation) in the end basically becomes a trivial and naïve stringification of the lowest level representation.

For converting from host to target languages, SC can make use of the same infrastructure. To do this conversion, a DSL developer only has to express the constructs and the necessary data-structure API of the target language as a library inside the host language. Then, there is no need for the DSL developer to *manually* provide code generation for the target language using internal compilers APIs as is the case with most existing solutions. In contrast, Alchemy automatically generates the transformation phases needed to convert from host language IR nodes to target language IR nodes (e.g., from Scala to C).

An important side-effect of our design is that since the plain Scala code of a system does not require *any* specific syntax, type or IR-related information from SC, this code is *directly* executable using the normal Scala compiler. In this case, the Scala compiler will ignore all Alchemy annotations, and interpret the code of the system using plain Scala. Alchemy can thus be seen as a system for converting a *system interpreter* (which executes the systems code unoptimized) into the corresponding *system compiler* along with its *optimizations*.

Next, we briefly provide more details about the two categories of transformations that SC supports.

8.5.2 SC Transformations

SC classifies the transformations into two categories, which we present in more detail next while also highlighting differences from previous work in each class.

[4]We note that Yin-Yang, in contrast to our work, handles only the conversion from plain Scala code to IR, without providing any functionality related to code optimization of the systems library.

```
class MyTransformer extends RuleBasedTransformer {
  analysis += rule { case Pattern =>
    // gather necessary information for analysis
  }
  rewrite += rule { case Pattern =>
    /* use analysis information while generating
       the appropriate transformed node */
  }
  rewrite += remove { case Pattern =>
    /* use analysis information to remove node */
  }
}
```

Figure 8.12 – Offline Transformation API of SC.

Online transformations are applied while the IR nodes are generated. Every construct of a DSL is mapped to a method invocation, which in turn results in the generation of an IR node [50, 153]. By overriding the behavior of that method, an online transformation can result in the generation of a different (set of) IR node(s) than the original IR node. Even though a large set of optimizations (such as constant folding, common subexpression elimination, and others) can be expressed using online transformations, some optimizations need to be preceded by analysis over the whole program.

For a restricted set of control-flow constructs, namely *structured loops*, it is possible to use the Speculative Rewriting [218] approach in order to combine the data-flow analysis with an online transformation, thus bypassing the need for a separate analysis pass. However, we have observed that there exists an important class of transformations in which the corresponding analysis *cannot* be combined with the transformation phase. This class of optimizations, which cannot be handled by existing extensible optimizing compilers, is presented next.

Offline transformations need whole program analysis before applying any transformation. Figure 8.12 shows the SC offline transformation API. The `analysis` construct specifies the information that should be collected during the analysis phase of a transformation. The `rewrite` construct specifies the transformation rules based on the information gathered during the analysis phase. Finally, the `remove` construct removes the pattern specified in its body.

The Alchemy annotation processor takes care of converting the Scala annotations of the systems library, which express optimizations, into IR transformers which manipulate the intermediate representation of SC. This is explained in more detail in Section 8.5.4.

8.5.3 SC Annotations

In this section, we present in more detail the different categories of annotations implemented for SC.

Side-Effects. These are annotations that guide the effect system of the optimizing compiler.

```
@deep
@inline
abstract class Operator[A] {
  abstract def init(): Unit
}

@deep
@inline
class ScanOp[A](table: Array[A]) extends Operator[A]{
  var i = 0
  @inline
  def init() = {
    while (i < table.length) {
      child.consume(table(i))
      i += 1
    }
  }
}

@deep
@inline
class HashJoinOp[A,B,C](val leftParent: Operator[A],
  val rightParent: Operator[B], ...) extends
    Operator[CompositeRecord[A, B]] {
  @inline var mode = 0
  @inline
  def init() = {
    mode = 0
    // phase 1: leftParent will call this.consume
    leftParent.init()
    mode = 1
    // phase 2: rightParent will call this.consume
    rightParent.init()
    mode = 2
  }
  @inline
  def consume(tuple: Record): Unit = {
    if (mode == 0) {
      /*phase1 -- elided code for left side of join*/
    } else if (mode == 1) {
      /*phase2 -- elided code for right side of join*/
    }
  }
}
```

Figure 8.13 – Inline annotations of two operators in our analytical query engine.

For example, a method annotated with `@pure` denotes that this method does not cause any side effects and the expressions that call this method can be moved freely throughout the program. In addition, Alchemy provides more fine-grained effect annotations that keep track of read and write mutations of objects. More precisely, if a method is annotated with `read` or `write` annotations, then there exists a mutation effect over the specific object (i.e., `this`) of that particular class. Similarly, an annotated argument may include read or write effects over that argument.

Inline. The `@inline` annotation guides the inlining decisions of the compiler. This annotation can be applied to methods, whole classes as well as class fields, with different semantics in each case. Methods annotated with the `@inline` annotation specify that every invocation of that method should be inlined by the compiler. For classes, the `@inline` annotation removes

the abstraction of the specific class during compilation time. In essence, this means that the methods of an inlined class are implicitly annotated with the inline annotation and are subsequently inlined. This makes inlined classes in Alchemy semantically similar to value classes [291]. Finally, a mutable field of a class can also be annotated with `@inline`, which means that all the usages of this field are partially evaluated during compilation time.

Figure 8.13 shows the scan and hash-join operators annotated with the `@inline` annotation. In this example, all methods of the `HashJoinOp` class are automatically inlined, as the `HashJoinOp` class is marked with the `@inline` annotation. Furthermore, the mutable field `mode` is partially evaluated at compilation time and, as a result, the corresponding branch in the `consume` method is also partially evaluated at compilation time. More concretely, both `leftParent.init` and `rightParent.init` invoke the `consume` method of the `HashJoinOp` class. However, the former inlines the code in the `phase1` block whereas the latter inlines the `phase2` code block. This is possible as `mode` is evaluated during compilation time and, thus, there is no need to generate any code for it and the corresponding `if` condition checks. We have found that there are multiple examples where such `if` conditions can be safely removed in our analytical query engine (e.g., in the case of configuration variables whose values are known in advance at startup time).

Algebraic Structure. These are annotations for specifying the common algebraic rules that occur frequently for different use cases. For example, `@monoid` specifies a binary operation of a type that has a monoid structure. In the case of natural numbers, `@monoid(0)` over the `+` operator represents that `a+0=0+a=a`. The annotation processor generates several constant folding optimizations which benefit from such algebraic structure and significantly improve the performance of systems that use them.

Furthermore, the `@commutative` annotation specifies that the order of the operands of a binary operation can be changed without affecting the result. This property is useful for applying constant folding on cases in which static arguments and dynamic arguments are mixed in an arbitrary order, thus hindering the constant folding process. For example, in the expression `1 + a + 2`, constant folding cannot be performed without specifying that the commutativity property of addition on natural numbers is applicable in this case. However, if we push the static terms to the left side of the expression while we generate the nodes, we generate the IR which represents the expression `1 + 2 + a` instead of the previous expression. Then, it becomes possible to apply constant folding and get the expression `3 + a`.

8.5.4 Generating Transformation Passes

As discussed in Section 8.5.2, these transformation passes are classified into two categories: *online* and *offline* transformations. In this section, we demonstrate how Alchemy generates online and offline transformation passes.

```
@deep
@reflect[Int]
class AnnotatedInt {
  @commutative
  @monoid(0)
  @pure
  def +(x: Int): Int
}
```

Figure 8.14 – Alchemy annotations of the `Int` class. The `AnnotatedInt` class is a mirror class for the original `Int` Scala class.

Generating Online Transformations. In general, Alchemy uses node generation (online transformation) in order to implement the appropriate rewrite rules for most annotations. As we discussed in Section 8.4.5, every construct of a DSL is mapped to a method invocation, which in turn results in the generation of an IR node [50, 153].

For example, in the case of addition on natural numbers, the default behavior for the method `int_plus` is shown in lines 1-3 of Figure 8.15. This method generates the `IntPlus` IR node, which is also automatically generated by Alchemy. However, when this method is annotated with the `@monoid` and `@commutative` annotations, this results in the generation of an online transformation. More specifically, the annotated method automatically generates the code shown in lines 5-14 of the same figure. First, as the method is pure, SC checks if both arguments are statically known. This is achieved by checking if the expressions are of `Constant` type or not. In this case, SC performs partial evaluation by computing the result through the addition of the arguments. Second, if only one of the arguments is statically known and it is equal to 0, the monoid property of this operator returns the dynamic operand. Third, if only one of the arguments is statically known (but it is not zero), then the static argument is pushed as the left operand, as we know that this operator is commutative. Finally, if none of the previous cases is true, then the default behavior is used and the original `IntPlus` IR node is generated.

Alchemy also generates an online transformation out of the `@inline` annotation. For methods with this annotation, instead of generating the corresponding node, Alchemy generates the nodes for the body of that method. In the special case of dynamic dispatch, the concrete type of the object is looked up and based on its value Alchemy invokes the appropriate method.

For example, the annotated code for the scanning operator of the analytical query engine, shown in Figure 8.13, generates the compiler code shown in Figure 8.16. There the `scanOpInit` method represents the corresponding method which is invoked in order to generate an appropriate IR. As is the case with integer addition, the default behavior of this method, which results in creating the `ScanOpInit` IR node, is shown in lines 1-3. The rest of the code presents the implementation of the `@inline` annotation for this operator, which results in inlining the body of this method while generating the IR node. The method `scanOpInit` is automatically generated by Alchemy which generates the body of the `init` method. As described earlier, all method invocations lead to the generation of the corresponding IR nodes. For example, `__whileDo` results in creating an IR node for a `while` loop. Finally, for inlining the `init` method of

```
trait Base {
  type Rep[T]
}

trait BaseExp extends Base {
  type Rep[T] = Exp[T]
}

trait IntOps extends Base {
  def int_plus(a: Rep[Int], b: Rep[Int]): Rep[Int]
}

trait IntExp extends IntOps with BaseExp {
  // default IR generation
  def int_plus(a: Exp[Int], b: Exp[Int]): Exp[Int] =
    IntPlus(a, b)
}

trait IntExpOpt extends IntExp {
  // optimized IR generation
  override
  def int_plus(a: Exp[Int], b: Exp[Int]): Exp[Int] =
  (a, b) match {
    case (Constant(aStatic), Constant(bStatic)) =>
      Constant(aStatic + bStatic)
    case (Constant(0), bDynamic) => bDynamic
    case (aDynamic, Constant(0)) => aDynamic
    case (aDynamic,Constant(bStatic)) => int_plus(b,a)
    case (_, _) => super.int_plus(a,b)
  }
}
```

Figure 8.15 – The generated online transformation by Alchemy for addition on `Int`.

the `Operator` class, we need to handle dynamic dispatch, as we described earlier. We do so by redirecting to the appropriate method based on the type of the caller object. An alternative design is to use multi-stage programming for encoding the fact that the objects of Operator class are staged away. This is achieved by generating the deep embedding interface of all operator classes as partially static. With a similar design, one can support staging for other libraries implemented using design patterns that require abstraction overheads such as generic programming [210, 372].

Generating Offline Transformations. The generated transformations are not limited to online transformations. Alchemy also generates offline transformation passes. Figure 8.17 shows the implementation of three different transformations for the `Seq` class[5], in plain Scala code. The first implementation uses a linked list for storing the elements of the sequence. The second implementation stores the elements in an array data-structure.[6] Finally, the third implementation uses a `g_list` data-structure, provided by GLib. The generated transformation from this class can be used for using data structures provided by GLib in the generated C code.

These implementations can be used for debugging the correctness of the transformers. For

[5] By a `Seq` data type, we mean a collection where the order of its elements does not matter.

[6] This implementation assumes that the number of the elements in the collection does not exceed `MAX_BUCKETS`. In cases where this assumption does not hold, one has to make the corresponding field mutable, and add an additional check while inserting an element.

```
trait OperatorOps extends Base {
  def operatorInit[A:Type](self:Rep[Operator[A]]):Rep[Unit]
}

trait ScanOpOps extends OperatorOps {
  def scanOpInit[A:Type](self: Rep[ScanOp[A]]): Rep[Unit]
}

trait OperatorExp extends OperatorOps with BaseExp {
  def operatorInit[A:Type](self: Exp[Operator[A]]) =
    OperatorInit(self)
}

trait ScanOpExp extends ScanOpOps with OperatorExp {
  // the default behavior of scanOp.init operation
  def scanOpInit[A:Type](self: Exp[ScanOp[A]]) =
    ScanOpInit(self)
}

trait ScanOpInline extends ScanOpExp {
  // the inlined behavior of scanOp.init operation
  override
  def scanOpInit[A:Type](self: Exp[ScanOp[A]]) =
    __whileDo(self.i < (self.table.length), {
      self.child.consume(self.table.apply(self.i))
      self.i = self.i + unit(1)
    })
  // handling of dynamic dispatch for operator.init
  override
  def operatorInit[A:Type](self: Exp[Operator[A]]) =
    self.tpe match {
      case ScanOpType(_) =>
        scanOpInit(self.asInstanceOf[Exp[ScanOp[A]]])
      // the rest of the operator types ...
    }
}
```

Figure 8.16 – The generated online transformations by Alchemy for the scan operator of the analytical query engine.

using them in the DSL compiler, Alchemy generates offline transformations based on the SC API (cf. Figure 8.12). Figure 8.18 shows the generated offline transformation for the implementation of the `Seq` data-structure using an array. This transformation lowers the objects of a `Seq` data structure into records with two fields: 1) the underlying array, 2) the current size of the collection. The nodes corresponding to each method of this data structure are then rewritten to the IR nodes of the implementation body provided in the reflected type.

Many offline transformations require inspecting the generated IR nodes to check their applicability. In some of these cases, compiler experts can provide annotations to generate the required analysis passes. However, in many cases, the analysis requires more features than the ones provided by the existing annotations. Implementing such analysis passes can be facilitated by using quasi-quotations [268, 270, 297]. More details about the implementation of quasi-quotations and their usages are beyond the scope of this thesis.

The aforementioned design provides several advantages over previous work. First, the Alchemy annotation processor uses Scala annotations. This means that there is no need to provide spe-

```
@offline
@reflect[Seq[_]]
class SeqLinkedList[T] {
  var head: Cont[T] = null


  def +=(elem: T) =
    head = Cont(elem, head)


  def foreach(f: T => Unit)
    = {
    var current = head
    while (current != null) {
      f(current.elem)
      current = current.next
    }
  }
}
```

```
@offline
@reflect[Seq[_]]
class SeqArray[T: Manifest]
   {
  val array =
    new Array[T](MAX_BUCKETS)
  var size: Int = 0
  def +=(elem: T) = {
    array(size) = elem
    size += 1
  }
  def foreach(f: T=>Unit) =
    for (i <- 0 until size) {
      val elem = array(i)
      f(elem)
    }


}
```

```
@offline
@reflect[Seq[_]]
class SeqGlib[T] {
  var gHead:
    Pointer[GList[T]] = null


  def +=(x: T) =
    gHead =
      g_list_append(gHead,
        &(x))

  def foreach(f: T=>Unit) =
    {
    var current = gHead
    while (current != NULL) {
      f(*(current.data))
      current =
        g_list_next(current)
    }
  }
}
```

Figure 8.17 – Different transformations for the Scala Seq class. The transformations are written using plain Scala code.

cific infrastructure for an external DSL, as opposed to the approach of Stratego/Spoofax [186]. Second, developers can annotate the source code with appropriate annotations, without the need to port it into another DSL, as opposed to the approach taken in Forge [322]. In other words, developers use the signature of classes and methods as the meta-data needed for specifying the DSL constructs, whereas in a system like Forge the DSL developer must use Forge DSL constructs to specify the constructs of the DSL. Third, as we aim to give systems developers the ability to write their systems in plain Scala code, we designed Alchemy so that developers can place the annotations on the systems code itself, whereas an approach like Truffle [159] focuses on self-optimizing AST interpreters. Thus, the latter annotates the AST nodes of the language itself.

8.5.5 Productivity Evaluation

We use Alchemy to automatically generate the compiler interface for a subset of the standard Scala library and two database engines: 1) an analytical query engine [300, 301], and 2) a transactional query engine [83]. Table 8.1 compares the number of LoCs[7] of the library classes with the generated compiler interfaces. We make the following observations.

First, for the Scala standard library classes, the LoCs of the reflected classes are mentioned in the table. These classes provide the method signatures of their original classes and are annotated with appropriate effect and algebraic structure annotations (Section 8.5.3). However, in most cases, developers do not need to provide the implementation of the methods of these classes. As a result, the compiler interfaces of the Scala standard library classes can be gener-

[7] We used CLOC [77] to compare the number of LoCs.

```
class SeqArrayTransformer extends RuleBasedTransformer{
  rewrite += rule { case SeqNew[T]() =>
    val _maxSize = ("maxSize", true, unit(0))
    val _array = ("array", false, __newArray[T](MAX_BUCKETS))
    record[Seq[T]](_maxSize, _array)
  }
  rewrite += rule { case SetPlusEq[T](self, elem) =>
    self.array.update(self.maxSize, elem)
    self.maxSize_=(self.maxSize.+(unit(1)))
  }
  // Provides access to the fields of the
  // generated record for Seq
  implicit class SeqArrayOps[T](self: Rep[Seq[T]]) {
    def maxSize_=(x: Rep[Int]): Rep[Unit] =
      fieldSetter(self, "maxSize", x)
    def maxSize: Rep[Int] =
      fieldGetter[Int](self, "maxSize")
    def array: Rep[Array[T]] =
      field[Array[T]](self, "array")
  }
}
```

Figure 8.18 – The generated offline transformations by Alchemy for `Seq` based on arrays.

ated with only tens of LoCs. The exception is the reflected classes responsible for generating offline transformations (e.g., Seq Transformation and HashMap Transformation), where the developer provides the implementation to which every method should be transformed into.

Furthermore, observe that the `Int` class contains more LoCs than the many other standard library classes. This is because each operation of this class encodes different combinations of arguments in its methods with other numeric classes (e.g., Int with Double, Int with Float, and so on). Furthermore, the generated compiler interface of this class is also longer than expected. This is because the generated compiler code contains the constant-folding optimization (Section 8.5.3), which is encoded by Alchemy annotations. In addition, for the query operators of the analytical query engine, the generated compiler interface encodes all online partial evaluation processes annotated using the `@inline` annotation. This results in the partial evaluation of mutable fields, function inlining, and virtual dispatch removal.

8.6 Conclusions

In this chapter, we have presented Alchemy, a compiler generator for DSLs embedded in Scala. Alchemy automatically generates the boilerplate code necessary for building a DSL compiler using the infrastructure provided by existing extensible optimizing compilers. Furthermore, Alchemy provides an extensible set of annotations for encoding domain-specific knowledge in different optimizing compilers. Finally, we have shown how to extend the Alchemy annotations to generate the boilerplate code required for building two different query compilers on top of an extensible optimizing compiler.

Type	Library	Compiler
Analytical Query Engine		
Query Operators	541	3456
Monadic Interface	156	407
File Manager	254	291
Aux. Classes	100	749
Transactional Query Engine		
In-Memory Storage	45	294
Indexing Data-Structures	69	394
Aux. Classes	58	364
Scala Library		
Boolean	18	255
Int	85	970
Seq	39	334
Seq Trans.	176	329
Array	39	306
ArrayBuffer	52	453
HashMap	32	259
HashMap Trans.	162	305
C GLib	181	729
Other Classes	936	7007
Total	2943	16902

Table 8.1 – The comparison of LoCs of the *(reflected)* classes of the Scala standard library and a preliminary implementation of two query engines together with the corresponding automatically generated compilation interface.

9 Related Work

In this chapter, we discuss the related work for different systems presented in this thesis.

9.1 Compilation Frameworks

In this section we present and compare with related work in four categories: a) Existing frameworks for defining domain-specific languages, b) The usage of annotations in order to guide the optimizations performed by an optimizing compiler, c) The role and execution model of partial evaluation, and, finally, d) The usage of DSLs and examples of language embedding. We briefly discuss these areas below.

Frameworks for defining DSLs. Forge [322] is an embedded DSL in Scala for specifying other DSLs. Forge is used by the Delite [215] and LMS [288, 290] compilation frameworks. This approach requires DSL developers to learn a new specification language before implementing DSLs. In contrast, SC developers write a DSL specification using *plain* Scala code. Then, domain-specific knowledge is encoded using simple annotations provided by Alchemy.

In addition, there are numerous systems [105, 186, 307] for defining external DSLs. Stratego/Spoofax [186] uses strategy-based term-rewrite systems for introducing domain-specific optimizations for DSLs. Stratego uses an approach similar to quasi-quotation [353] to hide the expression terms from the user. For the same purpose, Alchemy uses annotations for specifying domain-specific optimizations.

Yin-Yang [177] uses Scala macros for automatically converting shallow EDSLs to corresponding deep EDSLs. Thus, it completely removes the need of providing the definition of a deep DSL library from the DSL developer. However, contrary to our work, Yin-Yang does not generate any code related to optimizations of the DSL library. We have identified the task of automatically generating the optimizations to be not only a crucial requirement for system programmers but also one that is significantly more complicated than the one handled by Yin-Yang.

Guiding Compiler Optimizations via Annotations. Telescoping languages [191] as well as Broadway [140] both use annotations on libraries in order to expose the DSL semantics to an optimizing compiler. Similarly, Truffle [159] is an embedded DSL used to annotate the AST nodes of a programming language. Similarly to Alchemy, Truffle annotations are converted to optimization code. Truffle focuses mostly on dynamically typed languages and just-in-time compilation based on runtime profile information collected by the Graal VM [369]. In contrast, our work focuses mostly on providing a user-friendly way for expressing domain-specific optimization opportunities for embedded DSLs, where the corresponding IR nodes are automatically generated by Alchemy. Our framework could use the Truffle/Graal framework in order to benefit from runtime profiling information, e.g. for more precise information about the preconditions encoded with Alchemy annotations.

Partial Evaluation and Staging. Multi-stage programming [329] performs partial evaluation by using explicit annotations (i.e. the quotation mechanism for the MetaOCaml [328] and the Terra [89] languages or the Rep type classes of LMS [288]). SC follows a similar approach to LMS and distinguishes between static and dynamic variable binding time using type information, as was first proposed by [79]. Moreover, SC uses a construct similar to exo-types [91] in order to use staging for defining high-performance record types.

Online partial evaluation can be performed by specializing the code while the corresponding IR is reified, as was first suggested in Finally Tagless [50]. With this approach function inlining is possible while reifying the IR by simply performing beta-reduction on lambda expressions followed by a function application. Furthermore, constant propagation and domain-specific optimizations such as algebraic laws (e.g. distributivity) can be both encoded by defining rewrite rules using pattern matching on the IR nodes. LMS [288, 290] and polymorphic embedding [153] both follow this approach to perform the aforementioned optimizations.

SC uses the same method for performing online partial evaluation. However, the pattern matching rewrite rules are not written by DSL developers but are automatically generated by Alchemy. In addition, through the API exposed by SC, DSL developers can decide: a) the set of rewrite rules to be applied as well as b) the order in which these rules are applied during compilation. With this mechanism, DSL developers have full control over the generated rules by properly annotating a library. More importantly, automatically generating the rewrite rules means that DSL developers do not have to work with SC IR for encoding domain-specific optimizations (as is the case with LMS).

Language Embedding and Usage of DSLs. Js.Scala [206] deeply embeds JavaScript in Scala. However, Js.Scala handles only this particular combination of host and object languages, while SC offers a more general language embedding mechanism. We use this mechanism to build C.Scala which embeds the C language in Scala.

DSLs have been successfully used to generate highly optimized applications in the context of

linear algebra and query operators by the Spiral [277] and OCAS [201] systems, respectively. Our idea of considering different system components as different DSLs is similar to [336], which defines languages as libraries in Racket as well as to Language-oriented programming [361].

9.2 Compilation for Query Engines

We outline related work on compilation in query engines in seven areas: (a) Previous compilation approaches, (b) optimizing database applications, (c) compiler optimizations in high-performance computing, (d) language integrated queries in managed runtimes, (e) orthogonal techniques to speed up query processing, (f) a brief summary of work on Domain Specific Compilation in the Programming Languages (PL) community, and, finally, (g) a comparison with a previous realization of the abstraction without regret vision. We briefly discuss these areas below.

Previous Compilation Approaches. Historically, System R [53] first proposed code generation for query optimization. However, query interpretation replaced compilation early (before the first version of System R was released), since the additional effort of generating code for an algorithm rather than to directly implement an algorithm substantially slowed down prototyping in this pioneering project. The Daytona system [130] revisited compilation in the late nineties.

The shift towards pure *in-memory* computation in databases, evident in the space of data analytics and transaction processing has lead developers to revisit compilation. The reason is that, as more and more data is put in memory, query performance is increasingly determined by the effective throughput of the CPU. In this context, compilation strategies aim to remove unnecessary CPU overhead. Examples of industrial systems in the area since the mid-2000s include SAP HANA [106, 123], VoltDB [320, 178] and Oracle's TimesTen [262].

Recent industrial systems that employ query compilation include Microsoft's Hekaton [212], Netezza [378], and MemSQL. In the academic context, interest in query compilation has also been renewed since 2009 and continues to grow [10, 134, 208, 202, 255, 203, 85, 73, 252, 352, 205, 19, 72, 240].

All these works aim to improve database systems by removing unnecessary abstraction overheads. However, none embrace generative programming and source-to-source translation as we do. As a consequence, we are able to, thoroughly and in a novel way, separate a high-level system implementation from code transformers that are responsible for generating high-performance code, and to automatically obtain a query compiler from an implementation of a query interpreter. This solves the key problem of low productivity that caused the System R team to abandon query compilation.

The compilation approach confirmed for System R and Hekaton is to achieve code generation by *template expansion*. This refers to expanding every operator of a query plan (optimized by a classical query optimizer) by a low-level code template, the composition of which yields a low-level program for a query. It is hard to create template expanders that implement sophisticated code optimizations, as multiple transformations that conceptually could be implemented separately and applied in sequence have to be composed manually in all possible ways, causing a code explosion [301]. Our approach solves this problem. The strong points of template expanders are conceptual simplicity and that code generation is very fast.

Rao et al. propose to remove the overhead of virtual functions in the Volcano iterator model by using a compiled execution engine built on top of the Java Virtual Machine (JVM) [283]. The HIQUE system takes a step further and completely eliminates the Volcano iterator model in the generated code [208]. It does so by translating the algebraic representation to C++ code using templates. In addition, Zane et al. have shown how compilation can also be used to additionally improve operator internals [378].

DBToaster [10, 202, 205] is one of the first academic systems to employ query compilation and compilation to LLVM. Its goal is to perform incremental view maintenance, and thus it is an instance of a system that may safely assume that queries are known in advance and code can be specialized for them (in non TPC-H compliant ways). DBToaster uses a functional intermediate language in which optimizations such as loop fusion are performed, avoiding the disadvantages of template expansion.

The query compiler of the HyPer database system also uses query compilation [255]. This work targets improving on the CPU overhead of the Volcano model while maintaining low compilation times. This is achieved by a push-based operator interface. Such operators, when composed and inlined, yield highly integrated code with fewer indirections than in Volcano-style engines. The author uses a mixed LLVM/C++ execution engine where the algebraic representation of the operators is mapped to low-level LLVM code, while the remaining database code (e.g. management of data structures and memory allocation) is *precompiled* C++ code called from the LLVM code whenever needed. The paper argues that optimizations should happen completely before code generation (e.g. in the algebraic representation), which suggests that HyPer does not perform code optimizations of its own that cannot be expressed in the high-level plan language and that it uses template expansion to obtain LLVM code from the optimized plan.

Of course, LLVM performs certain standard low-level optimizations out of the box, from which all the code generators that employ LLVM (which includes HyPer [255], MemSQL, Peloton [271, 240], and a number or recent and unpublished industrial projects) can profit.

In addition, LLVM as a framework for manipulating intermediate representations allows to add code transformers similarly to SC, and it can even be "tricked" into accepting somewhat high-level LLVM code through calls to external interfaces. This solves the code explosion problem of template expanders. However, the ability to support high-level code is limited

and pattern matching is not available for the implementation of transformers, making it impractical to follow our approach by replacing SC and Scala by LLVM and C.

We note that other than for the systems mentioned, we are not aware of any publication confirming or denouncing the choice of template expansion, but it appears that most systems achieve code generation by template expansion.

There has recently been extensive work on how to specialize the code of query operators in a systematic way by using an approach called Micro-Specialization [379, 380]. In this line of work, the authors propose a framework to encode DBMS-specific intra-operator optimizations, like unrolling loops and removing if conditions, as precompiled templates in an extensible way. All these optimizations are performed by default by the SC compiler in DBLAB/LB. However, in contrast to our work, there are two main limitations in Micro-Specialization. First, the low-level nature of the approach makes the development process very time-consuming: it can take days to code a single intra-operator optimization [379]. Such optimizations are very fine-grained, and it should be possible to implement them quickly: for the same amount of time we are able to provide much more coarse-grained optimizations in DBLAB/LB. Second, the optimizations are limited to those that can be statically determined by examining the DBMS code and cannot be changed at runtime. Our architecture maintains all the benefits of Micro-Specialization, while it is not affected by these two limitations.

Optimizing Database Applications. Database applications are generally written using a combination of procedural and declarative languages. More precisely, the business logic is usually implemented in an imperative language such as Java, whereas the data access part is implemented using SQL. Having two different environments can cause a performance penalty. This is because, in our experience, neither of the environments can easily leverage opportunities available in the other. For example, the Java environment does not know about the indexes in the database system, whereas the database system does not see the loops in the Java code [302].

One way to optimize such programs is by using program analysis techniques to extract declarative queries from the imperative code [59, 58, 56, 364, 365]. As a result, the extracted code can benefit from the optimizations offered by the underlying database system. Furthermore, it is possible to partition database applications between the application runtime and the database system [56, 57], merge several related queries into a single query [139, 232], and prefetch the query results [279, 54]. However, as SQL is not as expressive as an imperative language, this approach is not applicable to all database applications. In addition, for applying optimizations available at a lower level of abstraction (e.g. operator inlining, inter-operator optimization, etc.), one should rely on the database system.

Compiler Optimizations in High-Performance Computing. There has been a large body of work in the high-performance computing (HPC) community since the 1970s for opti-

mizing data-intensive programs [11]. The key optimizations for improving parallelization and data locality are loop transformations such as loop fusion [14], loop interchange [366], loop tiling [235], and loop-invariant code motion [11], as well as data layout transformations [75, 382].

Similar techniques can be used for optimizing database applications by rewriting both the application logic and the data access part into an intermediate language that is built around looping constructs, such as UniQL [302] and forelem [286]. This way, all the optimizations happening in *both* the application runtime (e.g., the underlying optimizing compiler of the application program) and the database system (e.g., query optimization) become applicable directly. As the intermediate language in such systems is expressive enough, these systems enable various optimizations such as classical compiler optimizations (e.g., DCE and CSE), loop transformations (e.g., loop fusion and loop invariant code motion) [302, 285, 286], inter-operator optimizations by merging query operators by removing the unnecessary intermediate materialized data [286], data-layout transformations [286], and even some forms of data-structure specialization (such as using flat arrays instead of hash tables) [285]. More recently, HPAT [341] and Weld [264] investigate the use of a similar loop-oriented intermediate language for optimizing mixed data-intensive workloads such as SQL and machine learning. In contrast, our approach utilizes multiple intermediate languages (DSLs) and, thus, makes it possible to plug in optimizations available across different abstraction levels, which in some cases leads to a simpler optimization than the one expressed in a single loop-oriented intermediate language. As an example, expressing pushdown predicates is simpler in relational algebra than loop-invariant code motion in a loop-oriented intermediate language.

Language Integrated Queries in Managed Runtimes. Developers can use language integrated queries (LINQ [238]) as an interface for accessing databases in managed runtimes such as JVM or CLR (Common Language Runtime). Recently, there were several efforts in order to boost the performance of the database applications written using this approach using database-inspired strategies and optimizations through code generation and just-in-time compilation [134, 251, 252, 352]. In general, all these techniques employ compilation techniques to convert high-level LINQ programs to more efficient, imperative low-level code. This line of work is related to LegoBase/SC, since it mostly targets making query processing of collections in the memory space of the application more efficient by leveraging database technology. As an example, [252] improves the performance of the standard .NET collection implementation behind LINQ (also known as LINQ-to-objects) by using code generation and modifying the memory layout of a collection of records, from a generic array of pointers to objects allocated on the managed heap, into an array of contiguous objects. However, due to the lack of multiple intermediate languages in these systems, it is not possible to support data-structure specialization. Having said that, it would be interesting to add a collection programming frontend, similar to LINQ, to DBLAB/LB and see how such techniques can be leveraged to improve the performance of our Scala-based query engines (e.g. those presented in Figure 2.12), however we leave this for future work.

Techniques to speed up query processing. There are many works that aim to speed up query processing in general, by focusing on improving the way data is processed. Examples of such works include block-wise processing [263], vectorized execution [308], compression techniques to provide constant-time query processing [282] or a combination of the above along with a column-oriented data layout [231]. These approaches are orthogonal to this work as our framework provides a high-level framework for encoding *all* such optimizations in a user-friendly way (e.g. we present the transition from row to column data layout in Section 2.3.3).

Domain-specific compilation, which admits domain-specific optimizations, is a topic of great current interest in multiple research communities. Once one limits the domain or language, program analysis can be more successful. More powerful and global transformations then become possible, yielding speedups that cannot be expected from classical compilers for general purpose languages. To this end, multiple frameworks and research prototypes [158, 105, 347, 191, 288, 7, 215, 177, 159], have been proposed to easily introduce and perform domain-specific compilation and optimization for systems. Of interest is the observation that domain-specificity has already benefited query optimization tremendously: Relational algebra is a domain-specific language, and yields readily available associativity properties that are the foundation of query optimization. Optimizing compilers can combine the performance benefits of classical interpretation-based query engines with the benefits of abstraction and indirection elimination by compilers. Finally, OCAS [201] has been developed within the context of domain-specific synthesis and attempts to automatically generate optimized out-of-core algorithms for a particular target memory hierarchy.

Previous realization of the abstraction without regret vision. In the context of database systems, we have previously realized this vision in [199]. In Chapter 2 of this thesis, intended as an expanded version of [199], we provide a from scratch implementation of the vision using a *new* optimizing compiler, called SC, developed specifically to handle the optimization needs of large-scale software systems. We also present a detailed analysis of the compiler interfaces of SC as well as a significantly more thorough list of the optimizations supported by the DBLAB/LB system in order to demonstrate the ease-of-use of our compiler framework for optimizing database components that differ significantly in granularity and scope of operations. Finally, we provide a more extensive evaluation where, along with a renewed analysis of the previous results, we also evaluate three additional query engine configurations. We do so in order to compare as fairly as possible the performance of our system with that of previous work.

9.3 Fusion and Pipelining

Fusion in Array Languages. There are many array programming languages in the literature, APL [165] being the pioneer among them. There are functional array languages such as Futhark [148] and SAC [131] with support for fusion.

In array languages fusion can be achieved by using functional arrays known as *push* and *pull arrays* [326, 15, 66]. A push-array is represented by an effectful function that, given an index and a value, will write the value into the array. A pull-array is represented by the length of the array and a function producing an element for a given index, similar to the `build` construct in $\widetilde{\text{F}}$.

Fusion in Functional Collections. Loop fusion or Deforestation [357] removes the intermediate collections in collection programs. By restricting the language to a pure functional DSL, the intermediate collections can be removed using local transformations instead of global transformations. This approach is known as *short-cut* deforestation. Short-cut deforestation has been successfully implemented in the context of Haskell [325, 71, 121] and Scala-based DSLs [175, 301]. This can be achieved either by pulling the stream of data [325, 71] or pushing it [121].

Pipelining in Query Engines. The iterator model is the most widely used pipelining technique in query engines, which was initially proposed in XRM [225], and was adopted in the Volcano system [128]. Push-based engines are widely used in streaming systems [150]. The query compilers of systems such as HyPer [255] and LegoBase [199, 300] (cf. Chapter 2) use a push-based query engine approach. In Chapter 4, we showed the connection between the push-based engines and the fold-fusion approach, as well as the pull-based engines and the unfold-fusion approach.

9.4 Memory Management

Programming Languages without GC. Functional programming languages without garbage collection dates back to Linear Lisp [24]. However, most functional languages (dating back to Lisp in around 1959) use garbage collection for managing memory.

Region-based memory management [339] was first introduced in ML and then in an extended version of C, called Cyclone [133], as an alternative or complementary technique to in order to remove the need for runtime garbage collection. This is achieved by allocating memory regions based on the liveness of objects. This approach improves both performance and memory consumption in many cases. However, in many cases the size of the regions is not known, whereas in our approach the size of each storage location is computed using the shape expressions. Also, in practice there are cases in which one needs to combine this technique with garbage collection [141], as well as cases in which the performance is still not satisfying [36, 337]. Furthermore, the complexity of region inference hinders the maintenance

of the compiler, in addition to the overhead it causes for compilation time.

Safe [247, 246] suggests a simpler region inference algorithm by restricting the language to a first-order functional language. Also, linear regions [111] relax the stack discipline restriction on region-based memory management, due to certain usecases which use recursion and need an unbounded amount of memory. A Haskell implementation of this approach is given in [198]. The situation is similar for the linear types employed in Rust; due to loops it is not possible to enforce stack discipline for memory management. However, $\widetilde{\mathrm{F}}$ offers a restricted form of recursion, which always enforces a stack discipline for memory management.

Push-Arrays There is a close connection between so-called *push arrays* [326, 15, 66] and destination-passing style. A push-array is represented by an effectful function that, given an index and a value, will write the value into the array. This function closure captures the destination, so a program using push arrays is also using a form of destination-passing style. There are many differences, however. Our *functions* are transformed to destination-passing style, rather than our *arrays*. Our transformation is not array-specific, and can apply to any large object. Even though our basic array primitives are based on explicit indices, they are referentially transparent and may be read purely functionally. Our focus is on efficient allocation and freeing of array memory, which is not mentioned in the push-array literature. It may not be clear when the memory backing a push-array can be freed, whereas it is clear by construction in our work, and we guarantee to run without a garbage collector. Unsurprisingly, this guarantee comes with a limitation on expressiveness: we cannot handle operations such as filter, whose result size is data-dependent (cf. Section 5.3.7). Happily a large class of important applications can be expressed in our language, and enjoy its benefits.

Estimation of Memory Consumption. One can use type systems for estimating memory consumption. Hofmann and Jost [154] enrich the type system with certain annotations and uses linear programming for the heap consumption inference. Another approach is to use sized types [348] for the same purpose.

Size slicing [146] uses a technique similar to ours for inferring the shape of arrays in the Futhark programming language. However, in $\widetilde{\mathrm{F}}$ we guarantee that shape inference is simplified and is based only on size computation, whereas in their case, they rely on compiler optimizations for its simplification and in some cases it can fall back to inefficient approaches which in the worst case could be as expensive as evaluating the original expression [154]. The FISh programming language [166] also makes shape information explicit in programs, and resolves the shapes at compilation time by using partial evaluation, which can also be used for checking shape-related errors [167]. Our shape translation (Section 5.3.3) is very similar to their shape analysis, but their purposes differ: theirs is an analysis, while ours generates for every function f a companion shape function that (without itself allocating) computes f's space needs; these companion functions are called at runtime to compute memory needs.

Optimizing Tail Calls. Destination-passing style was originally introduced in [213], then was encoded functionally in [242] by using linear types [359]. Walker and Morrisett [360] use

extensions to linear type systems to support aliasing which is avoided in vanilla linear type systems. The idea of destination-passing style has many similarities to *tail-recursion modulo cons* [115, 356].

9.5 Differentiation

Automatic Differentiation. There is a large body of work on automatic differentiation (AD) of imperative programming languages. Tapenade [143] performs AD for a subset of C and Fortran, whereas, ADIFOR [37] performs AD for Fortran programs. Adept [155] and ADIC [254] perform automatic differentiation for C++ by using expression templates. However, as we have seen in our experimental results, an AD tool such as Tapenade misses several optimization opportunities, mainly due to their limited support for loop fusion.

ADiMat [38], ADiGator [362], and Mad [112] perform AD for MATLAB programs, whereas MuPAD [151] computes the derivatives using symbolic differentiation. AutoGrad [228] performs AD for Python programs that use NumPy library for array manipulation, whereas Theano [31] uses symbolic differentiation. Tensorflow [5] performs source-to-source reverse-mode AD, and uses advanced heuristics to solve the memory inefficiencies. ForwardDiff [284] employs vector forward-mode AD [194] for differentiating Julia programs. This system keeps a vector of derivative values in the dual number instead of only a single derivative value. All these systems miss important optimization opportunities such as loop fusion.

DiffSharp [28] is an AD library implemented in F#. This library provides both forward-mode and reverse-mode AD techniques. As DiffSharp is a library implementation of AD (in contrast to Lago, which implements AD as source-to-source transformation rules), it cannot not support the simplification rules such as loop-invariant code motion, loop fusion, and partial evaluation. Furthermore, Lago can efficiently manage memory by generating C code using DPS, whereas DiffSharp should rely on the garbage collection provided by the .NET framework for memory management.

Stalingrad [273] is an optimizing compiler for a dialect of Scheme with a first-class AD operator, with the support for both forward mode and reverse mode of AD. One of the key challenges that Stalingrad addresses is perturbation confusion [306], which occurs for computing the derivative of the functions for which the derivatives are already computed, or the cases where we need the computation of nested differentiation [272]. We have shown in Section 6.3.3 how Lago resolves the perturbation confusion problem. One key limitation of Stalingrad is the lack of support for variable-size vectors; Stalingrad only supports a statically-known-size list of elements which are unfolded using Scheme macros.

Karczmarczuk [183] presents a Haskell implementation for both forward and reverse mode AD. Elliott [98] improves this work by giving a more elegant implementation for its forward mode AD. These implementations lack the optimizations offered by transformation rules, such as loop fusion.

Numerical DSLs and Differentiation. There are many DSLs for numerical workloads. These DSLs can be classified in three categories. The first category consists of mainstream programming languages used by data analysts such as MATLAB and R. These languages offer many toolboxes for performing a wide range of tasks, however, from a performance point of view the focus is only on the efficient implementation of the libraries. The second category consists of DSLs such as Lift [315], Opt [90], Halide [278], Diderot [62], and OptiML [321], which generate parallel code from their high-level programs. The third category is the DSLs which focus on generating efficient machine code for fixed size linear algbra problems such as Spiral [277] and LGen [310]. These DSLs exploit the memory hierarchy by relying on searching algorithms for making tiling and scheduling decisions. Except the first category, for which automatic differentiation tools exist, the other DSLs do not have any support for automatic differentiation. Moreover, parallel code generation and efficient machine code generation are orthogonal concepts and can be added to Lago in the future.

9.6 Incrementalization

Computer Algebra Systems. CAS are software programs for the automation of tedious and difficult algebraic manipulation tasks; some perform symbolic computations including differentiation and integration. Examples include Mathematica [367], MAPLE [245] and Theano [334]. Lago performs symbolic computation in a sense that it derives Δ expressions using the reduction rules that we present in this thesis. Lago differs from typical CAS in its ability to derive incremental programs; perform cost-based optimization; and generate efficient specialized code.

Automatic Differentiation. As we discussed, Automatic differentiation (AD) [187] tools automatically computie the derivative of a given program, which makes them an essential component of the current machine learning frameworks. Even though there is a strong analogy between forward-mode AD and our delta derivation process (e.g., delta/differentiation derivation rules such as $\Delta_{x \mapsto x+1}(x^2) \rightarrow 2x+1$ and $(x^2)' \rightarrow 2x$ and binding intermediate subexpressions to new variables), delta derivation is based on discrete differences rather than differentials. It would be interesting to see if there is an equivalent of backward-mode AD for incremental computation.

Abstract Interpretation. The idea of abstract interpretation dates back to the 70s [70]. In the context of databases, this technique has been used by [250] to derive provenance information in SQL programs. Lago uses abstract interpretation for guiding the synthesis of incremental analytics; abstract interpretation provides a powerful framework for assigning semantics to alternative optimizer options, allowing the generation of highly optimized code.

Scientific Databases. RasDaMan [26] and AML [233] represent database systems that are specialized in array processing. They provide infrastructure for expressing and optimizing queries over multidimensional arrays. Queries are translated into an array algebra and optimized using a large collection of transformation rules. ASAP [318] supports scientific computing

primitives on a storage manager optimized for storing multidimensional arrays. Additionally, RIOT [381] provides an efficient out-of-core framework for scientific computing. However, none of these systems support incremental computation. In contrast, Lago is specialized for supporting IVM of matrix programs. Moreover, it provides a generic unified framework for different semiring configurations of matrix algebra.

High Performance Computing. There is high demand for efficient matrix manipulation in numerical and scientific computing. BLAS [95] exposes a set of low-level routines that represent common linear algebra primitives for higher-level libraries including LINPACK, LAPACK, and ScaLAPACK for parallel processing. Hardware vendors such as Intel or AMD and code generators such as ATLAS [363] provide highly optimized BLAS implementations for dense linear algebra. Moreover, other works such as Combinatorial BLAS [48, 94] provide efficient BLAS implementations dedicated for sparse linear algebra. All of this work is orthogonal to Lago as it operates at a higher level of abstraction. In essence, IVM translates input matrix programs to trigger code that calls cheaper matrix BLAS primitives.

Iterative Computation. Recently, there has been a growing interest in designing frameworks for iterative and incremental computation. The differential dataflow model [236] presents a new methodology to model incremental computation for iterative algorithms. Their approach relies on the assumption that input changes result in small changes down the road. However, this assumption does not hold for matrix algebra programs because of the avalanche effect of input changes as described in this thesis. For iterative applications under the MapReduce framework, several systems [46, 96, 380] have been proposed. They present techniques that cache and index loop-invariant data on local disks and persist materialized views between iterations. Moreover, Dryad [164] and Spark [377] represent systems that support iterative computation under the general DAG execution model. Mahout, MLbase [207] and others [81, 351, 241] provide scalable machine learning and data mining tools. All these systems are orthogonal to Lago. Our work is concerned with the design and implementation of a compiler framework for the incremental view maintenance of matrix algebra. Moreover, the framework can be easily coupled with any of these underlying systems at the code generation layer as we illustrate in the evaluation section § 7.6 with Spark.

IVM and Stream Processing. Incremental View Maintenance techniques [39, 205, 138] support incremental updates of database materialized views by employing differential algorithms to re-evaluate the view expression. Chirkova *et al.* [61] present a detailed survey on this direction. Moreover, data stream processing engines [1, 249, 18] incrementally evaluate continuous queries as windows advance over unbounded input streams. In contrast to all the previous approaches, Lago targets incremental maintenance of linear algebra programs as opposed to classical database (SQL) queries. The linear algebra domain has different semantics and primitives; thus, the challenges and optimization techniques widely differ.

Graph Analytics. There is plethora of frameworks dedicated for graph processing including Powergraph [126], Pregel [230], GraphLab [226, 227], GraphChi [209], and Galois [256]. They

provide various programming models specialized for graph processing based on Bulk Synchronous Programming. Recently, there has been work on representing graph algorithms using sparse matrix manipulation operations including CombBLAS [48] and [94]. However, none of these systems support incremental computation. There have been several works that target incremental computation of specific graph problems [65, 132], including connectivity [156], minimum spanning tree [156], transitive closure [86, 87], and all-pairs shortest path [88, 196, 88]. However, each of these solutions aim at a particular graph problem and are not represented as matrix computations. In contrast, Lago provides a general matrix framework that supports graph IVM, cost-based optimization, and low-level specializations.

Linear Algebra DSLs. The Spiral [277] project provides a domain-specific compiler for synthesizing Digital Signal Processing kernels, e.g., Fourier transforms. The authors present the SPL [371] language that expresses recursion and formulas in a mathematical form. They present a framework that optimizes at the algorithmic and implementation level and that uses runtime information to guide the synthesis process. The LGen compiler [309] targets small scale basic linear algebra computations of fixed size linear algebra expressions which are common in graphics and media processing applications. The authors present two level DSLs, namely LL to perform tiling decisions and Σ-LL to enable loop level optimizations. The generated output is a C function that includes intrinsics to enable SIMD vector extensions. Orthogonally, Lago targets IVM of LA programs for different domains, i.e., semiring configurations, and is restricted to high-level optimizations. The closest to our work is the basic linear algebra compiler presented in [104]. It decomposes a linear algebra target equation into a sequence of computations provided by BLAS or LAPACK and generates associated Matlab code. Similar to our work, their approach exploits domain knowledge and properties of the operands by rewriting and inference rules. However, we focus on IVM and optimization under this setting.

PageRank. There is a large body of literature focusing on various aspects of the PageRank problem, including the Markov chain model, sensitivity and conditioning, and the updating problem. Langville *et al.* [211] and Berkhin *et al.* [32] provide detailed surveys about such topics. In particular, the updating problem examines the effect of delta perturbations on the various Markov chain and PageRank models, ranging from sensitivity analysis to approximation and exact evaluation methods. These methods and techniques particularly target specific models. On the other hand, this report introduces a novel framework for efficient incremental evaluation of *general* linear algebra programs under various semiring domains using domain-specific transformations, cost-based optimizations, and efficient code generation.

Incremental Statistical Frameworks. Bayesian inference [30] uses the Bayes' rule to update the hypothesis's probability estimate as additional evidence is acquired. A variety of applications can be built on top of these frameworks including pattern recognition and classification. Our work focuses on incrementalizing applications that can be expressed as linear algebra programs and generating efficient incremental programs for different runtime environments.

Incremental Computation in Programming Languages. The PL community has extensively explored the direction of incremental computation and information flow [55]. They have developed compilation techniques that translate high-level programs into executables that are amenable to dynamic changes. Moreover, self-adjusting computation supports incremental computation by exploiting dynamic dependency graphs and change propagation algorithms [6, 55]. However, these approaches differ from our work on several dimensions:*a*) Firstly, they target general purpose programs in comparison to our domain-specific approach. *b*) Secondly, they require developer knowledge and involvement by annotating the modifiable parts of the program. *c*) Finally, they cannot capture the propagation of deltas across statements and efficiently represent them in a compressed form as presented in this thesis.

10 Conclusions and Future Work

The PL and compilers communities have a massive literature on optimizing programs written in high-level languages. On the other hand, the DB community has introduced one of the most successful domain-specific languages, SQL, together with many techniques for optimizing queries in this language.

In this thesis, we showed how such multi-disciplinary techniques can be combined for building three different data analytics systems. We first showed the design of SC, a compiler framework for building efficient and high-level data analytics systems. Then, we presented the LegoBase system and the challenges that we tackled for building efficient query engines in a high-level language. Then, we presented the design and implementation of the $d\widetilde{F}$ system for efficiently computing the derivative of computer vision and machine learning programs. Finally, we presented Lago, for efficiently and incrementally computing complex analytics tasks expressed by linear algebra, such as some machine learning functions and all-pairs variants of several graph algorithms.

There are still many existing techniques in these communities that can be adapted for efficient data analytics. The idea of using search algorithms for optimizing programs has been successfully used in the query optimizer component of database systems. We showed how to use the cost-based optimization technique for the Lago system. Similary, this technique can be used for automating the process of choosing the right set of optimizations for both LegoBase and $d\widetilde{F}$ systems.

There are two key challenges for using the cost-based optimization technique. First, one has to have an estimatation of the run-time cost of a given program. Second, searching among different versions of the same program becomes close to impossible for large search spaces. The former problem has been the topic of research in the PL and formal methods communities [176, 154, 78, 13]. Most of the existing work focuses mainly on the soundness of the cost approximation, which is useful for verification purposes. However, in program optimization we are more interested in the expected run-time cost. For Turing-complete programming languages even determining the termination of a program is undeciable. By

restricting the programming language for which we perform the cost estimation, e.g., to a restricted DSL, one can achieve more precise cost estimation, as demonstrated in OCAS [201] and Lago. One solution to the latter problem is using randomized search algorithms such as the ones used for query optimization [163] and Monte-Carlo Tree Search [84]. Also, it is possible to use more advanced machine learning techniques such as deep reinforcement learning, which is used for solving challenging AI problems such as beating professional players in Atari games [243] and Go [304, 305].

Another future direction is to extend the incrementalization techniques used in Lago for other types of applications that can be formulated using linear algebra. Also, another direction is to support incrementalization for lower level languages such as $\widetilde{\text{F}}$. This way, one can leverage the benefits of incrementalization for a wider range of applications, such as differentiable programming.

Furthermore, thanks to the extensibility of the systems presented in this thesis, one can add other target backends. One possible direction is to generate JavaScript query engines similar to AfterBurner [97] for performing in-browser analytics.

In all of the presented systems, we have either assumed that we run on a single-threaded architecture, or we relied on the runtime environment for handling parallelism. As mentioned in Section 3.8 one can use the well-studied intra-operator parallelism from the DB literature [128, 237], without modifications to the rest of the components. Also, one can exploit the vectorization intrinsics offered by the underlying architecture (e.g., SIMD instructions [383]) for both LegoBase and d$\widetilde{\text{F}}$ systems.

Finally, it has been already noted in the DB community [316] that the "one size fits all" belief no longer holds; one should build a specialized DBMS for different types of workloads. The systems presented in this thesis, as well as the ones we propose in this section, can be built as a single framework. This enables sharing many components among these systems. By using a library of optimizations, we can rely on the optimizing compiler to perform the specialization process [204]. Similarly, one can extend the abstraction without regret vision for building other types of performance-critical systems such as operating systems and networked systems.

A Absolute Execution Times of LegoBase Experiments

For completeness, the following tables present the *absolute performance results* of all evaluated systems and metrics in the experimental chapter of thesis.

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
DBX	1790	396	1528	960	19879	882	969	2172	3346	985	461
Compiler of HyPer	779	43	892	622	338	198	798	493	2139	565	102
LegoBase (Naive/C) – LLVM	3140	755	5232	10742	3627	357	2901	23161	26203	3836	409
LegoBase (Naive/C) – GCC	3140	801	5204	10624	3652	423	2949	19961	25884	3966	445
LegoBase (Naive/Scala)	3972	6910	11118	30103	10307	654	114677	9852	137369	18367	1958
LegoBase(TPC-H/C)	593	55	767	445	440	199	975	2871	2387	546	98
LegoBase(StrDict/C)	592	47	759	402	439	197	781	346	2027	544	103
LegoBase(Opt/C)	426	42	110	134	126	47	104	18	530	439	49
LegoBase(Opt/Scala)	2174	871	352	306	413	356	9496	104	2296	775	197

System	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
DBX	881	13593	823	578	12793	1224	4535	6432	744	1977	447
Compiler of HyPer	485	2333	197	229	590	490	3682	1421	277	1321	212
LegoBase (Naive/C) – LLVM	3037	12794	1289	889	16362	18893	4135	2810	974	11648	1187
LegoBase (Naive/C) – GCC	3286	13149	1398	899	16159	18410	4174	4460	1055	11848	1396
LegoBase (Naive/Scala)	3565	7909	4424	1543	10568	3503	15798	4470	5301	50998	4207
LegoBase(TPC-H/C)	891	5106	244	550	2774	513	2725	2020	370	1992	453
LegoBase(StrDict/C)	688	910	204	535	702	445	2735	1222	370	1706	333
LegoBase(Opt/C)	120	516	11	46	695	11	133	19	130	388	79
LegoBase(Opt/Scala)	604	7743	136	234	2341	274	355	125	700	955	406

Table A.1 – Execution times (in milliseconds) of Figure 2.11 and Figure 2.12. The various configurations of LegoBase are explained in more detail in Table 2.2 of this thesis.

Appendix A. Absolute Execution Times of LegoBase Experiments

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
LegoBase (Naive/C) – LLVM	3140	755	5232	10742	3627	357	2901	23161	26203	3836	409
+Struct Field Removal	3104	734	4480	10346	2983	202	2394	18707	24125	3323	403
+Domain-Specific Code Motion	1047	794	4283	10435	2902	196	2203	18507	23854	3177	332
+Data-Structure Specialization	497	44	918	148	130	172	96	75	498	610	52
+Date Indices	497	47	213	140	131	52	96	60	568	553	49
+String Dictionaries	497	43	158	140	130	51	94	17	533	552	47
LegoBase(Opt/C)	426	42	110	134	126	47	104	18	530	439	49

	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
LegoBase (Naive/C) – LLVM	3037	12794	1289	889	16362	18893	4135	2810	974	11648	1187
+Struct Field Removal	2631	11291	812	420	16068	17953	4070	2550	736	10647	970
+Domain-Specific Code Motion	2553	9415	786	495	15251	18063	3050	2568	742	10386	985
+Data-Structure Specialization	467	2389	291	277	4243	47	2709	62	168	410	300
+Date Indices	308	2233	38	40	4737	39	2718	46	168	392	291
+String Dictionaries	125	1379	16	52	860	13	2730	20	136	389	299
LegoBase(Opt/C)	120	516	11	46	695	11	133	19	130	388	79

Table A.2 – Execution times (in milliseconds) of TPC-H queries with individual optimizations applied (as shown in Figure 2.14 of this thesis). Each listed optimization is applied additionally to the set of optimizations applied in the system specified above it.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Memory Consumption	7.86	6.20	10.45	6.39	7.56	10.88	14.51	8.72	15.30	14.35	7.53
Loading Time (No opt.)	34	7	44	42	43	33	43	46	45	44	5
Loading Time (All opt.)	38	10	52	47	49	39	55	56	61	52	10
SC Optimization	429	633	482	323	663	128	547	918	608	498	317
CLang C Compilation	354	509	482	359	418	179	332	346	320	507	378

	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Memory Consumption	9.73	8.72	11.06	11.64	1.81	9.26	10.92	7.81	11.77	7.86	5.36
Loading Time (No opt.)	41	9	36	34	7	34	42	35	38	41	9
Loading Time (All opt.)	53	135	42	38	10	47	47	52	53	52	13
SC Optimization	310	215	295	255	518	248	321	357	420	411	389
CLang C Compilation	449	386	454	329	563	461	382	552	566	507	365

Table A.3 – Memory consumption in GB, input data loading time in seconds, and optimization/compilation time in milliseconds as shown in Figure 2.15, Figure 2.16, and, Figure 2.18 of this thesis, respectively.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
DBX	87.56	80.01	79.85	82.23	83.37	78.35	84.83	87.22	79.97	80.49	86.82
HyPer	73.45	73.01	73.09	72.97	73.39	73.15	70.86	68.12	66.79	72.71	73.54
LegoBase	62.26	44.34	60.03	70.39	49.35	67.04	28.33	51.9	56.59	59.25	59.3
	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
DBX	82.25	81.11	87.1	88.15	80.83	94.37	88.34	86.45	79.44	96.49	96.77
HyPer	71.02	74.07	74.17	72.8	72.3	73.36	70.54	73.19	71.88	71.17	69.25
LegoBase	64.3	53.73	67.02	62.09	62.7	46.72	60.11	56.31	46.87	28.5	35.72

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
DBX	1.1	3.01	0.21	0.33	2.17	1.24	0.32	2.85	3.12	0.48	3.09
HyPer	2.26	2.41	2.37	2.38	2.47	2.38	2.36	2.41	2.37	2.46	2.44
LegoBase	1.66	0.95	1.55	1.83	1.71	2.19	1.85	2.4	1.77	1.69	0.59
	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
DBX	1.95	0.32	0.34	2.41	3.01	2.96	2.12	0.5	2.29	3.01	3.01
HyPer	2.38	2.57	2.81	2.41	2.55	2.59	2.32	2.35	2.43	2.59	2.59
LegoBase	2.22	0.46	1.85	2.38	0.88	1.7	1.28	2.22	1.47	1.8	2.58

Table A.4 – Cache Miss Ratio (%) and Branch Misprediction Rate (%) for DBX, HyPer and LegoBase, respectively, as shown in Figure 2.13 of this thesis.

B Code Snippet for the Partitioning Transformer of LegoBase

Next, we present a portion of the data partitioning transformation, an explanation of which was given in Section 2.3.2. This code corresponds to the join processing for equi-joins (and not the actual partitioning of input data), but similar rules are employed for other join types as well. The aim of this snippet is to demonstrate the ease-of-use of the SC compiler.

```
/* A transformer for partitioning and indexing MultiMap data-structures. As a result,
   this
    transformation converts MultiMap operations to native Array operations. */
class HashTablePartitioning extends RuleBasedTransformer {

  val allMaps = mutable.Set[Any]()
  var currentWhileLoop: While = _

  /* ---- ANALYSIS PHASE ---- */
  /* Gathers all MultiMap symbols which are holding a record as their value */
  analysis += statement {
    case sym -> code"new MultiMap[_, $v]" if isRecord(v) => allMaps += sym
  }

  /* Keeps the closest while loop in scope (used in the next analysis rule)*/
  analysis += rule {
    case whileLoop @ code"while($cond) $body" => currentWhileLoop = whileLoop
  }

  /* Maintain necessary information for the left relation */
  analysis += rule {
    case code"($mm: MultiMap[_,_]).addBinding(struct_field($struct,
      $fieldName),$value)"
        => mm.attributes("addBindingLoop") = currentWhileLoop
  }

  /* Maintain necessary information for the right relation */
```

```
analysis += rule {
  case code"($mm : MultiMap[_, _]).get(struct_field($struct, $fieldName))" =>
    mm.attributes("partitioningStruct") = struct
    mm.attributes("partitioningFieldName") = fieldName
}

/* ---- REWRITING PHASE ---- */
def shouldBePartitioned(mm: Multimap[Any, Any]) = allMaps.contains(mm)

/* If the left relation should be partitioned, then remove the 'addBinding' and 'get'
    function calls for this multimap, as well as any related loops. Notice that there is
    no need to remove the multimap itself, as DCE will do so once all of its dependent
    operations have been removed.*/
rewrite += remove {
  case code"($mm: MultiMap[Any, Any]).addBinding($elem, $value)" if
    shouldBePartitioned(mm) =>
}

rewrite += remove {
  case code"($mm: MultiMap[Any, Any]).get($elem)" if shouldBePartitioned(mm) =>
}
rewrite += remove {
  case node @ code"while($cond) $body" if allMaps.exists({
    case mm => shouldBePartitioned(mm) && mm.attributes("addBindingLoop") ==
      node
  }) =>
}

/* If a MultiMap should be partitioned, instead of the construction of that MultiMap
    object, use the corresponding partitioned array constructed during data-loading.
    This can be an 1D or 2D array, depending on the properties and relationships of the
    primary and foreign keys of that table (described in Section 3.2.1 in more detail). */
rewrite += statement {
  case sym -> (code"new MultiMap[_, _]") if shouldBePartitioned(sym) =>
    getPartitionedArray(sym)
}

/* Rewrites the logic for extracting matching elements of the left relation (initially
    using the HashMap), inside the loop iterating over the right relation. */
rewrite += rule {
  case code"($mm:MultiMap[_,_]).get($elem).get.foreach($f)" if
     shouldBePartitioned(mm) =>{
    val leftArray = transformed(mm)
    val hashElem = struct_field(mm.attributes("partitioningStruct"),
                        mm.attributes("partitioningField"))
    val leftBucket = leftArray(hashElem)
    /* In what follows, we iterate over the elements of the bucket, even though the
```

```
        partitioned array may be an 1D−array as discussed in Section 3.1.2. There is
        another optimization in the pipeline which flattens the for loop of this case. */
      for(e <− leftBucket) {
        /* Function f corresponds to checking the join condition and creating the join
            output. This functionality remains the same, thus, we can simply inline the
            related code here as follows */
        ${f(e)}
      }
    }

  /* For a partitioned relation, there is no need to check for emptiness, due to primary /
      foreign key relationship. The if (true) is later removed by another optimization. */
  rewrite += rule {
    case code"($mm: MultiMap[Any, Any]).get($elem).nonEmpty" if
      shouldBePartitioned(mm) =>
      true
  }
}
```

C TPC-H Schema and Queries

The TPC-H schema is shown in the following figure, which is taken from the original benchmark specification [343]. SF stands for *Scaling Factor*, and configures the cardinality of each relation. Attributes marked with the key symbol form the primary key of the corresponding relation. The arrows point in the direction of the one-to-many relationships between tables.

TPC-H Q1

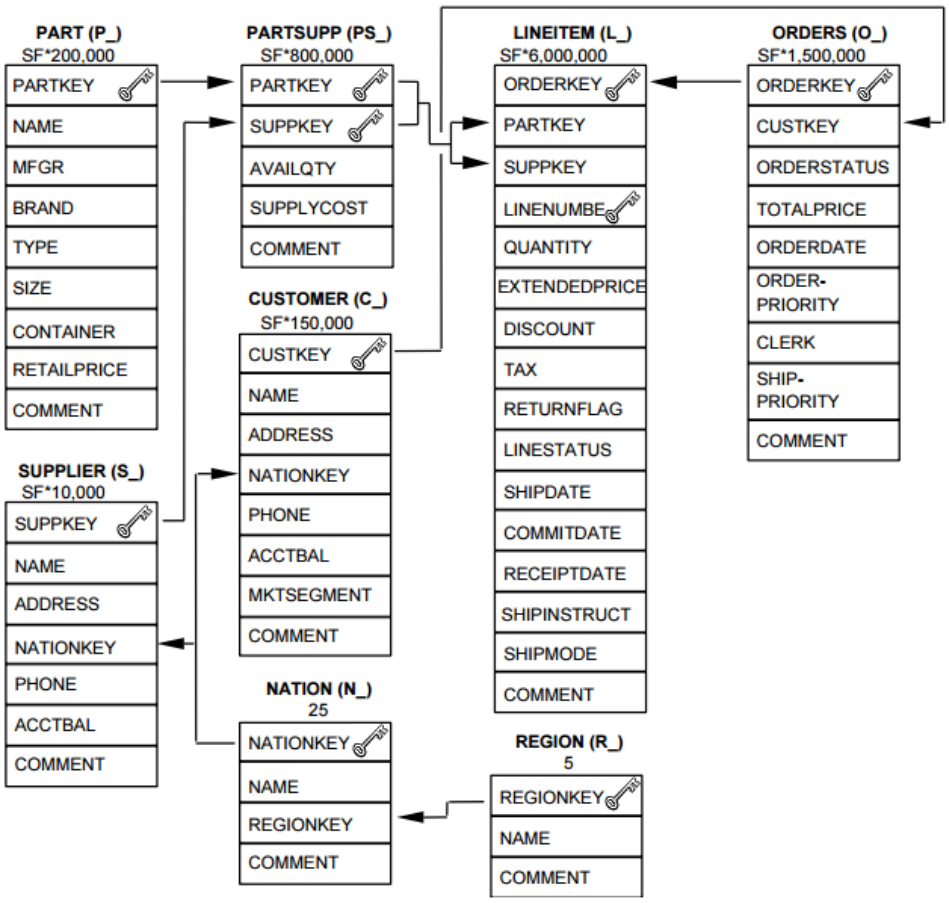


Figure C.1 – The TPC-H schema.

```
SELECT L_RETURNFLAG, L_LINESTATUS, SUM(L_QUANTITY) AS SUM_QTY,
   SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS SUM_DISC_PRICE,
   SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE, AVG(L_QUANTITY) AS AVG_QTY,
   AVG(L_EXTENDEDPRICE) AS AVG_PRICE, AVG(L_DISCOUNT) AS AVG_DISC, COUNT(*) AS COUNT_ORDER
FROM LINEITEM
WHERE L_SHIPDATE <= DATE '1998-09-02'
GROUP BY L_RETURNFLAG, L_LINESTATUS
ORDER BY L_RETURNFLAG, L_LINESTATUS
```

TPC-H Q2

```
SELECT TOP 100 S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY, P_MFGR, S_ADDRESS, S_PHONE, S_COMMENT
FROM SUPPLIER JOIN PARTSUPP ON S_SUPPKEY = PS_SUPPKEY
                JOIN NATION ON S_NATIONKEY = N_NATIONKEY
                JOIN PART ON PS_PARTKEY = P_PARTKEY
                JOIN REGION ON N_REGIONKEY = R_REGIONKEY
                JOIN (
    SELECT P_PARTKEY, MIN(PS_SUPPLYCOST) AS MIN_PS_SUPPLYCOST
        FROM SUPPLIER JOIN PARTSUPP ON S_SUPPKEY = PS_SUPPKEY
                      JOIN NATION ON S_NATIONKEY = N_NATIONKEY
                      JOIN PART ON PS_PARTKEY = P_PARTKEY
                      JOIN REGION ON N_REGIONKEY = R_REGIONKEY
    WHERE P_SIZE = 43 AND P_TYPE LIKE '%%TIN' AND R_NAME = 'AFRICA'
    GROUP BY P_PARTKEY
  ) AS TMP_VIEW ON P_PARTKEY = TMP_VIEW.P_PARTKEY AND PS_SUPPLYCOST = MIN_PS_SUPPLYCOST
WHERE P_SIZE = 43 AND P_TYPE LIKE '%%TIN' AND R_NAME = 'AFRICA'
ORDER BY S_ACCTBAL DESC, N_NAME, S_NAME, P_PARTKEY
```

TPC-H Q3

```
SELECT TOP 10 L_ORDERKEY, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)), O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER JOIN ORDERS ON C_CUSTKEY = O_CUSTKEY
              JOIN LINEITEM ON O_ORDERKEY = L_ORDERKEY
WHERE C_MKTSEGMENT = 'HOUSEHOLD' AND
      O_ORDERDATE < DATE '1995-03-04' AND L_SHIPDATE > DATE '1995-03-04'
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
```

TPC-H Q4

```
SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
FROM ORDERS LEFT SEMI JOIN LINEITEM ON O_ORDERKEY = L_ORDERKEY AND L_COMMITDATE < L_RECEIPTDATE
WHERE O_ORDERDATE >= DATE '1993-08-01' AND O_ORDERDATE < DATE '1993-11-01'
GROUP BY O_ORDERPRIORITY
ORDER BY O_ORDERPRIORITY
```

TPC-H Q5

```
SELECT N_NAME, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE
FROM REGION JOIN NATION ON R_REGIONKEY = N_REGIONKEY
JOIN CUSTOMER ON N_NATIONKEY = C_NATIONKEY
JOIN ORDERS ON C_CUSTKEY = O_CUSTKEY
JOIN LINEITEM ON O_ORDERKEY = L_ORDERKEY
JOIN SUPPLIER ON L_SUPPKEY = S_SUPPKEY AND N_NATIONKEY = S_NATIONKEY
WHERE R_NAME = 'ASIA' AND O_ORDERDATE >= DATE '1996-01-01' AND O_ORDERDATE < DATE '1997-01-01'
GROUP BY N_NAME
ORDER BY REVENUE DESC
```

TPC-H Q6

```
SELECT SUM(L_EXTENDEDPRICE * L_DISCOUNT) AS REVENUE FROM LINEITEM
WHERE L_SHIPDATE >= DATE '1996-01-01' AND L_SHIPDATE < DATE '1997-01-01'
  AND L_DISCOUNT BETWEEN 0.08 AND 0.1 AND L_QUANTITY < 24;
```

TPC-H Q7

```
SELECT N1.N_NAME, N2.N_NAME, YEAR(L_SHIPDATE), SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS VOLUME
FROM NATION N1 JOIN NATION N2
   JOIN SUPPLIER ON N1.N_NATIONKEY = S_NATIONKEY
   JOIN LINEITEM ON S_SUPPKEY = L_SUPPKEY
   JOIN ORDERS ON L_ORDERKEY = O_ORDERKEY
   JOIN CUSTOMER ON O_CUSTKEY = C_CUSTKEY AND N2.N_NATIONKEY = C_NATIONKEY
WHERE ((N1.N_NAME = 'UNITED STATES' AND N2.N_NAME = 'INDONESIA') OR
       (N1.N_NAME = 'INDONESIA' AND N2.N_NAME = 'UNITED STATES')) AND
L_SHIPDATE >= DATE '1995-01-01' AND L_SHIPDATE <= DATE '1996-12-31'
GROUP BY N1.N_NAME, N2.N_NAME, O_YEAR
ORDER BY N1.N_NAME, N2.N_NAME, O_YEAR
```

TPC-H Q8

```
SELECT YEAR(O_ORDERDATE),
  SUM(CASE WHEN N2.N_NAME = 'INDONESIA' THEN L_EXTENDEDPRICE*(1-L_DISCOUNT) ELSE 0.0 END) /
  SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
FROM NATION N1 JOIN NATION N2
   JOIN REGION ON N1.N_REGIONKEY = R_REGIONKEY
   JOIN SUPPLIER ON N2.N_NATIONKEY = S_NATIONKEY
   JOIN LINEITEM ON S_SUPPKEY = L_SUPPKEY
   JOIN PART ON L_PARTKEY = P_PARTKEY
   JOIN ORDERS ON L_ORDERKEY = O_ORDERKEY
   JOIN CUSTOMER ON O_CUSTKEY = C_CUSTKEY AND N1.N_NATIONKEY = C_NATIONKEY
WHERE R_NAME = 'ASIA' AND O_ORDERDATE >= DATE '1995-01-01' AND O_ORDERDATE < DATE '1996-12-31'
   AND P_TYPE = 'MEDIUM ANODIZED NICKEL'
GROUP BY O_YEAR
ORDER BY O_YEAR
```

TPC-H Q9

```
SELECT N_NAME, YEAR(O_ORDERDATE), SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)-PS_SUPPLYCOST *
   L_QUANTITY)
FROM LINEITEM JOIN PART ON L_PARTKEY = P_PARTKEY
   JOIN SUPPLIER ON L_SUPPKEY = S_SUPPKEY
   JOIN NATION ON S_NATIONKEY = N_NATIONKEY
   JOIN PARTSUPP ON L_PARTKEY = PS_PARTKEY AND L_SUPPKEY = PS_SUPPKEY
   JOIN ORDERS ON L_ORDERKEY = O_ORDERKEY
WHERE P_NAME LIKE '%%ghost%%'
GROUP BY N_NAME, O_YEAR
ORDER BY N_NAME, O_YEAR DESC
```

TPC-H Q10

```
SELECT TOP 20 C_CUSTKEY, C_NAME, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
                  C_ACCTBAL, N_NAME, C_ADDRESS, C_PHONE, C_COMMENT
FROM LINEITEM JOIN ORDERS ON L_ORDERKEY = O_ORDERKEY
   JOIN CUSTOMER ON O_CUSTKEY = C_CUSTKEY
   JOIN NATION ON C_NATIONKEY = N_NATIONKEY
WHERE O_ORDERDATE >= DATE '1994-11-01' AND O_ORDERDATE < DATE '1995-02-01' AND L_RETURNFLAG = 'R'
GROUP BY C_CUSTKEY, C_NAME, C_ACCTBAL, C_PHONE, N_NAME, C_ADDRESS, C_COMMENT
ORDER BY REVENUE DESC
```

TPC-H Q11

```
SELECT PS_PARTKEY, SUM(PS_SUPPLYCOST*PS_AVAILQTY) AS VALUE
FROM NATION JOIN SUPPLIER ON N_NATIONKEY = S_NATIONKEY
   JOIN PARTSUPP ON S_SUPPKEY = PS_SUPPKEY
WHERE N_NAME = 'UNITED KINGDOM'
GROUP BY PS_PARTKEY
HAVING VALUE > (
   SELECT SUM(PS_SUPPLYCOST * PS_AVAILQTY * 0.0001000000) AS TOTAL
   FROM NATION JOIN SUPPLIER ON N_NATIONKEY = S_NATIONKEY
      JOIN PARTSUPP ON S_SUPPKEY = PS_SUPPKEY
   WHERE N_NAME = 'UNITED KINGDOM'
)
```

```
ORDER BY VALUE DESC
```

TPC-H Q12

```
SELECT L_SHIPMODE,
  SUM(CASE WHEN O_ORDERPRIORITY = '1−URGENT' OR O_ORDERPRIORITY = '2−HIGH' THEN 1.0 ELSE 0.0 END)
    AS HIGH_LINE_COUNT,
  SUM(CASE WHEN O_ORDERPRIORITY <> '1−URGENT' AND O_ORDERPRIORITY <> '2−HIGH' THEN 1.0 ELSE 0.0 END )
    AS LOW_LINE_COUNT
FROM ORDERS JOIN LINEITEM ON O_ORDERKEY = L_ORDERKEY
WHERE (L_SHIPMODE = 'MAIL' OR L_SHIPMODE = 'SHIP')
  AND L_COMMITDATE < L_RECEIPTDATE AND L_SHIPDATE < L_COMMITDATE
  AND L_RECEIPTDATE >= DATE '1994−01−01'
  AND L_RECEIPTDATE < DATE '1995−01−01'
GROUP BY L_SHIPMODE
ORDER BY L_SHIPMODE
```

TPC-H Q13

```
SELECT C_COUNT, COUNT(*) AS CUSTDIST
FROM (
  SELECT C_CUSTKEY, COUNT(O_ORDERKEY) C_COUNT
  FROM CUSTOMER LEFT OUTER JOIN ORDERS ON C_CUSTKEY = O_CUSTKEY
  AND O_COMMENT NOT LIKE '%%customer%%complaints%%'
  GROUP BY C_CUSTKEY
) AS C_ORDERS
GROUP BY C_COUNT
ORDER BY CUSTDIST DESC, C_COUNT DESC
```

Note that there exists an efficient *imperative* implementation of this query that does not require any join processing. This implementation operates in two phases. First, we sequentially scan through the ORDERS table and extract which customers do not satisfy the predicate O_COMMENT NOT LIKE ´%%customer%%complaints%%; thus creating an 1-dimensional array indexed by O_CUSTKEY. This array stores how many orders a specific customer has (i.e. the C_COUNT aggregation of the query). This is feasible, since LegoBase collects statistics during data loading and infers that C_CUSTKEY has sequential values in the range [0, #NUM_CLIENTS], where C_CUSTKEY is a primary key. In the second phase, we simply iterate through this aggregation array, re-aggregating based on the counts. We also note that converting the join-based physical query plan to the imperative query plan (as described above) is not currently expressed as a compiler optimization. Instead, for all results reported in this thesis for Q13, we have implemented the aforementioned logic directly in the physical query plan.

TPC-H Q14

```
SELECT SUM(CASE WHEN P_TYPE LIKE 'PROMO%%' THEN L_EXTENDEDPRICE*(1−L_DISCOUNT) * 100 ELSE 0.0 END) /
       SUM(L_EXTENDEDPRICE*(1−L_DISCOUNT)) AS PROMO_REVENUE
FROM PART JOIN LINEITEM ON P_PARTKEY = L_PARTKEY
WHERE L_SHIPDATE >= DATE '1994−03−01' AND L_SHIPDATE < DATE '1994−04−01'
```

TPC-H Q15

```
SELECT S_SUPPKEY, S_NAME, S_ADDRESS, S_PHONE, TOTAL_REVENUE
FROM SUPPLIER JOIN (
  SELECT L_SUPPKEY,
         SUM(L_EXTENDEDPRICE*(1.0−L_DISCOUNT)) AS TOTAL_REVENUE
  FROM LINEITEM
  WHERE L_SHIPDATE >= DATE '1993−09−01' AND L_SHIPDATE < DATE '1993−12−01'
  GROUP BY L_SUPPKEY
```

```
) AS TMP_VIEW
ON S_SUPPKEY = L_SUPPKEY
ORDER BY TOTAL_REVENUE DESC
```

TPC-H Q16

```
SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(*) AS SUPPLIER_CNT
FROM (
   SELECT COUNT(*) AS CNT
   FROM PART JOIN PARTSUPP ON P_PARTKEY = PS_PARTKEY
      ANTI JOIN (
         SELECT S_SUPPKEY
         FROM SUPPLIER
         WHERE S_COMMENT LIKE '%%Customer%%Complaints%%'
      ) AS TMP_VIEW ON PS_SUPPKEY = S_SUPPKEY
   WHERE P_BRAND != 'Brand#21' AND
           P_TYPE NOT LIKE 'PROMO PLATED%%' AND
           (P_SIZE = 23 OR P_SIZE = 3 OR P_SIZE = 33 OR P_SIZE = 29 OR
            P_SIZE = 40 OR P_SIZE = 27 OR P_SIZE = 22 OR P_SIZE = 4)
   GROUP BY P_BRAND, P_TYPE, P_SIZE, PS_SUPPKEY
) AS TMP_VIEW
GROUP BY P_BRAND, P_TYPE, P_SIZE
ORDER BY SUPPLIER_CNT DESC, P_BRAND, P_TYPE, P_SIZE
```

TPC-H Q17

```
SELECT SUM(L_EXTENDEDPRICE) / 7
FROM PART JOIN LINEITEM ON P_PARTKEY = L_PARTKEY
   JOIN (
      SELECT P_PARTKEY,
               AVG(0.2 * L_QUANTITY) AS AVERAGE
      FROM LINEITEM JOIN PART ON L_PARTKEY = P_PARTKEY
      WHERE P_BRAND = 'Brand#15' AND
            P_CONTAINER = 'MED BAG'
      GROUP BY P_PARTKEY
   ) AS TMP_VIEW
ON P_PARTKEY = TMP_VIEW.P_PARTKEY AND L_QUANTITY < AVERAGE
WHERE P_BRAND = 'Brand#15' AND P_CONTAINER = 'MED BAG'
```

TPC-H Q18

```
SELECT C_NAME, C_CUSTKEY, O_ORDERKEY, O_ORDERDATE, O_TOTALPRICE,
         SUM(SUM_L_QUANTITY) AS TOTAL_L_QUANTITY
FROM ORDERS JOIN CUSTOMER ON O_CUSTKEY = C_CUSTKEY
   JOIN (
      SELECT L_ORDERKEY, SUM(L_QUANTITY) AS SUM_L_QUANTITY
      FROM LINEITEM
      GROUP BY L_ORDERKEY
      HAVING SUM_L_QUANTITY > 300
   ) AS TMP_VIEW ON O_ORDERKEY = TMP_VIEW.L_ORDERKEY
GROUP BY C_NAME, C_CUSTKEY, O_ORDERKEY, O_ORDERDATE, O_TOTALPRICE
ORDER BY O_TOTALPRICE DESC, O_ORDERDATE
```

TPC-H Q19

```
SELECT SUM(L_EXTENDEDPRICE* (1 - L_DISCOUNT)) AS REVENUE
FROM LINEITEM JOIN PART ON L_PARTKEY = P_PARTKEY
WHERE (P_BRAND = 'Brand#31' AND
  (P_CONTAINER = 'SM CASE' OR P_CONTAINER = 'SM BOX' OR P_CONTAINER = 'SM PACK' OR P_CONTAINER = 'SM
     PKG')
  AND L_QUANTITY >= 4 AND L_QUANTITY <= 14 AND P_SIZE<=5 AND
  (L_SHIPMODE = 'AIR' OR L_SHIPMODE = 'AIR REG') AND L_SHIPINSTRUCT = 'DELIVER IN PERSON'
) OR (P_BRAND ='Brand#43' AND
  (P_CONTAINER='MED BAG'OR P_CONTAINER='MED BOX'OR P_CONTAINER='MED PKG'OR P_CONTAINER='MED
```

```
      PACK')
  AND L_QUANTITY >=15 AND L_QUANTITY <= 25 AND P_SIZE<=10 AND
  (L_SHIPMODE = 'AIR' OR L_SHIPMODE = 'AIR REG') AND L_SHIPINSTRUCT = 'DELIVER IN PERSON'
) OR (P_BRAND = 'Brand#43' AND
  (P_CONTAINER = 'LG CASE' OR P_CONTAINER = 'LG BOX' OR P_CONTAINER = 'LG PACK' OR P_CONTAINER = 'LG
      PKG')
  AND L_QUANTITY >=26 AND L_QUANTITY <= 36 AND P_SIZE<=15 AND
  (L_SHIPMODE = 'AIR' OR L_SHIPMODE = 'AIR REG') AND L_SHIPINSTRUCT = 'DELIVER IN PERSON')
```

TPC-H Q20

```
SELECT S_NAME, S_ADDRESS
FROM SUPPLIER JOIN NATION ON S_NATIONKEY = N_NATIONKEY
  JOIN (
    SELECT SUM(0.5 * L_QUANTITY) AS TOTAL_L_QUANTITY
    FROM PART JOIN PARTSUPP ON P_PARTKEY = PS_PARTKEY
      JOIN LINEITEM ON PS_PARTKEY = L_PARTKEY AND PS_SUPPKEY = L_SUPPKEY
    WHERE L_SHIPDATE >= DATE '1996-01-01' AND L_SHIPDATE < DATE '1997-01-01' AND
        P_NAME LIKE 'azure%%'
    GROUP BY PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
    HAVING PS_AVAILQTY > TOTAL_L_QUANTITY
) AS TMP_VIEW ON S_SUPPKEY = PS_SUPPKEY
WHERE N_NAME = 'JORDAN'
ORDER BY S_NAME
```

TPC-H Q21

```
SELECT S_NAME, COUNT(*) AS NUMWAIT
FROM NATION JOIN SUPPLIER ON N_NATIONKEY = S_NATIONKEY
  JOIN LINEITEM L1 ON S_SUPPKEY = L_SUPPKEY
  LEFT SEMI JOIN LINEITEM L2 ON L1.L_ORDERKEY = L_ORDERKEY AND L1.L_SUPPKEY != L_SUPPKEY
  ANTI JOIN LINEITEM L3 ON L1.L_ORDERKEY = L_ORDERKEY AND L1.L_SUPPKEY != L_SUPPKEY
  JOIN ORDERS ON L1.L_ORDERKEY = O_ORDERKEY
WHERE N_NAME = 'MOROCCO' AND O_ORDERSTATUS = 'F' AND
    L1.L_RECEIPTDATE > L1.L_COMMITDATE AND L3.L_RECEIPTDATE > L3.L_COMMITDATE
GROUP BY S_NAME
ORDER BY NUMWAIT DESC, S_NAME
```

TPC-H Q22

```
SELECT SUBSTRING(C_PHONE,1,2) AS CNTRYCODE, COUNT(*) AS TOTAL, SUM(C_ACCTBAL) AS TOTACCTBAL
FROM (
  SELECT C_PHONE, C_ACCTBAL
  FROM CUSTOMER ANTI JOIN ORDERS ON C_CUSTKEY = O_CUSTKEY
  WHERE (
    C_PHONE LIKE '23%%' OR C_PHONE LIKE '29%%' OR C_PHONE LIKE '22%%' OR
    C_PHONE LIKE '20%%' OR C_PHONE LIKE '24%%' OR C_PHONE LIKE '26%%' OR C_PHONE LIKE '25%%'
  )
  HAVING C_ACCTBAL > (
    SELECT AVG(C_ACCTBAL) AS CNT
    FROM CUSTOMER
    WHERE C_ACCTBAL > 0.00 AND (
      C_PHONE LIKE '23%%' OR C_PHONE LIKE '29%%' OR C_PHONE LIKE '22%%' OR
      C_PHONE LIKE '20%%' OR C_PHONE LIKE '24%%' OR C_PHONE LIKE '26%%' OR C_PHONE LIKE '25%%')
  )
) AS TMP_VIEW
GROUP BY CNTRYCODE
ORDER BY CNTRYCODE
```

D Micro Benchmark Queries for Loop Fusion

In this section, we present the corresponding SQL queries for the micro benchmarks used in Section 4.7. These queries are presented in Table D.1. All these queries are using the LINEITEM and ORDERS tables of TPC-H.

Name	SQL Query
filter.count	**SELECT COUNT**(*) **FROM** LINEITEM **WHERE** L_SHIPDATE >= **DATE** '1995-12-01'
filter.sum	**SELECT SUM**(L_DISCOUNT * L_EXTENDEDPRICE) **FROM** LINEITEM **WHERE** L_SHIPDATE >= **DATE** '1995-12-01'
filter.filter.sum	**SELECT SUM**(L_DISCOUNT * L_EXTENDEDPRICE) **FROM** LINEITEM **WHERE** (L_SHIPDATE >= **DATE** '1995-12-01') **AND** (L_SHIPDATE < **DATE** '1997-01-01')
filter.filter.filter.sum	**SELECT SUM**(L_DISCOUNT * L_EXTENDEDPRICE) **FROM** LINEITEM **WHERE** (L_SHIPDATE >= **DATE** '1995-12-01') **AND** (L_SHIPDATE < **DATE** '1997-01-01') **AND** (L_SHIPMODE = 'MAIL')
filter.take.sum	**SELECT SUM**(L_DISCOUNT * L_EXTENDEDPRICE) **FROM** LINEITEM **WHERE** L_SHIPDATE >= **DATE** '1995-12-01' **LIMIT** 1000
filter.map.take	**SELECT** L_DISCOUNT * L_EXTENDEDPRICE **FROM** LINEITEM **WHERE** L_SHIPDATE >= **DATE** '1995-12-01' **LIMIT** 1000
take.sum	**SELECT SUM**(L_DISCOUNT * L_EXTENDEDPRICE) **FROM** LINEITEM **LIMIT** 1000
filter.XJoin(filter).sum	**SELECT SUM**(O_TOTALPRICE) **FROM** LINEITEM, ORDERS **WHERE** O_ORDERDATE >= **DATE** '1998-11-01' **AND** L_SHIPDATE >= **DATE** '1998-11-01' **AND** O_ORDERKEY = L_ORDERKEY

Table D.1 – SQL queries of microbenchmark queries.

E An of Example the Fusion Process

This section demonstrates the transformations applied for performing push and pull-based loop fusion on the working example. Note that, here, inlining a particular definition is always assumed together with β-reduction (inlining) of the accompanying function values. As an example, in Figure E.1, inlining `fold` means that we also inline the function value passed as the input parameter to `fold`.

```
val l1 = fromArray(R)
val l2 = l1.filter(r => r.A < 10)
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

(inline fromArray & filter)

```
val l1 = build { k1 =>
  var index = 0
  while(index < R.length) {
    k1(R(i))
  }
}
val l2 = build { k2 =>
  l1.foreach { e1 =>
    if(e1.A < 10)
      k2(e1)
  }
}
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

(fuse l1.foreach)

```
val l2 = build { k2 =>
  var index = 0
  while(index < R.length) {
    val e1 = R(i)
    if(e1.A < 10)
      k2(e1)
  }
}
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

(inline map)

```
val l2 = build { k2 =>
  var index = 0
  while(index < R.length) {
    val e1 = R(i)
    if(e1.A < 10)
      k2(e1)
  }
}
val l3 = build { k3 =>
  l2.foreach { e2 =>
    k3(e2.B)
  }
}
return l3.fold(0.0)((s, r) => s + r)
```

(fuse l2.foreach)

```
val l3 = build { k3 =>
  var index = 0
  while(index < R.length) {
    val e1 = R(i)
    if(e1.A < 10)
      k3(e1.B)
  }
}
return l3.fold(0.0)((s, r) => s + r)
```

(inline fold)

```
val l3 = build { k3 =>
  var index = 0
  while(index < R.length) {
    val e1 = R(i)
    if(e1.A < 10)
      k3(e1.B)
  }
}
var sum = 0.0
l3.foreach { e3 =>
  sum = sum + e3
}
return sum
```

(fuse l3.foreach)

```
var sum = 0.0
var index = 0
while(index < R.length) {
  val rec = R(index)
  index += 1
  if(rec.A < 10)
    sum += rec.B
}
return sum
```

Figure E.1 – Transformations needed for applying fold fusion on the example query.

```
val l1 = fromArray(R)
val l2 = l1.filter(r => r.A < 10)
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

(inline `fromArray` & `filter`)

```
var index = 0
val l1 = generate { () =>
 if(index < R.length) {
  val elem = R(index)
  index += 1
  elem
 } else { null }
}
val l2 = l1.destroy { n1 =>
 generate { () =>
  var elem: R = null
  do {
   elem = n1()
  } while (elem!=null && !(elem.A<10))
  elem
 }
}
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

(fuse `l1.destroy`)

```
var index = 0
val l2 = generate { () =>
 var elem: R = null
 do {
  if(index < R.length) {
   elem = R(index)
   index += 1
  } else { elem = null }
 } while (elem != null && !(elem.A < 10))
 elem
}
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

(inline `map`)

```
var index = 0
val l2 = generate { () =>
 var elem: R = null
 do {
  if(index < R.length) {
   elem = R(index)
   index += 1
  } else { elem = null }
 } while (elem != null && !(elem.A < 10))
 elem
}
val l3 = l2.destroy { n2 =>
 generate { () =>
  val elem = n2()
  if(elem == null) null
  else elem2.B
 }
}
return l3.fold(0.0)((s, r) => s + r)
```

(fuse `l2.destroy`)

```
var index = 0
val l3 = generate { () =>
 var elem: R = null
 do {
  if(index < R.length) {
   elem = R(index)
   index += 1
  } else { elem = null }
 } while (elem != null && !(elem.A < 10))
 if(elem == null) null
 else elem2.B
}
return l3.fold(0.0)((s, r) => s + r)
```

(inline `fold`)

```
var index = 0
val l3 = generate { () =>
 var elem: R = null
 do {
  if(index < R.length) {
   elem = R(index)
   index += 1
  } else { elem = null }
 } while (elem!=null && !(elem.A<10))
 if(elem == null) null
 else elem2.B
}
return l3.destroy { n3 =>
 var sum = 0.0
 while(true){
  val elem = n3()
  if(elem == null) break
  else sum = sum + elem
 }
 sum
}
```

(fuse `l3.destroy` & partial evaluation)

```
var sum = 0.0
var index = 0
while(true) {
 var elem: R = null
 do {
  if(index < R.length) {
   elem = R(index)
   index += 1
  } else { elem = null }
 } while (elem != null && !(elem.A < 10))
 if(elem == null) break
 else sum = sum + elem.B
}
return sum
```

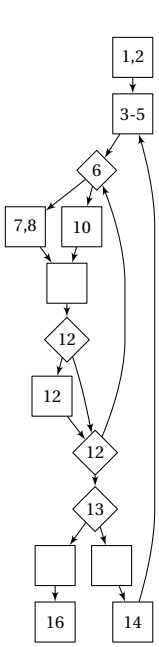
Figure E.2 – Transformations needed for applying unfold fusion on the example query.

F Impact of the Underlying Optimizing Compiler on Loop Fusion

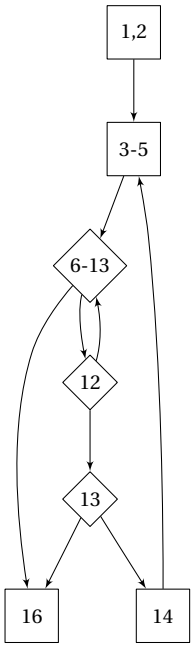
In Section 4.2.3, we have seen that the control flow of pull engines is more complicated than push engines. However, a careful examination of the optimized machine code generated by the CLang compiler shows that the optimizing compiler partially compensates this limitation of pull engines.

Figure F.1 shows the CFG of the specialized pull-engine for the `filter.map.sum` query. These graphs are obtained by using the `opt -dot-cfg` command for the generated LLVM code, which is generated by compiling C code using `clang -emit-llvm`.

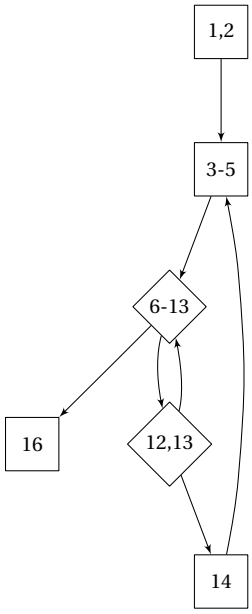
The CFG of the generated machine code when one does not use any optimization is shown in Figure F.1a. This CFG is as complicated as the one shown in Section 4.2.3. Figure F.1b shows the CFG of the generated machine code after performing the following two optimizations, which are both enabled by using the `-O1` and `-O3` optimization flags. First, the `-mem2reg` optimization is responsible for promoting memory references to register references. Second, the `-simplifycfg` is responsible for simplifying the CFG. The generated machine code has a much more simplified CFG than the machine code without any optimizations. Finally, Figure F.1c shows the CFG of the generated machine code by using the `-O1` or `-O3` optimization flags. We observed that adding the `-jump-threading` optimization flag, which is responsible for further simplification of CFG, to the existing set of optimization flags (`-mem2reg -simplifycfg`) achieves a similar CFG. This CFG is as simple as the CFG of the specialized push engine presented in Section 4.2.3.



(a) Without any optimization flags.



(b) With the memory to reference promotion and CFG simplification optimizations.



(c) With the most aggressive optimization flags.

Figure F.1 – Control flow graph of the specialized pull-based engine for the `filter.sum` query, compiled with different optimization flags in the CLang compiler.

G Loop Fusion for the Limit and Merge Join Operators

In this chapter, we investigate the loop fusion process for two problematic operators for push-based engines. First, in Section G.1 we show the translation of the limit operator. Then, in Section G.2 we give more details on the translation of the merge join operator.

G.1 Translating the Limit Operator

In this section, we show the translation of the limit operator in pull and push-based engines. The implementation of the limit operator for a pull-based engine is presented in Figure 4.4a. If the limit threshold was not reached, the limit operator returns the next element of its source operator. Otherwise, if the limit threshold was reached, the limit operator produces a `null` value, specifying that the end of stream is reached. However, in a push-based engine there is no straightforward way for the destination operator to send a signal to the source operator specifying that the limit was reached. Hence, the limit operator is implemented by not passing the element to the destination operator if the limit was reached. This means that the source operator continues producing more elements, even though these elements will be ignored by the subsequent operators (c.f. Figure G.1a).

Consider a query similar to the `take.sum` presented in Section 4.7. This query for a given collection of numbers (array of a thousand integers), returns the sum of the first five elements. The corresponding C code for pull and push-based engines can be found in Figure G.2a and Figure G.2b, respectively. In a pull-based engine, when the fifth element is reached, no further element is processed thanks to the `break` expression in line 10 of Figure G.2a. However, a push-based engine ignores the elements after the fifth element, without early termination of the loop, as it can be observed in Figure G.2b.

One could argue that a smart enough compiler can compensate the mentioned limitation of a push-based engine. However, examining the generated assembly code by CLang 3.9.1, when the `-O3` optimization flag is used, shows that this claim is not necessarily true. Figure G.2c demonstrates the generated assembly code for a pull-based engine. This figure shows that the

```
class LimitOp[R](n: Int) {
  var i = 0
  def consume(e: R): Unit =
    if(i < n) {
      dest.consume(e)
      i += 1
    }
}
```

(a) Push-based query engine.

```
class QueryMonad[R] {
  def take(n: Int) = build { k =>
    var i = 0
    for(e <- this)
      if(i < n) {
        k(e)
        i += 1
      }
} }
```

(b) Fold fusion of collections.

Figure G.1 – Push-based query engine and fold fusion of collections for the Limit operator.

optimizing compiler successfully unrolled the loop to process the sum of the first elements in five assembly instructions. However, the generated assembly code for a push-based engine is not as elegant as the one for a pull-based engine. The generated assembly code processes the elements of the array two-by-two, however it does not perform the early termination that is happening in a pull-based engine. The 12^{th} line of Figure G.2d corresponds to the 7^{th} line of Figure G.2b, which continues iterating the main loop, until *all* elements of the array have been processed (without terminating it early).

G.2 Translating the Merge Join Operator

In this section, we investigate in more detail the merge join operator in pull and push-based engines. The implementation of this operator for these engines is given in Figure G.3.

Figure G.3a presents the implementation of the merge join operator in a pull-based engine. The elements of both relations are iterated in parallel until either the elements of both relations can be joined or one of the relations reaches the end. The local variables `leftProceed` and `rightProceed` are introduced in order to have only one invocation for the `next` method of the source operators (c.f. lines 10 and 11 of Figure G.3a). This is in essence similar to the trick we used in the inline-friendly implementation of the Selection operator.

In a push-based engine, as opposed to a pull-based engine, one has to materialize the left relation. This is because there is no way for the destination operator to control which source operator should produce the next element. Hence, the merge join operator materializes the elements of the left source operator, when it is consuming those elements (c.f. line 8 of Figure G.3b). This way, the merge join operator can control how to consume the (materialized) elements of the left source operator.

Consider the following query which is similar to the `filter.mJoin(filter).sum` query presented in Section 4.7:

```
SELECT SUM(R.B * S.B) FROM R, S WHERE R.B > 10 AND R.A = S.A
```

The corresponding generated code for pull and push-based engines is presented in Figure G.4. In a pull-based engine the elements of each relation are processed on the fly, without materi-

```
 1 int mapTake(int* arr) {
 2    const int N = 1000;
 3    int res = 0;
 4    int cnt = 0;
 5    int i = 0;
 6    while(1) {
 7       if(i < N) {
 8         int rec = 0;
 9         if(cnt < 5) rec = arr[i];
10         else break;
11         res += rec;
12         cnt++;
13         i++;
14       } else break;
15    }
16    return res;
17 }
```

(a) C code for pull engine.

```
 1 int mapTake(int* arr) {
 2    const int N = 1000;
 3    int res = 0;
 4    int cnt = 0;
 5    int i = 0;
 6    while(i < N) {
 7       if(cnt < 5) {
 8          res += arr[i];
 9          cnt ++;
10       }
11       i++;
12    }
13    return res;
14 }
```

(b) C code for push engine.

```
1 mapTake(int*): # @mapTake(int*)
2    mov eax, dword ptr [rdi + 4]
3    add eax, dword ptr [rdi]
4    add eax, dword ptr [rdi + 8]
5    add eax, dword ptr [rdi + 12]
6    add eax, dword ptr [rdi + 16]
7    ret
```

(c) Generated assembly code for pull engine.

```
 1 mapTake(int*): # @mapTake(int*)
 2    xor ecx, ecx
 3    mov edx, 1
 4    xor eax, eax
 5 .LBB0_1: # =>This Inner Loop Header
 6    cmp ecx, 4
 7    jg .LBB0_3
 8    add eax, dword ptr [rdi+4*rdx-4]
 9    inc ecx
10 .LBB0_3: # in Loop: Header=BB0_1
11    cmp ecx, 5
12    jge .LBB0_5
13    add eax, dword ptr [rdi+4*rdx]
14    inc ecx
15 .LBB0_5: # in Loop: Header=BB0_1
16    add rdx, 2
17    cmp rdx, 1001
18    jne .LBB0_1
19    ret
```

(d) Generated assembly code for push engine.

Figure G.2 – The generated C and assembly code for a simple query which returns the sum of the first five elements of an array of a thousand elements in pull and push-based engines.

alizing the elements of any of the two relations (c.f. Figure G.4a). However, in a push-based engine the filtered elements of the left relation are materialized into an intermediate collection (c.f. lines 6-11 of Figure G.4b). The creation of this intermediate collection justifies the performance gap observed in Section 4.7 for the micro benchmark of the merge join operator and TPCH query 12.

```
1 class MergeJoinOp[R, S]
2   (cond: (R, S) => Int) {
3   var rec1: R = null
4   var rec2: S = null
5   var leftProceed = true
6   var rightProceed = true
7   def next(): (R, S) = {
8     var elem: (R, S) = null
9     while(true) {
10      if(leftProceed) rec1 = left.next()
11      if(rightProceed) rec2 = right.next()
12      if(rec1 != null && rec2 != null) {
13        leftProceed = cond(rec1, rec2) < 0
14        rightProceed = !leftProceed
15        if(cond(rec1, rec2) == 0) {
16          elem = rec1.concat(rec2)
17          rightProceed = true
18          break
19      } } else
20        break
21    }
22    return elem
23 } }
```

(a) Pull-based query engine for Merge Operator.

```
class MergeJoinOp[R, S]
  (cond: (R, S) => Int) {

  val leftBuf = new ArrayBuffer[R]()
  var leftIndex = 0

  def consumeLeft(e: R): Unit = {
    leftBuf += e
  }

  def consumeRight(e: S): Unit = {
    while(leftIndex < leftBuf.length &&
       cond(leftRel(leftIndex), e)<0) {
      leftIndex += 1
    }
    if(leftIndex < leftBuf.length &&
       cond(leftBuf(leftIndex), e) == 0) {
      val res = leftBuf(leftIndex).concat(e)
      dest.consume(res)
    }
  }
}
```

(b) Push-based query engine for Merge Operator

Figure G.3 – Pull and push-based query engines for Merge Join operator.

```
var sum = 0.0
var i1 = 0; var i2 = 0
var rec1 = null; var rec2 = null
var leftProceed = true
var rightProceed = true
while(true) {
  if(leftProceed) {
    do {
      if(i1 < R.length) {
        rec1 = R(i1)
        i1 += 1
      } else {
        rec1 = null
        break
    } } while (rec1.B <= 10)
  }
  if(rightProceed) {
    if(i2 < S.length) {
      rec2 = S(i2)
      i2 += 1
    } else
      rec2 = null
  }
  if(rec1 != null && rec2 != null) {
    leftProceed = rec1.A < rec2.A
    rightProceed = !leftProceed
    if(rec1.A == rec2.A)
      sum += rec1.B * rec2.B
  } else {
    break
} }
return sum
```

(a) Inlined query in pull engine.

```
var sum = 0.0
var i1 = 0
var i2 = 0
val leftBuf = new ArrayBuffer[R]()
var leftIndex = 0


// Materialize an intermediate collection
while(i1 < R.length) {
  if(R(i1).B > 10) {
    RB += R(i1)
  }
  i1 += 1
}


// Use the intermediate collection
while(i2 < S.length) {
  val rec2 = S(i2)

  while(leftIndex < leftBuf.length &&
        leftBuf(leftIndex).A < rec2.A) {
    leftIndex += 1
  }

  if(leftIndex < RB.length &&
     leftBuf(leftIndex).A == rec2.A)
    sum += leftBuf(leftIndex).B * rec2.B
  i2 += 1
}

return sum
```

(b) Inlined query in push engine.

Figure G.4 – Complied version of a query with a merge join operator in pull and push engines. Note that both versions are derived after several optimization passes.

Bibliography

[1] D. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[2] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.

[3] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD'08, pages 967–980, New York, NY, USA, 2008. ACM.

[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 466–475. IEEE, 2007.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[6] U. Acar, G. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *TOPLAS*, 32(1), 2009.

[7] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded DSL for high performance big data processing. In *Int'l Workshop on End-to-end Mgmt of Big Data*, 2012.

[8] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski. Bundle adjustment in the large. In *European conference on computer vision*, pages 29–42. Springer, 2010.

[9] E. Agichtein, E. Brill, and S. Dumais. Improving web search ranking by incorporating user behavior information. In *SIGIR*, 2006.

[10] Y. Ahmad and C. Koch. DBToaster: A SQL Compiler for High-performance Delta Processing in Main-Memory Databases. *Proc. VLDB Endow.*, 2(2):1566–1569, Aug. 2009.

[11] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, volume 2. Addison-Wesley Reading, 2007.

[12] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB'01, pages 169–180, 2001.

[13] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[14] F. Allen and J. Cocke. *A Catalogue of Optimizing Transformations*. 1971.

[15] J. Anker and J. Svenningsson. An EDSL approach to high performance haskell programming. In *ACM Haskell Symposium*, pages 1–12, 2013.

[16] A. W. Appel. SSA is functional programming. *SIGPLAN notices*, 33(4):17–20, 1998.

[17] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[18] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The Stanford data stream management system. Technical report, 2004.

[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD'15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[20] K. Asai, H. Masuhara, and A. Yonezawa. Partial evaluation of call-by-value λ-calculus with side-effects. PEPM '97, pages 12–21. ACM, 1997.

[21] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 169–178. IEEE, 2010.

[22] R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query recommendation using query logs in search engines. In *EDBT*, 2004.

[23] P. Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, University of California, Berkeley, USA, 2015.

[24] H. G. Baker. Lively linear lisp: 'look ma, no garbage!'. *ACM SIGPLAN notices*, 27(8):89–98, 1992.

[25] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A Tracing JIT for a Functional Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 22–34. ACM, 2015.

[26] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. In *SIGMOD*, 1998.

[27] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015.

[28] A. G. Baydin, B. A. Pearlmutter, and J. M. Siskind. Diffsharp: Automatic differentiation library. *arXiv preprint arXiv:1511.07727*, 2015.

[29] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *ICDT '90*, volume 470 of *Lecture Notes in Computer Science*, pages 72–88. 1990.

[30] J. Berger. *Statistical decision theory and Bayesian analysis*. Springer, 1985.

[31] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[32] P. Berkhin. A survey on PageRank computing. *Internet Mathematics*, 2(1), 2005.

[33] M. W. Berry, M. Browne, A. N. Langville, V. P. Pauca, and R. J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. In *Computational Statistics and Data Analysis*, 2006.

[34] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. Streams à la carte: Extensible Pipelines with Object Algebras. In *29th European Conference on Object-Oriented Programming*, page 591, 2015.

[35] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores. In *SIGMOD'09*, pages 283–296. ACM, 2009.

[36] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to Von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'96, pages 171–183, NY, USA, 1996. ACM.

[37] C. Bischof, P. Khademi, A. Mauer, and A. Carle. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32, 1996.

[38] C. H. Bischof, H. Bucker, B. Lang, A. Rasch, and A. Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pages 65–72. IEEE, 2002.

[39] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

[40] C. Böhm and A. Berarducci. Automatic Synthesis of Typed λ-Programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[41] P. Boncz, T. Neumann, and O. Erling. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*, pages 61–76. Springer International Publishing, Cham, 2014.

[42] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237, 2005.

[43] E. Book, D. V. Shorre, and S. J. Sherman. The cwic/36o system, a compiler for writing and implementing compilers. *SIGPLAN Notices*, 5(6):11–29, June 1970.

[44] V. Breazu-Tannen, P. Buneman, and L. Wong. *Naturally Embedded Query Languages*. Springer, 1992.

[45] V. Breazu-Tannen and R. Subrahmanyam. *Logical and Computational Aspects of Programming with Sets/Bags/Lists*. Springer, 1991.

[46] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1), 2010.

[47] P. Buchlovsky and H. Thielecke. A Type-theoretic Reconstruction of the Visitor Pattern. *Electronic Notes in Theoretical Computer Science*, 155:309 – 329, 2006.

[48] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, page 1094342011403516, 2011.

[49] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, Sept. 1995.

[50] J. Carette, O. Kiselyov, and C.-C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[51] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. *SIGPLAN Notices*, 45(10):835–847, Oct. 2010.

[52] M. M. Chakravarty, G. Keller, and P. Zadarnowski. A functional perspective on SSA optimisation algorithms. *Electronic Notes in Theoretical Computer Science*, 82(2):347–361, 2004.

[53] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. King III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A History and Evaluation of System R. *CACM*, 24(10):632–646, 1981.

[54] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, ICDE'11, pages 375–386. IEEE, 2011.

[55] Y. Chen, J. Dunfield, and U. Acar. Type-directed automatic incrementalization. In *PLDI*, 2012.

[56] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR'13*, 2013.

[57] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, July 2012.

[58] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. *IEEE Data Eng. Bull.*, 37(1):48–59, 2014.

[59] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'13, pages 3–14, New York, NY, USA, 2013. ACM.

[60] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *PVLDB*, 1(2):1313–1324, Aug. 2008.

[61] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4), 2012.

[62] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A Parallel DSL for Image Analysis and Visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'12, pages 111–120. ACM, 2012.

[63] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. *Acm SIGPLAN Notices*, 34(10):1–19, 1999.

[64] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. VLDB'07, pages 339–350. ACM, 2007.

[65] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. Theory of privacy and anonymity. In M. Atallah and M. Blanton, editors, *Algorithms and Theory of Computation Handbook (2nd edition)*. CRC Press, 2009.

[66] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, pages 21–30, NY, USA, 2012. ACM.

[67] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *TOPLAS*, 17(2):181–196, Mar. 1995.

[68] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Eng.*, 11(4):29–41, 2009.

[69] P. Cousot. Types as abstract interpretations, invited paper. POPL'97, pages 316–331, Paris, France, 1997. ACM Press, New York, NY.

[70] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. POPL'77, pages 238–252, New York, NY, USA, 1977. ACM.

[71] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP*, pages 315–326, New York, NY, USA, 2007. ACM.

[72] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An Architecture for Compiling UDF-centric Workflows. *PVLDB*, 8(12):1466–1477, 2015.

[73] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, abs/1406.6667, 2014.

[74] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*, 2015.

[75] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault. MPADS: Memory-Pooling-Assisted Data Splitting. In *Proceedings of the 7th International Symposium on Memory Management*, pages 101–110. ACM, 2008.

[76] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[77] A. Danial. Cloc–count lines of code. http://cloc.sourceforge.net/., 2018. Accessed: 2018-07-03.

[78] N. A. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'08, pages 133–144, New York, NY, USA, 2008. ACM.

[79] O. Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*, POPL'96, pages 242–257. ACM, 1996.

[80] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175 – 1193, 2000.

[81] S. Das, Y. Sismanis, K. Beyer, R. Gemulla, P. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.

[82] M. Dashti, S. Basil John, A. Shaikhha, and C. Koch. Transaction repair for multi-version concurrency control. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD'17, pages 235–250, New York, NY, USA, 2017. ACM.

[83] M. Dashti, S. B. John, T. Coppey, A. Shaikhha, V. Jovanovic, and C. Koch. Compiling database application programs. *CoRR*, abs/1807.09887, 2018.

[84] F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-based optimization on graphs with application to library performance tuning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 729–736. ACM, 2009.

[85] J. Dees and P. Sanders. Efficient many-core query execution in main memory column-stores. In *ICDE'13*, pages 350–361, April 2013.

[86] C. Demetrescu. Fully dynamic algorithms for path problems on directed graphs, 2001.

[87] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the o(n/sup 2/) barrier. FOCS, 2000.

[88] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *ACM*, 51(6), Nov. 2004.

[89] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013.

[90] Z. DeVito, M. Mara, M. Zollhöfer, G. Bernstein, J. Ragan-Kelley, C. Theobalt, P. Hanrahan, M. Fisher, and M. Nießner. Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *arXiv preprint*, 2016.

[91] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 11. ACM, 2014.

[92] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 1243–1254, New York, NY, USA, 2013. ACM.

[93] S. Doeraene and T. Schlatter. Parallel incremental whole-program optimizations for scala.js. *Acm SIGPLAN Notices*, 51(10):59–73, 2016.

[94] S. Dolan. Fun with semirings: a functional pearl on the abuse of linear algebra. In *ACM SIGPLAN Notices*, volume 48, pages 101–110. ACM, 2013.

[95] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *TOMS*, 16(1), 1990.

[96] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *HPDC*, 2010.

[97] K. El Gebaly and J. Lin. In-Browser Interactive SQL Analytics with Afterburner. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1623–1626. ACM, 2017.

[98] C. M. Elliott. Beautiful differentiation. In *ACM SIGPLAN Notices*, volume 44, pages 191–202. ACM, 2009.

[99] B. Emir, M. Odersky, and J. Williams. Matching Objects with Patterns. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 273–298, Berlin, Heidelberg, 2007. Springer-Verlag.

[100] EPFL DATA Laboratory. SC - Systems Compiler. http://data.epfl.ch/sc, 2018.

[101] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'11, pages 391–406, New York, NY, USA, 2011. ACM.

[102] S. Erdweg and F. Rieger. A framework for extensible languages. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE'13, pages 3–12, New York, NY, USA, 2013. ACM.

[103] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[104] D. Fabregat-Traver and P. Bientinesi. A domain-specific compiler for linear algebra operations. *CoRR*, abs/1205.5975, 2012.

[105] R. E. Faith, L. S. Nyland, and J. F. Prins. Khepera: A system for rapid implementation of domain specific languages. In *Proceedings of the 1997 Conference on Domain-Specific Languages*, volume 97 of *DSL' 97*, pages 19–19, Berkeley, CA, USA, 1997. USENIX Association.

[106] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database – data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2012.

[107] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *TODS*, 25(4):457–516, Dec. 2000.

[108] M. Felleisen. On the expressive power of programming languages. In *ESOP'90*, pages 134–151. Springer, 1990.

[109] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, July 1987.

[110] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *ACM SIGPLAN Notices*, volume 28, pages 237–247. ACM, 1993.

[111] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In *European Symposium on Programming*, ESOP '06, pages 7–21. Springer, 2006.

[112] S. A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):195–222, 2006.

[113] M. Fowler. Language workbenches: The killer-app for domain specific languages. 2005. Accessed: 2018-07-03.

[114] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'05, pages 315–326.

[115] D. Friedman and S. Wise. Unwinding stylized recursions into iterations. *Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep*, 19, 1975.

[116] Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[117] B. Gedik, H. Andrade, K.-L. Wu, P. Yu, and M. Doo. SPADE: the System S Declarative Stream Processing Engine. In *SIGMOD*, 2008.

[118] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

[119] J. Gibbons and B. Oliveira. The Essence of the Iterator Pattern. *Journal of Functional Programming*, 19(3-4):377–402, 2009.

[120] J. R. Gilbert, S. Reinhardt, and V. B. Shah. High-performance graph algorithms from parallel sparse matrices. In *PARA*, 2007.

[121] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[122] A. J. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.

[123] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: An architecture for scale-out of olxp workloads. *PVLDB*, 8(12):1716–1727, Aug. 2015.

[124] A. Goldberg and R. Paige. Stream processing. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 53–62, New York, NY, USA, 1984. ACM.

[125] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *ACM SIGMOD Record*, volume 30, pages 331–342. ACM, 2001.

[126] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[127] G. Graefe. Query Evaluation Techniques for Large Databases. *CSUR*, 25(2):73–169, June 1993.

[128] G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, Feb 1994.

[129] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[130] R. Greer. Daytona And The Fourth-Generation Language Cymbal. In *the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD'99, pages 525–526. ACM, 1999.

[131] C. Grelck and S.-B. Scholz. SAC—A functional array language for efficient multi-threaded execution. *Int. Journal of Parallel Programming*, 34(4):383–427, 2006.

[132] J. L. Gross, J. Yellen, and P. Zhang. *Handbook of Graph Theory, Second Edition*. 2nd edition, 2013.

[133] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI'02, pages 282–293, NY, USA, 2002. ACM.

[134] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY – Database-Supported Program Execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD'09, pages 1063–1066, New York, NY, USA, 2009. ACM.

[135] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-Safe LINQ Compilation. *PVLDB*, 3(1-2):162–172, Sept. 2010.

[136] T. Grust and M. Scholl. How to Comprehend Queries Functionally. *Journal of Intelligent Information Systems*, 12(2-3):191–218, 1999.

[137] G. Guennebaud, B. Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[138] A. Gupta and I. Mumick. *Materialized Views*. MIT Press, 1999.

[139] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proc. VLDB Endow.*, 1(1):1107–1123, Aug. 2008.

[140] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. *ACM SIGPLAN Notices*, 35(1):39–52, 2000.

[141] N. Hallenberg, M. Elsman, and M. Tofte. Combining Region Inference and Garbage Collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI'02, pages 141–152, NY, USA, 2002. ACM.

[142] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, VLDB'06, pages 487–498. VLDB Endowment, 2006.

[143] L. Hascoet and V. Pascual. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, May 2013.

[144] J. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12), 2012.

[145] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a Database System. *Foundations and Trends® in Databases*, 1(2):141–259, 2007.

[146] T. Henriksen, M. Elsman, and C. E. Oancea. Size Slicing: A hybrid approach to size inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 31–42, New York, NY, USA, 2014. ACM.

[147] T. Henriksen and C. E. Oancea. Bounds checking: An instance of hybrid analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY '14, NY, USA, 2014. ACM.

[148] T. Henriksen, N. G. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571. ACM, 2017.

[149] R. Hinze, T. Harper, and D. W. H. James. Theory and Practice of Fusion. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 19–37, Berlin, Heidelberg, 2011. Springer-Verlag.

[150] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, Mar. 2014.

[151] F. Hivert and N. Thiéry. MuPAD-Combinat, an open-source package for research in algebraic combinatorics. *Sém. Lothar. Combin*, 51:70, 2004.

[152] C. Hofer and K. Ostermann. Modular Domain-specific Language Components in Scala. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE'10, pages 83–92, New York, NY, USA, 2010. ACM.

[153] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*, pages 137–148. ACM, 2008.

[154] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'03, pages 185–197, New York, NY, USA, 2003. ACM.

[155] R. J. Hogan. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, July 2014.

[156] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *ACM*, 48(4), July 2001.

[157] Z.-K. Huang and K.-W. Chau. A new image thresholding method based on gaussian mixture model. *Applied Mathematics and Computation*, 205(2):899–907, 2008.

[158] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), Dec. 1996.

[159] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A domain-specific language for building self-optimizing ast interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 123–132, New York, NY, USA, 2014. ACM.

[160] R. Hundt. Loop Recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011*, 2011.

[161] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[162] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten, et al. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[163] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *ACM Sigmod Record*, volume 19, pages 312–321. ACM, 1990.

[164] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[165] K. E. Iverson. A Programming Language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 345–351. ACM, 1962.

[166] C. B. Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2(3):307–315, 1999.

[167] C. B. Jay and M. Sekanina. Shape checking of array programs. Technical report, In Computing: the Australasian Theory Seminar, Proceedings, 1997.

[168] JetBrains. Meta programming system. http://www.jetbrains.com/mps, 2018. Accessed: 2018-07-03.

[169] H.-J. Z. Ji-rong Wen, Jian-Yun Nie and. Query clustering using user logs. *ACM Transactions on Information Systems*, 20(1), 2002.

[170] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3hopp: A high-compression indexing scheme for reachability query. SIGMOD'09, pages 813–826, 2009.

[171] S. C. Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[172] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Peter Sestoft, 1993.

[173] S. Jones. Compiling haskell by program transformation: A report from the trenches. In H. Nielson, editor, *Programming Languages and Systems - ESOP '96*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44. Springer Berlin Heidelberg, 1996.

[174] S. P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93. Citeseer, 1993.

[175] M. Jonnalagedda and S. Stucki. Fold-based Fusion As a Library: A Generative Programming Pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pages 41–50. ACM, 2015.

[176] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'10, pages 223–236, New York, NY, USA, 2010. ACM.

[177] V. Jovanović, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky. Yin-Yang: Concealing the Deep Embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 73–82. ACM, 2014.

[178] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2):1496–1499, 2008.

[179] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[180] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, Jan. 1976.

[181] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. SIGMOD'11, pages 841–852, New York, NY, USA, 2011. ACM.

[182] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2):175 – 193, 1984.

[183] J. Karczmarczuk. Functional differentiation of computer programs. *ACM SIGPLAN Notices*, 34(1):195–203, 1999.

[184] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *Proceedings of the VLDB Endowment*, 9(12):972–983, 2016.

[185] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR*, 2015.

[186] L. C. L. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM SIGPLAN Notices*, volume 45, pages 444–463. ACM, 2010.

[187] G. Kedem. Automatic differentiation of computer programs. *ACM Trans. Math. Softw.*, 6(2):150–165, June 1980.

[188] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Notices*, volume 30, pages 13–22. ACM, 1995.

[189] A. Kennedy. Compiling with continuations, continued. In *ACM SIGPLAN Notices*, volume 42, pages 177–190, 2007.

[190] K. Kennedy. *A survey of data flow analysis techniques.* IBM Thomas J. Watson Research Division, 1979.

[191] K. Kennedy, B. Broom, A. Chauhan, R. J. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387–408, 2005.

[192] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320. Springer Berlin Heidelberg, 1994.

[193] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[194] K. A. Khan and P. I. Barton. A vector forward mode of automatic differentiation for generalized derivative evaluation. *Optimization Methods and Software*, 30(6):1185–1212, 2015.

[195] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst.*, 7(3):443–469, Sept. 1982.

[196] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS, 1999.

[197] O. Kiselyov, A. Biboudis, N. Palladinos, and Y. Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 285–299, New York, NY, USA, 2017. ACM.

[198] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *ACM SIGPLAN Notices*, volume 44, pages 1–12. ACM, 2008.

[199] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[200] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Errata for "Building Efficient Query Engines in a High-level Language": PVLDB 7(10):853-864. *PVLDB*, 7(13):1784–1784, Aug. 2014.

[201] Y. Klonatos, A. Nötzli, A. Spielmann, C. Koch, and V. Kuncak. Automatic Synthesis of Out-of-core Algorithms. In *ACM SIGMOD*, SIGMOD'13, pages 133–144. ACM, 2013.

[202] C. Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 87–98, New York, NY, USA, 2010. ACM.

[203] C. Koch. Abstraction without regret in data management systems. In *CIDR*, 2013.

[204] C. Koch. Abstraction without regret in database systems building: a manifesto. *IEEE Data Eng. Bull.*, 37(1):70–79, 2014.

[205] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *VLDBJ*, 23(2), 2014.

[206] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. Javascript as an embedded DSL. In *ECOOP 2012–Object-Oriented Programming*, pages 409–434. Springer, 2012.

[207] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. Franklin, and M. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, 2013.

[208] K. Krikellas, S. Viglas, and M. Cintra. Generating Code for Holistic Query Evaluation. In *ICDE*, ICDE '10, pages 613–624, Washington, DC, USA, March 2010. IEEE Computer Society.

[209] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[210] R. Lämmel and S. P. Jones. Scrap your boilerplate with class: Extensible generic functions. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP'05, pages 204–215, New York, NY, USA, 2005. ACM.

[211] A. Langville and C. Meyer. Deeper inside PageRank. *Internet Mathematics*, 1(3), 2004.

[212] P. Larson, M. Zwilling, and K. Farlee. The hekaton memory-optimized OLTP engine. *IEEE Data Eng. Bull.*, 36(2):34–40, 2013.

[213] J. R. Larus. *Restructuring symbolic programs for concurrent execution on multiprocessors*. PhD thesis, 1989.

[214] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society.

[215] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, Sept. 2011.

[216] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. SIGMOD'14, pages 743–754, New York, NY, USA, 2014. ACM.

[217] D. Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *FOCS*, pages 460–469, Nov 1983.

[218] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *ACM SIGPLAN Notices*, volume 37, pages 270–282. ACM, 2002.

[219] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[220] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, 2(2):164–168, 1944.

[221] Z. Li and K. A. Ross. Fast Joins Using Join Indices. *The VLDB Journal*, 8(1):1–24, 1999.

[222] L. Libkin. Expressive Power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, Mar. 2003.

[223] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th international conference on World wide web*, pages 681–690. ACM, 2010.

[224] D. H. Lorenz and B. Rosenan. Cedalion: A language for language oriented programming. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'11, pages 733–752, New York, NY, USA, 2011. ACM.

[225] R. A. Lorie. *XRM: An extended (N-ary) relational memory*. IBM, 1974.

[226] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[227] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[228] D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[229] G. Mainland, R. Leshchinskiy, and S. Peyton Jones. Exploiting Vector Instructions with Generalized Stream Fusion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP'13, pages 37–48, New York, NY, USA, 2013. ACM.

[230] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[231] S. Manegold, M. L. Kersten, and P. Boncz. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2):1648–1653, 2009.

[232] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic query transformations for dynamic web applications. ICDE'09, pages 1175–1178. IEEE, 2009.

[233] A. Marathe and K. Salem. Query processing techniques for arrays. *VLDBJ*, 11(1), 2002.

[234] D. W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[235] A. McKellar and E. G. Coffman Jr. Organizing matrices and matrix operations for paged memory systems. *Communications of the ACM*, 12(3):153–165, 1969.

[236] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[237] M. Mehta and D. J. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB'95, pages 382–394, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[238] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD'06, pages 706–706. ACM, 2006.

[239] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1), 2016.

[240] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, Sept. 2017.

[241] S. Mihaylov, Z. Ives, and S. Guha. REX: Recursive, delta-based data-centric computation. *PVLDB*, 5(11), 2012.

[242] Y. Minamide. A functional representation of data structures with a hole. In *In Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98*, POPL'98, pages 75–84, 1998.

[243] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[244] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11), 2011.

[245] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, 2005.

[246] M. Montenegro, R. Peña, and C. Segura. A type system for safe memory management and its proof of correctness. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '08, pages 152–162. ACM, 2008.

[247] M. Montenegro, R. Peña, and C. Segura. A simple region inference algorithm for a first-order functional language. In *International Workshop on Functional and Constraint Logic Programming*, pages 145–161. Springer, 2009.

[248] J. J. Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[249] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[250] T. Müller and T. Grust. Provenance for sql through abstract interpretation: Value-less, but worthwhile. *PVLDB*, 8(12):1872–1875, Aug. 2015.

[251] D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic Optimization of Declarative Queries. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, pages 121–131, New York, NY, USA, 2011. ACM.

[252] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. *Proc. VLDB Endow.*, 7(12):1095–1106, Aug. 2014.

[253] S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: Quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2016, pages 25–36, New York, NY, USA, 2016. ACM.

[254] S. H. K. Narayanan, B. Norris, and B. Winnicka. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science*, 1(1):1845–1853, 2010.

[255] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[256] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[257] F. Nielsen and K. Sun. Guaranteed bounds on information-theoretic measures of univariate mixtures using piecewise log-sum-exp inequalities. *Entropy*, 18(12):442, 2016.

[258] M. Nikolic, M. ElSeidy, and C. Koch. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *SIGMOD*, 2014.

[259] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.

[260] M. Odersky and M. Zenger. Scalable Component Abstractions. In *OOPSLA*, OOPSLA'05, pages 41–57, New York, NY, USA, 2005. ACM.

[261] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE'13, pages 125–134, New York, NY, USA, 2013. ACM.

[262] Oracle Corporation. TimesTen In-Memory Database Architectural Overview, 2006. http://download.oracle.com/otn_hosted_doc/timesten/603/TimesTen-Documentation/arch.pdf.

[263] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, pages 567–574, 2001.

[264] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[265] V. Pankratius, F. Schmidt, and G. Garreton. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In *ICSE 2012*, pages 123–133.

[266] J. Paredaens and D. V. Gucht. Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 21-23, 1988, Austin, Texas, USA*, pages 29–38, 1988.

[267] Y. Park, S. Seo, H. Park, H. K. Cho, and S. Mahlke. SIMD Defragmenter: Efficient ILP Realization on Data-Parallel Architectures. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 363–374. ACM, 2012.

[268] L. Parreaux, A. Shaikhha, and C. E. Koch. Quoted staged rewriting: A practical approach to library-defined optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017, pages 131–145, New York, NY, USA, 2017. ACM.

[269] L. Parreaux, A. Shaikhha, and C. E. Koch. Squid: Type-safe, hygienic, and reusable quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 56–66, New York, NY, USA, 2017. ACM.

[270] L. Parreaux, A. Voizard, A. Shaikhha, and C. E. Koch. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.*, 2(POPL):13:1–13:33, Dec. 2017.

[271] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, et al. Self-driving database management systems. In *CIDR'17*, 2017.

[272] B. A. Pearlmutter and J. M. Siskind. Lazy multivariate higher-order forward-mode ad. In *ACM SIGPLAN Notices*, volume 42, pages 155–160. ACM, 2007.

[273] B. A. Pearlmutter and J. M. Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

[274] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 2. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[275] B. C. Pierce. *Types and Programming Languages*. MIT press, 2002.

[276] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD'15, pages 1493–1508, New York, NY, USA, 2015. ACM.

[277] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[278] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'13, pages 519–530, New York, NY, USA, 2013. ACM.

[279] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 133–144. ACM, 2012.

[280] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, 2nd edition, 2000.

[281] G. Ramalingam. The undecidability of aliasing. *TOPLAS*, 16(5):1467–1471, Sept. 1994.

[282] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, ICDE '08, pages 60–69, 2008.

[283] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, ICDE '06, pages 23–34, Washington, DC, USA, 2006. IEEE Computer Society.

[284] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv preprint arXiv:1607.07892*, 2016.

[285] K. F. Rietveld and H. A. Wijshoff. Re-engineering compiler transformations to outperform database query optimizers. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 300–314. Springer, 2014.

[286] K. F. Rietveld and H. A. Wijshoff. Reducing layered database applications to their essence through vertical integration. *Transactions on Database Systems*, 40(3):18, 2015.

[287] T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.

[288] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Generative Programming and Component Engineering*, GPCE'10, pages 127–136, New York, NY, USA, 2010. ACM.

[289] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *CACM*, 55(6):121–130, June 2012.

[290] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'13, pages 497–510, New York, NY, USA, 2013. ACM.

[291] J. Rose. Value types and struct tearing. 2014. https://blogs.oracle.com/jrose/entry/value_types_and_struct_tearing.

[292] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'88, pages 12–27. ACM, 1988.

[293] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD'15*, pages 1311–1326, 2015.

[294] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2, 1994.

[295] M. Scherr and S. Chiba. Almost first-class language embedding: Taming staged embedded dsls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 21–30, New York, NY, USA, 2015. ACM.

[296] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. 2016.

[297] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for scala. Technical report, EPFL, 2013.

[298] A. Shaikhha, M. Dashti, and C. Koch. Push versus Pull-Based Loop Fusion in Query Engines. *Journal of Functional Programming*, 28:e10, 2018.

[299] A. Shaikhha, A. Fitzgibbon, S. Peyton Jones, and D. Vytiniotis. Destination-passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 12–23, New York, NY, USA, 2017. ACM.

[300] A. Shaikhha, Y. Klonatos, and C. Koch. Building efficient query engines in a high-level language. *ACM Transactions on Database Systems*, 43(1):4:1–4:45, Apr. 2018.

[301] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD'16, pages 1907–1922, New York, NY, USA, 2016. ACM.

[302] X. Shi, B. Cui, G. Dobbie, and B. C. Ooi. Towards unified ad-hoc data processing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD'14, pages 1263–1274. ACM, 2014.

[303] O. Shivers and M. Might. Continuations and Transducer Composition. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'06, pages 295–307. ACM, 2006.

[304] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[305] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[306] J. M. Siskind and B. A. Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode ad. 2005.

[307] A. M. Sloane. Lightweight language processing in kiama. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 408–425. 2011.

[308] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. Compilation in Query Execution. In *the Seventh Intern. Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40. ACM.

[309] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. CGO '14, pages 23:23–23:32. ACM, 2014.

[310] D. G. Spampinato and M. Püschel. A basic linear algebra compiler for structured matrices. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 117–127. ACM, 2016.

[311] S. Sra and I. S. Dhillon. Nonnegative matrix approximation: algorithms and applications. Technical report, 2006.

[312] F. Srajer, Z. Kukelova, and A. Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in machine learning and computer vision. 2016.

[313] J. Stanier and D. Watson. Intermediate representations in imperative compilers: A survey. *CSUR*, 45(3):26:1–26:27, July 2013.

[314] G. Steele. *Common LISP: the language*. Elsevier, 1990.

[315] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 205–217, New York, NY, USA, 2015. ACM.

[316] M. Stonebraker. Technical perspective one size fits all: an idea whose time has come and gone. *Communications of the ACM*, 51(12):76–76, 2008.

[317] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB'05, pages 553–564. VLDB Endowment, 2005.

[318] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all? Part 2: Benchmarking results. In *CIDR*, 2007.

[319] M. Stonebraker, J. Becla, D. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for science data bases and SciDB. In *CIDR*, 2009.

[320] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, VLDB'07, pages 1150–1160, 2007.

[321] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 609–616, 2011.

[322] A. K. Sujeeth, A. Gibbons, K. J. Brown, H. Lee, T. Rompf, M. Odersky, and K. Olukotun. Forge: Generating a high performance DSL implementation from a declarative specification. In *GPCE'13*, pages 145–154, New York, NY, USA, 2013. ACM, ACM.

[323] E. Sumii and N. Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher Order Symbol. Comput.*, 14(2-3):101–142, Sept. 2001.

[324] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. GraphMat: High Performance Graph Analytics Made Productive. *VLDB*, 8(11), 2015.

[325] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP'02, pages 124–132, New York, NY, USA, 2002. ACM.

[326] B. J. Svensson and J. Svenningsson. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 43–52, NY, USA, 2014. ACM.

[327] D. Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 43–54, New York, NY, USA, 2006. ACM.

[328] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[329] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, volume 32 of *PEPM '97*, pages 203–217, NY, USA, 1997. ACM, ACM.

[330] W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[331] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'09, pages 264–276. ACM.

[332] J. Taylor, R. Stebbing, V. Ramakrishna, C. Keskin, J. Shotton, S. Izadi, A. Hertzmann, and A. Fitzgibbon. User-specific hand modeling from monocular depth sequences. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pages 644–651, 2014.

[333] The GNOME Project. GLib: Library package for low-level data structures in C – the reference manual, 2013. https://developer.gnome.org/glib/2.38/.

[334] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[335] R. Tibbetts, S. Yang, R. MacNeill, and D. Rydzewski. StreamBase LiveView: Push-Based Real-Time Analytics. *StreamBase Systems (Jan 2012)*, 2011.

[336] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[337] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher Order Symbol. Comput.*, 17(3):245–265, Sept. 2004.

[338] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ml kit. Technical report, DIKU Rapport, 1997.

[339] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.

[340] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., 2nd edition, 2011.

[341] E. Totoni, T. A. Anderson, and T. Shpeisman. HPAT: High Performance Analytics with Scripting Ease-of-use. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 9:1–9:10, New York, NY, USA, 2017. ACM.

[342] S.-A.-A. Touati and D. Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 147–156, 2006.

[343] Transaction Processing Performance Council. TPC-H, an Ad-Hoc, Decision Support Benchmark, 1999.

[344] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon. Bundle adjustment—a modern synthesis. In *Inter. workshop on vision algorithms*, pages 298–372. Springer, 1999.

[345] P. Trinder. Comprehensions, a Query Notation for DBPLs. In *Proc. of the 3rd DBPL workshop*, DBPL3, pages 55–68, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[346] D. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

[347] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, June 2000.

[348] P. B. Vasconcelos. *Space cost analysis using sized types*. PhD thesis, University of St Andrews, 2008.

[349] T. L. Veldhuizen. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 96–106, 2014.

[350] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *EuroSys*, 2013.

[351] S. Venkataraman, I. Roy, A. AuYoung, and R. Schreiber. Using R for iterative and incremental processing. In *HotCloud*, 2012.

[352] S. Viglas, G. M. Bierman, and F. Nagel. Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes. *IEEE Data Eng. Bull.*, 37(1):12–21, 2014.

[353] E. Visser. Meta-programming with concrete object syntax. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE'02, pages 299–315, London, UK, 2002. Springer-Verlag.

[354] E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP'98, pages 13–26, 1998.

[355] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design Patterns: Elements of Reusable Object-Oriented Software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[356] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proc. of ACM Symp. on LISP and functional programming*, pages 45–52, 1984.

[357] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *ESOP'88*, pages 344–358. Springer, 1988.

[358] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[359] P. Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

[360] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Inter. Workshop on Types in Compilation*, pages 177–206. Springer, 2000.

[361] M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[362] M. J. Weinstein and A. V. Rao. Algorithm: ADiGator, a toolbox for the algorithmic differentiation of mathematical functions in MATLAB using source transformation via operator overloading. *ACM Trans. Math. Softw*, 2016.

[363] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *PPSC*, 1999.

[364] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'07, pages 199–210, New York, NY, USA, 2007. ACM.

[365] B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA'08, pages 19–36, New York, NY, USA, 2008. ACM.

[366] M. J. Wolfe. *Techniques for Improving the Inherent Parallelism in Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[367] S. Wolfram. *The mathematica*. Cambridge university press Cambridge, 1999.

[368] L. Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, Jan. 2000.

[369] T. Würthinger. Extending the Graal Compiler to Optimize Libraries. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 41–42. ACM, 2011.

[370] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.

[371] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI'01, pages 298–308, New York, NY, USA, 2001. ACM.

[372] J. Yallop. Staged generic programming. *Proceedings of the ACM on Programming Languages*, 1(ICFP):29, 2017.

[373] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI'03, pages 63–76, New York, NY, USA, 2003. ACM.

[374] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI' 08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[375] C. Zach. Robust bundle adjustment revisited. In *European Conference on Computer Vision*, pages 772–787. Springer, 2014.

[376] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12. USENIX Association, 2012.

[377] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[378] B. M. Zane, J. P. Ballard, F. D. Hinshaw, D. A. Kirkpatrick, and L. Premanand Yerabothu. Optimized SQL Code Generation (US Patent 7430549 B2). WO Patent App. US 10/886,011, 2008.

[379] R. Zhang, S. Debray, and R. T. Snodgrass. Micro-Specialization: Dynamic Code Specialization of Database Management Systems. In *the Tenth ACM International Symposium on Code Generation and Optimization*, CGO '12, pages 63–73, New York, NY, USA, 2012. ACM.

[380] R. Zhang, R. T. Snodgrass, and S. Debray. Micro-Specialization in DBMSes. In *ICDE*, pages 690–701, Washington, DC, USA, 2012. IEEE Computer Society.

[381] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-efficient numerical computing without SQL. In *CIDR*, 2009.

[382] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral. Forma: A framework for safe automatic array reshaping. *TOPLAS*, 30(1):2, 2007.

[383] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD'02, pages 145–156, New York, NY, USA, 2002. ACM.

[384] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.

Amir Shaikhha

CONTACT INFORMATION

Av. du Tir-Fédéral 92
1024 Ecublens, Switzerland

Cell: +41 78 7135670
amir.shaikhha@epfl.ch
http://people.epfl.ch/amir.shaikhha

RESEARCH INTERESTS

- Database Systems
- Programming Languages
- Compilers
- Domain-Specific Languages

EDUCATION

- **Ecole Polytechnique Fédérale de Lausanne (EPFL)**, Lausanne, Switzerland. Sep. 2013 - June 2018 (Expected)
  Ph.D. in Computer Science, **Overall GPA: 5.71/6**
  Thesis: Compilation and Code Optimization for Data Analytics
  Courses: Semester Project in Computer Science (6/6), Synthesis, Verification and Analysis (6/6), Advanced Compilers (5.5/6), Applied Data Analytics (5.5/6).
- **Ecole Polytechnique Fédérale de Lausanne (EPFL)**, Lausanne, Switzerland. Sep. 2011 - August 2013
  M.Sc. in Computer Science, **Overall GPA: 5.72/6**
  Thesis: An Embedded Query Language in Scala **(6/6)**
  Relevant Courses **(GPA: 5.72/6)**: Semester Project in Computer Science (6/6), Optional Project in Computer Science (6/6), Advanced Databases (5.5/6), Distributed Algorithms (6/6), Foundation of Software Systems (5.5/6), TCP/IP (5.5/6), Concurrent Algorithms (5.5/6), Program parallelization on PC clusters (5.5/6).
- **Sharif University of Technology**, Tehran, Iran. Sep. 2007 - June 2011
  B.S. in Information Technology, **Overall GPA: 18.15/20 (3.84/4)**
  Thesis: Exact Graph Coloring on Multicore Systems **(20/20)**

PROFESSIONAL EXPERIENCE

- **Research Intern**, Microsoft Research, Cambridge, UK, July-September 2016.
- **Software Engineering Intern**, Typesafe Inc., February-August 2013.
- **Software Engineering Intern**, Typesafe Inc., July-September 2012.

HONORS AND AWARDS

- Best Paper Award, GPCE 2017.
- Most Reproducible Paper Award, SIGMOD 2017.
- Google PhD Fellowship, 2017.
- Teaching Assistant Award, IC EPFL, 2016.
- Ranked $\mathbf{2}^{nd}$, in Iranian National Scientific Olympiad for University Students in Computer Engineering, September 2011.
- Ranked $\mathbf{1}^{st}$, in Cumulative GPA among more than 30 students of the Information Technology, Class of 2011 students, Sharif University of Technology, June 2011.
- Granted admission from *Talented Student Office* of Sharif University of Technology for graduate study.

PUBLICATIONS

- **Efficient Differentiable Programming in a Functional Array-Processing Language**; A. Shaikhha, A. Fitzgibbon, S. Peyton-Jones, D. Vytiniotis, C. Koch, Under submission.
- **Synthesis of Incremental Analytics**; A. Shaikhha, M. El Seidy, D. Espino, S. Mihaila, and C. Koch, Under submission.
- **Building Efficient Query Engines in a High-Level Language**; A. Shaikhha, Y. Klonatos, C. Koch, TODS 2018.

- **Push vs. Pull-Based Loop Fusion in Query Engines**; A. Shaikhha, M. Dashti, C. Koch, JFP 2018.
- **Unifying Analytic and Statically-Typed Quasiquotes**; L. Parreaux, A. Voizard, A. Shaikhha, and C. E. Koch, POPL 2018.
- **Quoted Staged Rewriting: a Practical Approach to Library-Defined Optimizations**; L. Parreaux, A. Shaikhha, and C. E. Koch, GPCE 2017 ***(Best Paper Award)***.
- **Squid: Type-Safe, Hygienic, and Reusable Quasiquotes**; L. Parreaux, A. Shaikhha, and C. E. Koch, SCALA 2017.
- **Destination-Passing Style for Efficient Memory Management**; A. Shaikhha, A. Fitzgibbon, S. Peyton-Jones, D. Vytiniotis, FHPC 2017.
- **Repairing Transaction Conflicts in Optimistic Multi-Version Concurrency Control**; M. Dashti, S. John, A. Shaikhha, C. Koch, SIGMOD 2017 ***(Most Reproducible Paper Award)***.
- **How to Architect a Query Compiler**; A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, C. Koch, SIGMOD 2016.
- **Yin-Yang: Concealing the Deep Embedding of DSLs**; V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, M. Odersky, GPCE 2014.
- **DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views**; C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, A. Shaikhha, The VLDB Journal, 2014.

Research Experience

- **SC: A Framework for Systems-Compiler Co-Design**; EPFL, Under supervision of Prof. Christoph Koch, in collaboration with Lionel Parreaux.
- **DBLAB: A Framework for Building Database Systems by High-Level Programming**; EPFL, Under supervision of Prof. Christoph Koch, in collaboration with Yannis Klonatos.
- **OCAS: Automatic Synthesis of Out-of-Core Algorithms**; EPFL, Under supervision of Prof. Christoph Koch, in collaboration with Yannis Klonatos.
- **LAGO: A Holistic Approach Towards Matrix Algebra**; EPFL, Under supervision of Prof. Christoph Koch, in collaboration with Mohammed ElSeidy.
- **Yin-Yang: A Library for Deep Embedding of DSLs in Scala**; EPFL, Under supervision of Prof. Martin Odersky and Prof. Christoph Koch, in collaboration with Vojin Jovanovic.

Teaching Experience

Teaching Assistant *Master in Data Science – EPFL*

- Systems for Data Science; (Spring 2018, $\sim$ 40 students)

Master in Computer Science – EPFL

- Database Systems; (Spring 2016, Spring 2017, $>$ 100 students)
- Foundation of Software; (Fall 2015, Fall 2016, Fall 2017, $\sim$ 50 students)
- Big Data; (Spring 2014, Spring 2015, $>$ 100 students)

Bachelor in Mathematics & Bachelor in Physics – EPFL

- Information, Computation, Communication; (Fall 2014, $>$ 250 students)

Bachelor in Computer Engineering – Sharif University of Technology

- Introduction to Programming (C++); (Spring/Fall 2008)
- Advanced Programming (Java); (Spring/Fall 2009, Spring/Fall 2010, Spring 2011)
- Data Structures and Algorithms; (Spring 2009, Spring/Fall 2010)
- Computer Structure and Language; (Spring 2009)

- Computer Networks; (Spring/Fall 2010, Spring 2011)
- Artificial Intelligence; (Fall 2010)

Teaching

- Domain-Specific Languages; Winter 2017; A workshop organized by the Students Scientific Chapter of Sharif University of Technology.
- J2ME Programming; Summer 2009, Summer 2010; The course maintained by the Students Scientific Chapter of Sharif University of Technology.

Advising & Mentoring

- **Michal Pleskowicz**, Bachelor Project, A Query Optimizer for DBLAB, Spring 2018.
- **Parand Alizadeh, Mohsen Ferdosi**, Summer Internship Project, A Frontend for DBToaster in DBLAB, Summer 2017.
- **Daniel Espino**, Master Thesis, Cost-Based Optimization of Iterative Linear Algebra, Spring 2016.
- **Matthieu Rudelle**, Master Semester Project, Optimization of a Distributed Information Retrieval System, Fall 2015.
- **Khayyam Guliyev**, Master Semester Project, Using SC as the Backend for DBToaster, Spring 2015.
- **Stefan Mihaila**, Master Thesis, Incremental Computation and Symbolic Differentiation of Matrices, Spring 2015.
- **Robin Hahling, Kevin Gillieron, Laurent Weingart**, Master Semester Project, DevMine: A Developer Search Engine and Source Code Analysis Framework, Fall 2014 & Spring 2015.
- **Lewis Brown**, Master Semester Project, C.Scala: Shallow Embedding of C in Scala, Fall 2014.

Projects

- **$\widetilde{\text{F}}$: A Subset of F# that Generates Fast Code for Numerical Workloads**; Microsoft Research, Cambridge, Under supervision of Dr. Andrew Fitzgibbon, Dr. Simon Peyton-Jones, Dr. Don Syme, and Dr. Dimitrios Vytiniotis.
- **Deep Embedding in Slick using Yin-Yang**; Typesafe Inc., Under supervision of Prof. Martin Odersky, Stefan Zeiger.
- **Type Providers in Slick**; Typesafe Inc., Under supervision of Prof. Martin Odersky, Stefan Zeiger.
- **Play Plugin for Scala IDE**; Typesafe Inc., Under supervision of Dr. Iulian Dragos.
- **Domain Maintenance in DBToaster**; EPFL, Under supervision of Prof. Christoph Koch.
- **Web-Based Virtual PCR-RFLP and Finding Maximum Correspondence in Phylogenetic Trees**; University of Tehran, Under supervision of Prof. A. Sharifi.
- **Design and Implementation of a Simulator and Assembler for IBM System/370**; Sharif University of Technology, Under supervision of Prof. H. Sarbazi-Azad.

Activities

- PhD Students' Representative, IC EPFL, 2017-2018.
- Member of EPIC PhD Student Association, IC EPFL, 2017.
- Elected Member and Head of Students Scientific Chapter (SSC), Department of Computer Engineering, Sharif University of Technology, 2010-2011.

Skills

- **Programming Languages:** ***Scala***, C/C++, Pascal, C#, F#, Java(J2SE, J2ME), Prolog, Assembly(x86, IBM S/370, Motorolla 68000), Verilog, MATLAB, GPGPU Programming with CUDA, Cell BE Programming, OCaml.
- **Operating Systems:** Linux, Dos, Windows, Mac OS.

- **Web/DB Technologies:** HTML, CSS, Javascript, PHP, MySQL, ASP.NET, Symfony, J2EE, Play.
- **Typesetting**: TEX, LATEX, Microsoft Word.

HOBBIES

- **Sports:** Ping Pong, Karate (Shito-Ryu).

REFERENCES Available upon request

Last update: June 15, 2018