BUILDING EFFICIENT QUERY ENGINES USING HIGH-LEVEL LANGUAGES

THÈSE N^{O} 7508 (2017)

PRÉSENTÉE LE 27 JANVIER 2017

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

Ioannis KLONATOS

acceptée sur proposition du jury:

Prof. V. Kuncak, président du jury
Prof. C. Koch, directeur de thèse
Prof. S. Viglas, rapporteur
Prof. A. Cheung, rapporteur
Prof. J. Larus, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2017

We all have our demons.
(I know I have many –
Anguish, stress and self doubt to name a few.)

Thus, I dedicate this to my parents.

To my mother, above all,
who despite my gazillion faults,
was always ready to lift her little fists
(with a serious twitch of the eyebrow)
to fight my demons for me.

To my father,
who valiantly fought his demons,
(being an only child is not easy after all)
and allowed me, despite his fears,
to experience a wonderful life abroad.

May my future journeys bring you
no more sorrow.

Acknowledgements

No man is an island.
– John Donne.

As you set out for Ithaka,
hope the voyage is a long one,
full of adventure, full of discovery.
– Constantine P. Cavafy.

My PhD journey has definitely been a long, bumpy road, with many great moments, but also a number of disappointing and depressing ones. I was unlucky (or, maybe, lucky) enough to have witnessed the best and the worst aspects of academia. For sure, EPFL is one of the best places to do research, but this frequently brings out the ugliest and most competitive side of people. Given this, there were frequently feelings of pure loneliness in my journey, maybe just to balance (in a supernatural way) the awesome moments of inspiration that come with doing research at EPFL.

But hold on a minute! one may say. These are the *acknowledgments* of your thesis, and you should be grateful, not miserable and gloomy like that. And he or she would be right to say so. But yet, I feel that such a "dark" introduction is necessary. Why?

Because of the bridges. The bridges that specific people kept building in order to reach me when all I wanted was to be left alone on my island. These people, who I am honored today to call mentors, friends, as well as my family, who patiently listened to me when I looked for a sounding board; they who not only threw a rope down the big dark hole that I have dug myself in but even climbed down themselves in my abyss to talk me out of it; they who calmed me down so many times when I was in anguish; they who with perseverance tolerated my emotional explosions when I needed to vent; they who patiently pushed me to continue; they who never allowed me to give up. My long journey, of which Cavafy spoke, today comes to an end – and it is because of them.

So here is a (probably incomplete) list of people that I am truly grateful to:

- My PhD advisor and mentor, Professor Christoph Koch, for continuously pushing me to improve myself, teaching me the difference between engineering and research, and helping me to grow far beyond my self-imposed limited aspirations. He is definitely the smartest man I have ever met, and I hope that I will carry some of his wisdom with me from now on.

- Professors Alvin Cheung, Jim Larus, Stratis Viglas, and Victor Kunčak for agreeing to be in my committee and for providing an interesting (yet very stressful) discussion during the defense. In particular, I am grateful to Professor Viglas for his help with some bureaucratic issues and to Professor Kunčak for always treating me with such absolute kindness.

- Nikolaos "the patient" Arvanitopoulos, simply because he was definitely the one to suffer the most from my constant moaning all these years. Nikos continuously (and patiently!) pushed me to have a healthier and more balanced lifestyle and has been a very loyal friend, without whose constant and *always* wise advice I wouldn't be writing here today.

- Elio "the vibrant" Abi Karam. When we first met, Elio and I were on diagonally opposite sides of optimism. Despite my stubbornness, Elio took it upon himself to make me enjoy life once again and become more positive and happy. He definitely succeeded, and some of his (seemingly endless) joyful energy has rubbed off on me over time. I truly owe him a lot.

- Daniel "the Vulcan" Lupei. Among other things, Daniel always helped me put things into perspective, setting negative feelings aside and analyzing problems in the most logical of ways. That one night when I was one step away from fleeing the PhD program, he sat with me over a beer and a burger and persuaded me to stay. Live long and prosper, Daniel.

- Aleksandar "the kind-hearted" Vitorovic. The famous writer Dostoyevsky once wrote of a man who is *pure* good, and – until I met Aleksandar – I always believed that such a person can exist only in fiction. He truly is the embodiment of kindness and selflessness and his support throughout my journey has been immense.

- Vasilis Agrafiotis and Vasilis Koukoulomatis, also known as V^2. When I first met them, some events have led me to be extremely reserved towards making new friends. Yet, their immediate support, openness, and friendship were so overwhelming that I soon changed my mind. Apparently, there is time for more friends and time for fewer friends :-).

- My coauthors Andres Nötzli and Amir Shaikkha for their extensive support and tolerance throughout all the long nights that led to the publications of my thesis. Andres is simply the best and funniest Swiss I have ever come across. Amir helped me to overcome a significant crisis in my career and gave me valuable insights about the Iranian history and culture.

- Predrag "the foosball teammate" Spasojevic and Danica "the rollercoasters co-rider" Porobic for significant psychological support during this PhD. Predrag never allowed me to give up on obtaining the diploma (with an often somewhat irritating persistence). Danica made my internship in California enjoyable with many awesome, rollercoaster-related, moments.

- The Credit Suisse "gang", and specifically (ladies first): Julie Djeffal, Simona Treykova, Jessica Putz, Mariza Lazic, Federico Petrucci, Enrico Ferrari, Alexandru Repede, Mihai "Mouhaha"

Gurban, and Sergey Golubev for their constant support. Truly, the last two years would have been so much less enjoyable and fun if I haven't made their acquaintance.

- My Greek colleagues and friends, Manos Karpathiotakis, Ioannis Alagiannis, and Matthaios Alexandros Olma for continuously pushing me to finish this PhD journey with a good ending. In particular, and among other things, I would like to thank Manos because he had the patience to listen to my 45 minute PhD presentation and provide valuable feedback.

- My supervisors Paul Keenan and Nicolas Debons from Credit Suisse. Among other things, Paul has persistently worked to regenerate my (back then) depleted self-confidence. They both kindly allowed me to work towards finishing the PhD, particularly during the last months. It would have been a much harder road in the end without their support.

- My "Heraklion friends" Vangelis Kafentarakis, Michail Alvanos, Kostas Mousikos and Nikiforos Manalis for always finding some opportunity to communicate and discuss all these years that we have been afar. During my PhD days, I would often reminisce back to my days in Heraklion and the joyful and full of happiness memories that these friends helped shape.

- To the *"The friends who met here and embraced are gone, each to his own mistake"*: Andrew Becker, Marina Boia, Vasilis Kalofolias, Anna Kalokairinou, and Mohammed El Seidy. Somehow life brought our paths together, we bonded and shared good moments, and then simply (and mysteriously) departed on separate journeys. I would like to thank them here anyway.

- My "old" mentors, Professor Angelos Bilas, Manolis Marazakis and Michalis Flouris, even though the word "old" is not really fit for people that influenced and continue to influence one person so much. Their support seems never-ending, and despite being many miles away, they have always been there to offer me sound advice whenever I needed it.

- The dear family friend, Eleni Papadogeorgopoulou, for always providing a shoulder to cry on during the most difficult moments of this PhD, but also – and more importantly – because she always *truly* shared on the joy and happiness (and that is *very* rare in life) of the good moments of my career so far.

- I am constantly told that I have been lucky in life. Lucky because, among other reasons, I have had people in my life that – while not related to me by blood – have always treated me like their own child. I am truly grateful to my "second parents" Ioannis Antonakos and Efstathia Salamaliki for always giving me such unconditional love, affection, and care.

- My parents Ioannis Klonatos and Evanthia Megagianni. There are no words that can possibly convey how much I owe to them and their upbringing. My PhD journey took ugly turns at times and often brought out an uncharacteristically harsher, cynical and much more egocentric side of me. Their constant support and love helped me to always put things into perspective, maintain my sanity and survive this journey while being true to myself. I may be a "doctor" now, but I am definitely the *man* I am today because of them.

Lausanne, 29 December 2017 Y. K.

Abstract

We are currently witnessing a shift towards the use of *high-level* programming languages for systems development. Success stories can be found in the areas of operating and distributed systems as well as GPU programming. These approaches collide with the traditional wisdom which calls for using low-level languages for building efficient software systems.

This shift is necessary as billions of dollars are spent annually on the maintenance and debugging of performance-critical software. High-level languages promise faster development of higher-quality software; by offering advanced software features, they allow the same functionality to be implemented with significantly less code, thus helping to reduce the number of software errors of the systems and facilitate their verification.

Despite these benefits, database systems development seems to be lagging behind as DBMSes are *still* written in low-level languages. The reason is that the increased productivity offered by high-level languages comes at the cost of a pronounced negative performance impact.

In this thesis, we argue that it is now time for a radical rethinking of how database systems are designed. We show that, by using high-level languages, it is indeed possible to build databases that allow for both productivity and high performance, instead of trading-off the former for the latter. By programming databases in a high-level style, the time saved can be spent implementing more interesting database features and optimizations.

More concretely, in this thesis we follow this *abstraction without regret* vision and use high-level programming languages to address the following two problems encountered while developing database systems.

First, the introduction of a new storage or memory technology typically requires the development of new versions of most out-of-core algorithms employed by the database system. This is because performance-critical software always needs to be specialized to best match the underlying architecture. Given the rapid rate of hardware innovation and the increasing popularity of hardware specialization, this leads to an arms race for the developers. To make things even worse, there exists no clear methodology for creating such algorithms and we must rely on significant creative effort to serve our need for out-of-core algorithms.

To address this issue, we present the OCAS framework for the automatic synthesis of efficient out-of-core algorithms. These are specialized for a particular memory hierarchy and a set of

storage devices. The developer provides two independent inputs: 1) an algorithm, expressed using a high-level specification language, that ignores memory hierarchy and external storage aspects; and 2) a description of the target memory hierarchy, including its topology and parameters. Using these specifications, our system is then able to automatically synthesize memory-hierarchy and storage-device-aware algorithms for tasks such as joins and sorting. The framework is extensible and allows developers to quickly synthesize custom out-of-core algorithms as new storage technologies become available.

Second, from a software engineering point of view, years of performance-driven DBMS development have led to complicated, *monolithic*, low-level code bases, which are hard to maintain and extend. In particular, the introduction of new innovative approaches or optimizations in existing systems can be a very time-consuming and challenging task.

To overcome such limitations, we present LegoBase, a query engine written in the *high-level* programming language, Scala. LegoBase realizes the abstraction without regret vision in the domain of analytical query processing. We show how by offering sufficiently powerful abstractions our system allows to *easily* implement a broad spectrum of optimizations which are difficult to achieve with existing approaches. Then, the key technique to regain efficiency is to apply *generative* programming: LegoBase performs source-to-source compilation and converts the high-level Scala code to specialized, low-level C code. Our architecture significantly outperforms a commercial in-memory database system as well as an existing query compiler, while programmers need to provide just a few hundred lines of high-level code for building and optimizing the *entire* query engine. LegoBase is the first step towards providing a full DBMS written in a high-level language.

Key words: High-level programming languages, Out-of-core algorithms, Program synthesis, Memory and storage hierarchies, Query processing, Generative programming, Optimizing compilers, Abstraction without regret, Database optimization.

Résumé

Nous assistons actuellement à une transition vers l'utilisation des langages de haut niveau pour le développement de systèmes. On trouve des réussites dans les domaines des systèmes distribués et d'exploitation ainsi que la programmation des GPUs. Ces approches sont en contradiction avec la pensée traditionnelle qui appelle à l'utilisation de langages de bas niveau pour la construction de systèmes informatiques efficaces.

Cette transition est nécessaire car des milliards de dollars sont dépensés chaque année pour la maintenance et le débogage de logiciels nécessitant une haute performance. Les langages de haut niveau promettent un développement plus rapide de logiciels de meilleure qualité. En offrant des fonctionnalités logicielles avancées, elles permettent l'implémentation de la même fonctionnalité avec beaucoup moins de code. Cela contribue à la réduction du nombre d'erreurs informatiques et à faciliter la vérification des fonctionnalités.

En dépit de ces avantages, le développement des systèmes de bases de données traîne toujours parce que les DBMS sont encore écrits avec des langages de bas niveau. En effet, l'augmentation de la productivité provenant des langages de haut niveau est accompagnée d'un impact négatif sur la performance.

Dans cette thèse, nous soutenons qu'il est temps de radicalement repenser la façon dont les systèmes de base de données sont conçus. Nous montrons qu'en utilisant des langages de haut niveau, il est possible de construire des bases de données qui permettent à la fois la productivité et la haute performance. En programmant des bases de données dans un style de haut niveau, le temps économisé peut être utilisé pour le développement de nouvelles optimisations et fonctionnalités de bases de données.

Plus concrètement, dans cette thèse nous suivons le concept de l'abstraction sans regret et nous utilisons des langages de programmation de haut niveau pour résoudre les deux problèmes présentés ci-dessous. Ces derniers sont souvent rencontrés dans les systèmes de bases de données.

Tout d'abord, l'introduction d'une nouvelle technologie de stockage ou de mémoire exige généralement le développement de nouvelles versions de la plupart des algorithmes hors-cœur utilisés par le système de base de données. En effet, un logiciel nécessitant une haute performance doit toujours être spécialisé pour mieux correspondre à l'architecture sous-

jacente. Etant donné le rythme accéléré de l'innovation en hardware et l'augmentation de la popularité de la spécialisation en hardware, cela conduit à une course aux armements pour les développeurs. De plus, il n'existe pas de méthodologie claire pour créer de tels algorithmes et donc nous devons compter sur un effort créatif considérable pour répondre à notre besoin d'algorithmes hors-cœur.

Pour résoudre ce problème, nous présentons le cadre du OCAS pour la synthèse automatique des algorithmes hors-cœur. Ceux-ci sont spécialisés pour une hiérarchie de mémoire particulière et un ensemble d'appareils de stockage. Le développeur fournit deux entrées indépendantes : 1) un algorithme, exprimé avec un langage de spécification de haut niveau, qui ignore la hiérarchie de mémoire et les aspects de stockage externe. 2) une description de la hiérarchie de la mémoire cible, y compris sa topologie et paramètres. A partir de ces spécifications, notre système est alors capable de synthétiser automatiquement des algorithmes tenant compte de la hiérarchie de mémoire et des appareils de stockage, pour des tâches telles que les jointures et le tri. Le cadre est extensible et permet aux développeurs, avec la disponibilité de nouvelles technologies de stockage, de synthétiser rapidement des algorithmes hors-cœur personnalisés .

Deuxièmement, du point de vue d'ingénierie informatique, des années de développement de DBMS concentrés sur la performance ont conduit à des bases de code de bas niveau, compliquées, et monolithiques, qui sont difficiles à maintenir et étendre. En particulier, l'introduction de nouvelles méthodes innovatrices et d'optimisations dans les systèmes existants peut être une tâche très longue et difficile.

Pour surmonter de telles limitations, nous présentons LegoBase, un moteur de requêtes écrit avec le langage de programmation de haut niveau, Scala. LegoBase réalise la vision de l'abstraction sans regret dans le domaine du traitement analytique des requêtes. Nous montrons qu'en offrant des abstractions suffisamment puissantes, notre système permet de facilement mettre en œuvre un vaste choix d'optimisations qui sont difficiles à atteindre avec les approches actuelles. Ensuite, la technique clé pour regagner l'efficacité est d'appliquer la programmation générative : LegoBase effectue la compilation source-à-source et convertit le code Scala, de haut niveau, en code C spécialisé, de bas niveau. Notre architecture surpasse un système de base de données en mémoire commerciale ainsi qu'un compilateur de requêtes existant. Notre système nécessite en contrepartie que les programmeurs fournissent juste quelques centaines de lignes de code de haut niveau pour la construction et l'optimisation du moteur de requête tout entier. LegoBase est la première étape vers un DBMS écrit dans un langage de haut niveau.

Mots clefs : Langages de programmation de haut niveau, algorithmes hors-cœur, synthèse de programmes, mémoire et hiérarchies de stockage, traitement des requêtes, programmation générative, compilateurs d'optimisation, abstraction sans regret, optimisation de bases de données.

Zusammenfassung

Wir verzeichnen derzeit einen Wandel in der Programmiersprache für Systementwicklungen von den noch allgegenwärtigen low-level Sprachen hin zu high-level Programmiersprachen. Unter Anderem und vor Allem sind in den Bereichen Operating und Distributed Systems und in der GPU Programmierung enorme Erfolge erzielt worden.

Dieser Wandel ist unvermeidlich in Anbetracht der Milliardenbeträge die jährlich für die Instandhaltung und Fehlerbehebung an performancekritischer Software aufgebracht werden müssen. Durch high-level languages kann höherwertige Software schneller entwickelt werden. Fortschrittliche Softwareeigenschaften ermöglichen gleichwertige Funktionalität mit deutlich reduzierter Kodierung wodurch die Zahl der Softwarefehler enorm reduziert und es erleichtert wird diese zu verifizieren.

Ungeachtet dieser Vorteile hinsichtlich Effizienz und Effektivität hinken High-Level- Programmierungen hinterher, denn nach wie vor werden DBMSes in Low-Level-Programmiersprachen geschrieben, da die erhöhte Produktivität einen negativen Effekt auf das Betriebsverhalten ausübt.

In dieser Arbeit soll dargestellt werden, wie wichtig das radikale Umdenken für das zukünftige Design von Database Systemen ist. Es wird demonstriert, dass bei dem Gebrauch von high-level Sprachen anstatt der bisherigen Denkweisen, weder auf Produktivität, noch auf hohe Performance verzichtet werden muss. Durch die Programmierung in high-level Programmierungssprachen kann nicht nur Zeit gespart, sondern der Fokus auf interessantere Database Funktionalitäten und Optimierungsmethodiken gelegt werden.

Im Einzelnen soll diese Arbeit darlegen, wie durch die Nutzung der high-level Sprachen zwei Problematiken bei der Kodierung von Datasbase Systemen konkret angesprochen werden: Erstens benötigt die Einführung einer neuen Speichertechnologie normalerweise die Entwicklung von neuen Versionen eines Out-of-Core Algorithmus, der von Database Systemen verwendet wird, da performance-kritische Software jedes Mal den Gegebenheiten bestmöglich angepasst werden muss. Die rasante Entwicklung immer neuer Hardware sowie die zunehmende Nachfrage nach spezieller und individueller Hardware führt daher zu einem sehr starken Konkurrenzkampf. Zudem gibt es bislang keine präzise Methodik derartige Algorithmen zu entwickeln. Die Entwicklung solcher Algorithmen ist daher sehr anspruchsvoll und

zeitaufwendig.

Eine mögliche Vorgehensweise um diesen Aspekt anzugehen, ist die Nutzung eines sogenannten OCAS Frameworks, welcher die Synthese dieses Out-of-Core Algorithmuses erlaubt. Diese Frameworks spezialisieren sich auf die bestimmte Speicherhierarchie und eine Reihe von Speichervorrichtungen. Für die Entwicklung werden zwei unabhängige Inputs benötigt: 1. Ein auf high-level language basierter Algorithmus der sowohl Speicherhierarchien, als auch externe Speichervorrichtungen ausser Acht lässt. Und 2. Eine Beschreibung der gewünschten Speicherhierarchie welche die genaue Netzstruktur und deren Parameter bestimmt. Ein System das beide Voraussetzungen erfüllt, wäre folglich fähig, Speicherhierarchien und Algorithmen, die auf spezielle Speichervorrichtungen gepolt sind, automatisch zu kombinieren. Diese Funktionalität würde Ausführung wie Joins und Sorting um ein Vielfaches vereinfachen. Der Framework ist beliebig erweiterbar und ermöglicht eine rasche Entwicklung eines Out-of-Core Algorithmus der neue Speichervorrichtungstechnologien zur Verfügung stellt.

Zweitens hat, aus Sicht eines Softwareingenieurs oder -entwicklers, die jahrelange DBMS Entwicklung zu komplexen, monotholistischen, anspruchslosen Code Bases geführt. Diese sind schwer aufrechtzuerhalten und zu erweitern. Besonders die Einführung von neuen, innovativen Denkansätzen oder der Optimierungsprozess von bereits existierenden Systemen ist herausfordernd und äusserst zeitaufwendig.

Um diese Anwendungsgrenzen zu überwinden, wird im Folgenden LegoBase erläutert, ein Abfrage Mechanismus der in der high-level Programmiersprache Scala verfasst ist. LegoBase realisiert die Vision der Abstraktion ohne Bedauern im Bereich der analytischen Abfrageverarbeitung. Es wird dargestellt, wie einfach die Implementierung eines breiten Spektrums von Optimierungen mit Hilfe von ausreichend leistungsstarken Rückhaltungen des LegoBase Systems ist. Des Weiteren wird aufgeführt, wie durch das sogenannte „generative Programming“ die Effizienz wieder hergestellt wird: LegoBase verübt ein source-to-source Kompilat und konvertiert dabei high-level Scala Code in einen speziellen low-level C Code. Die Struktur, die hierbei verwendet wird, übertrifft die Struktur kommerzielleren, in-memory Database Systemen, sowie bestehende query compiler. Gleichzeitig benötigt es nur einige hundert Zeilen high-level Code für die Entwicklung und Optimierung eines kompletten Abfragemechanismus.Schlussfolgernd ist LegoBase der erste Schritt in die Entwicklung eines vollständigen DBMS, basiert auf high-level Programmierungssprache.

Stichwörter: High-level Programmierungssprachen, „Out-of-core Algorithmen”, Programmaufbau, „Speichervorrichtungen und Hierarchien,” „Query Processing”, „Generative Programming“, Optimierungscompiler, Database Optimierung, “Abstraction without regret“

Contents

List of Figures

List of Tables

1 Introduction

During the last decade, we have witnessed a shift towards the use of high-level programming languages for systems development. Examples include the Singularity Operating System [Hunt and Larus, 2007], the Spark [Zaharia et al., 2010] and DryadLINQ [Yu et al., 2008] frameworks for efficient, distributed data processing, the FiST platform for specifying stackable file systems [Zadok et al., 2006] and GPUs programming [Holk et al., 2013]. All these approaches collide with the traditional wisdom which calls for using low-level languages like C for building high-performance systems.

This shift is necessary as the productivity of developers is severely diminished in the presence of complicated, monolithic, *low-level* code bases, making their debugging and maintenance very costly. Studies indicate that, currently, maintenance costs range from 50% up to 90% of the total costs of a software product, while the annual cost of addressing software bugs rises to billions of dollars [Bhattacharya and Neamtiu, 2011].

High-level programming languages can remedy this situation in two ways. First, by offering advanced software features (modules, interfaces, collections, object orientation, etc.), they allow the same functionality to be implemented with significantly less code (compared to low-level languages). Second, by providing powerful type systems and well-defined design patterns, they allow programmers not only to create abstractions and protect them from leaking but also to quickly define system modules that are truly *reusable* (even in contexts very different from the one these were created for) and easily *composable* [Odersky and Zenger, 2005]. All these properties can reduce the number of software errors of the systems and facilitate their verification.

Yet, despite these benefits, database systems are still written using low-level languages.

The reason is that increased productivity comes at a cost: high-level languages increase indirection, which in turn has a pronounced negative impact on performance. For example, abstraction generally necessitates the need of containers, leading to costly object creation and

destruction operations at runtime. Encapsulation is provided through object copying rather than object referencing, thus similarly introducing a number of expensive memory allocations on the critical path. Even primitive types such as integers are often converted to their object counterparts for use with general-purposes libraries. Given that high performance has always been the holy grail of database management systems, such performance overheads make the use of high-level languages for developing high-performance databases to seem (deceptively) prohibited.

The *abstraction without regret* vision [Koch, 2013, 2014] argues that it is indeed possible to use high-level languages for building database management systems that allow for *both* productivity and high performance, instead of trading off the former for the latter. 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. In addition, the language features of high-level languages can grant great flexibility to developers so that they can easily experiment with various design choices when building database systems.

In this thesis, we realize this vision and argue that it is now time for a radical rethinking of how database management systems and their optimizations are designed. More concretely, we take advantage of high-level programming languages to address the following two problems frequently encountered while developing database systems.

1.1 Problem Statement

Let us first consider the case where a new hardware platform becomes available.

It is common knowledge that the design of performance-critical software systems depends on the hardware on which these systems run; program optimization dictates that high-performance software must always match to the properties of the underlying architecture. For example, we must ensure that the CPU does not remain idle waiting on memory, by structuring algorithms so that they make sufficient use of data locality. This is particularly true for data-intensive computations.

This means that the introduction of a new storage or memory technology requires the development of new versions of most out-of-core algorithms utilized by a database management system. The research literature describes numerous out-of-core algorithms designed and optimized for a variety of hardware and storage device configurations [Ramakrishnan and Gehrke, 2002; Govindaraju et al., 2006; Cederman and Tsigas, 2008; Sintorn and Assarsson, 2008; Andreou et al., 2009; Park and Shim, 2009; Ye et al., 2010; Kim et al., 2010; Liu et al., 2011]. These are some case studies of how understanding memory hierarchies and data locality can drive algorithm design.

However, given the rapid rate of hardware innovation and the increasing popularity of hardware specialization, we are currently experiencing an arms race between the developers of

hardware on the one hand and of software systems and out-of-core algorithms on the other. Each new development in hardware calls for numerous research contributions on the software side, to update a multitude of algorithms and systems. To make things even worse, to this day no methodology exists for creating out-of-core algorithms and we must rely on significant creative talent and effort to serve our need for such algorithms. □

Let us now turn our attention to the role of the database developer today.

As Dennard's law has already failed [Esmaeilzadeh et al., 2011], sequential computing hardware is not getting faster anymore, and developers have to look for specialization opportunities for continued performance growth in software systems [Stonebraker and Cetintemel, 2005].

However, traditional general-purpose compilers are not, to date, trusted to sufficiently deliver on performance out-of-the-box. Thus, a considerable aspect of the developer's job is to act as a substitute (pre-)compiler, who is trusted to deliver fast code and who manually optimizes the DBMS code by eliminating abstraction and indirection overheads. To do so, the developer has to work with the highly complicated and largely monolithic design of existing database systems. Such monolithic implementations are mostly driven by the dictate of performance: there conceptually separate components such as the storage manager, query engine, concurrency control, and recovery subsystems are all blended together into a single giant monolithic component. In addition, alternative implementations of data structures are manually inlined for performance, leading to a great deal of redundancy. Further development of such large code bases is, thus, a very difficult task [Lomet et al., 2011; Koch, 2014].

In addition, and as illustrated in Figure 1, query compilation – a prominent technique to boost the performance of a database system – further sacrifices productivity for performance. In general, query compilation approaches perform source-to-source compilation[1] in order to optimize away the overheads of traditional database abstractions like the Volcano operator model [Graefe, 1994]. However, existing solutions do so by using low-level code templates. This introduces an additional level of complexity when coding database optimizations for two reasons. First, providing low-level templates – essentially in stringified form – makes it hard or impossible to automatically typecheck the code. Second, since the templates are directly emitted by the generator, developers have to deal with a number of low-level concerns which make templates very difficult to implement and get right. For example, when generating LLVM code [Lattner and Adve, 2004] developers must handle register allocation themselves. Code maintenance definitely becomes more complicated in the presence of query compilation; the System R team reported that complicated maintenance was one of the main reasons that query compilation was abandoned in favor of interpretation [Chamberlin et al., 1981].

To summarize, the introduction of new innovative approaches or optimizations in existing systems can be a very time-consuming and challenging task for database developers. □.

[1] Typically from C/C++ – with which the DBMS system is originally written – to optimized C/LLVM code.

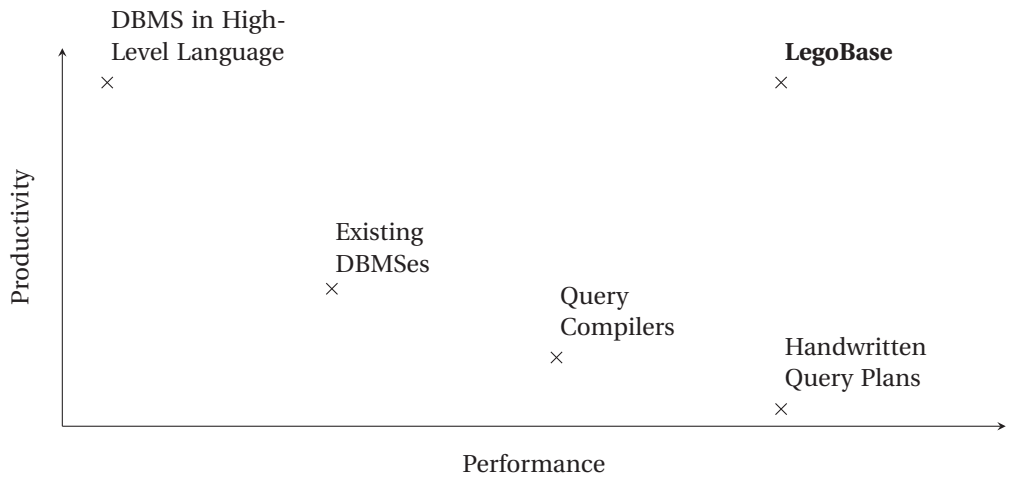

Figure 1.1 – Comparison of the performance/productivity trade-off for all approaches presented in the second part of this thesis.

1.2 Thesis Contributions

We present solutions to the two aforementioned problems, which at the same time form the main contributions of this thesis.

To address the first problem, we present the OCAS framework for the automatic synthesis of specialized out-of-core algorithms. The input is a) a naive, memory-hierarchy-oblivious algorithm, expressed using a high-level specification language and b) a description of the target hardware setup and memory hierarchy. We encode fundamental principles of out-of-core algorithm design, many of which aim at the maximization of data locality, as transformation rules. The application of such a rule to the high-level algorithm results in a *functionally equivalent* algorithm which *may* have better performance on the underlying hardware. By applying transformation rules, we create a navigable search space of equivalent algorithms. To be able to choose an optimal algorithm, we develop a cost-estimation procedure, which is based on the given hardware description and is an approximation of the program's running time. The objective is then to find the program with the minimal cost. The framework is extensible and allows developers to quickly synthesize out-of-core algorithms as new technologies become available, for fundamental database operations such as joins and sorting.

To address the second issue, we present LegoBase, an in-memory query execution engine written in the high-level programming language, Scala. LegoBase realizes the abstraction without regret vision in the domain of analytical query processing (where queries are typically known in advance and which process huge amounts of data) and offers a productivity/performance combination not provided by existing database systems or previous query compilers in this domain. To avoid the overheads of a high-level language (e.g. complicated memory management) while maintaining well-defined abstractions, we opt for using *generative* programming [Taha and Sheard, 2000], a technique that allows for programmatic removal of

abstraction overhead through source-to-source compilation. In particular, we show how generative programming can be used to optimize *any* piece of Scala code. This property allows LegoBase to perform *whole-system* specialization and source-to-source compile *all* components, data structures and auxiliary functions used inside the query engine to efficient, low-level C code. We demonstrate how generative programming allows to *easily* implement a broad spectrum of optimizations which are difficult to achieve with existing query compilation approaches; in our approach developers need to provide just a few hundred lines of high-level code for building and optimizing the *entire* query engine. LegoBase is the first step towards providing a full analytical DBMS written in a high-level language and outperforms both a commercial in-memory database system as well as an existing query compiler.

1.3 Thesis Outline

The rest of this thesis is organized into two main parts: from Chapter 2 to Chapter 5 we focus on the architecture of the OCAS framework, while from Chapter 6 to Chapter 9 we provide more details on the design and implementation of the LegoBase query engine. More specifically:

- Chapter 2 provides a high-level overview of the OCAS synthesis framework, with particular emphasis on the software components necessary to be able to efficiently synthesize out-of-core algorithms. It also highlights the contributions of this work in more detail.
- Chapter 3 presents OCAL (Out-of-Core Algorithm Language) which OCAS uses to represent data processing algorithms. It also analyzes how OCAS automatically costs semantically equivalent OCAL programs in order to detect which one has the best performance for the provided memory hierarchy.
- Chapter 4 discusses the set of rules that transform a given program to another one with equivalent functionality that may have better performance with respect to a given memory hierarchy. We also discuss how these transformation rules are derived based on commonly known data locality principles.
- Chapter 5 concludes our discussion about OCAS with an experimental evaluation of the synthesis framework. We show that through accurate cost estimations of programs, OCAS can adapt its generated algorithms to changes in the memory hierarchy and can quickly produce optimized versions of out-of-core algorithms.
- Chapter 6 provides a high-level introduction to the LegoBase query engine. We outline how our system performs source-to-source compilation, as well as the properties that differentiate our approach from existing general-purpose compilers and previous work on query compilation.
- Chapter 7 discusses the overall system architecture of LegoBase in more detail, with particular emphasis on the interfaces provided by the optimizing compiler. We also address various special issues encountered when one performs source-to-source compilation.

- Chapter 8 presents examples of compiler optimizations in a number of domains, demonstrating 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.
- Chapter 9 concludes our discussion of the LegoBase query engine through an extensive experimental evaluation of our approach in the domain of ad-hoc, analytical query processing. We show that LegoBase can significantly outperform existing approaches, while still allowing developers to be highly productive.

Finally, in Chapter 10 we discuss related work while Chapter 11 concludes and outlines some possible future research directions.

2 Automatic Synthesis of Efficient Out-of-Core Algorithms

In this chapter, we provide a high-level overview of the OCAS framework for the automatic software synthesis of specialized out-of-core algorithms. We discuss the software components needed in order to be able to quickly navigate the search space of semantically equivalent programs and generate optimized algorithms. We also highlight the individual contributions of this work in more detail.

The next example illustrates our approach:

Example 1 The simplest way to implement a join algorithm on relations R and S is with two nested for loops:

```
for (x ← R)
  for (y ← S)
    if joinCond(x,y) then
      [⟨x,y⟩]
    else []
```

This program is an intuitive description of the programmer's intention. Let us now assume a scenario where the input is stored on a hard disk and the output is not written anywhere (e.g., it is consumed by the CPU). Then, ignoring any buffering of the hard disk and the operating system, this program transfers every tuple of R and S from the hard disk separately and hence performs at least twice as many seeks as there are tuples in R.

The efficiency of the algorithm can be significantly improved if we reduce the number of disk seeks by accessing the relations in larger contiguous blocks. Also, the semantics of the program does not change if the loops are reordered so that the outer relation is the smaller, but this further reduces the amount of seeking.

By expressing such knowledge as transformation rules, we can automatically transform the above program into one that implements these two optimizations:

```
(λ⟨R, S⟩.
  for (xBlock [k1] ← R)
    for (yBlock [k2] ← S)
      for (x ← xBlock)
        for (y ← yBlock)
          if joinCond(x,y) then
            [⟨x,y⟩]
          else [])
(if length(R) ≤ length(S) then ⟨R, S⟩ else ⟨S, R⟩)
```

When the block-size k1 of *xBlock* is maximized, this is the canonical Block Nested Loops Join typically found in traditional database textbooks (e.g. in [Ramakrishnan and Gehrke, 2002]). □

In the above example, we used the following transformation rule that turns a naive for loop into a buffered scan with block-based transfers of block size k:

```
for (x ← S) e
⇓
for (xBlock [k] ← S)
  for (x ← xBlock) e
```

This rule says that the program at the top is *functionally equivalent* to the one at the bottom and suggests that, subject to the targeted memory hierarchy, the latter is likely to be more efficient than the former. Indeed, the latter program requires less seeking on the hard disk.

OCAS explores the space of equivalent programs created by the application of such transformation rules, assigns a cost metric to each one (an estimation of the program's actual execution time), and finally selects the one with the minimum cost for the provided memory hierarchy. To realize this vision of automatic synthesis of out-of-core algorithms, we have addressed the following challenges, which at the same time form the main contributions of our approach in this domain:

Design of a new language. We have designed a high-level, domain-specific language (DSL) called OCAL (Out-of-Core Algorithm Language). The primary design goals of OCAL are (i) to be *expressive* enough for a variety of out-of-core algorithms, (ii) to be *succinct* enough to keep typical algorithms short and the search space of program synthesis manageable, and (iii) to keep the *syntax and semantics of the language simple* in order to facilitate program analysis and transformation. More concretely, to make it reasonably easy to cost programs and apply transformations, we should avoid constructs such as unrestricted recursion, mutable values, and side effects[1]. As a consequence, we avoid imperative and low-level languages such as C for program representation during synthesis and, instead, opt for using a high-level language

[1] It is long known that the optimization of low-level, imperative programs with side effects is notoriously hard [Asai et al., 1997] because the compiler has to reason about aliased mutable locations, a problem that has been shown to be intractable in general [Ramalingam, 1994].

for expressing out-of-core algorithms.

OCAL is defined as Monad Calculus on lists [Breazu-Tannen et al., 1992; Breazu-Tannen and Subrahmanyam, 1991] with a fold expression. It satisfies the aforementioned three design desiderata: (i) It is expressive, extending the power of nested relational algebra by the ability to process collections sequentially and exploiting order, which is central to capturing the essence of most out-of-core algorithms. (ii) High-level composition of expressions are represented as named OCAL function *definitions*; these can be used to keep OCAL programs short and are treated like language extensions in our synthesis system. (iii) OCAL is a simple purely functional language without side-effects in which recursion is confined to fold (and flatMap). Transformations in OCAL can be applied locally due to its functional and algebraic qualities. Finally, since full recursion is excluded and for comprehensions (functional for loops as in, say, XQuery and Scala) are straightforward to define in OCAL, even users familiar only with imperative languages can read most OCAL programs without great difficulty. It is relatively easy to map OCAL programs to imperative (C) code.

We discuss the design of OCAL in more detail in Chapter 3.

Cost Estimation and Cost Minimization. We need a systematic cost estimation framework to reason about the efficiency of OCAL programs. This requires an easily computable cost measure for evaluating the performance of each program that we explore. This measure is a function of the algorithm, the memory hierarchy and statistics about the input.

One contribution of this thesis is the demonstration that in the domain of out-of- core algorithms it is possible to efficiently and automatically perform such estimation and that the estimates are predictive enough to differentiate more efficient from less efficient algorithms on a given memory hierarchy.

There are two orthogonal aspects of cost estimation: structural program transformations and parameter selection. For the former, we use a breadth-first search strategy to explore the space of structurally different programs, and we use constants derived from the given memory hierarchy to build a cost function. To perform the latter, each program is also parameterized with values such as sizes of blocks and buffers. In Example 1, k1 and k2 are two such parameters. We use our cost estimation rules to characterize the running time estimate as a (possibly non-linear) function of those parameters. We have also implemented the non-linear optimization solver described in [Liuzzi et al., 2010] to tune the values of parameters in order to minimize the cost estimate. We have found this strategy to be computationally feasible and to yield efficient programs for various memory hierarchies.

Developers also have the ability to override the costing formulas used for any language expression. This allows developers to tune the costing engine for more precise cost estimates and is particularly important for the costing of *OCAL* definitions.

Costing of OCAL programs is discussed in Section 3.1.

Development of a program synthesizer. Based on the language and the costing framework, we have implemented OCAS, the Out-of-Core Algorithm Synthesizer. The input to OCAS consists of two orthogonal items: (1) a naive, memory-oblivious algorithm given in OCAL; and (2) the structure and parameters of the memory hierarchy and storage devices.

From this input, OCAS automatically derives efficient algorithms that have the same functional behavior as the initial specification algorithm, but whose performance is tuned to the given memory hierarchy. To do so, our tool uses a library of transformation rules, which we discuss in Chapter 4. This set of rules is derived from design and data locality principles commonly used in efficient out-of-core algorithms. Finally, OCAS then generates C code out of the optimized algorithm, using an OCAL-to-C code generator.

Our approach also necessitates a technique, presented in Section 3.1.1, for describing memory hierarchies, such that device properties can be expressed in a sufficiently abstract manner. We use this technique for expressing a number of characteristics and details of usage, such as the speed of read and write operations, different speeds of sequential and random data accesses, and block-wise access to data.

We use OCAS to derive C code for algorithms such as Block Nested Loops Join, GRACE Hash Join and the External Merge-Sort in their canonical textbook forms starting from naive specifications of joins and sorting. We also present examples of algorithms specialized for memory hierarchies that are not yet found in textbooks, such as a join algorithm for flash drives. We present these case studies and their evaluation in Chapter 5.

Finally, because OCAS operates automatically, it is possible to deploy it even in environments where the system configuration changes dynamically, such as cloud infrastructures. OCAS can be used at installation time to adapt a piece of data management software to a computer, or at deployment time via just-in-time compilation to make the best use of fresh information on the availability of system resources.

Providing an extensible architecture. Extensibility is an important property of the design of OCAS. Developers should be able to easily adapt OCAS as new hardware platforms become available and new algorithms are proposed. The library of program transformation rules of OCAS can be extended to implement new ways of using data locality considerations to create better algorithms. Furthermore, we can create named definitions in OCAL that can subsequently be used like new language operations. For each such definition, we can extend OCAS by matching code generator and cost function plugins to allow the synthesizer to make use of a particularly efficient implementation of that new language feature. Thus, definitions (in conjunction with code generator and cost function extensions) do *not* increase the expressiveness of the language but the efficiency of the algorithms created.

To summarize, we believe that the design of OCAS provides the so far missing methodology for designing efficient out-of-core algorithms, and even automatizes algorithm creation. Develop-

ers may need to make use of the extensibility of OCAS to adapt to unforeseen developments, but there is no need to “reinvent the wheel”; the basic machinery of OCAS will remain unchanged. This machinery, along with its evaluation, is presented in the next three chapters of this thesis.

3 OCAL: The Out-of-Core DSL

In this chapter, we present OCAL (Out-of-Core Algorithm Language) which OCAS uses to represent data processing algorithms. The design of OCAL provides enough expressive power to describe commonly used algorithms ranging from traditional relational algebra operators, such as selection, projection, and joins, to additional aspects of data processing such as sorting. At the same time, OCAL also allows easy application of costing and transformation rules as it will be discussed in more detail in Section 3.1 and Chapter 4, respectively.

The base language. OCAL extends Monad Calculus on lists [Breazu-Tannen et al., 1992; Breazu-Tannen and Subrahmanyam, 1991] with a fold expression. Consequently, the proposed language is more expressive than nested relational calculus. Starting from a totally ordered set D of atomic values that includes integers, booleans and strings, values are built inductively from D using list and tuple construction as formalized by the following grammar:

$$\tau ::= \mathsf{D} \mid \langle \tau, \ldots, \tau \rangle \mid [\tau]$$

The typing rules of the language are presented in Figure 3.1 where $e, e_1, \ldots, e_n$ range over expressions, x over variables, c over primitive constants and $\tau, \tau_1, \ldots, \tau_n$ over types. Each value x is assigned a $\mathsf{Type}(x)$ and, similarly, constants c have a type $\mathsf{Type}(c)$. Functions in OCAL are of type $\tau_1 \to \tau_2$ where τ_1 and τ_2 are value types. As an example, the type of a join operator for two binary relations on D is:

$$\langle [\langle \mathsf{D}, \mathsf{D} \rangle], [\langle \mathsf{D}, \mathsf{D} \rangle] \rangle \to [\langle \mathsf{D}, \mathsf{D}, \mathsf{D}, \mathsf{D} \rangle]$$

In the same figure, p ranges over primitive functions including boolean connectives ($\wedge$, $\vee$, $\neg$); equality of values of various types and comparison of basic data types D ($==$, $\leq$, $\geq$); a list union operator $\sqcup$, and further functions on tuples of values of D that only require a constant amount of memory (e.g., arithmetic operations). IType and OType are the input and output types of p, respectively.

$$\frac{}{x : \mathsf{Type}(x)} \qquad \frac{}{c : \mathsf{Type}(c)} \qquad \frac{}{p : \mathsf{IType}(p) \to \mathsf{OType}(p)} \qquad \frac{e : \tau}{\lambda x.e : \mathsf{Type}(x) \to \tau}$$

$$\frac{e_1 : \tau_2 \to \tau_1 \quad e_2 : \tau_2}{e_1 e_2 : \tau_1} \qquad \frac{e_1 : \tau_1 \ldots\ e_n : \tau_n}{\langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \qquad \frac{e : \langle \tau_1, \ldots, \tau_n \rangle \quad i : \mathsf{Int}}{e.i : \tau_i}$$

$$\frac{e : \tau}{[e] : [\tau]} \qquad \frac{e : \tau_1 \to [\tau_2]}{\mathsf{flatMap}(e) : [\tau_1] \to [\tau_2]} \qquad \frac{c : \mathsf{Bool},\ e_1 : \tau,\ e_2 : \tau}{\mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \frac{c : \tau_2,\ f : \langle \tau_2, \tau_1 \rangle \to \tau_2}{\mathsf{foldL}(c, f) : [\tau_1] \to \tau_2}$$

Figure 3.1 – The type system of OCAL.

The addition of a fold expression to Monad Calculus adds the ability to express sequential computation, which is essential for data processing algorithms including sorting. Folding from the left – foldL(c,f) – encodes a restrictive recursion pattern, an iterative application of the binary function f to elements of an input list and the result of a previous iteration. When using an infix operator $\oplus$, foldL is defined as follows:

$$\mathsf{foldL}(c,\oplus)([v_1, v_2, \ldots, v_n]) = (\cdots((c \oplus v_1) \oplus v_2) \oplus \cdots \oplus v_n)$$

Next, we discuss the extensibility and code generation properties of OCAL in more detail.

Extensibility. Developers have the ability to provide additional *definitions*, expressed in terms of the base language. Figure 3.2 presents schemes of definitions, where we use symbol __ as a placeholder for an unused function argument. We make the following observations regarding these definitions.

The head and tail constructs are used to extract elements from a list. They are undefined when the list is empty. Aggregate functions are also expressed as definitions in OCAL – for example avg calculates the average value of the elements of a given list. Other aggregate functions (e.g. sum, min, max) can be defined similarly. length returns the size of a list, as expected.

The unfoldR function iterates over a tuple of n lists simultaneously. In every iteration the n-ary function f is applied, which computes part of the output and removes at most one element from the beginning of each list. The computation terminates when all lists are empty, a condition that is satisfied for a number of iterations smaller than the sum of the lengths of the lists. The result of each iteration is appended to the intermediate result from the previous iteration starting with an empty list, thus constructing the output from left to right. The direction of the construction is the reason for the "R" in unfoldR. We can use unfoldR to express the merging of two sorted lists as unfoldR(mrg) and the zipping of n lists as unfoldR(z).

The treeFold construct generates a tree-shaped bracketing for the applications of a function f which takes k arguments. This construct is used to represent divide and conquer strategies, as found in e.g. Merge-Sort. It uses a queue to store the initial elements and the intermediate results. For example, for a ternary f we have:

head : $[\tau] \to \tau$
$:= \lambda l.\text{foldL}(\ \langle \text{true}, 0\rangle,\ \lambda\langle a, x\rangle.$
$\quad \text{if } a.1 \text{ then} \langle \text{false}, x\rangle$
$\quad \text{else } a$
$)(l).2$

tail : $[\tau] \to \tau$
$:= \lambda l.\text{foldL}(\ \langle \text{true}, []\rangle,\ \lambda\langle a, x\rangle.$
$\quad \text{if } a.1 \text{ then } \langle \text{false}, []\rangle$
$\quad \text{else } \langle \text{false}, a.2 \sqcup [x]\rangle$
$)(l).2$

avg : [D] → D
$:= (\lambda x.(x.1/x.2))($
$\quad \text{foldL}(\ \langle 0, 0\rangle,\ \lambda\langle a, x\rangle.\langle a.1 + x, a.2 + 1\rangle)$

z : $\langle [\tau_1], \ldots, [\tau_n]\rangle \to \langle [\langle \tau_1, \ldots, \tau_n\rangle], \langle [\tau_1], \ldots, [\tau_n]\rangle\rangle$
$:= \lambda\langle l_1, \ldots, l_n\rangle.$
$\langle [\langle \text{head}(l_1), \ldots, \text{head}(l_n)\rangle], \langle \text{tail}(l_1), \ldots, \text{tail}(l_n)\rangle\rangle$

length : $[\tau] \to \text{Int}$
$:= \text{foldL}(0,\ \lambda\langle sum, \_\_\rangle.sum + 1)$

funcPow$[1](f) : \langle \tau_1, \tau_2\rangle \to \tau_3$
$:= f$

unfoldR$(f) : \langle [\tau_1], \ldots, [\tau_n]\rangle \to [\tau_r]$
$:= \lambda seed.\text{foldL}(\langle [], seed\rangle,\ \lambda\langle a, \_\_\rangle.$
$\quad \text{if } a.2 == \langle [], \ldots, []\rangle \text{ then} \langle a.1, \langle [], \ldots, []\rangle\rangle$
$\quad \text{else } \langle a.1 \sqcup f(a.2).1, f(a.2).2\rangle$
$)(seed.1 \sqcup \ldots \sqcup seed.n)$

funcPow$[k+1](f) : \langle \tau_1, \ldots, \tau_{2^k}\rangle \to \tau_r$
$:= \lambda\langle a.1, \ldots, a.2^{k+1}\rangle.$
$\quad f(\text{funcPow}[k](f)(a.1, \ldots, a.(2^k)),$
$\quad \text{funcPow}[k](f)(a.(2^k + 1), \ldots, a.2^{k+1}))$

treeFold$[k](c, f) : [\tau_1] \to [\tau_2]$
$:= \lambda seed.\text{foldL}(\ \langle [], seed\rangle,\ \lambda\langle a, \_\_\rangle.$
$\quad \text{if length}(a.2) == 1 \wedge a.1 == [] \text{ then}$
$\quad\quad a$
$\quad \text{else if length}(a.1) == k \text{ then}$
$\quad\quad \langle [], a.2 \sqcup f(a.1)\rangle$
$\quad \text{else if tail}(a.2) \mathrel{!}= [] \text{ then}$
$\quad\quad \langle a.1 \sqcup \text{head}(a.2), \text{tail}(a.2)\rangle$
$\quad \text{else } \langle a.1 \sqcup \text{head}(a.2), [c]\rangle$
$)(seed \sqcup seed)$

mrg : $\langle [\tau], [\tau]\rangle \to \langle [\tau], \langle [\tau], [\tau]\rangle\rangle$
$:= \lambda\langle l_1, l_2\rangle.$
$\quad \text{if length}(l_1) == 0 \wedge \text{length}(l_2) == 0 \text{ then}$
$\quad\quad \langle [], \langle [], []\rangle\rangle$
$\quad \text{else if length}(l_1) == 0 \text{ then}$
$\quad\quad \langle [\text{head}(l_2)], \langle [], \text{tail}(l_2)\rangle\rangle$
$\quad \text{else if length}(l_2) == 0 \text{ then}$
$\quad\quad \langle [\text{head}(l_1)], \langle \text{tail}(l_1), []\rangle\rangle$
$\quad \text{else if head}(l_1) < \text{head}(l_2) \text{ then}$
$\quad\quad \langle [\text{head}(l_1)], \langle \text{tail}(l_1), l_2\rangle\rangle$
$\quad \text{else } \langle [\text{head}(l_2)], \langle l_1, \text{tail}(l_2)\rangle\rangle$

for $(x\ [k] \leftarrow \text{R})\ e : [\tau_1] \to [\tau_2]$
$:= \text{foldL}(\ \langle [], []\rangle,\ \lambda\langle a, x\rangle.$
$\quad \text{if length}(a.1) - 1 == \text{k then}$
$\quad\quad \langle [], a.2 \sqcup f(a.1 \sqcup x)\rangle$
$\quad \text{else } \langle a.1 \sqcup x, a.2\rangle$
$)$

partition : $[\langle \tau_1, \ldots, \tau_n\rangle] \to [\tau_1, [\langle \tau_2, \ldots, \tau_n\rangle]]$
$:= \text{foldL}(\ [],\ \lambda\langle ps, x\rangle.$
$\quad (\lambda nps.\text{if } nps.1 \text{ then } nps \text{ else } ps \sqcup \langle x.1, [x.2]\rangle)$
$\quad (\text{foldL}(\ \langle \text{false}, []\rangle,\ \lambda\langle nps, xs\rangle.$
$\quad\quad \text{if } xs.1 == x.1 \text{ then } \langle \text{true}, nps \sqcup [xs \sqcup [x.2]]\rangle$
$\quad\quad \text{else } \langle \text{false}, nps \sqcup [xs]\rangle$
$\quad)(ps)))$

Figure 3.2 – Examples of definitions in OCAL.

treeFold[3]$(c,f)([v_1,v_2,\ldots,v_6]) = f(f(v_1,v_2,v_3),f(v_4,v_5,v_6),c)$

The functional for loop returns a value of a list type, which is the concatenation of list-typed values computed by its body at each iteration. This is similar to the for loop in XQuery and to flatMap/ext in other languages [XQuery; Breazu-Tannen et al., 1992]. The parameter k concerns blocking and is explained in detail in Chapter 4. Whenever omitted, its value is assumed to be equal to 1.

For a given fixed k, the funcPow[k](f) definition scheme yields a definition to obtain a 2^k-ary function using multiple applications of a binary function f. Stated otherwise, the funcPow[k+1](f) is a function of 2^{k+1} inputs where the construct is recursively applied to the left and right half of the 2^{k+1} input tuple (each of size 2^k) as funcPow[k](f).

Finally, the partition function groups a set of tuples by their first elements. The function iterates over the tuples of a list, and uses the first element of each tuple as a key to map it to a partition. The number of partitions is not known in advance but their set is built progressively: if there is no partition for some key, a new empty partition is created.

Generating C code from OCAL. As we mentioned earlier, OCAS generates C code out of programs written in OCAL by translating each expression to an appropriate sequence of C statements. We choose C as the target language since it is currently widely used in database systems development. By default, OCAS expands definitions and generates code for each individual expression of the base language.

In order to increase efficiency, developers can overwrite the default code generators for expressions and definitions using generator plugins. OCAS contains efficient generator plugins for all definitions in Figure 3.2. For instance, our partition definition as shown in Figure 3.2 has $O(n^2)$ complexity, even though there exists a linear implementation with the same semantics. By providing a code generator plugin for this construct, the linear implementation can be used. Similarly, the definitions of the head and length functions have linear time complexity, even though there exist suitable implementations for constant time execution. Finally, because the inner function of unfoldR can only access the head of the lists and the output is produced sequentially, we can transfer *blocks* of elements at once, as we present in Chapter 4.

Next, we analyze how we can accurately perform cost estimation of OCAL programs.

3.1 Automated Cost Estimation of OCAL programs

Sufficiently accurate cost estimation of OCAL programs is essential because it is used by OCAS to compare programs in terms of efficiency. In the domain of out-of-core algorithms, we are mainly interested in costs introduced by moving data around the memory hierarchy. Thus, we currently neglect the actual computation cost of a program in our system. Instead, we opt for modeling only the two aspects of data transfers: initiating the transfer and actually

transferring the requested data.

This section provides a stepwise description of the communication cost computation. First, we describe how we model memory hierarchies in our system. Second, we analyze *how* to compute the result size of each OCAL expression. This is needed since the input typically represents structured data, and thus we need to estimate not only the total size, but also the sizes of the nested components that may be separately used in subcomputations. Third, we present *when* data transfers are introduced in our cost model and analyze how the cost estimator separately computes two aspects of data transfers in order to provide the final cost formula which takes into account the characteristics of the memory hierarchy. Finally, we briefly discuss the extensibility of the costing in OCAS.

Note that the costing of a program in OCAS does not require to actually run the program. This is important, since actual execution may be very costly. This aspect enables our methodology to be used to compare a large number of programs efficiently, which is essential when exploring variations of a program by applying transformations. We discuss transformation rules in greater detail in Chapter 4 and in this section we focus on how to cost one single program.

3.1.1 Memory and Storage Model

Automated transformations in OCAS are driven by a model of the memory hierarchy. For this purpose, the developer must specify a tree-shaped hierarchy where every node represents a hardware component able to store data and an edge represents the ability to transfer data between two nodes. For example, a basic memory hierarchy consists of a main memory node at the root with a single child node representing the hard disk.

Every node is attributed a set of properties that provide information about its characteristics. This is merely an abstract description of each node's characteristics, since precise modeling of the architectural and physical attributes of nodes is beyond the scope of this work. Examples of such properties for a number of devices are presented in Figure 3.3.

Our model makes three assumptions. First, events between distinct hierarchy levels do not interfere with each other (we assume DMA transfers). Second, there exists a single processing unit which executes all computation and can only access data that is stored at the root node of the tree. Third, we assume synchronous I/O and that the hardware properties, such as the throughput and seek time of hard disks, remain constant.

For a program, the location of the input data, as well as the output node, must both be specified. If the output node is not set, we assume that the output is consumed by the CPU. Each data value resides in a node. In order to perform computation on those values, they must be transferred to the root node. Thus, for a given program, OCAS has to infer transfers for the set of values that have to be accessible by the processing unit throughout the execution of the program. For our basic memory hierarchy presented above, all data have to be transferred to RAM before performing any operations on them.

Size. The size of the device. This property must be set for all nodes.

Pagesize. The data at this node must be accessed by pages of this size. If it is possible to address every byte individually then pagesize = 1.

Maximum length of a write sequence (maxSeqW). The maximum amount of data that it is possible to write in a sequence, using a single I/O request. For flash drives this is equal to the erase block size.

Maximum length of a read sequence (maxSeqR). The maximum amount of data that it is possible to read in a sequence, using a single I/O request.

Edge properties: Weights of InitCom[m1 → m2] and UnitTr[m1 → m2] cost events, where m1 and m2 are nodes of the memory hierarchy.

Figure 3.3 – Examples of abstract properties for a number of devices of a memory hierarchy.

Moving a data value v from one hierarchy level to another induces *costs*. We leave the specifics of cost computation of OCAL expressions for Section 3.1.4, but we note here that the final cost depends on the paths actually used for data transfers. The act of transferring data concerns not only the input and the output but intermediate results as well.

In order to model the cost of moving data along an edge in the memory hierarchy, each edge has two cost metrics associated with it. Using different costs for different edges enables more accurate cost estimation. First, we consider the cost of initiating a transfer between the two hierarchy nodes (InitCom event). If either of the nodes is a hard disk, this corresponds to a seek in our model. Similarly, in order to transfer data to a flash drive, a block has to be erased before data can be written. The second metric is the cost of transferring a unit of data between the two hierarchy levels (UnitTr event). If the developer chooses to ignore certain cost events, he can set their value to zero. This allows our system to, for example, ignore the cost of InitCom for RAM when considering I/O intensive workloads. Both costs can be collected either from the device specifications or using standard tools like e.g. Seeker [Seeker] for hard disk seeks. We follow this approach in our evaluation in Chapter 5.

Because we model memory hierarchies as trees whose leaves are storage devices and whose root is the fastest level of the hierarchy, we cannot model, say, general parallel computation. We leave the extensive hardware modeling for future work, with the goal of ultimately being able to automatically infer program transformation rules and cost functions from the hardware description. Still, it is our experience that the current memory model is adequate to explore a variety of interesting algorithms.

3.1.2 Estimating the Result Size of Expressions

Given that OCAL programs are compositions of expressions, and that each expression may increase the amount of output, we must estimate the result size of every expression in OCAL. To do that, we introduce the notion of *annotated types*, which annotate lists types with cardi-

$$\mathsf{card}([\alpha]^x) := x \qquad \mathsf{elem}([\alpha]^x) := \alpha \qquad \mathsf{size}([\alpha]^x) := x \cdot \mathsf{size}(\alpha) \qquad \mathsf{size}(c) := c$$

$$\mathsf{size}(\langle \alpha_1, \ldots, \alpha_n \rangle) := \mathsf{size}(\alpha_1) + \ldots + \mathsf{size}(\alpha_n) \qquad \mathsf{R}(\Gamma, x) := \Gamma(x) \qquad \mathsf{R}(\Gamma, [e]) := [\mathsf{R}(\Gamma, e)]^1$$

$$\mathsf{R}(\Gamma, c) := \mathsf{sizeof}(c) \qquad \mathsf{R}(\Gamma, e.i) := \mathsf{R}(\Gamma, e).i \qquad \mathsf{R}(\Gamma, e_1 \sqcup e_2) := \mathsf{R}(\Gamma, e_1) + \mathsf{R}(\Gamma, e_2)$$

$$\mathsf{R}(\Gamma, \langle e_1, \ldots, e_n \rangle) := \langle \mathsf{R}(\Gamma, e_1), \ldots, \mathsf{R}(\Gamma, e_n) \rangle \qquad \mathsf{R}(\Gamma, (\lambda x.e_1)(e_2)) := \mathsf{R}(\Gamma \cup \{x \mapsto \mathsf{R}(\Gamma, e_2)\}, e_1)$$

$$\mathsf{R}(\Gamma, \mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) := \max(\mathsf{R}(\Gamma, e_1), \mathsf{R}(\Gamma, e_2))$$

$$\mathsf{R}(\Gamma, \mathsf{for}(x\ [k] \leftarrow e_1)\ e_2) := \frac{\mathsf{card}(\mathsf{R}(\Gamma, e_1))}{k} \cdot \mathsf{R}(\Gamma \cup \{x \mapsto [\mathsf{R}(\Gamma, \mathsf{elem}(e_1))]^k\}, e_2))$$

$$\mathsf{R}(\Gamma, \mathsf{foldL}(c, \lambda\langle a, x\rangle.e_1)(e_2)) :=$$

$$\mathsf{R}(\Gamma, c) + \mathsf{card}(\mathsf{R}(\Gamma, e_2))\Big(\mathsf{R}(\Gamma \cup \{a \mapsto \mathsf{R}(\Gamma, c), x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) - \mathsf{R}(\Gamma, c)\Big)$$

Figure 3.4 – Data size estimation rules for every expression of OCAL.

nalities. The corresponding grammar is:

$$\alpha ::= [\alpha]^x \mid \langle \alpha_1, \ldots, \alpha_n \rangle \mid c$$

An annotated type α is either a list of form $[\alpha]^x$ where x is the cardinality, a tuple of annotated types or a constant size c. This notation allows us to represent the size of values while retaining their structure. It is worth mentioning that the length of a list is not restricted to integer constants but can be described by an arithmetic expression containing variables. As an example of an annotated type, $\langle [[1]^y]^x, [\langle 1, 1\rangle]^z\rangle$ represents a tuple composed of a list of lists and a list of tuples. By using variables we can express the result size as a function of the input sizes and other parameters without having to recompute the cost of a program every time the size of its inputs or other parameters change.

By using annotated types, the result size of expressions can be then estimated as shown in Figure 3.4. In what follows, we sometimes write $x \cdot [b]^y$ to denote $[b]^{x \cdot y}$. The recursive function R defines the result size as an annotated type for an expression in a context Γ, which is a set that maps symbols to annotated types. This context is extended every time new symbols are referenced. In order to turn the estimate of a result size into a single arithmetic expression, we define the function size which turns an annotated type to an integer-valued arithmetic expression representing the size of the annotated type in bytes. In addition, we define card and elem to extract information about lists. This is necessary, since as we mentioned, we want to be able to operate on nested data. Since function definitions do not produce any results until they are applied to a value, in our costing we assume that all of them are matched with corresponding function applications. The cost of the flatMap construct is the same as that of for with k set to 1.

Observe that we perform *worst-case* analysis of the result size of each expression. For instance,

for nested lists, we take the maximum of the lengths of the inner lists. This design choice may lead to overestimation of result sizes, e.g. in the case of if-then-else, the branch that gives the largest result may not be the one that will actually be taken during execution. However, as we show in our evaluation, even with this overestimation, OCAS can still differentiate more efficient programs from less efficient ones, with respect to the given memory hierarchy. Finally, we also allow the programmer to annotate any expression with a custom result size estimate. This may be needed since the static rules that OCAS uses for data size estimation may not capture specific algorithm semantics. A fold which produces a very small output in its last iteration, but very large outputs in all others is one example where these annotations allow programmers to explicitly express the intention of their algorithm.

3.1.3 Determining Data Transfer Occurrences

OCAS models data transfers implicitly: whenever the execution context is extended with a new value, we account for an appropriate amount of transfers for this value. After modeling the transfer, this value is then considered to be in its new location. Furthermore, as soon as a value is not in the context anymore, it does not consume space for the hierarchy level it belonged to. This means that the next time this value is needed, it has to be brought again from the input device, all the way up to the processing node. By implicitly modeling data transfers, we enable separation between a program and its execution environment (memory hierarchy). This alleviates the need for programmers to annotate where intermediate values are stored throughout the execution of a program.

We use the following notation and semantics for data transfers. First, values are transferred from a hierarchy node m_s, where they originally reside, to a memory node m_d. In order to simplify costing of a program, we assume that data transfers happen only between adjacent memory nodes (m_s is directly connected to m_d in the tree). Furthermore, if m_d is the root node, then an expression will be executed to process the fetched data, thus producing an output written at a node m_o, which has to be a child of m_d, possibly different from m_s.

Finally, data transfers between adjacent hierarchy levels are constrained by the physical size of the participating nodes. Given that modeling replacement algorithms at each level of the memory hierarchy is a very complicated task, we choose a simpler solution. We use dedicated space for input (b_{in}) and output (b_{out}) buffers at each level, per value, so that their combined size does not exceed the size of the specific level. These buffers determine the amount of transfers necessary to process each value and will be utilized by the transformation rules presented in Chapter 4. Furthermore, when the output buffer is filled, it is completely evicted to the output memory level. This is in accordance with the fact that we perform worst-case analysis. Choosing good values for input and output buffer sizes is a critical aspect of designing high-performance out-of-core algorithms. It is also a non-trivial task for developers, since choosing locally optimal solutions at each node may not give a globally optimal solution for the whole hierarchy. Thus, the automation that our system provides in that respect is very

Expression e	Cost of InitCom events $\mathsf{C}(\Gamma, e)$	Cost of UnitTr events $\mathsf{T}(\Gamma, e)$
$(\lambda x.e_1)(e_2)$	$\mathsf{C}(\Gamma \cup \{x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) + \mathsf{C}(\Gamma, e_2)$ $+\mathsf{size}(\mathsf{R}(\Gamma, e_2))\ \mathsf{InitCom}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{InitCom}[m_d \to m_o]$	$\mathsf{T}(\Gamma \cup \{x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) + \mathsf{T}(\Gamma, e_2)$ $+\mathsf{size}(\mathsf{R}(\Gamma, e_2))\ \mathsf{UnitTr}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{UnitTr}[m_d \to m_o]$
$(\mathsf{for}(x\ [\mathsf{k}] \leftarrow e_2)\ e_1)$	$\frac{\mathsf{card}(\mathsf{R}(\Gamma, e_2))}{k}\mathsf{C}(\Gamma \cup \{x \mapsto [\mathsf{R}(\Gamma, \mathsf{elem}(e_2))]^k\}, e_1)$ $+\mathsf{C}(\Gamma, e_2) + \frac{\mathsf{size}(\mathsf{R}(\Gamma, e_2))}{k}\mathsf{InitCom}[m_s \to m_d]$ $+\frac{\mathsf{size}(\mathsf{R}(\Gamma, e))}{b_{out}}\ \mathsf{InitCom}[m_d \to m_o]$	$\frac{\mathsf{card}(\mathsf{R}(\Gamma, e_2))}{k}\mathsf{T}(\Gamma \cup \{x \mapsto [\mathsf{R}(\Gamma, \mathsf{elem}(e_2))]^k\}, e_1)$ $+\mathsf{T}(\Gamma, e_2) + \mathsf{size}(\mathsf{R}(\Gamma, e_2))\ \mathsf{UnitTr}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{UnitTr}[m_d \to m_o]$
$(\mathsf{foldL}(c, \lambda\langle a, x\rangle.e_1))(e_2)$	$\sum_{i=0}^{\mathsf{card}(\mathsf{R}(\Gamma, e_2))-1} \mathsf{C}\Big(\Gamma \cup \{a \mapsto i \cdot (\mathsf{R}(\Gamma \cup$ $\{a \mapsto \mathsf{R}(\Gamma, c), x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) - \mathsf{R}(\Gamma, c)),$ $x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1\Big) + \mathsf{C}(\Gamma, e_2)$ $+(\mathsf{size}(\mathsf{R}(\Gamma, c)) + \mathsf{size}(\mathsf{R}(\Gamma, e_2)))\ \mathsf{InitCom}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{InitCom}[m_d \to m_o]$	$\sum_{i=0}^{\mathsf{card}(\mathsf{R}(\Gamma, e_2))-1} \mathsf{T}\Big(\Gamma \cup \{a \mapsto i \cdot (\mathsf{R}(\Gamma \cup$ $\{a \mapsto \mathsf{R}(\Gamma, c), x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) - \mathsf{R}(\Gamma, c)),$ $x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1\Big)$ $+\mathsf{T}(\Gamma, e_2) + (\mathsf{size}(\mathsf{R}(\Gamma, c))$ $+\mathsf{size}(\mathsf{R}(\Gamma, e_2)))\ \mathsf{UnitTr}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{UnitTr}[m_d \to m_o]$

Table 3.1 – Rules for computing the cost of InitCom and UnitTr events for various OCAL functions.

helpful to developers.

3.1.4 Estimating Cost Events

The core of cost computation concerns estimating the cost of InitCom and UnitTr transfer events occurring between adjacent nodes. OCAS estimates these events independently. Table 3.1 shows how our system counts the amount of data transferred for various kinds of functions. For function definitions (along with their applications), the size of the argument determines how many bytes are transferred from m_s to m_d. The size of the result tells how many bytes are written out. In addition, as flatMap executes its inner function for every element, we have to multiply the number of the elements caused by this function by the length of the list. For foldL the situation is very similar but we also have to take into account that c has to be transferred to m_d as well.[1] Every expression other than function application basically just aggregates the number of events of its subexpressions. Calculating the amount of InitCom events is similar to computing the amount of data transferred. The total cost is then found if we add up the two separate costs, which gives a single expression depicting the cost of a program as a function of various parameters like block and input sizes.

We end this section with two remarks. First, our system also allows the developer to define custom costs for definitions by extending the mechanisms for counting events and estimating result sizes with special cases. This feature allows to specify tighter bounds for special cases using the developer's expertise. If the developer does not specify a cost formula for his definitions, OCAS extends each definition and costs its inner expressions in order to get a cost estimation metric. Second, observe that when the target memory hierarchy changes, the costing formulas are changed accordingly, based on the above analysis. This may make a different set of transformation rules applicable and may, as a result, generate a different program as output.

3.2 Putting it all together – An example of automatic synthesis

Figure 3.5 presents an example that illustrates the costing methodology introduced in the previous section. It shows the different steps carried out by the cost estimation engine for every expression of a block nested loop join that reads two relations R and S from HDD into RAM, joins them there and finally writes the result back to HDD. We make the following observations.

Initially, the context Γ is empty. As discussed, starting from top to bottom, expressions lookup their context for any values they may reference, and they accordingly extend it whenever they define new ones. In this case, the first for loop extends the context with symbols R and xB. Then, the current context is passed as a parameter to the nested for expression, and the

[1] The presented cost function is simplified and adapted to our examples. The general cost function used in practice by OCAS is more complicated.

Expression	Context	Result size	UnitTr $m_{\text{HDD}} \rightarrow m_{\text{RAM}}$	UnitTr $m_{\text{RAM}} \rightarrow m_{\text{HDD}}$	InitCom $m_{\text{HDD}} \rightarrow m_{\text{RAM}}$	InitCom $m_{\text{RAM}} \rightarrow m_{\text{HDD}}$
for $(xB\ [k_1] \leftarrow \mathsf{R})$	$\Gamma_1 = \mathsf{R} \mapsto [1]^x, xB \mapsto [1]^{k_1}$	$[\langle 1,1\rangle]^{x \cdot y}$	$x + \frac{x}{k_1} y$	$2xy$	$x/k_1 + \frac{xy}{k_1 k_2}$	$2xy/k_o$
for $(yB\ [k_2] \leftarrow \mathsf{S})$	$\Gamma_2 = \Gamma_1 \cup \mathsf{S} \mapsto [1]^y$, $yB \mapsto [1]^{k_2}$	$[\langle 1,1\rangle]^{k_1 \cdot y}$	y	$2k_1 y$	y/k_2	$2k_1 y/k_o$
for $(x \leftarrow \mathrm{xB})$	$\Gamma_3 = \Gamma_2 \cup x \mapsto 1$	$[\langle 1,1\rangle]^{k_1 \cdot k_2}$	0	$2k_1 k_2$	0	$2k_1 k_2/k_o$
for $(y \leftarrow yB)$	$\Gamma_4 = \Gamma_3 \cup y \mapsto 1$	$[\langle 1,1\rangle]^{k_2}$	0	$2k_2$	0	$2k_2/k_o$
if joinCond(x,y)	Γ_4	$[\langle 1,1\rangle]^1$	0	0	0	0
then $[\langle x,y\rangle]$	Γ_4	$[\langle 1,1\rangle]^1$	0	0	0	0
else []	Γ_4	0	0	0	0	0

Figure 3.5 – Costing of an example of joining two unary relations R and S of type [Int]. The hierarchy has two nodes, an HDD and a RAM (root node), and we assume that the size of Int is 1. k_o is the size of the output buffer.

context is recursively extended at each step, whenever new values are first referenced.

The result size of the top level expression is built in a similarly recursive fashion, by first evaluating the result size of primitive expressions of the language (using the result size estimation formulas of Figure 3.4) and then composing the total result size bottom-up. Observe that, as discussed, OCAS performs worst-case analysis for some expressions. In the case of this example, OCAS will estimate the result size of the if−then−else as the maximum result size of the two branches.

Finally, for the calculation of InitCom and UnitTr events we use the costing rules listed in Table 3.1. Notice that, for the last five expressions of the example figure, there is no transfer initiated for the input, as all values are already existing in the top-level RAM node of the hierarchy. In other words, m_s is the same as m_d and, thus, both InitCom$[m_S \rightarrow m_d]$ and UnitTr$[m_s \rightarrow m_d]$ are zero for these expressions. In contrast, when costing the first two for loops, we must take into account that the input needs to be transfer from HDD to RAM, thus producing the reported InitCom and UnitTr costs. A similar analysis is true for the output of this example program as well.

In the next chapter, we discuss the transformation rules in more detail, and show how they can generate a better program based on the cost formulas we presented here.

4 Program Transformation Rules

In the previous chapter, we discussed how to estimate the cost of an OCAL program based on the amount of data transfers it performs. Since it is rarely possible to determine analytically whether the application of a rule results in a performance improvement, we opted for cost-based optimization rather than using a deterministic recipe for obtaining efficient programs. OCAS exhaustively searches the space of equivalent programs, estimates the cost of each and then selects one with the best performance. In this chapter, we discuss the set of rules that transform a given program to another one with equivalent functionality that may have better performance with respect to a given memory hierarchy. For example, consider the following sequence of equivalent join algorithms between relations R and S of type list of tuples:

```
for (x ← R)
  for (y ← S)
    if joinCond(x,y) then [⟨x,y⟩] else []

⇓ [rule apply−block applied twice, for R and S respectively]

for (xBlock [k1] ← R) for (x ← xBlock)
  for (yBlock [k2] ← S) for (y ← yBlock)
    if joinCond(x,y) then [⟨x,y⟩] else []

⇓[rules swap−iter and seq−ac]

for (xBlock [k1] ← R) for[HDD⇝RAM] (yBlock [k2]← S)
  for (x ← xBlock) for (y ← yBlock)
    if joinCond(x,y) then [⟨x,y⟩] else []

⇓[rule order−inputs]

(λ⟨R, S⟩.for (xBlock [k1] ← R) for[HDD⇝RAM] (yBlock [k2]←S)
  for (x ← xBlock) for (y ← yBlock)
    if joinCond(x,y) then [⟨x,y⟩] else [])
(if length(R) ≤ length(S) then ⟨R, S⟩ else ⟨S, R⟩)
```

The first program is a naive implementation of a Nested Loops Join algorithm that issues a disk read every time it accesses a tuple from either of the two relations. The final program is a Block Nested Loops Join that uses the smaller relation in the outer loop. Every step in the derivation is annotated with a transformation rule presented in Section 4.2 and we assume that all programs discard the output. Then, the final program has a smaller cost metric than all of the intermediate programs. Thus, OCAS will select it as the final, optimized out-of-core algorithm, out of which the C code will be generated. Next, we discuss how we derive our collection of transformation rules from fundamental design principles of out-of-core algorithms.

4.1 From Principles to Transformation Rules

We have identified three main principles that drive the transformations of our system:

Data locality and block-based transfers. One heuristic that OCAS uses is to fetch the largest possible block of data to the processing unit *at once*. The justification is that, unless the input contains data that is never looked at by the algorithm, every data element has to be eventually fetched. Thus, if fetching is performed in larger chunks, then the number of InitCom events, which represent disk seeking and the costly erasure on flash drives, decreases. The transformation rule that applies this optimization is called apply−block.

The order in which individual data elements are accessed can also be changed by rules swap−iter and order−inputs. This is a class of optimizations whose effectiveness depends on the interaction of several levels of the memory hierarchy, rather than the properties of each individual level. Therefore, there does not exist a generally valid characterization of the cases when these optimizations are effective, other than suggesting that the performance after the application of the rule should improve.

Sequential versus random access. Some devices perform significantly better if data on them is accessed sequentially rather than in random order. A notable example are hard disks, but also writing sequentially to flash drives is more efficient because an erased block can be filled before another one has to be erased somewhere else. Pre-fetching data by blocks into a level that does not have a performance penalty for random access can improve performance in programs where random access is confined to happen within blocks. Blocking is introduced by OCAS through the apply−block rule.

Minimizing the number of passes through the input. Some programs require accessing every element of the input several times to compute the result. There are abundant examples of this behavior in data management systems: join algorithms may have to consider all pairs of members from two relations, comparison-based sorting algorithms have a theoretical bound of at least $\log n$ accesses to each input element, etc.

OCAS uses two techniques to minimize the number of passes through the input: Partitioning

data by hash, represented by the rule hash−part; and Divide-and-Conquer, represented by the rule inc−branching. Both rules have the effect that the individual input elements need to be accessed fewer times, in fact only two times in the case of hash−part(once for constructing subsets of the input and once for performing the computation on them and recombining the results[1]), but in a more random order. Therefore, there is a trade off between the total amount of data transferred and the amount of seeking that this rule introduces.

In addition to the previous ideas, another class of optimization rules target improving the asymptotic computational complexity of the algorithm, such as the fldL−to−trfld rule. However, we do not make this kind of rules a priority of this thesis, because they are rather independent of the usage of the memory hierarchy and they form a broad enough research topic on their own. Application of functional-style transformations to improve the asymptotic complexity of programs has been studied in e.g. [Bird, 1989] and [Augusteijn, 1999], although not in the context of *automatic* synthesis of programs.

4.2 List of Transformation Rules

We write our rules as $e_1 \Rightarrow e_2$ where e_1 and e_2 are OCAL expressions. This means that whenever a part of a program matches e_1 then this part is equivalent to and can be replaced by e_2, leading to a new program.

Most rules come with additional conditions on e_1 that determine when the rule can be applied. For example, the swap−iter rule, which exchanges the outer and the inner loop of a nested for loop, requires as a side condition that the range of the inner loop to be independent of the looping variable of the outer loop. Such conditions can pose a challenge: some are undecidable in the general case or deciding them is too computation intensive. In such cases, we implement a conservative estimation procedure that returns no false positives by deciding a stronger but simpler condition. This approach may lead our tool to fail to notice opportunities when a rule could be correctly applied but it never allows it to apply a rule in a non-valid context. We now describe the motivation of each rule, the conditions under which it can be applied and some examples of usage.

We show in the experimental evaluation of OCAS (Chapter 5) that this set of rules already covers a rich collection of programs. There are other principles that OCAS does not yet deal with. However, our tool can be easily extended with such principles in the form of new transformation rules that follow the same pattern as the ones presented in this chapter.

Increasing the Block Size (apply−block). The fold and flatMap constructs, as specified in Chapter 3, iterate over the elements of a list one by one, as they appear, in a sequential fashion. However, OCAS provides the for construct which allows iterating over blocks of elements,

[1] This assumes that the subsets are small enough to perform the computation on them in the memory level into which they are read (thus requiring no additional I/O operations). OCAS models the memory hierarchy, and thus, can take such constraints into account when deciding the subset size.

instead of one by one. Using the blocked for in place of the more granular counterpart is the aim of the following transformation rule:

for $(x\ [1] \leftarrow \mathsf{R})\ [1]\ e$
$\Downarrow$
for $(xBlock\ [k_1] \leftarrow \mathsf{R})$
  for $(x \leftarrow xBlock)\ [k_2]\ e$

This rule can be applied both when fetching data towards the processing unit as well as to the data that is written as a result of evaluating expression *e*. To use it, we introduce the new annotation $[k_2]$ (in the place shown) for buffering the output. In general, apply−block increases the amount of data read or written in a single I/O request, from the default single element to blocks of size k_1 and k_2, respectively. The value of k_2 is limited by the space and the maxSeqW property of the node (explained in Figure 3.3) where the elements are being written to, and k_1 by the maxSeqR property of the source node (also explained in the same figure). The actual values of k_1 and k_2 are determined by the non-linear optimizer that we have implemented based on [Liuzzi et al., 2010]. In short, for a single loop, a good heuristic is that both k_1 and k_2 should be as big as possible, subject to the aforementioned restrictions. However, if several nested loops over different ranges compete for space at the same node, this trivial heuristic does not work and we use the optimization solver to determine the block sizes.

If R is originally stored at node m_0, is fetched at m_1 and the output is written to node m_2, this rule reduces the number of InitCom$[m_0 \rightarrow m_1]$ cost events k_1-fold, and the number of InitCom$[m_1 \rightarrow m_2]$ events k_2-fold, as long as $m_0 \neq m_2$. If some of these nodes are hard disks, this rule decreases the number of disk seeks. In general, our system aims to replace every list-iterative construct with block size 1 with as many levels of nested equivalent constructs with larger block size as there are levels in the memory hierarchy. We note that we also use an analogous rule to introduce bigger blocks to our implementation of unfoldR.

Swapping The Order Of Iterative Constructs (swap−iter). Given two for or two flatMap constructs that iterate over two different lists, we can then change the order in which these two constructs are applied, as follows:

for $(x_1\ [k_{11}] \leftarrow range_1)\ [k_{12}]$
  for $(x_2\ [k_{21}] \leftarrow range_2)\ [k_{22}]$ e
$\Downarrow$
for $(x_2\ [k_{21}] \leftarrow range_2)\ [k_{22}]$
  for $(x_1\ [k_{11}] \leftarrow range_1)\ [k_{12}]$ e

This rule can be applied provided that the value of $range_2$ does not depend on x_1. We also have an analogous rule for loops with a condition:

```
for (x1 [k11] ← range1) [k12]
 if c then
    for (x2 [k21] ← range2) [k22] e1
 else e2
⇓
for (x2 [k21] ← range2) [k22]
 for (x1 [k11] ← range1) [k12]
    if c then e1
    else e2
```

Ordering Input Lists by Length (order−inputs)

$$f \Rightarrow \lambda\langle x_1, x_2\rangle.\ f(\text{if length}(x_1) \leq \text{length}(x_2) \text{ then } \langle x_1, x_2\rangle \text{ else } \langle x_2, x_1\rangle)$$
$$f \Rightarrow \lambda\langle x_1, x_2\rangle.\ f(\text{if length}(x_1) \leq \text{length}(x_2) \text{ then } \langle x_2, x_1\rangle \text{ else } \langle x_1, x_2\rangle)$$

The target of this rule are applications where the input is a tuple of lists whose order does not matter for the calculated result but may matter for efficiency. For instance, a Block Nested Loops join is more efficient if the outer relation is the smaller. These two rules can be applied if f is of type $\langle[\tau_1],[\tau_1]\rangle \to \tau_2$. However, it is easy to generalize this rule for functions whose input is a tuple of type $\langle[\tau_1],\ldots,[\tau_1]\rangle \to \tau_2$.

Hash Partitioning of Input (hash−part). The following procedure can sometimes improve the performance of an algorithm. Given a tuple of lists, we distribute the elements of each of the lists into subsets, each containing elements that hash into a particular range. We thus obtain a tuple of lists of lists, where each list of lists represents the set of hash partitions of one of the original lists. These are then zipped together to form a value L of type list of tuples of lists, that has length s and contains all the tuples of corresponding partitions. The original algorithm is then mapped over the tuples. OCAS provides an efficient implementation of the partition definition, which is executed in linear time. The following transformation rule captures this idea:

$$\begin{array}{l} f \Rightarrow \lambda\langle x_1,\ldots,x_k\rangle.(\\ \quad \text{flatMap}(f)(\\ \quad\quad \text{zip}(\langle \text{partition}(x_1),\ldots,\text{partition}(x_k)\rangle) \\ \quad) \\ ) \end{array}$$

This rule works for any s when f is a function that iterates over a tuple of lists, for example a join. Most importantly, f must be such that when one takes the union of results of f applied to the s partitions, one gets the same result as applying f to the original input lists. This means

that we do not care about the order in which the function processes the input elements. Stated otherwise, while the function performs computation on r lists, every value is only considered in the context of other values that hash close to it from all lists.

If f is a program that accesses every element of its input more than once, applying this rule has the effect that all of the data is read only twice: once during the partitioning phase and once when applying f to the partitions, provided they are small enough to fit in the node into which f reads the data to from their original location. This rule is needed for synthesizing hash joins.

Increasing the Branching of treeFold (inc−branching).

$$\begin{array}{l}\mathsf{treeFold}[2^k](c,\ \mathsf{funcPow}[k](f)) \\ \Downarrow \\ \mathsf{treeFold}[2^{k+1}](c,\ \mathsf{funcPow}[k+1](f))\end{array}$$

The condition for this rule to work is that f has to be associative. When this rule is applied, the number of applications of the function inside the treeFold decreases but the function becomes more complicated as it accepts more arguments.

For example, when converting treeFold[2] into treeFold[4] for a list of 8 elements, we get a reduction on the number of functions calls of funcPow from seven down to three; however, each such call in the former case has two arguments, while in the latter case it has four. In general, we get approximately $\frac{n-1}{2^k-1}$ calls to funcPow for an input list of size n, where each such function call receives 2^k arguments.

In Chapter 5, we provide the example of deriving 2^k-way External Merge-Sort. There, the sorting algorithm operates on a list of lists, which necessitates the usage of the unfold definition. In this particular case, it is more efficient to actually execute the above transformation rule as follows:

$$\begin{array}{l}\mathsf{treeFold}[2^k](c,\ \mathsf{unfoldR}(\mathsf{funcPow}[k](f))) \\ \Downarrow \\ \mathsf{treeFold}[2^{k+1}](c,\ \mathsf{unfoldR}(\mathsf{funcPow}[k+1](f)))\end{array}$$

Change of Folding Pattern (fldL−to−trfld).

$$\begin{array}{l}\mathsf{foldL}(c,f) \\ \Downarrow \\ \mathsf{treeFold}[2](c,f)\end{array}$$

This rule works whenever f is associative and c is an identity element for f. The treeFold[2] pattern applies f the same number of times as foldL. However, if the size of the result of f and its computational complexity grow at least linearly with the size of its input, then treeFold[2] achieves better performance by balancing f's input sizes more equally.

Adding a Sequentiality Annotation (seq−ac). To enhance the precision of the cost estimation, we allow an expression to be annotated with a token $[m_1 \rightsquigarrow m_2]$. This notifies the costing engine that for this expression, all data transfers from m_1 to m_2 happen sequentially. This annotation serves only as an indicator for the costing engine to allow for more precise estimates and it does not change the semantics and implementation of the program. It can be applied when no other part of the program causes any communication to m_2. A syntactic check provides a sufficient condition.

For example, a for loop that reads a page of the hard disk to the main memory in every iteration and does not otherwise touch the hard disk is allowed to have this annotation. In this case, instead of counting one such event for every iteration (which is the result of ordinary cost inference), we only need to count an InitCom[m1 → m2] event every time that maxSeqR units have been read from m1 or maxSeqW units have been written to m2. Thus, the new InitCom cost is given by: $\max\left(1, \frac{\text{totaltransfers}}{\min(m_1.\text{maxSeqR}, m_2.\text{maxSeqW})}\right)$. Another natural interpretation of this cost function optimization is writing multiple blocks of data to a flash drive after a block, usually much larger, has been erased.

5 Experimental Evaluation of OCAS

In this chapter, we first present our experimental platform and then evaluate our approach with respect to the following points:

1. The **quality** of synthesized algorithms. We evaluate this aspect in two ways. First, we manually inspect the generated C code obtained from OCAS and check whether the code matches our expectations. Particularly for disk-based joins and sorting, we check whether we obtain exactly the standard textbook algorithms given our library of program transformation rules. Then, we evaluate the performance of the synthesized algorithms by running their generated C code on actual data on a hardware configuration that matches our memory hierarchy description and check whether their actual running times match our expectations.

2. The **accuracy** of the predictions compared to the actual execution times of the algorithms. We examine two aspects of this issue, the imprecision of the estimations caused by the fact that the cost formulas do not currently consider CPU costs, and the degree of overestimation caused by the fact that OCAS performs worst-case analysis.

3. The **adaptiveness** of the synthesized algorithms generated by OCAS, whenever the memory or storage configuration changes. This is important, as it proves that OCAS can generate different algorithms for the same naive input program when a different hardware specification is provided. We demonstrate this property using traditional join and sorting operations in three dimensions, by showing that OCAS (a) updates the cost formulas when the hierarchy changes, (b) adapts the generated programs and (c) updates the estimation of parameter values used by specific transformation rules.

4. The **execution time** of the synthesizer, given that the search space grows as longer chains of transformation rules are evaluated on larger programs.

Hard disk: size = 1T	pagesize = 4K
Flash drive: size = 512G	maxSeqW = 256K
Cache: size = 3M	pagesize = 512B
InitCom[HDD ↦ RAM] = 15ms	InitCom[RAM ↦ HDD] = 15ms
InitCom[RAM ↦ SSD] = 1.7ms	InitCom[RAM ↦ Cache] = 0.1ms
UnitTr[HDD ↦ RAM] = 1s/30M	UnitTr[RAM ↦ HDD] = 1s/30M
UnitTr[SSD ↦ RAM] = 1s/120M	UnitTr[RAM ↦ SSD] = 1s/120M

Figure 5.1 – Node properties and associated cost units of a modeled memory hierarchy.

5.1 Experimental Platform

Our platform[1] is a Mac OS X machine with an i7-2620M processor, standard commodity 1TB Western Digital hard disk drives and one 500GB Apple SSD TS512C. Input and RAM buffer sizes are reported in bytes, and are specifically chosen for each experiment. The properties of our devices and the cost of unit events are listed in Figure 5.1. Costs not included are assumed to be zero. For all experiments, we assume exclusive usage of all devices. OCAS is implemented in Scala, so we use a Java Virtual Machine with 256MB of heap space. The C programs generated by OCAS are compiled using GCC 4.2. In what follows, and unless otherwise mentioned, the running time refers to the *actual execution time* of the generated programs.

Table 5.1 presents results for most of our experiments and it also contains the cost of the naive algorithm the user provides, which assumes one I/O (and one seek) per tuple processed. The code of each program in this table can be found in Appendix A.

5.2 Inspection and Quality Evaluation

The aim of this work is to automatically generate algorithms tuned for a particular memory hierarchy. To that end, we do not claim the optimality of the generated algorithms. We have instead manually verified that the generated algorithms are the same as those in textbooks [Ramakrishnan and Gehrke, 2002].

Next, we show how OCAS optimizes the naive join algorithm presented in Example 1 of Chapter 2 for a more complicated memory hierarchy. We also explain in detail how OCAS automatically derives an External Merge-Sort of $n \cdot \log n$ complexity from a naive specification of an insertion sort of n^2 complexity. The purpose of these examples is to show that OCAS generates optimized algorithms, and that these algorithms are the textbook algorithms for the case of disk-based joins and sorting. We also similarly highlight a number of other examples to further validate our analysis.

Block Nested Loops (BNL) and Hash Join. So far, all examples in this thesis assumed a memory hierarchy consisting of *only* one hard disk drive, where the input is stored, and the

[1] This is for all experiments other than measuring cache misses later in this chapter.

Program	Spec. [s]	Opt. [s]	Act. [s]	Relation size R	Relation size S	Total buffer size	Search space	Steps	OCAS Runtime [s]
BNL - No writeout	4×10^9	411	545	1G	32M	8M	9287	6	17
BNL with cache - No writeout	4×10^9	445	533	1G	32M	8M	54202	7	370
(GRACE) hash join - No writeout	4×10^9	356	491	1G	32M	8M	28471	7	78.5
BNL writing to HDD	1016144	5058	4704	32K	256M	20K	2566	6	8.2
BNL writing to other HDD	1016144	1689	2176	32K	256M	20K	7443	6	14.4
BNL writing to flash	561179	307	455	32K	256M	20K	7443	6	12.7
External sorting	1×10^9	157	272	1G	-	260K	130	10	2.9
Column Store Read 5 cols.	125965	197	196	4G	-	5M	7	3	0.01
Column Store Read 10 cols.	251931	395	382	8G	-	10M	7	3	0.01
Duplicate Removal from a Sorted List	503862	546	882	16G	-	16K	7	3	0.16
Aggregation	125965	136	168	4G	-	32K	7	3	0.25
Set Union	251931	396	499	2G	2G	48K	21	3	0.07
Multiset Union (sorted list)	251931	396	479	2G	2G	48K	21	3	0.06
Multiset Union (value-multiplicity)	251931	396	487	2G	2G	48K	21	3	0.07
Multiset Difference (sorted list)	126033	266	137	2G	2G	48K	21	3	0.07
Multiset Difference (value-multiplicity)	126033	266	153	2G	2G	48K	21	3	0.07

Table 5.1 – Cost estimates for the naive specification algorithm (Spec) and the synthesized algorithm (Opt), actual running times of the generated C programs for the synthesized algorithms (Act), input data sizes, and various statistics on synthesis (search space size and depth, and synthesizer running time). The OCAL programs for these examples can be found in Appendix A.

main memory.

However, when this memory hierarchy is extended with one additional level of CPU cache, OCAS generates a version of Block Nested Loops join with additional for loops that make use of the available cache, as explained in the description of the apply−block transformation rule in Chapter 4.

The reader can verify that by applying this transformation, which corresponds to loop tiling, the program becomes more cache-friendly. As a result, there is a small performance improvement, as shown in Table 5.1. Using the *perf* tool [perf] we measured the number of data cache misses. This number is reduced by 98.2%, compared to the previous example (the non-cache conscious BNL join). However, the execution time does not reflect this significant improvement, since this experiment is I/O bound.[2]

Furthermore, by applying the partitioning rule from Chapter 4, OCAS is capable of transforming the Block Nested Loops Join into a variant of the GRACE hash join (while the memory hierarchy remains the same). Our experiments show that, as expected, the hash join performs better than the BNL join. In addition, the underestimation of cost is now more significant, since hash join is more CPU intensive than the cache variant, and OCAS does not currently model CPU costs.

Finally, we notice that in order for OCAS to find and apply these two transformation rules (as described above), it must examine a significantly larger space of semantically equivalent programs, leading to a significant increase in the execution time of the synthesizer (as the depth of the tree is now increased). However, we argue that this increase in synthesis time is well spent, given the performance improvement obtained from using our optimized algorithm versus a non optimized algorithm.

External Merge-Sort. This example uses the fact that folding merge over a list of singleton lists of integers yields a sorting algorithm, and, thus, demonstrates how our rules for changing the folding patterns can automatically bring us from Insertion Sort to a version of the External Merge-Sort. As a starting point, Insertion Sort of a list stored on the hard disk can be represented in OCAL as:

$$\mathsf{foldL}([], \mathsf{unfoldR}(\mathsf{mrg}))(\mathsf{R})$$

where mrg is the supporting function presented in Chapter 3 and the input is a list R of length x of singleton lists of integers. In this naive program, the elements are transferred one by one from HDD to RAM and back, assuming the basic memory hierarchy containing only one HDD

[2] We have acquired the cache miss ratio of this experiment on a Linux server with an Intel Xeon E5-2620 CPU. The relative performance speedup between the Block Nested Loops join and the cache example in the new server is the same as the one calculated using the original Mac machine.

and RAM. The worst-case cost thus is:

$$\sum_{j=0}^{x-1} (\mathsf{InitCom}[\mathsf{HDD} \mapsto \mathsf{RAM}] + (j+1)(\mathsf{UnitTr}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \mathsf{UnitTr}[\mathsf{RAM} \mapsto \mathsf{HDD}] + \mathsf{InitCom}[\mathsf{RAM} \mapsto \mathsf{HDD}]))$$

Our system includes a basic engine for simplifying arithmetic expressions, capable of finding closed forms of some sums, which automatically simplifies the above formula to:

$$x\,\mathsf{InitCom}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \frac{x(x+1)}{2}(\mathsf{UnitTr}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \mathsf{UnitTr}[\mathsf{RAM} \mapsto \mathsf{HDD}] + \mathsf{InitCom}[\mathsf{RAM} \mapsto \mathsf{HDD}])$$

Notice that this formula captures the worst-case asymptotic complexity of $\Theta(n^2)$.

Then, by applying rule fldL−to−trfld (replacing foldL with treeFold[2]), rule inc−branching (replacing treeFold[2^{k}] with treeFold[$2^{\mathsf{k}+1}$]) and finally rule apply−block (applying blocking to unfoldR), we obtain 4-way External Merge-Sort. If we then apply rule inc−branching k − 1 more times, we get to 2^{k}-way External Merge-Sort, whose code is:

$$\mathsf{treeFold}[2^k]([], \mathsf{unfoldR}(\mathsf{funcPow}[k](\mathsf{mrg})))$$

The cost of running this program is, after simplification:

$$\left\lceil \frac{\lceil \log x \rceil\, x}{\mathsf{k}} \right\rceil \left(\mathsf{UnitTr}[\mathsf{RAM} \mapsto \mathsf{HDD}] + \mathsf{UnitTr}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \frac{1}{b_{in}}\mathsf{InitCom}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \frac{1}{b_{out}}\mathsf{InitCom}[\mathsf{RAM} \mapsto \mathsf{HDD}]\right)$$

which captures the asymptotic complexity of $n \cdot \log n$ of external merge-sort.

Our non-linear optimization solver determines that this cost is minimal when the all the input blocks and the output block are as large as possible, which means $b_{out} = b_{in} = \frac{m_s.\mathsf{size}}{2^k+1}$, and hence the number of units transferred decreases with k and is proportional to $1/k$, while the amount of seeking *increases* with k and is proportional to $2^k/k$. Choosing the right k is again accomplished using the optimization solver and depends on the ratio between the seek time and the reading speed of the hard disk. Our experience with the generated external merge-sort is that the optimizer is capable of choosing optimal values for parameters k and b_{in}. For the result presented in Table 5.1, we initially experimented with $k = 1$ for processing 1GB of data but the optimizer recommended $k = 2$ which led to the reported better execution time.

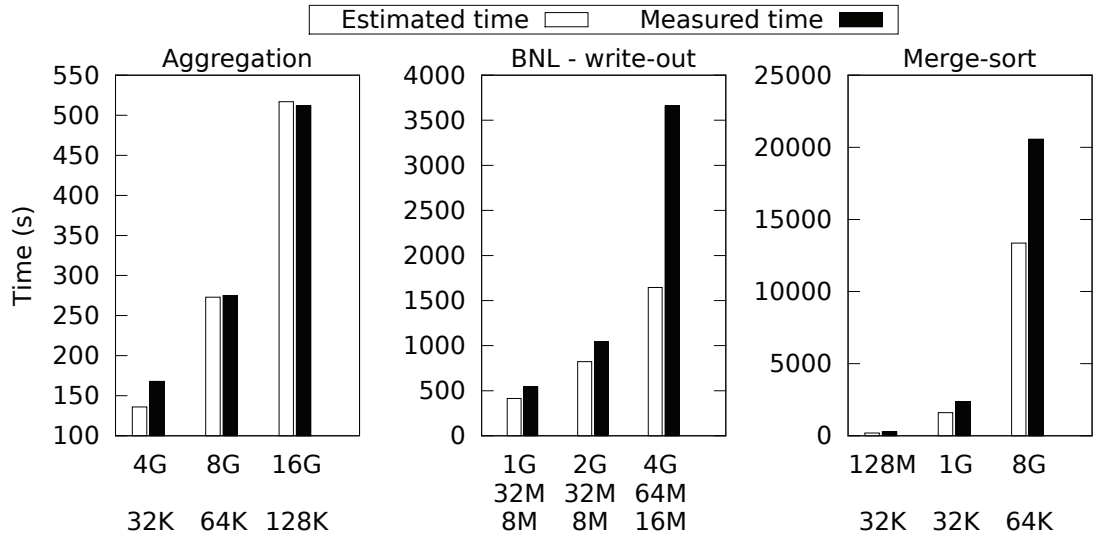

Figure 5.2 – Estimated and actual running times for varying input and buffer sizes. The x-axis label shows the size of the first input relation, the second input relation (if applicable) and the buffer size.

Other miscellaneous examples: To further validate our analysis, we have also expressed a couple of other miscellaneous algorithms using OCAL, such as column store scan (zipping r lists to obtain a single list of r-tuples) with a varying number of columns and removing duplicates from a sorted list. The results, presented in Table 5.1, verify that the predictions of OCAL are accurate and that our tool can generate efficient out-of-core algorithms quickly.

To sum up, the output algorithm of OCAS is always better, performance wise, than the specification algorithm provided by the user. In addition, manual inspection of the generated C programs shows that OCAS produces exactly the standard textbook (disk-based) BNL and hash join and external sorting algorithms. This fact confirms the correctness of our approach.

5.3 Accuracy of Cost Formulas

General Overview. In Table 5.1, we present the actual execution times of the generated C code for the optimized algorithms. As we can see, the estimates of OCAS are in general not far from the actual execution time, and in some cases our tool underestimates. This is because, as we explain below, the cost formulas are simplistic in nature and they do not completely represent the actual execution properties, especially for CPU-dominated workloads. The important point to notice is that, when comparing equivalent algorithms, the predictions follow the same trend as the actual execution times. Next, we examine two different aspects of accuracy in more detail: the effects of not modeling computation cost and of performing worst-case analysis on the estimations of OCAS.

Impact of Computation Costs. OCAS does not currently model computation costs. This can cause our system to underestimate, as shown in Table 5.1. Moreover, underestimation should grow the more CPU intensive a task is. To examine this hypothesis, we run a set of experiments with a variety of different algorithms, input and buffer sizes. The results for these experiments, presented in Figure 5.2, confirm the initial assumption: For tasks that are not CPU-intensive, such as aggregation, the estimations are very accurate. However, for tasks like joins or sorting, which consume a significant amount of CPU cycles, underestimation grows with the input size. However, we leave the development of a more precise CPU modeling for future work.

Impact of Worst-Case Analysis. The worst case analysis that OCAS performs can lead to significant overestimation of the output size, resulting in overestimation of the number of write operations. This is important, since this amount proportionally affects the reported estimations. To better understand this behavior, we present three examples in Table 5.1: one that calculates the union of sets represented as a sorted list of unique values, another that returns the union of multisets represented as a list of value-multiplicity pairs, and finally one that calculates their difference. For the union examples, the estimated output is equal to $\mathsf{length}(L_1)+\mathsf{length}(L_2)$ and for difference it is $\mathsf{length}(L_1)$. The latter follows because in the worst case there is no element that is the same amongst the two relations. The results of Table 5.1 show that there is overestimation due to predicting more write operations than those actually happening for the difference example. The union algorithm, however, is estimated correctly in both examples.

The join operator has a similar behavior due to selectivity: the higher the selectivity, the closer the estimation is to the actual running time. As Table 5.1 shows, when selectivity is 100%, which corresponds to the relational product computed by all join variants presented in this chapter, the predictions become very accurate.

5.4 Adaptivity Evaluation

In this section, we demonstrate how OCAS adapts its cost formulas and generated algorithms when the memory hierarchy changes.

Recalculating Cost Formulas. Accurate cost prediction is important in OCAS since it allows our system to differentiate between more efficient and less efficient programs. Adapting the cost formulas to a particular memory hierarchy is also important, as algorithms that are sub-optimal in one memory hierarchy may be optimal in another.

To examine this aspect of adaptivity in OCAS, we use joins as an example. So far, all join variants presented in this thesis were discarding any output produced. Next, we turn our attention to three examples where the output of the join is instead written to a device and not discarded. We do so using the following three storage hierarchy configurations.

First, when the output is written back to the same hard disk that stores the input, sequential

reading from the hard disk is no longer possible, because the write operations interfere with the reads. Table 5.1 shows that the cost formula successfully depicts the considerably increased running time, even though the total size of the input relations is significantly smaller compared to the original BNL join with no write-out.

If the memory hierarchy changes so that another hard disk HDD2 stores the output, reading and writing do not interfere with each other, so both can be executed sequentially. By doing so, even though the amount of data transfers remains the same as before, hard disk seeking is significantly reduced, as indicated by the updated cost formulas. As a result, Table 5.1 shows that both the estimated and the actual execution times are reduced by more than 50%. Note that we use the join condition "true" (thus we compute a Cartesian product between the two relations) in the BNL join examples which write their output to a drive. Thus, write cost dominates read cost, which explains why the BNL join writing to a different disk is much slower than the BNL join discarding its output.

Finally, we consider a memory hierarchy where a flash drive is used in place of the second hard disk. In this case, OCAS generates the same program as before, since the memory hierarchy has the same form. However, both the estimated and the actual execution times are reduced due to the significantly better sequential write speed of SSDs. This is true, as the factor of the InitCom events changes to depict their different meaning on flash. They do not correspond to seeks, but rather to an erasure occurring before each sequence of write operations, the length of which is given by the maxSeqW property of the flash drive. OCAS estimates better execution time for the example with flash and, thus, it accurately captures this trade-off between sequential writing and erase operations. The actual execution time of this experiment presents a similar behavior.

Adapting the Generated Programs. Now that we have established that OCAS adapts the cost formulas of programs whenever the underlying memory hierarchy changes, we need to also verify that the generated programs actually change and adapt for new memory hierarchies. To demonstrate this, we again consider joins as the driving example, and we examine which join variants are generated for various memory hierarchies.

As we mentioned before in Section 5.2, when the memory hierarchy changes from the basic one (containing a single HDD and RAM) to include also a level of CPU cache, OCAS will generate a more cache-friendly version that contains additional for loops in order to best utilize the cache, leading to improved performance.

Then, in the same section, we mentioned that OCAS also employs the partitioning rule to generate the GRACE hash join. However, OCAS should generate this program *only if* the RAM buffer size is enough to hold the partitions in memory (This was indeed the case for the dataset and buffer size used in Table 5.1). Otherwise continuously evicting partitions from memory to disk can cause significant overhead to performance. To validate this analysis, we run OCAS to generate efficient join algorithms for memory hierarchies with various RAM buffer sizes.

RAM Buffer Size	Generated Optimal Program	Estimated Exec. Time of Optimal Program [s]	Non-Optimal Program	Estimated Exec. Time of Non-Optimal Program [s]
4M	GRACE Hash Join	362	BNL Join	549
2M	GRACE Hash Join	767	BNL Join	1099
1M	BNL Join	2205	GRACE Hash Join	2348
512K	BNL Join	4444	GRACE Hash Join	8638

Table 5.2 – Generated join variants and various performance characteristics (estimated) for memory hierarchies with different RAM buffer sizes. For all examples, the memory hierarchy contains a single hard disk, storing both the input and the output, RAM and a CPU cache.

Our results are presented in Table 5.2. We can see that as the RAM buffer size gets smaller and smaller, the GRACE hash join does not always yield better performance compared to a textbook Block Nested Loops join. Thus, we can see that depending on the size of the RAM buffer, OCAS will always optimally choose between the GRACE hash join and the BNL join.

Updating the Estimation of Parameter Values. A final consideration regarding the adaptivity of our approach is whether OCAS adapts the values of parameters when the underlying memory hierarchy changes. To validate this fact, we use sorting as an example and examine whether the branching factor in our inc−branching transformation rule is changed when we change the size of the RAM buffer.

Our results indicate that the non-linear optimization solver employed by OCAS successfully adapts the value of this parameter. For example, when OCAS is run with a memory buffer of 1M to 4M, then it calculates $k = 3$, while for values smaller than 1M, it calculates $k = 2$ (as was also reported in Table 5.1). Similarly, when allowing for even larger buffers, it will further increase the value of k. For example, for a 16M buffer, the non-linear optimization solver will provide $k = 8$.

These results indicate that OCAS successfully takes into account architectural characteristics when evaluating the values of parameters using the non-linear optimization solver.

5.5 Running Time of OCAS

Finally, Table 5.1 presents the time required for OCAS to generate the optimized algorithms. As we can see, our tool is practical, since its execution time is small for all examples. This is true even for long-running OCAL programs.

We observe that the size of the search space depends on the number of steps needed for the derivation, the complexity of the input program, as well as the memory model used in the experiment. As expected, the search space is growing roughly exponentially with the number of transformation steps and the execution time is linked to the size of the search space.

However, it is not dependent on the input size because OCAS uses cost-based optimization, which does not need to execute the programs in order to estimate their cost.

6 Building Efficient Query Engines in a High-Level Language

We now turn our attention to the realization of the abstraction without regret vision on the domain of ad-hoc, analytical query processing. We present LegoBase, an in-memory query execution engine written in the high-level programming language, Scala, being the first step towards providing a full DBMS written in a high-level language. As shown in Figure 1.1, LegoBase offers a productivity/performance combination not provided by existing database systems or query compilers written using low-level languages. In order to achieve this behavior, we address the following challenges and make the following contributions:

- First, to avoid the overheads of a high-level language (e.g. complicated memory management) while maintaining well-defined abstractions, we opt for using *generative* programming [Taha and Sheard, 2000], a technique that allows for programmatic removal of abstraction overhead through source-to-source compilation. This is a key benefit as, in contrast to traditional, general-purpose compilers – which need to perform complicated and sometimes brittle analyses before *maybe* optimizing programs – generative programming in Scala takes advantage of the type system of the language to provide programmers with strong *guarantees* about the structure of the generated code. For example, developers can specify optimizations that are applied during compilation in order to ensure that certain abstractions (e.g. generic data structures and function calls) are definitely optimized away during compilation.

  Generative programming can be used to optimize *any* piece of Scala code. This allows LegoBase to perform *whole-system* specialization and compile *all* components, data structures and auxiliary functions used inside the query engine to efficient C code. This design significantly contrasts our approach with existing *query* compilation approaches (e.g. the one proposed in [Neumann, 2011]) for three reasons. First, a compiler that handles *only* queries cannot optimize and inline their code with the remaining code of the database system (which is typically *precompiled*), thus missing a number of optimization opportunities. Second, in their purest form, query compilation approaches simply optimize or inline the code of *individual* operators in the physical query plan, thus

making cross-operator code optimization inside the query compiler impossible. Finally, existing approaches perform compilation using low-level code generation templates. These essentially come in stringified form, making their development and automatic type checking very difficult[1].

- The LegoBase query engine uses a *new* optimizing compiler called SC. When performing *whole-system* compilation, 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 applying generative programming.

  First, to achieve maximum efficiency, developers must have tight control on the compiler's phases – admitting custom optimization phases and phase orderings. This is necessary as code transformers with different optimization objectives may have to be combined in every possible ordering, depending on architectural, data, or query characteristics. However, existing generative programming frameworks do not offer much control over the compilation process[2]. 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[3].

  Second, 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 and, more importantly, the introduction of the IR as an additional level of abstraction, both significantly increase the difficulty of debugging as developers cannot easily track the relationship between the source code, the compiler optimization for it – expressed using IR constructs (instead of the source language) – and the final, generated code [Jovanović et al., 2014; Sujeeth et al., 2013].

[1] For example, templates can be used to convert the code of individual query operators – typically written today in C/C++ – to optimized LLVM code. In that case, developers must handle a number of low-level concerns themselves, like register allocation.

[2] For instance, Lightweight Modular Staging (LMS) [Rompf and Odersky, 2010] applies *all* user-specified, domain-specific optimizations in a *single* optimization step. It does so to avoid the well-known *phase-ordering* problem in compilers [Touati and Barthou, 2006], where applying two (or more) optimizations in an improper order can lead not only to suboptimal performance but also to programs that are semantically incorrect [Rompf, 2012]. We analyze how the design of the new optimizing compiler, SC, differs from that of LMS in Chapter 7 of this thesis.

[3] 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 [Ailamaki et al., 2001]) 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.

Instead, 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 along with all optimizations are both written in plain Scala, thus allowing developers to be highly productive when optimizing all components of the query engine.

- 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 providing (i) an in-depth presentation of the optimizations applied to the LegoBase query engine and (b) a description of the *high-level* compiler interfaces that database developers need to interact with when coding optimizations.

  We show that the design and interfaces of our optimizing compiler provide a number of nice 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 later in more detail in Chapter 7) by applying them in multiple, *distinct* optimization phases. Optimizations are (a) adjustable to the characteristics of workloads and architectures, (b) configurable, so that 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 such optimization, we present: (a) the *domain-specific* conditions that need to be satisfied in order to apply it (if any) and (b) possible trade-offs (e.g. improved execution time versus increased memory consumption). Finally, 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 [Transaction Processing Performance Council, 1999]. 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 (that does not employ compilation) and the code generated by the query compiler of the HyPer database [Neumann, 2011]. 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 on 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 conclude our analysis by demonstrating that our *whole-system* compilation approach incurs negligible overhead to query execution.

```
SELECT * FROM
  (SELECT S.D,
    SUM(1-S.B) AS E,
    SUM(S.A*(1-S.B)),
    SUM(S.A*(1-S.B)*(1+S.C))
  FROM S
  GROUP BY S.D) T, R
WHERE T.E=R.Z AND R.Q=3
```

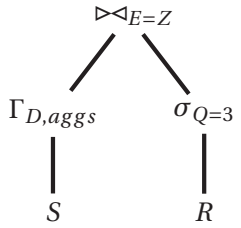

Figure 6.1 – Motivating example showing missed optimizations opportunities by existing query compilers that use template expansion.

Motivating Example. To better understand the differences of our work with previous approaches, consider the SQL query shown in Figure 6.1. This query first calculates some aggregations from relation *S* in the group by operator Γ. Then, it joins these aggregations with relation *R*, the tuples of which are filtered by the value of column *Q*. The results are then returned to the user. Careful examination of the execution plan of this query, shown in the same figure, reveals the following three basic optimization opportunities missed by existing query compilers that use template expansion:

- First, the limited scope of existing approaches usually results in performing the evaluation of aggregations in *precompiled* DBMS code. Thus, each aggregation is evaluated *consecutively* and, as a result, common sub-expression elimination cannot be performed in this case (e.g. in the calculation of expressions `1-S.B` and `S.A*(1-S.B)`). This shows that, if we include the evaluation of all aggregations in the *compiled* final code, we can get an additional performance improvement. This motivates us to extend the scope of compilation in this work.

- Second, template-based approaches may result in unnecessary computation. This is because operators are not aware of each other. 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. By high-level programming, we can instead easily pattern match on the operators, as we show in Section 8.1.

- Finally, the data structures have to be *generic* enough for all queries. As such, they incur significant abstraction overhead, especially when these structures are accessed millions of times during query evaluation. Current query compilers cannot optimize the data structures since these belong to the precompiled part of the DBMS. Our approach eliminates these overheads as it performs *whole-program* optimization and compiles, along with the operators, the data structures employed by a query. This significantly contrasts our approach with previous work.

The next chapters of this thesis are organized as follows. Chapter 7 presents the overall design of LegoBase, along with a detailed description of the APIs provided by the new SC optimizing compiler. Chapter 8 gives an in-depth presentation of all supported compiler optimizations of our system in multiple domains. Chapter 9 presents our evaluation, where we experimentally show that our approach using the SC optimizing compiler can lead to significant benefits compared to (i) a commercial DBMS that does not employ compilation and (ii) a database system that uses low-level, code-generation templates during query compilation. We also give insights about the memory footprint, data loading time and programming effort required when working with the LegoBase system.

7 System Design of LegoBase and SC

In this chapter, we describe the design of the LegoBase system. First, we present the overall system architecture of our approach (Section 7.1). Then, we discuss in detail the SC compiler that is the core of our proposal (Section 7.2) as well as how we efficiently convert the *entire* high-level Scala code of the query engine (not just that of individual operators) to optimized C code for each incoming query (Section 7.3). While doing so, we present how (a) physical query operators, (b) physical query plans, and, (c) compiler interfaces look like in our system. Finally, we provide a concrete example of source-to-source compilation of an SQL query to efficient C code (Section 7.4) and briefly discuss about the extensibility of our approach (Section 7.5).

7.1 Overall System Architecture

LegoBase implements the typical query plan operators found in traditional database systems, including equi, semi, anti, and outer joins, all on a high level. In addition, LegoBase supports both a classical Volcano-style [Graefe, 1994] query engine as well as a push-style query interface [Neumann, 2011][1].

The overall system architecture of LegoBase is shown in Figure 7.1. First, for each incoming SQL query, we must get a query plan which describes the physical query operators needed to process this query. For this work, we consider traditional query optimization (e.g. determining join ordering) as an orthogonal problem and we instead focus more on experimenting with the different optimizations that can be applied *after* traditional query optimization. Thus, to obtain a physical query plan, we pass the incoming query through *any existing* query optimizer. For example, for our evaluation, we choose the query optimizer of a commercial, in-memory database system.

Then, we pass the generated physical plan to LegoBase. Our system, in turn, parses this plan and instantiates the corresponding Scala implementation of the operators. Figure 7.2 presents

[1] In a push engine, the meaning of child and parent operators is reversed compared to the usual query plan terminology: Data flows from the leaves (the ancestors, usually being scan operators) to the root (the final descendant, which computes the final query results that are returned to the user).

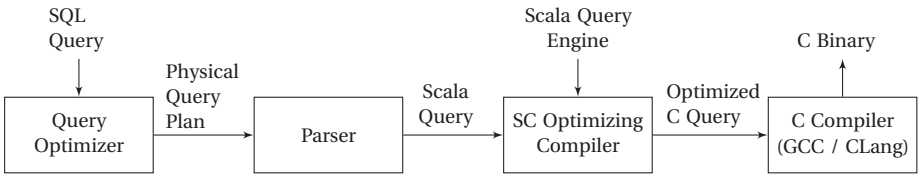

Figure 7.1 – Overall system architecture. The domain-specific optimizations of LegoBase are applied during the SC compiler optimization phase.

an example of how query plans and operators are written in LegoBase, respectively. That is, the Scala code example shown in Figure 7.2a loads the data, builds a functional tree from operator objects and then starts executing the query by calling the `next` function of the root operator in the tree.

It is important to note that operator implementations like the one presented in Figure 7.2b are exactly what one would write for a simple 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 7.2b) and involves expensive memory allocations on the critical path[2], both properties that can significantly affect performance.

However, in our system, the SC optimizing compiler specializes the code of the *entire* query engine 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 domain-specific optimizations (described in detail in Chapter 8). For example, it optimizes away the `HashMap` abstraction and transforms it to efficient low-level constructs (Section 8.2). In addition, SC utilizes the available *query-specific* information during compilation. For instance, it will inline the code of all individual operators and, for the example of Figure 7.2b, 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 they improve branch prediction.

Finally, our system generates the optimized C code which is compiled using any existing C compiler (e.g. we use the CLang[3] frontend of LLVM [Lattner and Adve, 2004] for compiling the generated C code in our evaluation). We also note that, in this work, we choose C as our code-generation language simply because 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). We then return the query results to the user.

[2]Note that such memory allocations are not always explicit (i.e. at object definition time through the **`new`** keyword in object-oriented languages like Java and Scala). For instance, in line 15 of Figure 7.2b, the HashMap data structure 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 talk more about this issue and its consequences to performance later in this thesis.

[3]http://clang.llvm.org/

```
def Q6() {
  val lineitemTable = loadLineitem()
  val scanOp = new ScanOp(lineitemTable)
  val startDate = parseDate("1996-01-01")
  val endDate = parseDate("1997-01-01")
  val selectOp = new SelectOp(scanOp)
    (x =>
      x.L_SHIPDATE >= startDate &&
      x.L_SHIPDATE < endDate &&
      x.L_DISCOUNT >= 0.08 &&
      x.L_DISCOUNT <= 0.1 &&
      x.L_QUANTITY < 24
    )
  val aggOp = new AggOp(selectOp)
    (x => "Total")
    ((t, agg) => { agg +
      (t.L_EXTENDEDPRICE * t.L_DISCOUNT)
    })
  val printOp = new PrintOp(aggOp)(
    kv => printf("%.4f\n", kv.aggs(0))
  )
  printOp.open
  printOp.next
}
```

(a)

```
class AggOp[B](child:Operator, grp:Record=>B,
           aggFuncs:(Record,Double)=>Double*)
  extends Operator {
    val hm = HashMap[B, Array[Double]]()
    def open() { parent.open }
    def process(aggs:Array[Double], t:Record){
      var i = 0
      aggFuncs.foreach { aggFun =>
        aggs(i) = aggFun(tuple, aggs(i))
        i += 1
      }
    }
    def consume(tuple:Record) {
      val key = grp(tuple)
      val aggs = hm.getOrElseUpdate(key,
         new Array[Double](aggFuncs.size))
      process(aggs, tuple)
    }
    def next() {
      hm.foreach { pair => child.consume(
        new AGGRecord(pair._1, pair._2)
      ) }
    }
}
```

(b)

Figure 7.2 – 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 the Push-style interface [Neumann, 2011].

7.2 The SC Compiler Framework

LegoBase makes key use of the SC framework, which provides runtime compilation and code generation facilities for the Scala programming language, as follows.

To begin with, in contrast to 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. SC offers two *high-level* programming primitives named `analyze` and `rewrite` for this purpose, which are illustrated in Figure 7.3a and which analyze and manipulate statements and expressions of the input program, respectively. For example, our data-structure specialization (Section 8.2.2) replaces operations on hash maps with operations on native arrays. 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[4]. We use this optimization interface to provide database-specific optimizations as a library and to aggressively optimize our query engine.

[4]Of course, every compiler needs to represent code through an intermediate representation. 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 [Jovanović et al., 2014].

```
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 7.3 – (a) The analysis and transformation APIs provided by SC. (b) The SC transformation pipeline used by LegoBase. Details for the optimizations listed in this pipeline are presented in Chapter 8.

Then, to allow for maximum efficiency when specializing all components of the query engine, developers must be able to easily experiment with different optimizations and optimization 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 transformation pipeline of LegoBase, shown in Figure 7.3b, Parameter Promotion, Dead Code Elimination and Partial Evaluation are all applied at the end of each of the custom, domain-specific optimizations. Through this transformation pipeline, developers can easily turn optimizations on and off at demand (e.g. by making their application dependant on simple runtime or configuration conditions) as well as specifying which optimizations should be applied *only* for specific hardware platforms.

Even though it has been advocated in previous work [Rompf et al., 2013] that having multiple transformers can cause phase-ordering problems [Touati and Barthou, 2006], our experience is that system developers are empowered by the control they have when coding optimizations with SC and rise to the challenge of specifying a suitable order of transformations as they design their system and its compiler optimizations. As we show in Chapter 9, 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

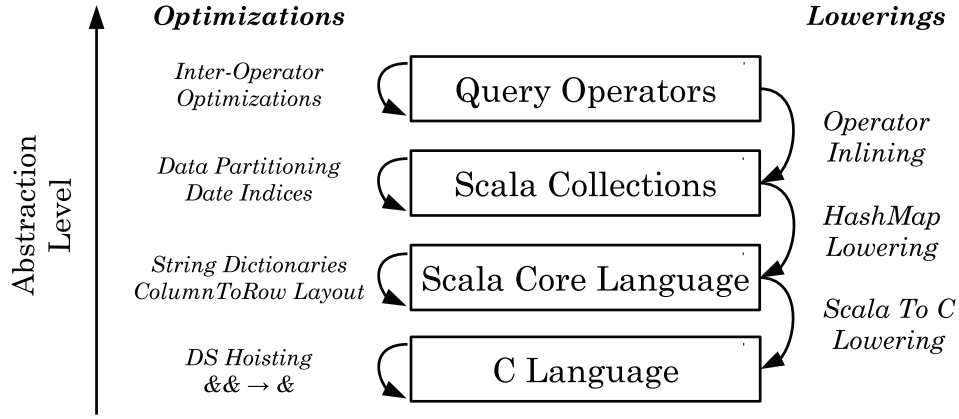

Figure 7.4 – Source-to-source compilation expressed through the progressive lowering approach – there different optimizations are applied in different optimization stages, thus guaranteeing the notion of separation of concerns.

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 and specializing all data structures). We describe these optimizations in more detail in Chapter 8.

7.3 Efficiently Compiling High-Level Query Engines

Database systems comprise many components of significantly different nature and functionality, thus typically resulting in very big code bases. To efficiently optimize those, developers must be able to express new optimizations without the having to modify neither (i) the base code of the system nor (ii) previously developed optimizations. As discussed in the introduction of this thesis, compilation techniques based on template expansion do not scale to the task, as their single-pass approach makes individual optimizations interdependent and forces developers to deal with a number of low-level concerns, making their debugging and development costly.

To this end, the SC compiler framework 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 possible level of representation, and only then generating the final, low-level code. This design is illustrated in Figure 7.4.

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 generally more declarative, thus allowing for increased productivity, while lower levels are closer to the underlying architecture, thus making it possible to more easily perform low-level

Paradigm	Advantages
Declarative	✓ Concise programs
	✓ Simple to analyze and verify
	✓ Simple to parallelize
Imperative	✓ Efficient data structures
	✓ Precise control of execution flow
	✓ More predictable performance

Table 7.1 – Comparison of declarative and imperative language characteristics. We use both paradigms for different steps of our progressive lowering compilation approach.

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 very low abstraction levels. This design provides the nice property that generation of the final code basically becomes a trivial and naive stringification of the lowest level representation. Table 7.1 provides a brief summary of the benefits of imperative and declarative languages in general.

More precisely, in order to reach the abstraction level of C code in LegoBase (the lowest level representation for the purposes of this thesis), transformations in SC also 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 more detail in Chapter 8), 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 programming language. For example, classes are converted to structs, strings to arrays of bytes, etc. In general, composite types are handled in a recursive way, by first lowering their fields and then wrapping the result in a C struct. The final result is a struct of only primitive C constructs.

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 be in turn optimized away in subsequent optimization passes. After all lowering steps have been performed, developers can now apply low-level, architecture-dependent optimizations, as the code is now close to the semantics offered by low-level programming languages (e.g. it includes pointers for explicitly referencing memory locations). Then, a final iteration emits the actual C code.

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 function calls. For example, all collection data structures like hash maps are converted in LegoBase to primitive arrays (Section 8.2). Thus, lowering such library calls to C is not a big issue. 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. We have found GLib [The GNOME Project, 2013] to be efficient enough for this purpose.

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.

7.4 Putting it all together – A compilation example

To better illustrate the various steps of our progressive lowering, we analyze how LegoBase converts the example SQL query shown in Figure 7.5a to efficient C code.

To begin with, the query plan, shown in Figure 7.5b, is parsed and converted to the program shown in Figure 7.5c. This step inlines the code of all relational operators present in the query plan and implements the equijoin using a hash table. This is the natural way database developers would typically implement a join operator using high-level collections programming.

Then, this hash-table data structure is lowered to an array of linked lists (Figure 7.5d). However, these lists are not really required, as we can chain the records together using their *next* pointer. This optimization, which is presented in more detail in Section 8.2, takes place in the next step (Figure 7.5e). Finally, the code is converted to an embedded [Hudak, 1996] version of the C language in Scala (Figure 7.5f) and, only then, SC generates the final C program out of this embedding (Figure 7.5g).

This example clearly illustrates that our optimizing compiler applies different optimizations in distinct transformation phases, thus guaranteeing the separation of concerns among different optimizations. For example, operator inlining is applied in the very first, high-level representation, which only describes operator objects. Performance concerns for data structures are then handled in subsequent optimization steps. Finally, memory management and low-level

```
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 physical plan of the example query.

```
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
```

(c)

```
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
```

(d)

```
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
```

(e)

```
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
```

(f)

```
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;
```

(g)

Figure 7.5 – Progressively lowering an example query to C code with SC.

code generation concerns are addressed only in the last two, low-level representations.

7.5 Extensibility of LegoBase

One of the main advantages of our progressive lowering and optimization design is its extensibility in various dimensions. Next, we first highlight three such extensibility opportunities in LegoBase. Then, we present an outlook of extending our architecture to support parallelism as a more concrete case study.

To begin with, given that *collection programming*[5] APIs [Meijer et al., 2006; Grust et al., 2010, 2009] are growing in popularity, one may consider expressing relational operators and queries using the functionality provided by those APIs. For example, our previous example can be written using collection programming as follows:

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

where the `filter` method is a higher-order function that corresponds to the selection operator in relational algebra, the `hashJoin` method performs the traditional database operation using high-level collection operations[6], and the `count` method returns the number of elements in a collection (e.g. a list).

Then, by providing a lowering transformation from the new, collection-based representation to one of our intermediate representations (e.g. the one in Figure 7.5c), the existing infrastructure can generate optimized C code for that new representation out-of-the-box. In addition, we can reuse all transformations provided by the lower-level representations of our stack *for free*, without any modifications required.

Second, generative programming in LegoBase allows us to optimize any piece of Scala code. This is particularly useful for introducing and optimizing user-defined functions (UDFs). Since SC makes no distinction between these functions and any other piece of code of the system, they can be easily integrated either (a) to modify the functionality of the query engine or (b) as constructs of the input queries. Then, the only requirement may is that users have to explicitly lower their new UDF constructs to the target code, through our progressive lowering approach described in this chapter. This is, however, necessary *only if* those UDFs introduce new constructs; otherwise our existing mechanism and lowering libraries suffice to generate target code out-of-the-box.

Finally, and as we discussed previously, SC is not specifically designed to generate C code.

[5] We refer to *collection programming* as the practice of preferring generic operations defined on collections like lists (e.g. `map`, `fold`, `filter`, `groupBy`, `sum`, etc.) rather than writing them out as loops.

[6] e.g. as described in https://rosettacode.org/wiki/Hash_join#Scala.

On the contrary, the language of the final, generated code can be changed easily. The only requirement is to provide a new lowering from the last, target-language-agnostic representation (e.g. the one in Figure 7.5e of our previous example) to the desired new target language (for instance, replace C with Scala or LLVM in our example). The advantage of this design is that there is no need to perform *any* modification to the code of all optimizations provided by higher-level representations (e.g. those in Figures 7.5c to 7.5e in our example). Note that this approach works well as long as the underlying architecture is not changed. A case study of the extensibility of our compilation and optimization 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 more detail in the next subsection.

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 *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 [Graefe, 1994]. Next, we show how our compilation stack can be enriched with parallelization through the demonstration of the modifications needed for the various stack levels and their transformations.

First, we present the required modifications for the individual levels in our stack. To begin with, the parallelization logic is encoded in the highest-level representation by adding the split and merge operators [Mehta and DeWitt, 1995], which can be used by the physical query plans in LegoBase. As these two operators are not expressible by intermediate representations, we must progressively lower the new constructs. We do so by progressively adding new threading facilities (i.e. constructs for forking and joining threads) to one of the intermediate representations. Finally, the lowest-level representation of our compilation framework generates parallel code by unparsing the new parallel constructs to the corresponding C code (e.g. by using the pthreads library).

Second, we analyze what modifications are required for the transformations of our stack. We distinguish two cases. First, if LegoBase is configured so that input queries do not use parallel physical query plans at all (e.g. there is no use of split and merge operators), then there is no change required for any transformation. However, the generated code for a physical query plan with split and merge operators should use an appropriate set of parallelization constructs, as we described above. To do so, we add two lowering transformers that map these two operators to the newly introduced threading constructs of SC. Note that the introduction of the merge operator needs to be done with special care depending on the class of the aggregation (e.g. `SUM` is distributive and `AVG` is algebraic [Gray et al., 1997]).

We conclude this chapter by making three observations. First, apart from the merge and split operators, there is no other modification needed for any existing query operator (e.g. scans,

joins etc.). Second, the compilation stack needs to modified only once while the modification can actually be reused by all parallel physical query plans. Third, as the rest of the existing transformations do not need to be modified at all, the generated code for each thread benefits from the optimizations provided for the sequential version of our compilation stack *for free*.

In the next chapter, we provide more details about our individual compiler optimizations.

8 Compiler Optimizations

In this chapter, we present examples of compiler optimizations in six domains: (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 dead code elimination. The purpose of this chapter 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, the code of other compiler optimizations, or interaction with compiler internals. Throughout this chapter we use, unless otherwise stated, Q12 of TPC-H[1], shown in Figure 8.1, as a guiding example in order to better illustrate various important characteristics and design choices of our optimizations. The structure of this chapter closely follows the domains described above.

8.1 Inter-Operator Optimizations – Eliminating Redundant Materialization Points

Consider again the motivating example presented in Figure 6.1. We observed that existing query compilers use template-based generation and, thus, in such schemes operators are not aware of each other. This can cause redundant computation: in this example there are two materialization points (in the group by and in the left side of the hash join) where there could be only a single one.

By expressing optimizations at a higher level, LegoBase can optimize code across operator interfaces. For this example, we can treat operators as objects in Scala, and then 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 distinct data structures. The code of this optimization is shown in Figure 8.2.

This optimization operates as follows. First, we call the optimize function, passing it the top-

[1] A brief presentation of the TPC-H schema and queries can be found in Appendix E.

```
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 Next used by the HashJoinOp operator
    ((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 8.1 – Example of an input query plan (TPC-H Q12). We use this query to explain various characteristics of the domain-specific optimizations of LegoBase.

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. In the case of the example this chain is (as shown in line 2 of Figure 8.2) 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 8.2). 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 processing of the tuples of the right-side relation (relation R in Figure 6.1), alongside with checking the join condition and the rest of join-related processing, still takes place during the call of `next` function of the HashJoinOp operator, similarly to the original query operator code.

We observe that this optimization is programmed in the same level of abstraction as the rest of the query engine: as normal Scala code. This allows to completely avoid code duplication during development, but more importantly it 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

```
def optimize(op: Operator): Operator = op match {
  case joinOperator@HashJoinOp(aggOp:AggOp, rightChild, joinPred, leftHash, rightHash) =>
    new HashJoinOp(aggOp.leftChild, rightChild, joinPred, leftHash, rightHash) {
      override def open() {
        // leftChild is now the child of aggOp (relation S)
        leftChild foreach { t =>
          // leftHash hashes according to the attributes referenced in the join condition
          val key = leftHash(aggOp.grp(t))
          // Get aggregations from the hash map of HashJoin
          val aggs = hm.getOrElseUpdate(key, new Array[Double](aggOp.aggFuncs.size))
          // Process all aggregations using the original code of Aggregate Operator
          aggOp.process(aggs,t)
        }
      }
    }

  case op: Operator =>
    op.leftChild = optimize(op.leftChild)
    op.rightChild = optimize(op.rightChild)

  // Operators with only one child have leftChild set, but rightChild null.
  case null => null
}
```

Figure 8.2 – Removing redundant materializations by high-level programming (here between a group by and a join). The semantics (child-parent relationships) of this code segment are adapted to a Volcano-style engine. However, the same optimization logic can be similarly applied to a push engine. The code of the Aggregate Operator is given in Figure 7.2b. The `next` function of the HashJoinOp operator remains the same.

pipeline. Both these properties raise the productivity of developers working with our system, showing the merit of developing database systems using high-level programming languages.

8.2 Data-Structure Specialization

Data-structure optimizations contribute significantly to the complexity of database systems today, as they tend to be heavily specialized to be workload, architecture and (even) query-specific. Our experience with the PostgreSQL[2] database management 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 significantly contributes to a *maintenance nightmare* as in order to apply any code update, many different pieces of code have to be modified.

In addition, even though data-structure specialization is important when targeting high-performance systems, it is not provided, to the best of our knowledge, by any existing query compilation engine. Since our approach can be used to optimize the *entire* Scala code, and not only the operator interfaces, it allows for various degrees of specialization in data structures, as has been previous shown in [Rompf et al., 2013].

[2] http://www.postgresql.org

In this section, we demonstrate such possibilities by explaining how the SC optimizing 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.

8.2.1 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 throughout query execution. We have observed that 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 Appendix C, next.

To begin with, 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, LegoBase creates a data structure which is accessed through the primary key specified for that relation. There are two possibilities:

- For single-attribute primary keys, the value of this attribute in each tuple is used to place the tuple in a continuous 1D-array. For example, for the relations of the TPC-H workload this is a straightforward task as the primary keys are typically integer 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 input data into a sparse array.
- For composite primary keys (e.g. those of the LINEITEM table of TPC-H), creating an 1D array does not suffice, as there may be multiple tuples with the same value for *any one* of the attributes of the primary key (thus causing conflicts when accessing the array). One possible solution would be to hash *all* attributes of the primary key and guarantee that we get a unique index value to access the 1D-array. However, deriving such a function in full generality requires knowledge of the whole dataset in advance (in order to know all possible combinations of the primary key). More importantly, it introduces additional computation on the critical path in order to perform the hash, a fact that, according to our observations, can lead to significant, negative impact on performance. For this reason, LegoBase does not create an 1D array and, instead, handles such primary keys similarly to the handling of foreign key, as we discuss shortly.

For the example given in Figure 8.1, LegoBase creates a 1D array for the ORDERS table, indexed through the O_ORDERKEY attribute, but does not create a 1D array for LINEITEM (as this relation has a composite primary key of the L_ORDERKEY, L_LINENUMBER attributes).

```
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   }
8 }
```

Figure 8.3 – Using primary and foreign key information in order to generate code for high-performance join processing. This optimization replaces the code generated for traditional joins using HashMaps, which operates by first *building* the map by copying all tuples of the left-side operator, then probes it while scanning the tuples of the right-side operator for matches. 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.

Second, LegoBase replicates and repartitions the data of the input relations based on each specified foreign key. This basically leads to the creation of a two-dimensional array, indexed by the foreign key, where each bucket holds all tuples having a particular value for that foreign key. We also apply the same partitioning technique for relations that have composite primary keys, as we mentioned above. We resolve the case where the foreign key is not in a contiguous value range by trading-off system memory, in a similar way to how we handled 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 data structures 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 mainly on the derived physical query execution plan (e.g. on the referenced attributes as well as on the select and join conditions of the input query), but also on simple to estimate statistics, like cardinality estimation of the input relations.

For the example of Q12, out of the two partitioned, foreign-key data structures presented above for LINEITEM, our optimized generated code for Q12 uses only the partitioned table on L_ORDERKEY, as there is no reference to attributes L_PARTKEY or L_SUPPKEY in the query.

These data structures can be used to significantly improve join processing, as they allow to quickly extract matching tuples on a join between two relations on attributes that have a primary-foreign key relationship. This is best illustrated through our running example of Q12 and the join between the LINEITEM and ORDERS tables. For this query, LegoBase (a) infers that the ORDERKEY attribute actually 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 8.3 in order to directly get

the corresponding bucket of the array of LINEITEM (by using the value of the ORDERKEY attribute), thus avoiding the processing of a possibly significant number of LINEITEM tuples[3].

LegoBase uses this approach for multi-way joins as well, to completely eliminate the overhead of intermediate data structures for most TPC-H queries. This results in significant performance improvement as the corresponding tuple copying between these intermediate data structures (e.g. the MultiMap of Figure 7.5c) is completely avoided, thus reducing memory pressure and improving cache locality. In addition, a number of expensive system calls responsible for the tuple copying is also avoided by applying this optimization.

After the aforementioned optimization has been performed, LegoBase has removed the overhead of using generic data structures for join processing, but there are still some hash maps remaining in the generated code. These are primarily hash maps which correspond to aggregations, as in this case there is no primary/foreign key information that can be used to optimize these data structures away, but also hash maps which process 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.

8.2.2 Optimizing Hash Maps

Next, we show how hash maps, which are the most commonly used data structures along with Trees in DBMSes, can be specialized for significant performance improvement by using schema and query knowledge.

By default, LegoBase uses GLib [The GNOME Project, 2013] 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 7.2b) 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[4]. Finally, the data-structures may have to be resized

[3] Performing the join in this way makes sense from a *data locality* point of view as well. Since ORDERS is represented by a 1D array, each cache miss during its scan (the outer `for` loop in Figure 8.3) brings elements to the cache that are *definitely* examined by the join operator. On the other hand, if we were to scan the (possibly fragmented) 2D array of the LINEITEM table, we would have to check the counts array of possibly empty buckets; this would introduce a number of unnecessary `if` conditions which can negatively affect branch prediction.

[4] The overhead of using virtual function calls in C++ has been studied in [Driesen and Hölzle, 1996]. In this work, the authors use micro-benchmarks to demonstrate that there can be up to 29% median overhead simply for executing dispatch code. Given this observation, there have been efforts to automatically eliminate virtual functions in C++ through source-to-source compilation, e.g. in [Aigner and Hölzle, 1996].

```
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
13  rewrite += rule {
14   case code"($hm: HashMap[K, V]).getOrElseUpdate($key, $value)" => {
15    val arr = transformed(hm) // Get the array representing the original hash map
16    // Extract functions
17    val hashFunc = arr.attributes("hash")
18    val equalFunc = arr.attributes("equals")
19    code"""
20     // Get bucket
21     val h = $hashFunc($value) // Inlines hash function
22     var elem = $arr(h)
23     // Search for element & inline equals function
24     while (elem != null && !$equalFunc(elem, $key))
25      elem = elem.next
26     // Not found: create new elem / update pointers
27     if (elem == null) {
28      elem = $value
29      elem.next = $arr(h)
30      $arr(h) = elem
31     }
32     elem
33    """
34   }
35  }
36
37  // Lowering of remaining operations is performed in a similar way.
38 }
```

Figure 8.4 – Specializing HashMaps by converting them to native arrays. The corresponding operations are mapped to a set of primitive C constructs.

during runtime in order to efficiently accommodate more data. These operations typically correspond to (a) allocating a bigger memory space, (b) copying the old data over to the new memory space and, finally, (c) freeing the old space. These resizing operations are a significant bottleneck, especially for long-running, computationally expensive queries.

Next, we resolve all these issues with our compiler, 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 a high-level compiler API, is practical for specializing DBMS components. The transformation, shown in Figure 8.4, is applied during the *lowering* phase of the compiler (Section 7.3), 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 making use of 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[5] (e.g. at data loading, *outside the critical path*[6]). 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 optimizing 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 wherever they are called (lines 20 and 23 of Figure 8.4). This significantly reduces the number of function calls (to almost zero), thus considerably improving query execution performance.

3. Finally, to avoid costly maintenance operations on the critical path, we preallocate in advance 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 a given query, which means that we possibly allocate much more memory space that what is actually needed. However, database statistics can make this estimation very accurate, as we show in our experiments (Chapter 9) where we evaluate the overall 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 this particular value of `L_SHIPMODE` exists and if a match is found, the aggregation entries are updated accordingly, or a new entry is initialized otherwise.

In addition to the above optimizations, the SC optimizing compiler also detects hash table data structures that receive only a single, *statically-known* key and converts each such structure to a single value, thus completely eliminating the unnecessary abstraction overhead of these tables. In this case, this optimization maps all related HashMap operations to operations in the single value. For example, we convert a `foreach` to a single value lookup. An example of such a lowering is in aggregations which calculate one single global aggregate (in this case `key = ''TOTAL''`). This happens for example in Q6 of the TPC-H workload.

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 8.4.

[5] Stated otherwise, we use *intrusive linked lists* for this optimization.

[6] The transformer shown in Figure 8.4 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.

```
// 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 8.5 – Using `date` incides to speed up selection predicates on large relations.

8.2.3 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.

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 8.5, 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 [Transaction Processing Performance Council, 1999]. In general, date indices are very beneficial for queries that apply select predicates on date attributes with very low selectivity.

8.3 Changing Data Layout

A long-running debate in database literature is the one between row and column stores [Stonebraker et al., 2005; Harizopoulos et al., 2006; Abadi et al., 2008]. Even though there are many significant differences between the two approaches in all levels of the database stack, the central contrasting point is the *data-layout*, i.e. the way data is organized and grouped together. By default LegoBase uses the row layout, since this intuitive data organization facilitated fast development of the relational operators. However, we quickly noted the benefits of using a

```
1 class ColumnarLayoutTransformer extends RuleBasedTransformer {
2
3   rewrite += rule {
4     case code"new Array[T]($size)" if typeRep[T].isRecord => typeRep[T] match {
5       case RecordType(recordName, fields) => {
6         val arrays =
7           for((name, tp: TypeRep[Tp]) <- fields) yield
8             name -> code"new Array[Tp]($size)"
9         val rec = record(recordName, arrays)
10        // Keep size of original array on attributes to answer size and length calls quickly
11        rec.attributes += "size" -> $size
12        rec
13      }
14    }
15  }
16
17  rewrite += rule {
18    case code"(arr:Array[T]).update($idx,$v)" if typeRep[T].isRecord => typeRep[T] match {
19      case RecordType(recordName, fields) => {
20        val columnarArr = transformed(arr) // Get the record of arrays
21          for((name, tp: TypeRep[Tp]) <- fields) {
22            code """
23              val fieldArr: Array[Tp] = record_field($columnarArr, $name)
24              fieldArr($idx) = record_field($v, $name)
25            """
26          }
27      }
28    }
29  }
30
31  rewrite += rule {
32    case code"(arr:Array[T]).apply($index)" if typeRep[T].isRecord => typeRep[T] match {
33      case RecordType(recordName, fields) => {
34        val columnarArr = transformed(arr) // Get the record of arrays
35        val elems = for((name, tp: TypeRep[Tp]) <- fields) yield {
36          name -> code """
37            val fieldArr: Array[Tp] = record_field($columnarArr, $name)
38            fieldArr($index)
39          """
40        }
41        record(recordName, elems)
42      }
43    }
44  }
45
46  // Fill remaining operations accordingly
47 }
```

Figure 8.6 – 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).

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[7].

The optimization of Figure 8.6 performs a conversion from an array of records (row layout) to

[7]We must note that changing the data layout does not mean that LegoBase becomes a column store. There are other important aspects which we do not yet handle, and which we plan to investigate in future work.

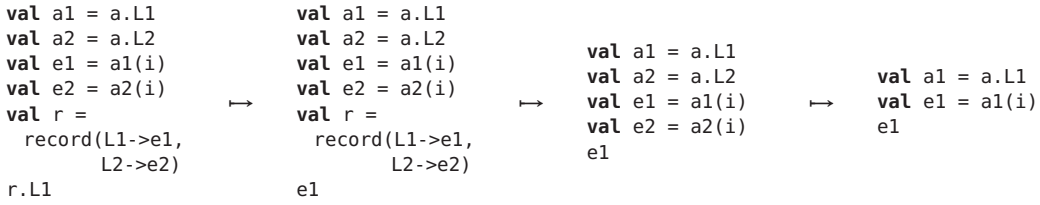

Figure 8.7 – Dead code elimination (DCE) can remove intermediate materializations, e.g. row reconstructions when using a column layout. Here *a* is a record of arrays (column-layout) and *i* is an integer. The records have only two attributes *L1* and *L2*. The notation `L1->v` associates the label (attribute name) L1 with value *v*.

a record of arrays (column layout), where each array in the column layout stores the values for *one* attribute. The optimization basically 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 and the original code is generated (e.g. an array of Integers remains unchanged).

Each optimized operation is basically a straightforward rewriting to a set of operations on the record of arrays. 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 the elements of `arr`. So, for each attribute we extract the corresponding array from `arr` (line 18) and field from `v` (line 19); then we can perform the update operation 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, array_apply results in *record reconstruction* by extracting the individual record fields from the record of arrays (lines 29-34) and then building a new record to hold the result (line 35). This intermediate record can be *automatically* removed using dead code elimination (DCE), as shown in Figure 8.7. 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 8.7).

We notice that, as was the case with previously presented optimizations, 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 distinct optimization phase that handles *only* the lowering 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 (as was described in this section).

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 8.1 – 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.

8.4 String Dictionaries

Operations on non-primitive data types, such as strings, incur a very high performance overhead. This is true for two reasons. First, there is typically a function call required. Second, most of these operations typically 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, even though they seem straightforward, can actually introduce a number of auxiliary CPU instructions [8].

LegoBase uses String Dictionaries to remove the abstraction overhead of Strings. Our system maintains one dictionary 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. 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. Then, at query execution time, string operations are mapped to their integer counterparts, as shown in Table 8.1. This mapping allows to significantly improve the query execution performance, as it typically eliminates underlying loops and, thus, significantly reduces the number of CPU instructions executed. 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

[8] The importance of these overheads to query execution performance becomes more clear if one considers that string attributes often comprise a large portion of the database data. For example, the TPC-H schema contains 61 attributes, 26 of which are of string type, constituting 60% of the total database size [Chen et al., 2001].

first compute the list of distinct values, as mentioned above, then sort this list lexicographically, and afterwords 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 sorted list of distinct values and finding the first and last strings which start or end with that particular string. This range is then used when lowering the operation, as shown in Table 8.1. This *two-phase* string dictionary allows to map all operations that require some notion of ordering in string operations.

In addition, there is one additional special case where the string attributes need to be tokenized on a word granularity. This happens for example in Q13 of TPC-H. This is because queries like that one 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. Then, searching for a word slice is equal to looking through all the integer-typed words in that string for a match during query execution. This is the only case where the integer counterparts of strings operations contain a loop. It is however our experience, that even with this loop through the integer vales, the obtained performance significantly outperforms that of the `strstr` function call of the C library. This may be because such loops can be more easily vectorized by an underlying C compiler like CLang, compared to the corresponding loops using the string types.

Finally, it is important to note that string dictionaries, even though they significantly improve query execution performance[9], they have an even more pronounced negative impact on the performance of data loading. This is particularly true for the word-tokenizing string dictionaries, as the impact of tokenizing a string is significant. 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 Chapter 7.

8.5 Domain-Specific Code Motion

Domain-Specific code motion includes optimizations that remove code segments that have a 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-off increased loading time for improved query execution performance. There are two main optimizations in this category, described next.

[9] In addition to reducing the number of function calls and CPU instructions executed, string dictionaries can also significantly reduce the number of cache misses. This is because these structures reduce the total amount of data required to be transferred from the main memory. For example, it has been shown that string dictionaries can reduce cache misses by 7.5× in an in-memory database on a micro-benchmarking query with only 5% selectivity [Krüger et al., 2012].

8.5.1 Hoisting Memory Allocations

Memory allocations can cause significant performance degradation in query execution. Our experience shows that, by taking advantage of type information available in each SQL query, we can completely eliminate such allocations from the critical path. The LegoBase system provides the following optimization for this purpose.

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 executed occurring during the 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).

We also note that it is not sufficient to naively generate one pool per data type in the order of their appearance, as there may be dependencies between data types. This is particularly true for composite types, which need to reference the pools of the native types (e.g. the pool for Strings). We resolve such dependencies by first applying topological sorting on the obtained type information and only then generating the pools in the proper order.

Finally, we must mention that the size of the memory pools is estimated by performing worst-case analysis on a given query. This means that LegoBase may allocate much more space than needed. However, we have confirmed that our estimated statistics are accurate enough so that the pools do not unnecessarily create memory pressure, thus negatively affecting query performance. In fact, as we show in Chapter 9, for our workloads LegoBase does not so far require more than twice the size of the input data as memory footprint; yet the majority of this additional memory requirement is introduced from the partitioning optimization rather that the optimization for hoisting the memory allocations described here.

8.5.2 Hoisting Data-Structure Initialization

Performing operations on any data structure needed during query execution generally requires specific code to be executed in the critical path regading the proper initialization and maintenance of these structures. 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 *domain* of that 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 is not possible in its full generality, as it directly depends on the ability to predict the possible key values in advance, during data loading. In addition, this value range should be adequately dense (e.g. sequential values), since otherwise the unused values can significantly (and unnecessarily) increase memory pressure, thus eliminating any performance benefit obtained from removing the data structure initialization code segment.

However, we note three things. First, once our partitioning optimization (Section 8.2.1) has been applied, LegoBase requires intermediate data structures mostly for aggregate operators, whose initialization code segment we remove, as described above. Second, particularly for TPC-H, there is no key that is the result of an intermediate join operator deeply nested in the query plan. Instead, TPC-H uses attributes of the original relations to access most data structures, attributes whose value range can be accurately estimated during data loading through statistics, as we discussed previously. Finally, for TPC-H queries the key value range is very small, typically ranging up to a couple of thousand sequential key values[10]. These three properties allow to completely remove initialization overheads and the associated unnecessary computation for all TPC-H queries.

[10] 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.

8.6 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 optimizing compiler.

8.6.1 Removal of Unused Relational Attributes

In Section 8.3 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.

8.6.2 Removing Unnecessary Let-Bindings

The SC compiler uses the Administrative Normal Form (ANF) [Flanagan et al., 1993] when generating code. This simplifies code generation for the compiler. However, it has the negative effect of introducing many unnecessary intermediate variables. We have observed that this form of code generation not only affects code compactness but also significantly increases register pressure. To improve upon this situation, SC uses a technique first introduced by the programming language community [Sumii and Kobayashi, 2001], 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 possibly expensive computation). SC then replaces any appearance of this variable later in the code with its initialization value. We have noticed that this optimization makes the generated code much more compact and reduces register pressure, resulting in improved performance. Moreover, we have observed that since the variable initialization time may take place significantly earlier in the code of the program than its actual use, this does not allow for this optimization opportunity to be detected by low-level compilers like LLVM.

Finally, our compiler applies a technique known as *parameter promotion*[11]. 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. We have observed that this optimization significantly reduces the number of memory accesses occurring during query execution.

[11] This technique is also known as Scalar Replacement in the PL community.

8.6.3 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 range 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.

8.7 Classification of LegoBase Optimizations

In this section, we classify the LegoBase optimizations according to (a) their generality and (b) whether they follow the rules of the TPC-H benchmark, which we use in our evaluation. These two metrics are important for a more thorough understanding of which categories of database systems can benefit from these optimizations. We detect six groups of optimizations, illustrated in Figure 8.8, described next in the order they appear from left to right in the figure.

Generic Compiler Optimizations: In this category, we include optimizations which are also applied by traditional compilers, such as LLVM. These include Dead Code Elimination (DCE), Common Subexpression Elimination (CSE), Partial Evaluation (PE) and the Scalar Replacement optimization presented in Section 8.6.2. These optimizations are TPC-H compliant and do not require any particular domain-specific knowledge; thus they can be applied for optimizing any input query as well as the code of the query engine.

Fine-grained Optimizations: In this TPC-H compliant category we include, as described in Section 8.6.3, fine-grained optimizations that aim to transform and improve the performance

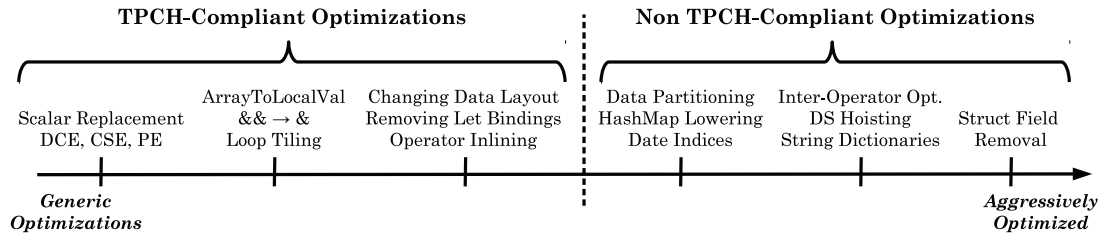

Figure 8.8 – Classification of LegoBase optimizations, based on (a) whether or not they adhere to the implementation rules of the TPC-H workload and (b) the amount of domain-specific and query-specific information they utilize.

of individual statements (or a small number of contiguous statements). We do not list this category alongside the generic compiler optimizations, as whether they improve the performance or not depends on the characteristics of the input query. Thus, SC needs to analyze the program before detecting whether the application of one of the optimizations in this group is beneficial.

Optimizing Data Accesses: The two optimizations presented in Sections 8.3 and 8.6.2, alongside the generic *operator inlining* optimization, aim to improve performance by minimizing the number of function calls and optimizing data accesses and code compactness. Even though they are coarse-grained in nature, affecting large code segments, they are still TPC-H compliant, as they are neither query specific nor depend on type information.

Partitioning and Indexing Optimizations: This class of optimizations, presented in detail in Section 8.2, can significantly improve query execute performance. However, even though they provide significant performance improvement (as we show in our evaluation), they are not TPC-H compliant, as this workload does not allow logical replication of data using more than one primary or foreign key. Similarly, our HashMap lowering optimization requires knowledge of the domain of the aggregation keys in advance. Still, there is a class of database systems that can greatly benefit from such indexing and partitioning transformations. These include systems that have all their data known in advance (e.g. OLAP style processing) or systems where we can introduce pre-computed indexing views, as in the case of Incremental View Maintenance (IVM).

Inter-Operator, String Dictionaries, and Domain-Specific Code Motion Optimizations: The three optimizations in this category, presented in Sections 8.1, 8.4 and 8.5 respectively, aim to remove unnecessary materialization points and computation from the critical path. However, they are *query specific*, as they can only be applied if a query is known in advance. This is the primary characteristic that differentiates this category of optimizations from the previous one. They also depend on type information and introduce auxiliary data structures. Thus, they are not TPC-H compliant.

Struct Field Removal Optimization: The most aggressive optimization that LegoBase applies removes unnecessary relational attributes from C structs. This optimization is query specific and is highly dependent on type information. It also requires specializing the data structures during data loading (to remove the unnecessary fields). Thus, it is not TPC-H compliant.

9 Experimental Evaluation of LegoBase

In this chapter, 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 9.2).

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 9.3).

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 9.4).

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 9.5).

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 9.6).

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 9.7).

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[1]	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 9.1 – 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.

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 9.8).

9.1 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 9.1). For compiling the generated programs throughout the evaluation section, we use version 2.11.4

[1] We note that according to the TPC-H specification rules, a database system can employ data partitioning (as described in Section 8.2.1) 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).

of the Scala compiler and version 3.4.2 of the CLang front-end for LLVM [Lattner and Adve, 2004], with the default 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) [The GNOME Project, 2013] whenever we need to generate generic data structures in C.

For our evaluation, we use the TPC-H benchmark [Transaction Processing Performance Council, 1999]. 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 9.1) 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 E.

As a reference point for most results presented in this chapter, 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 Chapter 7, LegoBase uses query plans from the DBX database. We also use the query compiler of the HyPer system [Neumann, 2011] (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.

9.2 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[2]).

As shown in Figure 9.1, the achieved performance is very poor: the unoptimized query engine, LegoBase(Naive/C)–LLVM, is significantly slower for all TPC-H queries, requiring more than

[2] [Shaikhha et al., 2016] presents an easy-to-follow example and an analysis of why general-purpose compilers need to operate in this fashion.

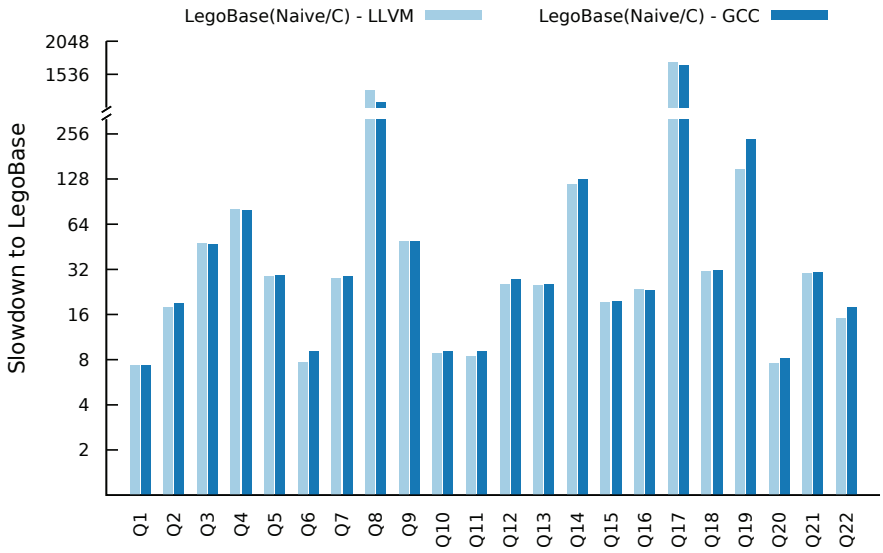

Figure 9.1 – Performance of a push-style engine compiled with LLVM and GCC. These engines are generated using *only* operator inlining. The baseline is the performance of the optimal generated code, LegoBase(Opt/C), with all optimizations enabled.

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[3]. 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.

9.3 Optimizing Query Engines Using Template Expansion

Next, we compare our approach – which compiles the *entire* query engine and utilizes *query-specific* information – with the compiler of the HyPer database [Neumann, 2011]. HyPer

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

performs template expansion through LLVM in order to inline the relational operators of a query executed on a push engine[4]. The results are presented in Figure 9.2.

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 8.7, have all queries or data known in advance.

To begin with, Figure 9.2 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 [Boncz et al., 2014], 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 9.2 also shows that by using all other optimizations of LegoBase (as they were presented in Chapter 8), 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).

[4]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.

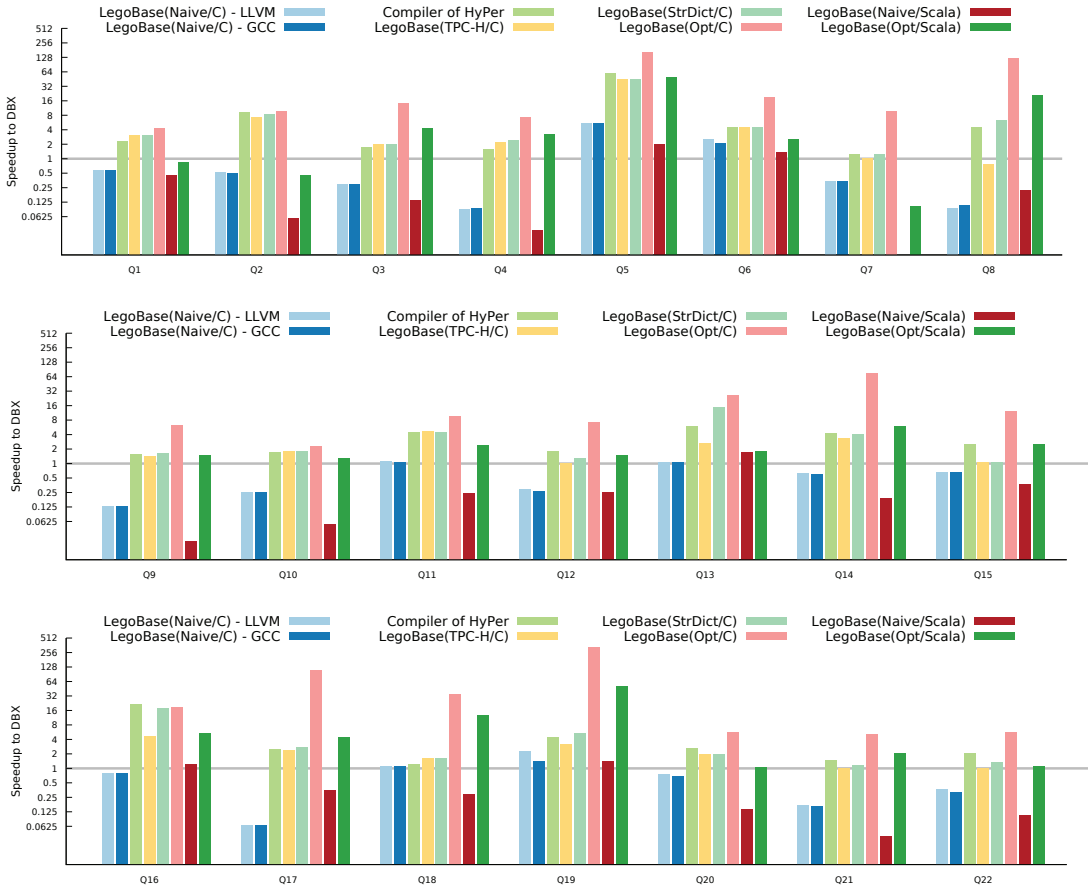

Figure 9.2 – Performance comparison of various LegoBase configurations (C and Scala programs) with the code generated by the query compiler of [Neumann, 2011]. The baseline for all systems is the performance of the DBX commercial database system. The absolute execution times for this figure can be found in Appendix B.

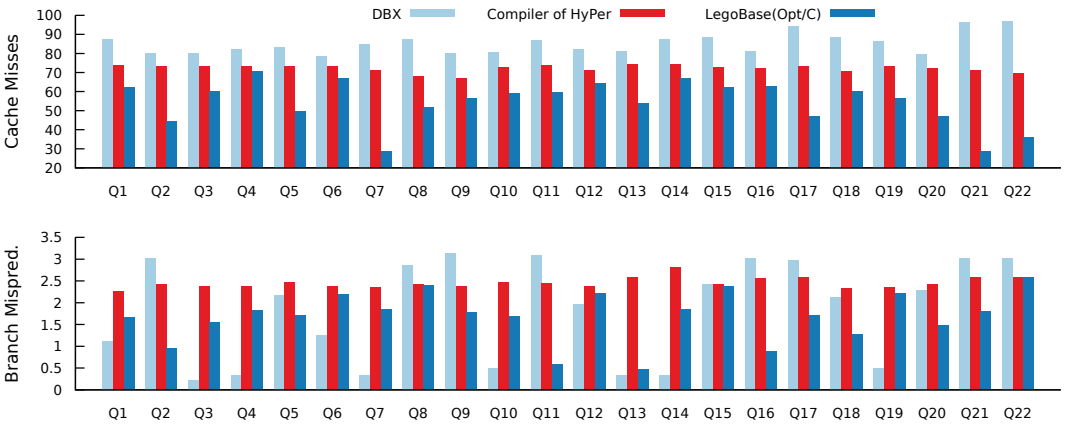

Figure 9.3 – Percentage of cache misses and branch mispredictions for DBX, HyPer and LegoBase(Opt/C) for all 22 TPC-H queries.

Such optimizations result in improved cache locality and branch prediction, as shown in Figure 9.3. 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.

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.

9.4 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 9.2 also presents performance results for both a naive and an optimized Scala query engine, named LegoBase(Naive/Scala) 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*[5] 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[6]. 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.

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

[6] A publication from Google [Hundt, 2011] 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.

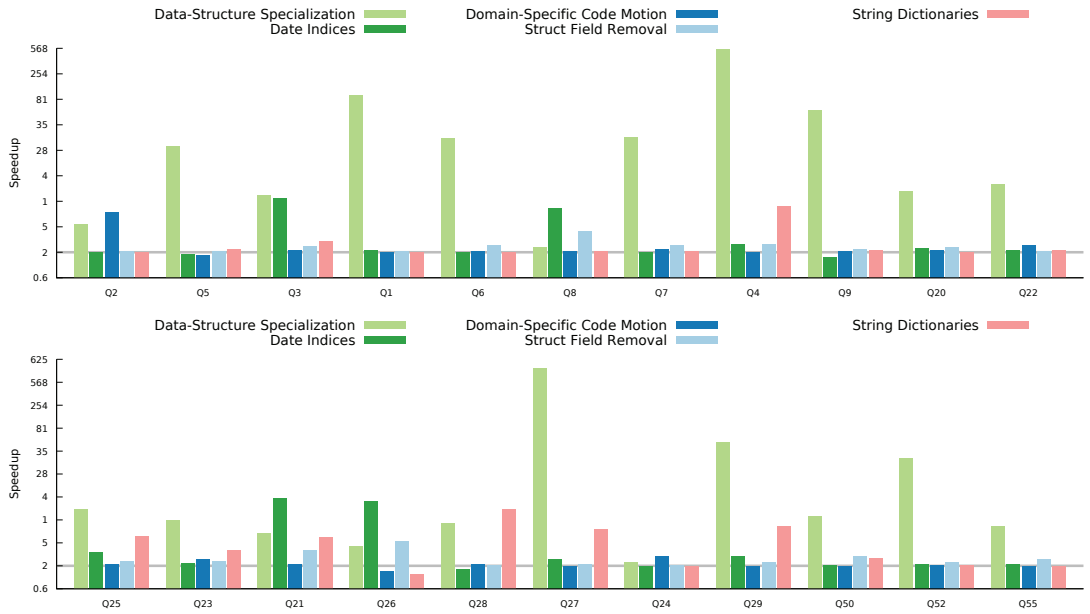

Figure 9.4 – Impact of different LegoBase optimizations on query execution time. The baseline is an engine that does not apply this optimization.

9.5 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 9.4, 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 8.2.1 and 8.2.2). 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 8.2.3) 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×[7]. 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

[7] 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.

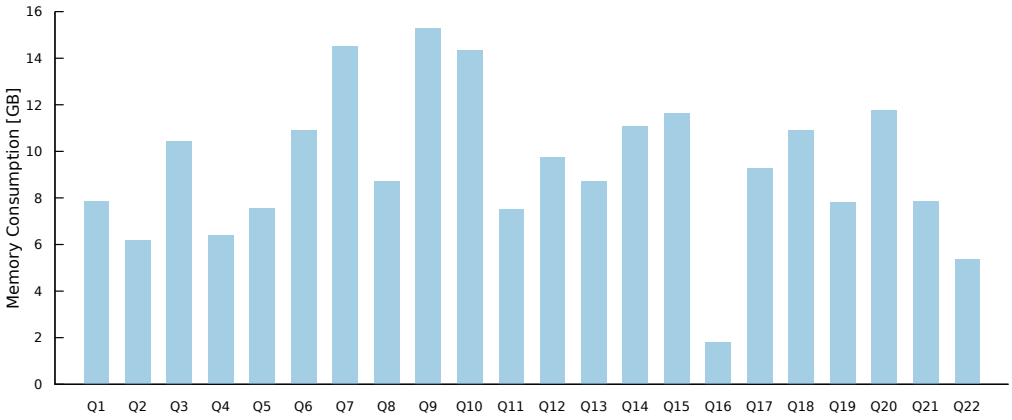

Figure 9.5 – Memory consumption of LegoBase(Opt/C) for the TPC-H queries.

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.

9.6 Memory Consumption and Overhead on Input Data Loading

Figure 9.5 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 8.2) in order to achieve optimal performance.

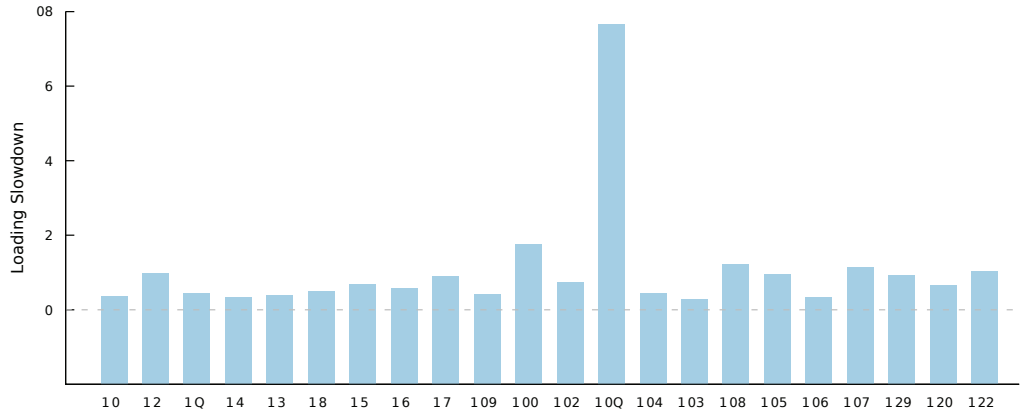

Figure 9.6 – Slowdown of input data loading occurring from applying all LegoBase optimizations to the C programs of the TPC-H workload (scaling factor 8).

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 9.6 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 B, 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.

9.7 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 9.2.

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

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 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 9.2 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[8] 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

[8]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 [Beeri and Kornatzky, 1990; Goldberg and Paige, 1984]. A decent loop fusion is still an open topic in the PL community [Svenningsson, 2002; Coutts et al., 2007; Gill et al., 1993].

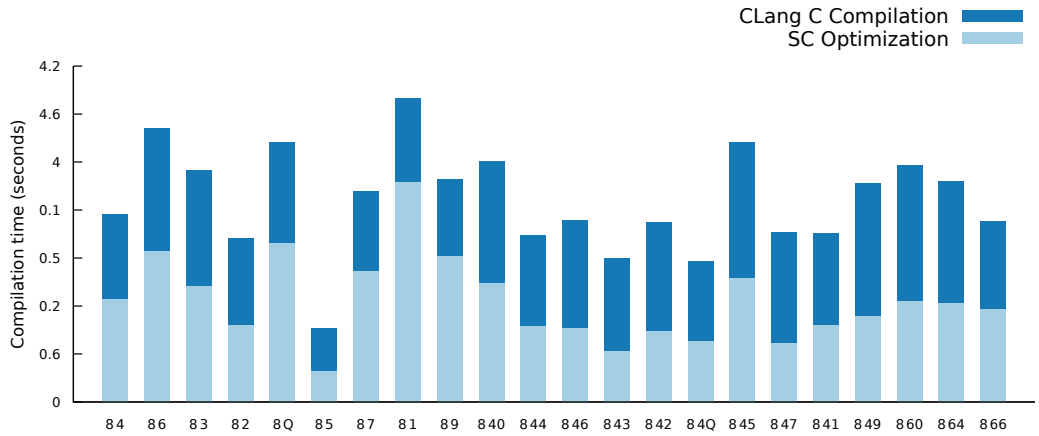

Figure 9.7 – Compilation time (in seconds) of all LegoBase(Opt/C) programs.

introduced in SC (e.g. those required for custom data types and operations on those types).

9.8 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 9.7, where the y-axis corresponds to 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.

10 Related Work

We outline related work in six areas:

(a) Existing approaches in the domain of program synthesis and automatic synthesis of out-of-core algorithms,

(b) Work on manual optimization of out-of-core algorithms for specific memory hierarchies,

(c) Work on algebraic manipulation and costing of algorithms in order to produce more efficient programs,

(d) Previous compilation frameworks for optimizing query engines and data processing systems,

(e) Frameworks for applying intra-operator optimizations,

(f) Orthogonal techniques to speed up query processing and, finally,

(g) A brief summary of work on domain-specific compilation in the Programming Languages (PL) community, a field of study that closely relates to ours.

We discuss these areas in more detail below.

Program Synthesis and Synthesis of Out-of-Core Algorithms. OCAS is closely related to the general field of program synthesis, and more specifically to that of *transformational program synthesis*, which generally aims to transform an inefficient specification program to a more efficient program using general rewrite rules that capture the logical laws of a particular *domain*. The basic motivation behind work in this field – which is also the motivation behind OCAS – is that for many programming problems it is relatively easy to produce and verify a correct, prototype implementation in a comprehensible, abstract style. Yet, these typically lack efficiency, making them unacceptable for practical purposes. Thus, we can only consider them as an abstract specification of the "real", efficient program [Kreitz, 1998]. Stated otherwise, transformational program synthesis is concerned with the search for an *efficient*, optimized

program starting from an inefficient specification, rather than finding a provably *correct* program matching the specification.

The rewrite rules used by transformational program synthesis are typically domain specific, since automatic synthesis is a hard problem to solve in general. This is because general solutions to program synthesis force the synthesis system to derive an algorithm almost from scratch and to re-invent well-known algorithmic principles. In addition, solving complex programming problems heavily relies on knowledge about application domains and standard programming techniques. One can hardly expect a synthesis system to be successful if such expertise is not explicitly embedded. Thus, it is easier to make such knowledge explicit, and develop synthesis systems on the basis of such knowledge, rather that attempting to derive it [Kreitz, 1998]. This is often referred to as *Knowledge-based program synthesis* and is also performed by OCAS using its library of program transformation rules, which encode well-known data-locality principles, such as buffering and caching.

Even though the foundations of transformational synthesis have been set for decades [Manna and Waldinger, 1979; Darlington and Burstall, 1976], only with recent advances has a rather broad class of synthesis systems reached the level of practical applications [Bodik and Jobstmann, 2013]. To this end, there is actually very little work on automating out-of-core algorithm design. One notable example in this direction is the StreamBit system [Solar-Lezama et al., 2005] which synthesizes efficient bit-streaming programs. However, in contrast to our approach, StreamBit performs semi-automatic program generation and requires a sketch of the optimized algorithm. In addition, the idea of automatic program transformation for loop-based, out-of-core algorithms is also present in [Krishnan et al., 2004]. However, this work does not take into account the characteristics of the architecture as OCAS does. This makes the approach limited as new architectures become available.

Synthesis appears in domains other than data management as well. For instance, hardware-specific synthesis of linear transforms and other mathematical functions is the aim of SPIRAL [Püschel et al., 2011]. Our system addresses a different domain of programs, that of data management. Similarly to OCAS – which uses OCAL for encoding data processing algorithms such as joins – SPIRAL uses its own domain-specific language, based on linear algebra, for expressing the specifications. However, the main difference between SPIRAL and our work is that the former needs to *actually* execute each generated program during the search space exploration (vs the cost-based optimization of OCAS). In our domain, such actual execution may be prohibitive, given that traditional out-of-core algorithms such as joins may run for very long periods (e.g. up to hours or days). Finally, in the domain of commercial scheduling algorithms, transformational synthesis has long been shown to be able to generate algorithms that are much more efficient than any hand-coded implementation [Smith and Parra, 1993; Gomes et al., 1996].

An approach that is related to both the LegoBase and OCAS systems presented in this thesis is the P2 Lightweight DBMS Generator [Batory and Thomas, 1997]. Similarly to OCAS, P2

addresses the problem of designing efficient programs based on some abstract specification provided by developers. In addition, and similarly to LegoBase, P2 argues for the decomposition of a database system into well-defined components. To this end, P2 uses a custom, low-level language which is a super-set of C, with the primary objective being reuse identification rather than productivity concerns of developers (the latter is, however, the main motivation behind LegoBase). In addition, P2 is mostly a component-based generator for the domain of container data structures employed by the database system, and thus, in contrast to LegoBase, this framework cannot encode well-known database optimizations such as dictionary encoding for attributes of string type.

Finally, program synthesis can be also used to optimize the code of database applications or automatically synthesize efficient code for specific data structures from declarative specifications. As an example of the former, the QBS system [Cheung et al., 2013] shows how synthesis can be used to automatically transform fragments of application logic into SQL queries with lifting, which are then optimizable by an off-the-shelf database query planner. For the latter, synthesis has been shown to be able to generate efficient, imperative, low-level code from high-level, declarative specifications both for the single-threaded [Smaragdakis and Batory, 1997; Hawkins et al., 2010, 2011] and concurrent setting [Hawkins et al., 2012].

Manual Optimization of Out-of-Core Algorithms. A great amount of work has been done on *manually* developing specialized out-of-core algorithms for various tasks and memory hierarchies. In our work, we often refer to canonical algorithms for certain database management tasks. Their descriptions can be found in the standard textbooks. For example, our join variants are presented in [Ramakrishnan and Gehrke, 2002], while our version of External Merge-Sort was originally introduced by Don Knuth for sorting on tapes [Knuth, 1998].

More recently, effort has been invested on designing algorithms for flash memory [Park and Shim, 2009; Liu et al., 2011; Andreou et al., 2009], the intricate memory hierarchies of graphics cards [Cederman and Tsigas, 2008; Govindaraju et al., 2006; Sintorn and Assarsson, 2008; Ye et al., 2010], and multi-level memory hierarchies [Kim et al., 2010]. These papers demonstrate that the state-of-the-art in developing out-of-core algorithms is to manually carry out ad-hoc effort; one cannot yet rely on automation or a clear design methodology.

In the Sequoia project [Ren et al., 2008; Fatahalian et al., 2006; Houston et al., 2008] a general-purpose C-like language is presented that has *explicit* knowledge of the topology of the machine, and allows writing programs that efficiently utilize the hierarchy and the available parallelism. The Sequoia system does not perform software synthesis, so the programmers must specify the out-of-core algorithms themselves. However, it still handles other aspects of out-of-core algorithms like our tool, such as parameter selection and an abstract representation of the memory hierarchy.

Algebraic Manipulation and Costing of Algorithms. OCAS performs algebraic manipulations to obtain more efficient equivalent programs; this idea can also be found in work on functional programming [Meijer et al., 1991; Bird, 1989]. Recursion schemas like folding [Gibbons, 2003]

also play an important role in our work, especially for our sorting algorithms. Discussions of more general recursion schemas are presented in [Augusteijn, 1999] and [Vene and Uustalu, 1998].

A language construct that has received particular attention in the domain of program transformations is the *for loop*. In this context, there exist a number of approaches that attempt to derive desirable loop organizations using a cost model and/or a compiler in order to improve data locality [McKinley et al., 1996; Kandemir et al., 1998; Lam et al., 1991; Krishnan et al., 2004]. In contrast, our synthesis framework proposes the use of a new, high-level language that (i) supports a wider set of constructs and (ii) is extensible to allow for easy addition of new definitions.

On a more fine-grained granularity, *superoptimization* is the problem of finding an optimal sequence of instructions that implements a certain function, which is usually specified with a sub-optimal implementation. The Denali superoptimizer [Gulwani et al., 2011] is one of the systems that tackles this problem by producing all programs (up to a bound) derivable from the specification using a given set of expression equality axioms and selecting the fastest program. OCAS uses a similar search strategy, however it performs a more coarse-grained optimization using program rewrite transformation rules that can completely change the structure and instructions used by an OCAL program (versus the basic instruction re-ordering rules of Denali).

Costing of programs is a fundamental component when designing frameworks for the transformation and algebraic manipulation of algorithms. For static analysis of the running costs of functional programs, we draw inspiration from [Jost et al., 2010; Hofmann and Jost, 2003; Chin and Khoo, 1999; Danielsson, 2008]. COSTA [Albert et al., 2011] is a general-purpose cost estimation system for Java byte-code. This makes it applicable to languages more powerful than the one described in this thesis. However, for the same reason, COSTA often fails to deduce bounds as tight as those of our system which works with a restricted, custom-designed language. In particular, for the Merge-Sort algorithm that we use in one of our examples, we could not bring COSTA to estimate the asymptotically correct cost bound of $O(n \log n)$.

Previous Compilation Approaches for Optimizing Query Engines and Data Processing Systems. Historically, System R [Chamberlin et al., 1981] first proposed code generation for query optimization. However, the Volcano iterator model eventually dominated over compilation, since code generation was very expensive to maintain. The Daytona system [Greer, 1999] revisited compilation in the late nineties; however, it heavily relied on the operating system for functionality that is traditionally provided by the DBMS itself, like buffering.

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, by optimizing away the overheads of traditional database abstractions

like the Volcano operator model [Graefe, 1994]. Examples of in-memory industrial systems in the area since the mid-2000s (with or without compilation) include SAP HANA [Färber et al., 2012], VoltDB [Stonebraker et al., 2007; Kallman et al., 2008] and Oracle's TimesTen [Oracle Corporation, 2006]. In the academic context, interest in query compilation has also been renewed since 2009 and continues to grow [Rao et al., 2006; Zane et al., 2008; Ahmad and Koch, 2009; Grust et al., 2009; Koch, 2010; Krikellas et al., 2010; Neumann, 2011; Koch, 2013; Koch et al., 2014; Koch, 2014; Nagel et al., 2014; Viglas et al., 2014; Armbrust et al., 2015; Goel et al., 2015].

Despite the differences between the individual approaches, all compilation frameworks generate *on-the-fly* an *optimized* query evaluation engine for each incoming SQL query. More importantly, most existing query compilers are, to the best of our knowledge, *template expanders* at heart. As we discussed in Chapter 1 of this thesis, a template expander is a procedure that, simply speaking, generates *low-level* code in *one* direct macro expansion step. This means that, while a query interpreter immediately *calls* the operator implementations listed in a query plan, the template expander first *inlines* the code of each operator, to obtain low-level code for the *entire* plan. While inlining, the template expander may also apply specific optimizations to the code of each *individual* operator, before calling the final program. We briefly discuss most of the aforementioned systems next.

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) [Rao et al., 2006]. The HIQUE system takes a step further and completely eliminates the Volcano iterator model in the generated code [Krikellas et al., 2010]. 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 [Zane et al., 2008].

The query compiler of the HyPer database system also uses query compilation, as described in [Neumann, 2011]. This work targets minimizing the CPU overhead of the Volcano operator model while maintaining low compilation times. The authors use a mixed LLVM/C++ execution engine where the algebraic representation of the operators is first translated to low-level LLVM code, while the complex part of the database (e.g. management of data structures and memory allocation) is still *precompiled* C++ code called periodically from the LLVM code whenever needed. Two basic optimizations are presented: operator inlining and reversing the data flow (to a push engine).

All these works aim to improve database systems by removing unnecessary abstraction overhead. However, these *template-based* approaches require writing low-level code which is hard to maintain and extend. This fact significantly limits their applicability. Furthermore, their static nature makes them miss significant optimization opportunities that exist in the precompiled components of the database system. In contrast, our approach advocates a new methodology for programming query engines where the query engine and its optimizations are written in a *high-level* language. This provides a programmer-friendly way to express opti-

mizations and allows extending the scope of optimization to cover the whole query engine. In addition, and in contrast to previous work, we separate the optimization and code generation phases. Even though [Neumann, 2011] argues that optimizations should happen completely before code generation (e.g. in the algebraic representation), there exist many optimization opportunities that occur only *after* one considers the complete generated code, e.g. after operator inlining. Our compiler can detect such optimizations, thus providing additional performance improvement over existing techniques.

Compilation has also been used to optimize systems for Incremental View Maintenance (IVM). In particular, the DBToaster project [Ahmad and Koch, 2009; Koch, 2010; Koch et al., 2014] uses compiled C++ or Scala code to incrementally maintain internal representations of materialized views. Experimental results show that through compilation, DBToaster can improve the performance of IVM by several orders of magnitude compared to state-of-the-art alternatives. LegoBase targets the optimization (through compilation) of a different domain, that of analytical query processing.

Furthermore, with the increasing popularity of language-intergrated query languages, such as LINQ, work has been carried out in order to boost the performance of these languages and that of their managed runtimes using database-inspired strategies and optimizations [Grust et al., 2009; Murray et al., 2011; Nagel et al., 2014; Viglas et al., 2014]. In general, all these techniques employ compilation to convert high-level LINQ programs to more efficient, low-level code. We believe that this line of work, despite making several contributions, is orthogonal to the approach of LegoBase. This is because all systems in this category do not target the optimization of a database system, but rather making query processing of collections in the memory space of the application more efficient by leveraging database technology. Still, it would be interesting to examine how each of these two research directions could benefit from applying methodologies develop for the other.

Finally, in the distributed setting, Tupleware [Crotty et al., 2014, 2015] targets the optimization of workflows of user-defined functions (UDFs) that specify the individual algorithmic steps of complex analytics tasks such as those encountered in machine learning and advanced statistics. Similar to LegoBase, Tupleware allows developers to express their workflows using specific high-level languages and employs compilation for optimizing them. However, Tupleware heavily utilizes the low-level, compilation framework LLVM for expressing most of the optimizations. This fact severely limits the productivity of software developers, inheriting all drawbacks of the template-expansion-based query compilation approaches as described so far in this thesis. Finally, as we discussed in Section 7.5, LegoBase compiles the *entire* system; thus, it makes no distinction between the code of UDFs and that of the remaining system, thus allowing for a broader scope of compilation and optimization.

Frameworks for applying intra-operator optimizations. 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 [Zhang et al., 2012a,b,c]. 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 LegoBase.

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 [Zhang et al., 2012a]. 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 LegoBase. 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.

Techniques to speed up query processing. There are many works that aim to speed up query processing in general, by focusing mostly on improving the way data is processed rather than individual operators. Examples of such works include block-wise processing [Padmanabhan et al., 2001], vectorized execution [Sompolski et al., 2011], compression techniques to provide constant-time query processing [Raman et al., 2008] or a combination of the above along with a column-oriented data layout [Manegold et al., 2009]. We believe all these approaches are orthogonal to this work since our framework aims to provide 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 8.3).

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 [Hudak, 1996; Faith et al., 1997; van Deursen et al., 2000; Kennedy et al., 2005; Rompf and Odersky, 2010; Ackermann et al., 2012; Lee et al., 2011; Jovanović et al., 2014; Humer et al., 2014] 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.

11 Conclusions and Future Work

In this thesis, we draw inspiration from the recent usage of high-level languages for building high-performance computer systems to argue that it is now time for a radical rethinking of the methodologies and techniques used when developing database management systems.

We show that the advanced software features offered by high-level programming languages can be leveraged to significantly boost the productivity of database developers without experiencing a negative impact on performance. This approach, which was previously called *abstraction without regret*, makes it easier to introduce innovative techniques and optimizations into the code-base of the database system, as new hardware architectures become available. More concretely, in this thesis we make the following two contributions:

First, we describe OCAS, the out-of-core algorithm synthesizer. By providing a memory hierarchy oblivious algorithm expressed in a high-level language and an abstract representation of the memory hierarchy, OCAS can automatically generate an efficient out-of-core version of the naive algorithm by exploiting the characteristics of the memory hierarchy. OCAS applies transformation rules to a program and exhaustively searches the space of semantically equivalent generated programs to locate the one with the best performance metric. This metric is an estimation of the data transfers occurring at execution time, is derived by analyzing the individual expressions in it, and is an approximation of the program's actual execution time.

Our preliminary results show that OCAS adapts the generated algorithms to changes in the memory hierarchy and that it produces optimized versions of out-of-core algorithms quickly. Its estimations are accurate when I/O cost dominates CPU cost. Otherwise, the underestimation increases proportionally with the CPU costs. However, this underestimation does not affect the correctness of the approach, as OCAS can always differentiate between more efficient and less efficient algorithms. Finally, OCAS efficiently performs estimation of parameters like buffer sizes, a task which is non-trivial for developers.

Second, we realize the abstraction without regret vision in the domain of ad-hoc, analytical query processing. We present LegoBase, a new analytical database system currently under

development at EPFL. In this thesis, we focus on the query execution subsystem of LegoBase. We show that the key technique to admit the productivity/efficiency combination of the abstraction without regret vision 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 a number of database-specific optimizations naturally at a high level and use it to optimize the *entire* query engine. 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.

Future Work. Given the easy extensibility offered by both frameworks presented in this thesis, there is a multitude of opportunities and future research directions. We briefly discuss some of these directions next.

First, with respect to the OCAS synthesizer, we noticed that our tool underestimates the execution costs of the program in some cases. To overcome this limitation, we need to develop more precise cost estimations and modeling of the computational costs. This is admittedly a challenging task, as there are simply too many parameters that affect CPU costs, and a combination of experimental and analytical methods needs to be developed [Manegold, 2002]. In addition, it is worth investigating how we could use OCAS to tune algorithms deployed in heterogeneous deployments automatically; such environments typically use, along with traditional CPUs, other types of processing units like GPU or FPGAs. Finally, OCAS could be used to optimized the operators employed by the LegoBase query engine, for a specific memory architecture. Given that both OCAS and LegoBase use high-level languages for expressing the system and related algorithms, but more importantly, because both approaches generate C code, the combination of these two techniques is certainly feasible.

Second, there are a number of extensions and optimizations that can be introduced in the LegoBase query engine in order to further boost its performance. To begin with, since LegoBase is currently single-threaded, one natural first extension would be to introduce parallelism into the system. In particular, intra-operator (or partitioned) parallelism has already been widely studied (e.g. in [Graefe, 1994; Mehta and DeWitt, 1995]) and LegoBase could be easily extended in this direction as we already outlined in Section 7.5, without requiring modifications to the rest of the components or transformations of the query engine that are oblivious to parallelism (e.g. join and aggregate operators should remain the same). Second, performance could be similarly improved by introducing other types of data partitioning schemes, such as range partitioning, or SIMD instructions (e.g. in the spirit of [Zhou and Ross, 2002]). LegoBase already uses range partitioning for attributes of *date* type; this technique

should be extended to handle attributes of other types as well. SIMD-based processing could, in fact, be a more memory-efficient alternative to the string dictionaries used by our query engine.

In general, the performance improvement obtained by applying optimizations like the ones presented in this thesis and the ones highlighted above is *always* subject to the characteristics of the input data and the queries executed. We view LegoBase as a platform for easy experimentation of database optimizations. Thus, the development of a framework that estimates the potential benefits and trade-offs of applying an optimization would be beneficial to developers. The insights gained by our work with the cost optimization of the OCAS synthesizer are certainly helpful in this direction.

A OCAL programs of Table 5.1

BNL – No writeout

```
(λ⟨R, S⟩.
  for (xBlock [k1] ← R)
    for[HDD⇝RAM] (yBlock [k2]←S)
      for (x ← xBlock)
        for (y ← yBlock)
          if joinCond(x,y) then [⟨x,y⟩] else [])
(if length(R) ≤ length(S) then ⟨R, S⟩ else ⟨S, R⟩)
```

BNL with cache – No writeout

```
(λ⟨R, S⟩.
  for (xBlock [k1] ← R)
    for[HDD⇝RAM] (yBlock [k2]←S)
      for (xCacheLine ← xBlock)
        for[RAM⇝Cache] (yCacheLine ← yBlock)
          for (x ← xCacheLine)
            for (y ← yCacheLine)
              if joinCond(x,y) then [⟨x,y⟩] else [])
(if length(R) ≤ length(S) then ⟨R, S⟩ else ⟨S, R⟩)
```

BNL writing to HDD

```
(λ⟨R, S⟩.
  for (xBlock [k1]← R) [RAM⇝HDD][k3]
    for (yBlock [k2]← S)
      for (x ← xBlock)
        for (y ← yBlock)
          if joinCond(x,y) then [⟨x,y⟩] else [])
(if length(R) ≤ length(S) then ⟨R, S⟩ else ⟨S, R⟩)
```

BNL writing to other HDD

```
(λ⟨R, S⟩.
  for (xBlock [k1]← R) [RAM⇝HDD2][k3]
    for[HDD1⇝RAM] (yBlock [k2]← S)
      for (x ← xBlock)
        for (y ← yBlock)
          if joinCond(x,y) then [⟨x,y⟩] else [])
(if length(R) ≤ length(S) then ⟨R, S⟩ else ⟨S, R⟩)
```

BNL writing to flash

Essentially the same as before, with only changes to the devices specified

```
(λ⟨R, S⟩.
  for (xBlock [k1]← R) [RAM⇝FD][k3]
    for[HDD⇝RAM] (yBlock [k2]← S)
      for (x ← xBlock)
        for (y ← yBlock)
          if joinCond(x,y) then [⟨x,y⟩] else [])
(if length(R) ≤ length(S) then ⟨R, S⟩ else ⟨S, R⟩)
```

We note that, even though some of the block nested loops join variants presented above are structurally similar, the sequentiality annotations for the input and output actually differ, thus leading to different cost formulas (as was discussed in Chapter 5.

(GRACE) hash-join – No writeout

As discussed in Section 4.2, the hash-join in OCAS is implemented by applying the hash−part program transformation rule to any Block Nested Loops join program, like those presented before.

External Sorting

Implemented as:

$$\mathsf{treeFold}[2^k]([], \mathsf{unfoldR}(\mathsf{funcPow}[k](\mathsf{mrg})))$$

where all functions used are defined in Figure 3.2.

Set Union

Implemented as unfoldR(uni) where uni is a function of type $\langle[\tau],[\tau]\rangle \to \langle[\tau],\langle[\tau],[\tau]\rangle\rangle$ defined as follows.

$\lambda\langle l_1,\ l_2\rangle$.
  if (length(l_1)==0 ∧ length(l_2)==0) then ⟨[], ⟨[],[]⟩⟩
  else if (length(l_1)==0) then ⟨[head(l_2)], ⟨[], tail(l_2)⟩⟩
  else if (length(l_2)==0) then ⟨[head(l_1)], ⟨tail(l_1),[]⟩⟩
  else if (head(l_1)<head(l_2)) then ⟨[head(l_1)],⟨tail(l_1), l_2⟩⟩
  else if (head(l_1)==head(l_2)) then ⟨[head(l_1)],⟨tail(l_1), tail(l_2)⟩⟩
  else ⟨[head(l_2)],⟨l_1, tail(l_2)⟩⟩

Note that the difference between uni and mrg (defined in Figure 3.2) is that, as expected, the former removes duplicate values while the latter maintains them.

Multiset Union (sorted list)

Implemented as unfoldR(mrg) where mrg is the function defined in Figure 3.2.

Multiset Union (value-multiplicity)

Implemented as unfoldR(multiuni) where multiuni is a function of type $\langle[\langle\tau,\mathsf{Int}\rangle],[\langle\tau,\mathsf{Int}\rangle]\rangle \rightarrow \langle[\langle\tau,\mathsf{Int}\rangle],\langle[\langle\tau,\mathsf{Int}\rangle],[\langle\tau,\mathsf{Int}\rangle]\rangle\rangle$ defined as follows.

$\lambda\langle l_1,\ l_2\rangle$.
  if (length(l_1)==0 ∧ length(l_2)==0) then ⟨[], ⟨[],[]⟩⟩
  else if (length(l_1)==0) then ⟨[head(l_2)],⟨[],tail(l_2)⟩⟩
  else if (length(l_2)==0) then ⟨[head(l_1)],⟨tail(l_1),[]⟩⟩
  else if (head(l_1).1<head(l_2).1) then ⟨[head(l_1)],⟨tail(l_1),l_2⟩⟩
  else if (head(l_1).1==head(l_2).1) then
    ⟨[⟨head(l_1).1, head(l_1).2 + head(l_2).2⟩], ⟨tail(l_1),tail(l_2)⟩⟩
  else ⟨[head(l_2)],⟨l_1,tail(l_2)⟩⟩

where type Int belongs to D, according to our language specification described in Chapter 3.

Multiset Difference (sorted list)

Implemented as unfoldR(diff) where diff is a function of type $\langle[\tau],[\tau]\rangle \rightarrow \langle[\tau],\langle[\tau],[\tau]\rangle\rangle$ defined as follows.

$\lambda\langle l_1,\ l_2\rangle$.
  if (length(l_1)==0) then ⟨[], ⟨[],[]⟩⟩
  else if (length(l_2)==0) then ⟨[head(l_1)], ⟨tail(l_1),[]⟩⟩
  else if (head(l_1)<head(l_2)) then ⟨[head(l_1)],⟨tail(l_1), l_2⟩⟩
  else if (head(l_1)==head(l_2)) then ⟨[],⟨tail(l_1), l_2⟩⟩
  else ⟨[],⟨l_1, tail(l_2)⟩⟩

Multiset Difference (value-multiplicity)

Implemented as unfoldR(multidiff) where multidiff is a function of type $\langle[\langle\tau,\mathsf{Int}\rangle],[\langle\tau,\mathsf{Int}\rangle]\rangle \rightarrow$

$\langle[\langle\tau,\mathsf{Int}\rangle],\langle[\langle\tau,\mathsf{Int}\rangle],[\langle\tau,\mathsf{Int}\rangle]\rangle\rangle$ defined as follows.

```
λ⟨l1, l2⟩.
  if (length(l1)==0) then ⟨[], ⟨[],[]⟩⟩
  else if (length(l2)==0) then ⟨[head(l1)],⟨tail(l1),[]⟩⟩
  else if (head(l1).1==head(l2).1) then
    if (head(l1).2−head(l2).2>0) then
      ⟨[⟨head(l1).1,head(l1).2−head(l2).2⟩],⟨tail(l1),tail(l2)⟩⟩
    else ⟨[],⟨tail(l1),tail(l2)⟩⟩
  else if (head(l1).1<head(l2).1) then ⟨[head(l1)],⟨tail(l1),l2⟩⟩
  else ⟨[],⟨l1,tail(l2)⟩⟩
```

Notice that, as expected, these two last definitions add only elements of list l_1 to the result, excluding those elements that exist in list l_2 as well. In addition, the type Int belongs to D, as before.

Removing Duplicates From a Sorted List

Implemented as unfoldR(dp) where dp is a function of type $\langle[\tau],\tau\rangle \rightarrow \langle[\tau],[\tau]\rangle$ defined as follows.

```
λ⟨l, last⟩.
  if (l.length==0) then ⟨[],[]⟩
  else if (head(l) == last) then ⟨[], tail(l)⟩
  else ⟨head(l), tail(l)⟩
```

Column Store Read (5 and 10 columns)

Implemented as unfoldR(cs) where cs is a function of type $\langle[\tau],\ldots,[\tau]\rangle \rightarrow \langle[\tau],\langle[\tau],\ldots,[\tau]\rangle\rangle$ defined as follows.

```
λ⟨l1,...,ln⟩.
  if (length(l1)==0) ⟨[], ⟨[],...,[]⟩⟩
  else ⟨[⟨head(l1),..., head(ln)⟩], ⟨tail(l1),..., tail(ln)⟩⟩
```

Note, that it is not possible for one of the attributes in a column store to have less rows than the other attributes for the same relation. This is why it suffices to just check the length of only the first attribute while performing row reconstruction.

Computing Aggregates

Most of the aggregate functions can be implemented in our synthesizer using the foldL construct of OCAL. Here, we present a more complex example of an aggregate function which calculates the maximum value of a relation, computed across all columns of every tuple of that relation. For simplicity we assume here that all columns are of type Int, but this assumption

can be easily relaxed. This function, called globalMax, is of type:

[[Int]]→Int

where each nested list represents one tuple and type Int belongs to D. Then, globalMax is defined as follows:

```
foldL( 0, λ⟨globalAgg, x⟩.
  (λagg.if agg > globalAgg then agg else globalAgg)
  (foldL( 0, λ⟨localAgg, xs⟩.
    if xs > localAgg then xs
    else localAgg
  )(x)))
```

where the local foldL calculates the maximum per tuple and the outer one updates the global maximum whenever needed.

B 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 B.1 – Execution times (in milliseconds) of Figure 9.1 and Figure 9.2. The various configurations of LegoBase are explained in more detail in Table 9.1 of this thesis.

Appendix B. 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 B.2 – Execution times (in milliseconds) of TPC-H queries with individual optimizations applied (as shown in Figure 9.4 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 B.3 – Memory consumption in GB, input data loading time in seconds, and optimization/compilation time in milliseconds as shown in Figure 9.5, Figure 9.6, and, Figure 9.7 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 B.4 – Cache Miss Ratio (%) and Branch Misprediction Rate (%) for DBX, HyPer and LegoBase, respectively, as shown in Figure 9.3 of this thesis.

C 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 8.2.1. 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
  }
}
```

D Converting a Volcano-style Query Engine to a Push-style Query Engine

As we discussed in Section 7.1 of this thesis, LegoBase supports both a classical Volcano-style [Graefe, 1994] query engine as well as a push-style query interface [Neumann, 2011]. The latter changes the flow of data processing in query engines. More specifically, it argues that operators should not *pull* data from other operators whenever needed (Volcano-style processing), but instead operators should *push* data to consumer operators. Data should then be continuously pushed until we reach a materialization point. It has been shown that this organization significantly improves cache locality and branch prediction [Neumann, 2011].

Let us assume for the moment that a DBMS is initially developed using the Volcano interface. This was indeed the case when the first version of the LegoBase query engine was developed. In this chapter, we present two ways to convert the Volcano-style operator interface to a push-style query engine, so that we take advantage of the performance benefits of the latter as were described above.

First, we can implement a push-style engine from scratch (thus switching from an iterator to a consumer/producer model). This, in turn, necessitates rewriting the implementation of all operators: with traditional, low-level approaches (which typically use the C programming language), this is a challenging and error-prone task considering that the logic of each operator is likely spread over multiple code fragments of complicated low-level software [Neumann, 2011].

However, when using high-level languages, this task becomes significantly easier, as the advanced software features of Scala allow us to write an implementation of the query operator interface that does not affect other, semantically independent components of the query engine. In essence, with these features, we are able to simply swap the Volcano-style implementation for the newly implemented push-style engine. Then, the transformation pipeline of SC allows us to easily turn off any Volcano-specific transformations (if any). In addition, all of the lower, operator-independent optimizations are still applicable and require no modifications, since they are oblivious to differences in operator semantics between the two engines.

```
case class HashJoin[B](leftChild: Operator,
   rightChild: Operator, hash: Record=>B,
   cond: (Record,Record)=>Boolean) extends
        Operator {
  val hm = HashMap[B,ArrayBuffer[Record]]()
  var it: Iterator[Record] = null
  def next() : Record = {
    var t: Record = null
    if (it == null || !it.hasNext) {
      t = rightChild.findFirst { e =>
        hm.get(hash(e)) match {
          case Some(hl) => it = hl.iterator; true
          case None => it = null; false
        }
      }
    }
    if (it == null || !it.hasNext) return null
    else return it.collectFirst {
      case e if cond(e,t) => conc(e, t)
    } get
  }
}
```

(a) The starting Volcano-style implementation.

```
case class HashJoin[B](leftChild: Operator,
   rightChild: Operator, hash: Record=>B,
   cond: (Record,Record)=>Boolean) extends
        Operator {
  val hm = HashMap[B,ArrayBuffer[Record]]()
  var it: Iterator[Record] = null
  def next(t: Record) {
    var res: Record = null
    while (res = {
      if (it == null || !it.hasNext) {
        hm.get(hash(t)) match {
          case Some(hl) => it = hl.iterator
          case None => it = null
        }
      }
      if (it == null || !it.hasNext) null
      else it.collectFirst {
        case e if cond(e,t) => conc(e, t)
      } get
    } != null) parent.next(res)
  }
}
```

(b) After the first two steps of the algorithm.

```
case class HashJoin[B](leftChild: Operator,
   rightChild: Operator, hash: Record=>B,
   cond: (Record,Record)=>Boolean) extends
        Operator {
  val hm = HashMap[B,ArrayBuffer[Record]]()
  var it: Iterator[Record] = null
  def next(t: Record) {
    if (it == null || !it.hasNext) {
      hm.get(hash(t)) match {
        case Some(hl) => it = hl.iterator
        case None => it = null
      }
    }
    while (it!=null && it.hasNext)
        it.collectFirst {
      case e if cond(e,t) =>
          parent.next(conc(e,t))
    }
  }
}
```

(c) After the third step of the algorithm.

```
case class HashJoin[B](leftChild: Operator,
   rightChild: Operator, hash: Record=>B,
   cond: (Record,Record)=>Boolean) extends
        Operator {
  val hm = HashMap[B,ArrayBuffer[Record]]()
  def next(t: Record) {
    hm.get(hash(t)) match {
      case Some(hl) => hl.foreach { e =>
        if (cond(e,t)) parent.next(conc(e,t))
      }
      case None => {}
    }
  }
}
```

(d) The final result after additional optimizations.

Figure D.1 – Transforming a HashJoin from a Volcano engine to a Push Engine. The lines highlighted in red and blue are removed and added, respectively. All branches and intermediate iterators are automatically eliminated. The *open* function (not shown) is handled accordingly.

Second, given the two types of engines, there actually exists a methodological way to obtain one from the other and to express this as a compiler transformation. We present the high-level ideas of this conversion next, using the HashJoin operator as an example (Figure D.1).

A physical query plan consists of a set of operators in a tree structure. For each operator, we can extract its children as well as its (single) parent. Operators call the *next* function of other children operators in the Volcano model to make progress in processing a tuple. An

operator can be the *caller*, the *callee* or even both depending on its position in the tree (e.g. an operator with no children is only the callee, but an operator in an intermediate position is both). Given a set of operators, we must take special care to (a) reverse the dataflow (turning callees to callers and vice versa) as well as (b) handle *stateful* operators in a proper way. The optimization outlined here handles these cases in the following three steps:

Turning callees to callers: When calling a *next* function in the Volcano model, a single tuple is returned by the callee[1]. In contrast, in a push model, operators call their parents whenever they have a tuple ready, as we explained previously. The necessary transformation is straightforward: instead of letting callees return a single tuple, we remove this return statement. Then, we put the whole operator logic inside a while loop which continues until the value that would be returned to the *original* callee operator is null (operator has completed execution). For each tuple encountered in this loop, we call the *next* function of the original parent. For scan operators, who are only callees, this step is enough to port these operators to the push-style query engine.

Turning callers to callees: The converse of the above modification should be performed: the original callers should be converted to callees. To do this, we remove the call to the *next* function of the child in the original caller, since in the push engine the callee calls the next function of the parent. However, we still need a tuple to process. Thus, this step changes all *next* functions to take a record as an argument, which corresponds to the value that would be returned from a callee in the Volcano engine. Observe that the call to *next* may be explicit or implicit through functional abstractions like the *findFirst* in line 9 of Figure D.1(a). In addition, calls to the *next* function may happen in the *open* function of the Volcano model for purposes of *state-initialization*. We handle the *open* function similarly. This step ports the Sort, Map, Aggregate, Select, Window, View and Print operators of LegoBase to the push engine[2].

Managing state: Finally, special care should be taken for *stateful* operators. The traditional example of such operators is the join variants (semi-join, hash-join, anti-join etc). For these operators, the tuples from the left child are organized in hash lists, matched on the join condition with tuples from the right child. Then, to avoid materialization, the join operator must keep state about how many elements have already been output from this list whenever there is a match. A nice abstraction for this is the *iterator* interface[3], where for each *next* call in the Volcano model the iterator is advanced by one (and one output tuple is produced). In this optimization, we change this behavior so that after the iterator is initialized, we exhaust it by calling the *next* function of the parent for each tuple in it.

[1] This assumes no block-style processing, where multiple tuples are first materialized and then returned as a unit. In general, LegoBase avoids materialization whenever possible.

[2] All operators initialize their state (if any) from one child in the open function, and call their other child (if any) in the next function. The only exception is the nested loop joins operator which calls both children in the next function. We handle this by introducing phases where each phase handles tuples only from one child.

[3] Observe that the *iterator* itself is an abstraction which introduces overheads during execution. Our compiler maps this high-level construct to efficient native C loops.

In addition, after this optimization, the staging compiler can further optimize the generated code, as shown in Figures D.1(c) and D.1(d). There, the compiler detects that both the iterator abstraction and some while loops can be completely removed, and automatically removes them, thus improving branch prediction.

However, it is more important to note that – despite the fact that the optimization's code closely follows the human-readable description given above – there are still plenty of corner cases that need to be handled on top of this baseline implementation. This is particularly true for stateful operators since each one of them manipulates its state in significantly different ways compared to the others. This, in turn, means that, while the conceptual description is straightforward, the optimization code becomes hard to develop and maintain.

To conclude, the important observation to be made at this point is that not all LegoBase optimizations need to be *compiler optimizations*. Every time developers want to introduce a new optimization into the LegoBase query engine, they must analyze the number of components that are affected by the new optimization. If this number is relatively small and/or the component to be optimized is expressed at a high-level of abstraction – as is the case with the query operator interface of this chapter – then it is highly probable that the new optimization can simply be written by swapping the old implementation with new, high-level Scala code. However, if the optimization is shared by multiple components and/or optimizes more intermediate language constructs – as is the case with our HashMap optimization – then it is probably better to express it using the compiler API of SC. We argue that this a simple to make decision, which further increases the productivity of developers, as they are not necessarily bound to use our compiler interfaces, depending on their optimization goals.

E TPC-H Schema and Queries

The TPC-H schema is shown in the following figure, which is taken from the original benchmark specification [Transaction Processing Performance Council, 1999]. 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.

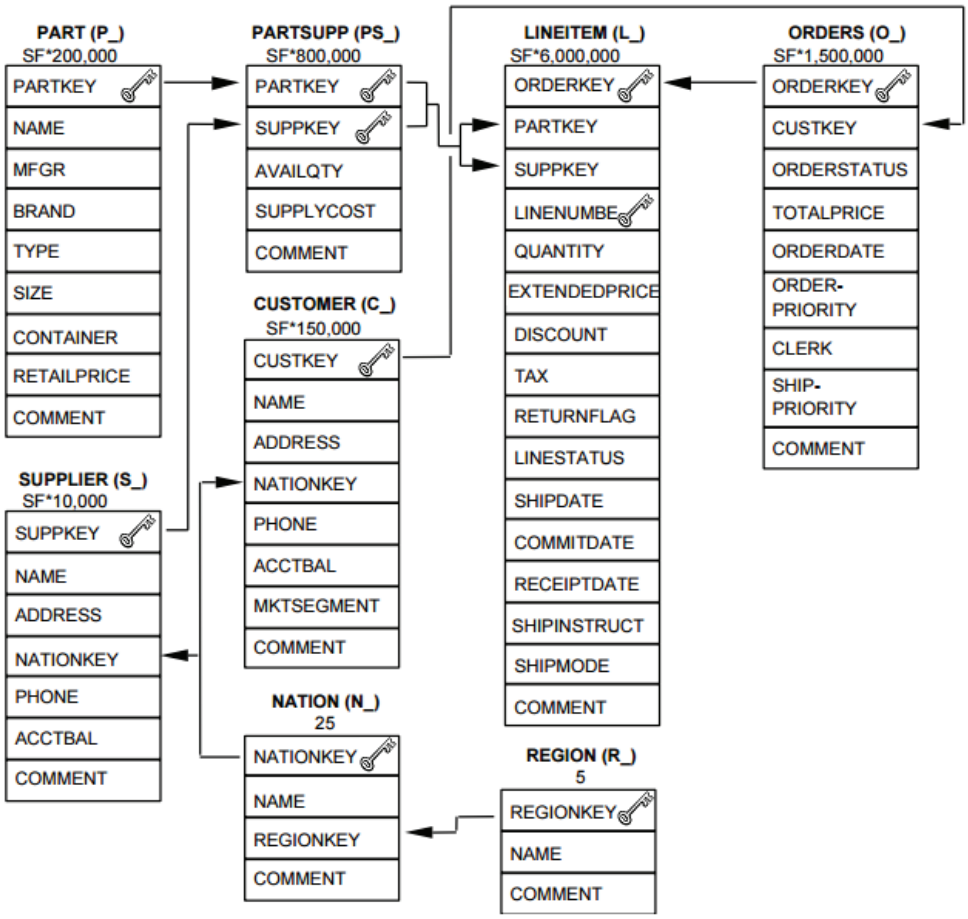

Figure E.1 – The TPC-H schema.

TPC-H Q1

```
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
```

Bibliography

Daniel J. Abadi, Samuel R. Madden, and Nabil 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. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376712. URL http://doi.acm.org/10.1145/1376616.1376712.

Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An Embedded DSL for High Performance Big Data Processing. In *International Workshop on End-to-end Management of Big Data (BigData 2012)*, 2012. URL http://infoscience.epfl.ch/record/181673/files/paper.pdf.

Yanif Ahmad and Christoph Koch. DBToaster: A SQL Compiler for High-performance Delta Processing in Main-Memory Databases. *Proc. VLDB Endow.*, 2(2):1566–1569, August 2009. ISSN 2150-8097. doi: 10.14778/1687553.1687592. URL http://dx.doi.org/10.14778/1687553.1687592.

Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *the 10th European Conference on Object-Oriented Programming*, ECOOP '96, pages 142–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68570-8. doi: 10.1007/BFb0053060. URL http://dx.doi.org/10.1007/BFb0053060.

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4. http://research.cs.wisc.edu/multifacet/papers/vldb01_pax.pdf.

Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011. ISSN 1573-0670. doi: 10.1007/s10817-010-9174-1. URL http://dx.doi.org/10.1007/s10817-010-9174-1.

Panayiotis Andreou, Orestis Spanos, Demetrios Zeinalipour-Yazti, George Samaras, and Panos K. Chrysanthis. FSort: External Sorting on Flash-based Sensor Devices. In *Proceedings of the Sixth International Workshop on Data Management for Sensor Networks*,

DMSN '09, pages 10:1–10:6, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-777-6. doi: 10.1145/1594187.1594201. URL http://doi.acm.org/10.1145/1594187.1594201.

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742797. URL http://doi.acm.org/10.1145/2723372.2742797.

Kenichi Asai, Hidehiko Masuhara, and Akinori Yonezawa. Partial Evaluation of Call-by-value λ-calculus with Side-effects. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '97, pages 12–21, New York, NY, USA, 1997. ACM. ISBN 0-89791-917-3. doi: 10.1145/258993.258997. URL http://doi.acm.org/10.1145/258993.258997.

Lex Augusteijn. Sorting Morphisms. In *the Third International School on Advanced Functional Programming*, AFP'98, pages 1–27, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48506-3. doi: 10.1007/10704973_1. URL http://dx.doi.org/10.1007/10704973_1.

Don Batory and Jeff Thomas. P2: A lightweight dbms generator. *Journal of Intelligent Information Systems*, 9(2):107–123, 1997. doi: 10.1023/A:1008617930959. URL http://dx.doi.org/10.1023/A:1008617930959.

Catriel Beeri and Yoram Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In Serge Abiteboul and Paris C. Kanellakis, editors, *ICDT '90*, volume 470 of *Lecture Notes in Computer Science*, pages 72–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990. ISBN 978-3-540-53507-2. doi: 10.1007/3-540-53507-1_71. URL http://dx.doi.org/10.1007/3-540-53507-1_71.

Pamela Bhattacharya and Iulian Neamtiu. Assessing Programming Language Impact on Development and Maintenance: A Study on C and C++. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 171–180, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985817. URL http://doi.acm.org/10.1145/1985793.1985817.

Richard S. Bird. Algebraic Identities for Program Calculation. *Comput. J.*, 32(2):122–126, 1989. doi: 10.1093/comjnl/32.2.122. URL http://dx.doi.org/10.1093/comjnl/32.2.122.

Rastislav Bodik and Barbara Jobstmann. Algorithmic program synthesis: introduction. *International Journal on Software Tools for Technology Transfer*, 15(5):397–411, 2013. ISSN 1433-2787. doi: 10.1007/s10009-013-0287-9. URL http://dx.doi.org/10.1007/s10009-013-0287-9.

Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *the 5th TPC Technology Conference on Performance, Characterization, and Benchmarking*, TPCTC 2013, pages 61–76. Springer

International Publishing, 2014. ISBN 978-3-319-04936-6. doi: 10.1007/978-3-319-04936-6_5. URL http://dx.doi.org/10.1007/978-3-319-04936-6_5.

Val Breazu-Tannen and Ramesh Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *the 18th International Colloquium on Automata, Languages and Programming*, pages 60–75, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-47516-3. doi: 10.1007/3-540-54233-7_125. URL http://dx.doi.org/10.1007/3-540-54233-7_125.

Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally Embedded Query Languages. In *Proceedings of 4th International Conference on Database Theory (ICDT)*, pages 140–154, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-47360-2. doi: 10.1007/3-540-56039-4_38. URL http://dx.doi.org/10.1007/3-540-56039-4_38.

Daniel Cederman and Philippas Tsigas. A Practical Quicksort Algorithm for Graphics Processors. In *Proceedings of the 16th Annual European Symposium on Algorithms*, ESA '08, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87743-1. doi: 10.1007/978-3-540-87744-8_21. URL http://dx.doi.org/10.1007/978-3-540-87744-8_21.

Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A History and Evaluation of System R. *Comm. ACM*, 24(10): 632–646, 1981. ISSN 0001-0782. doi: 10.1145/358769.358784. URL http://doi.acm.org/10.1145/358769.358784.

Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query Optimization in Compressed Database Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 271–282, New York, NY, USA, 2001. ACM. ISBN 1-58113-332-4. doi: 10.1145/375663.375692. URL http://doi.acm.org/10.1145/375663.375692.

Alvin Cheung, Armando Solar-Lezama, and Samuel 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. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462180. URL http://doi.acm.org/10.1145/2491956.2462180.

Wei-Ngan Chin and Siau-Cheng Khoo. Calculating Sized Types. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '00, pages 62–72, New York, NY, USA, 1999. ACM. ISBN 1-58113-201-8. doi: 10.1145/328690.328893. URL http://doi.acm.org/10.1145/328690.328893.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on*

Functional Programming, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291199. URL http://doi.acm.org/10.1145/1291151.1291199.

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Cetintemel, and Stan Zdonik. Tupleware: Redefining Modern Analytics. *CoRR*, abs/1406.6667, 2014. URL http://arxiv.org/abs/1406.6667.

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An Architecture for Compiling UDF-centric Workflows. *Proc. VLDB Endow.*, 8(12):1466–1477, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824045. URL http://dx.doi.org/10.14778/2824032.2824045.

Nils Anders 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. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328457. URL http://doi.acm.org/10.1145/1328438.1328457.

J. Darlington and R. M. Burstall. A System Which Automatically Improves Programs. *Acta Inf.*, 6(1):41–60, March 1976. ISSN 0001-5903. doi: 10.1007/BF00263742. URL http://dx.doi.org/10.1007/BF00263742.

Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 306–323, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: 10.1145/236337.236369. URL http://doi.acm.org/10.1145/236337.236369.

Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108. URL http://doi.acm.org/10.1145/2000064.2000108.

Rickard E. Faith, Lars S. Nyland, and Jan F. Prins. KHEPERA: A System for Rapid Implementation of Domain Specific Languages. In *Proceedings of the 1997 Conference on Domain-Specific Languages*, DSL' 97, pages 19–19, Berkeley, CA, USA, 1997. USENIX Association. URL https://www.usenix.org/legacy/publications/library/proceedings/dsl97/full_papers/faith/faith.pdf.

Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database – data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2012. ISSN 0163-5808. doi: 10.1145/2094114.2094126. URL http://doi.acm.org/10.1145/2094114.2094126.

Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188543. URL http://doi.acm.org/10.1145/1188455.1188543.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155113. URL http://doi.acm.org/10.1145/155090.155113.

Jeremi Gibbons. Origami programming. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, page chapter 3. Palgrave, 2003.

Andrew Gill, John Launchbury, and Simon 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. ISBN 0-89791-595-X. doi: 10.1145/165180.165214. URL http://doi.acm.org/10.1145/165180.165214.

Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. Towards Scalable Real-time Analytics: An Architecture for Scale-out of OLxP Workloads. *Proc. VLDB Endow.*, 8(12):1716–1727, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824069. URL http://dx.doi.org/10.14778/2824032.2824069.

Allen Goldberg and Robert 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. ISBN 0-89791-142-3. doi: 10.1145/800055.802021. URL http://doi.acm.org/10.1145/800055.802021.

Carla P. Gomes, Douglas R. Smith, and Stephen J. Westfold. Synthesis of Schedulers for Planned Shutdowns of Power Plants. In Doug White and Chris Welty, editors, *Proceedings of the 11th Annual Knowledge-Based Software Engineering Conference*, page 6. IEEE Computer Society, 1996. URL http://www.ase-conferences.org/ase/past/abstracts-96/gomes.txt.

Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142511. URL http://doi.acm.org/10.1145/1142473.1142511.

Goetz Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, Feb 1994. ISSN 1041-4347. doi: 10.1109/69.273032. URL http://dx.doi.org/10.1109/69.273032.

Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, January 1997. ISSN 1384-5810. doi: 10.1023/A:1009726021843. URL http://dx.doi.org/10.1023/A:1009726021843.

Rick Greer. Daytona And The Fourth-Generation Language Cymbal. In *the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 525–526, New York, NY, USA, 1999. ACM. ISBN 1-58113-084-8. doi: 10.1145/304182.304242. URL http://doi.acm.org/10.1145/304182.304242.

Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom 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. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559982. URL http://doi.acm.org/10.1145/1559845.1559982.

Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-safe LINQ Compilation. *Proc. VLDB Endow.*, 3(1-2):162–172, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920866. URL http://dx.doi.org/10.14778/1920841.1920866.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993506. URL http://doi.acm.org/10.1145/1993498.1993506.

Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance Tradeoffs in Read-optimized Databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 487–498. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164170.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. *Data Structure Fusion*, pages 204–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-17164-2. doi: 10.1007/978-3-642-17164-2_15. URL http://dx.doi.org/10.1007/978-3-642-17164-2_15.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data Representation Synthesis. *SIGPLAN Not.*, 46(6):38–49, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993504. URL http://doi.acm.org/10.1145/1993316.1993504.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent Data Representation Synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 417–428, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254114. URL http://doi.acm.org/10.1145/2254064.2254114.

Martin Hofmann and Steffen 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. ISBN 1-58113-628-5. doi: 10.1145/604131.604148. URL http://doi.acm.org/10.1145/604131.604148.

Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D. Matsakis. GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language. In *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 315–324, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4979-8. doi: 10.1109/IPDPSW.2013.173. URL http://dx.doi.org/10.1109/IPDPSW.2013.173.

Mike Houston, Ji-Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William Dally, and Pat Hanrahan. A Portable Runtime Interface for Multi-level Memory Hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 143–152, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345229. URL http://doi.acm.org/10.1145/1345206.1345229.

Paul Hudak. Building Domain-specific Embedded Languages. *ACM Comput. Surv.*, 28(4es), December 1996. ISSN 0360-0300. doi: 10.1145/242224.242477. URL http://doi.acm.org/10.1145/242224.242477.

Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas 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. ISBN 978-1-4503-3161-6. doi: 10.1145/2658761.2658776. URL http://doi.acm.org/10.1145/2658761.2658776.

Robert Hundt. Loop Recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011*, 2011. URL https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf.

Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. ISSN 0163-5980. doi: 10.1145/1243418.1243424. URL http://doi.acm.org/10.1145/1243418.1243424.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin 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. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706327. URL http://doi.acm.org/10.1145/1706299.1706327.

Vojin Jovanović, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin 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, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3161-6. doi: 10.1145/2658761.2658771. URL http://doi.acm.org/10.1145/2658761.2658771.

Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2):1496–1499, 2008. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1454159.1454211.

M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving Locality Using Loop and Data Transformations in an Integrated Framework. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 285–297, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3. URL http://dl.acm.org/citation.cfm?id=290940.290999.

Ken Kennedy, Bradley Broom, Arun Chauhan, Robert J. Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping Languages: A System for Automatic Generation of Domain Languages. *Proceedings of the IEEE*, 93(2):387–408, 2005. doi: 10.1109/JPROC.2004.840447.

Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807206. URL http://doi.acm.org/10.1145/1807167.1807206.

D. Knuth. Volume 3: Sorting and searching, 2nd ed. In *The Art of Computer Programming*, pages 248–379. Addison-Wesley, 1998.

Christoph 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. ISBN 978-1-4503-0033-9. doi: 10.1145/1807085.1807100. URL http://doi.acm.org/10.1145/1807085.1807100.

Christoph Koch. Abstraction without regret in data management systems. In *the Sixth Biennial Conference on Innovative Data Systems Research*, CIDR 2013. www.cidrdb.org, 2013. URL http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper149.pdf.

Christoph Koch. Abstraction Without Regret in Database Systems Building: a Manifesto. *IEEE Data Eng. Bull.*, 37(1):70–79, 2014. URL http://sites.computer.org/debull/A14mar/p70.pdf.

Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, 2014. ISSN 1066-8888. doi: 10.1007/s00778-013-0348-4. URL http://dx.doi.org/10.1007/s00778-013-0348-4.

Christoph Kreitz. *Program Synthesis*, pages 105–134. Springer Netherlands, Dordrecht, 1998. ISBN 978-94-017-0437-3. doi: 10.1007/978-94-017-0437-3_5. URL http://dx.doi.org/10.1007/978-94-017-0437-3_5.

Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *Proceedings of the 26th International Conference on Data Engineering*, ICDE '10, pages 613–624, Washington, DC, USA, March 2010. IEEE Computer Society. doi: 10.1109/ICDE.2010.5447892.

Sandhya Krishnan, Sriram Krishnamoorthy, Gerald Baumgartner, Chi-Chung Lam, J. Ramanujam, P. Sadayappan, and Venkatesh Choppella. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. In *18th International Symposium on Parallel and Distributed Processing, 2004. Proceedings.*, pages 34–, April 2004. doi: 10.1109/IPDPS.2004.1302948.

Jens Krüger, Johannes Wust, Martin Linkhorst, and Hasso Plattner. Leveraging Compression in In-Memory Databases . In *Proceedings of the DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 147 – 153, 2012. ISBN 978-1-61208-185-4. URL https://www.thinkmind.org/download.php?articleid=dbkda_2012_6_20_30160.

Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM. ISBN 0-89791-380-9. doi: 10.1145/106972.106981. URL http://doi.acm.org/10.1145/106972.106981.

Chris Lattner and Vikram 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. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31(5):42–53, September 2011. ISSN 0272-1732. doi: 10.1109/MM.2011.68. URL http://dx.doi.org/10.1109/MM.2011.68.

Yang Liu, Zhen He, Yi-Ping Phoebe Chen, and Thi Nguyen. External Sorting on Flash Memory Via Natural Page Run Generation. *Comput. J.*, 54(11):1882–1990, November 2011. ISSN 0010-4620. doi: 10.1093/comjnl/bxr051. URL http://dx.doi.org/10.1093/comjnl/bxr051.

Giampaolo Liuzzi, Stefano Lucidi, and Marco Sciandrone. Sequential Penalty Derivative-Free Methods for Nonlinear Constrained Optimization. *SIAM J. on Optimization*, 20(5):2614–2635, July 2010. ISSN 1052-6234. doi: 10.1137/090750639. URL http://dx.doi.org/10.1137/090750639.

David Lomet, Kostas Tzoumas, and Michael Zwilling. Implementing Performance Competitive Logical Recovery. *Proc. VLDB Endow.*, 4(7):430–439, April 2011. ISSN 2150-8097. doi: 10.14778/1988776.1988779. URL http://dx.doi.org/10.14778/1988776.1988779.

Stefan Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. PhD thesis, University of Amsterdam, December 2002.

Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2):1648–1653, 2009. ISSN 2150-8097. doi: 10.14778/1687553.1687618.

Zohar Manna and Richard Waldinger. Synthesis: Dreams => Programs. *IEEE Transactions on Software Engineering*, 5(undefined):294–328, 1979. ISSN 0098-5589. doi: doi.ieeecomputersociety.org/10.1109/TSE.1979.234198.

Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996. ISSN 0164-0925. doi: 10.1145/233561.233564. URL http://doi.acm.org/10.1145/233561.233564.

Manish Mehta and David 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. ISBN 1-55860-379-4. URL http://dl.acm.org/citation.cfm?id=645921.673299.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-54396-1. URL http://dl.acm.org/citation.cfm?id=127960.128035.

Erik Meijer, Brian Beckman, and Gavin 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, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142552. URL http://doi.acm.org/10.1145/1142473.1142552.

Derek Gordon Murray, Michael Isard, and Yuan 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. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993513. URL http://doi.acm.org/10.1145/1993498.1993513.

Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. Code Generation for Efficient Query Processing in Managed Runtimes. *Proc. VLDB Endow.*, 7(12):1095–1106, August 2014. ISSN 2150-8097. doi: 10.14778/2732977.2732984. URL http://dx.doi.org/10.14778/2732977.2732984.

Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011. ISSN 2150-8097. doi: 10.14778/2002938.2002940. URL http://dx.doi.org/10.14778/2002938.2002940.

Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In *the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094815. URL http://doi.acm.org/10.1145/1094811.1094815.

Oracle Corporation. TimesTen In-Memory Database Architectural Overview, 2006. http://download.oracle.com/otn_hosted_doc/timesten/603/TimesTen-Documentation/arch.pdf.

Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block oriented processing of Relational Database operations in modern Computer Architectures. In *Proceedings of the 17th International Conference on Data Engineering*, ICDE '01, pages 567–574, Washington, DC, USA, 2001. IEEE Computer Society. doi: 10.1109/ICDE.2001.914871. URL http://dx.doi.org/10.1109/ICDE.2001.914871.

Hyoungmin Park and Kyuseok Shim. FAST: Flash-aware External Sorting for Mobile Database Systems. *J. Syst. Softw.*, 82(8):1298–1312, August 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.02.028. URL http://dx.doi.org/10.1016/j.jss.2009.02.028.

perf. Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Spiral*, pages 1920–1933. 2011. doi: 10.1007/978-0-387-09766-4_244. URL http://dx.doi.org/10.1007/978-0-387-09766-4_244.

R. Ramakrishnan and J. Gehrke. *Database Management Systems, 3rd ed.* McGraw-Hill, 2002.

G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions Programming Languages and Systems*, 16(5):1467–1471, September 1994. ISSN 0164-0925. doi: 10.1145/186025.186041. URL http://doi.acm.org/10.1145/186025.186041.

Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-Time Query Processing. In *Proceedings of the 24th International Conference on Data Engineering*, ICDE '08, pages 60–69, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-1836-7. doi: 10.1109/ICDE.2008.4497414. URL http://dx.doi.org/10.1109/ICDE.2008.4497414.

Jun Rao, Hamid Pirahesh, C. Mohan, and Guy Lohman. Compiled Query Execution Engine using JVM. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 23–34, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9. doi: 10.1109/ICDE.2006.40. URL http://dx.doi.org/10.1109/ICDE.2006.40.

Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A Tuning Framework for Software-managed Memory Hierarchies. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 280–291, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454155. URL http://doi.acm.org/10.1145/1454115.1454155.

Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL), 2012. URL http://dx.doi.org/10.5075/epfl-thesis-5456.

Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314. URL http://doi.acm.org/10.1145/1868294.1868314.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin 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. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429128. URL http://doi.acm.org/10.1145/2429069.2429128.

Seeker. an utility to measure disk performance. http://www.linuxinsight.com/how_fast_is_your_disk.html.

Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph 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. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2915244. URL http://doi.acm.org/10.1145/2882903.2915244.

Erik Sintorn and Ulf Assarsson. Fast Parallel GPU-sorting Using a Hybrid Algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, October 2008. ISSN 0743-7315. doi: 10.1016/j.jpdc.2008.05.012. URL http://dx.doi.org/10.1016/j.jpdc.2008.05.012.

Yannis Smaragdakis and Don Batory. DiSTiL: A Transformation Library for Data Structures. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, DSL'97, pages 20–20, Berkeley, CA, USA, 1997. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267950.1267970.

Douglas Smith and Eduardo A. Parra. Transformational Approach to Transportation Scheduling. In Bruce Johnson, Mehdi Harandi, and Bill Sasso, editors, *Proceedings of the Eighth Annual Knowledge-Based Software Engineering Conference*, page 12, 1993.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. *SIGPLAN Not.*, 40(6):281–294, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065045. URL http://doi.acm.org/10.1145/1064978.1065045.

Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. Compilation in Query Execution. In *the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0658-4. doi: 10.1145/1995441.1995446. URL http://doi.acm.org/10.1145/1995441.1995446.

Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2285-8. doi: 10.1109/ICDE.2005.1. URL http://dx.doi.org/10.1109/ICDE.2005.1.

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325981.

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-oriented DBMS. In *the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005. ISBN 1-59593-154-6. URL http://dl.acm.org/citation.cfm?id=1083592.1083658.

Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 145–154, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2373-4. doi: 10.1145/2517208.2517220. URL http://doi.acm.org/10.1145/2517208.2517220.

Eijiro Sumii and Naoki Kobayashi. A Hybrid Approach to Online and Offline Partial Evaluation. *Higher Order Symbol. Comput.*, 14(2-3):101–142, September 2001. ISSN 1388-3690. doi: 10.1023/A:1012984529382. URL http://dx.doi.org/10.1023/A:1012984529382.

Josef 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. ISBN 1-58113-487-8. doi: 10.1145/581478.581491. URL http://doi.acm.org/10.1145/581478.581491.

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000. doi: http://dx.doi.org/10.1016/S0304-3975(00)00053-0.

The GNOME Project. GLib: Library package for low-level data structures in C – the reference manual, 2013. https://developer.gnome.org/glib/2.38/.

Sid-Ahmed-Ali Touati and Denis 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, New York, NY, USA, 2006. ACM. ISBN 1-59593-302-6. doi: 10.1145/1128022.1128042. URL http://doi.acm.org/10.1145/1128022.1128042.

Transaction Processing Performance Council. TPC-H, an ad-hoc, decision support benchmark., 1999. URL http://www.tpc.org/tpch.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000. ISSN 0362-1340. doi: 10.1145/352029.352035. URL http://doi.acm.org/10.1145/352029.352035.

Varmo Vene and Tarmo Uustalu. Functional Programming with Apomorphisms / Corecursion. In *9th Nordic Workshop on Programming Theory*, 1998.

Stratis Viglas, Gavin M. Bierman, and Fabian Nagel. Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes. *IEEE Data Eng. Bull.*, 37(1):12–21, 2014. URL http://sites.computer.org/debull/A14mar/p12.pdf.

XQuery. Documentation of XQuery v1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

Xiaochun Ye, Dongrui Fan, Wei Lin, Nan Yuan, and Paolo Ienne. High performance comparison-based sorting algorithm on many-core GPUs. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, April 2010. doi: 10.1109/IPDPS.2010.5470445.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon 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. URL http://dl.acm.org/citation.cfm?id=1855741.1855742.

Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P. Wright. On Incremental File System Development. *Transactions on Storage*, 2(2):161–196, May 2006. ISSN 1553-3077. doi: 10.1145/1149976.1149979. URL http://doi.acm.org/10.1145/1149976.1149979.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud' 10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

Barry M. Zane, James P. Ballard, Foster D. Hinshaw, Dana A. Kirkpatrick, and Less Premanand Yerabothu. Optimized SQL Code Generation (US Patent 7430549 B2). WO Patent App. US 10/886,011, September 2008. URL http://www.google.ch/patents/US7430549.

Rui Zhang, Saumya Debray, and Richard 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, 2012a. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259025. URL http://doi.acm.org/10.1145/2259016.2259025.

Rui Zhang, Richard T. Snodgrass, and Saumya Debray. Application of Micro-specialization to Query Evaluation Operators. In *Proceedings of the 28th International Conference on Data Engineering Workshops*, ICDEW '12, pages 315–321, Washington, DC, USA, 2012b. IEEE Computer Society. doi: 10.1109/ICDEW.2012.43.

Rui Zhang, Richard T. Snodgrass, and Saumya Debray. Micro-Specialization in DBMSes. In *ICDE*, pages 690–701, Washington, DC, USA, 2012c. IEEE Computer Society. doi: 10.1109/ICDE.2012.110.

Jingren Zhou and Kenneth 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. ISBN 1-58113-497-5. doi: 10.1145/564691.564709. URL http://doi.acm.org/10.1145/564691.564709.

Postlude

And as my PhD journey finally comes to an end, I cannot help but contemplate over the beautiful words of the famous poet, Dante Alighieri:

In that part of the book of my memory
before which little can be read,
there is a heading, which says:
"Incipit vita nova: Here begins the new life"

Yannis Klonatos

Avenue Victor Ruffy 12, CH-1012
Lausanne, Vaud, Switzerland
+41 78 837 35 34
klonatos@gmail.com
12/04/1986, Greek, Permit C

Strengths:

- Database, OS and Storage specialist
- Strong analytical and abstract thinking
- Hands-on and can-do systems developer

Education

2011 - Today — PhD. in the field of Database Systems, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

2009 - 2011 — MSc. with two majors in the fields of Parallel and Distributed Systems and Information Systems, Computer Science Department, University of Crete, Greece

2004 - 2008 — BSc. in Computer Science, Computer Science Department, University of Crete, Greece

Professional Experience

2015 – Today — **Credit Suisse**, Lausanne, Vaud, Switzerland

Working as a Senior Database Engineer on the area of regulatory reporting. My work concerns the development and maintenance of the database applications that gather and calculate the data reported to the various regulators. This is a task of vital importance as delays or inconsistencies can significantly affect the bank's reputation and may result in substantial fines and penalties. I have always delivered solutions in a timely manner and managed to improve the overall stability and performance of the applications.

2013 — **Oracle Labs**, Belmont, California, USA (3–month internship, July – October 2013)

Research intern in Project Q, which aims to boost the performance of RDBMSes by using just-in-time (JIT) compilation. My work concerned the design and implementation of a compiler that was creating a highly specialized query engine for an incoming SQL query at runtime. Despite the short period of the internship, I quickly came up to speed with the project and managed to integrate my software with a commercial OLAP database. My system improved the performance of an *optimally* configured database setup up to 7.7×.

2009 – 2011 — Institute of Computer Science, Foundation of Research and Technology Herakleion (FORTH), Crete, Greece

Research associate for the ERP project IOLanes. My work studied techniques for improving the performance and scalability of the I/O stack of the Linux kernel in multi-core environments. I solved technically challenging issues in the complex, multi-KLOC code-base of the Linux kernel, providing significant performance improvements. This required methodological profiling and complex, low-level programming. Despite the tight deadlines, I always successfully handled the project deliverables assigned to me.

Technical Skills

Programming Languages — C (Expert), C++ (Very good), Java (Very good), Scala (Very good), Bash Scripting (Expert)
SQL(Expert), PL/SQL(Very Good), Python (Good), MATLAB(Good)
7 years of experience in developing systems software

Operating Systems — Linux, Solaris, MacOS
In depth understanding of file-system, AIO and block-layer internals in Linux-based systems
Extensive driver development (2.5 years) during my MSc. project
Very good knowledge of methodologies for evaluating the performance of computer systems
Extensive experience on administrating Linux-based systems (especially RedHat) and setting up related utilities and protocols (e.g. RAID, NFS, DNS, FTP, SSH, RPM, Solaris Zones)

Database Systems — Relational: Oracle DB, Oracle TimesTen, Vertica, MySQL, PostgreSQL / NoSQL: HBase, Berkeley DB
Very good understanding of general database design
Very good knowledge of database optimization and configuration

Tools — Hadoop, SVN, Git, Maven, LaTeX, JUnit, ScalaTest, Scala Actors, MPI, JDBC
Knowledge of virtualization technologies (e.g. Xen, VMware) and their general design principles

Projects

PhD. Thesis — I am working on the *abstraction without regret* vision, a new, promising paradigm for systems programming, under which developers leverage high-level languages without paying a price in efficiency. To realize this vision for databases, I implemented from scratch LegoBase, an analytical DBMS written in Scala. LegoBase employs generative programming to regain efficiency: the Scala source is a generator that emits specialized C code. This allows to easily implement a number of optimizations that are difficult for existing systems. LegoBase significantly outperforms a *commercial* DBMS and won the **best paper award at VLDB 2014**.

MSc. Thesis I developed Linux kernel drivers that allow Solid State Disks (SSDs) to be used as HDD caches in the I/O path. I thoroughly examined several design aspects, such as data placement, that significantly affect the performance of SSD caches. My software allowed for a high degree of I/O concurrency, did not require *any* modifications to applications such as databases, and was achieving up to 14× better performance than a high-end HDD-based storage solution. **This software was bought by Nevex Virtual Technologies** (which was later acquired by Intel). It also has a **pending patent application** (#WO2014015409 A1).

Publications

PhD Thesis

2016 **Yannis Klonatos**, Amir Shaikhha, and Christoph Koch. "Building Efficient Query Engines in a High-Level Language". In: *Transactions on Database Systems (under submission)* (2016).

Amir Shaikhha, **Yannis Klonatos**, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. "How to Architect a Query Compiler". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 1907–1922. ISBN: 978-1-4503-3531-7. URL: `http://doi.acm.org/10.1145/2882903.2915244`.

2015 Tiark Rompf, Kevin J Brown, HyoukJoong Lee, Arvind K Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, **Yannis Klonatos**, Mohammad Dashti, et al. "Go meta! A case for generative programming and DSLs in performance critical systems". In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

2014 **Yannis Klonatos**, Christoph Koch, Tiark Rompf, and Hassan Chafi. "Building Efficient Query Engines in a High-level Language". In: *Proc. VLDB Endow.* 7.10 (June 2014), pp. 853–864. ISSN: 2150-8097. URL: `http://dx.doi.org/10.14778/2732951.2732959`.

Tiark Rompf, Nada Amin, Thierry Coppey, Mohammad Dashti, Manohar Jonnalagedda, **Yannis Klonatos**, Martin Odersky, and Christoph Koch. "Abstraction without regret for efficient data processing". In: *Data-Centric Programming Workshop*. 2014.

2013 **Yannis Klonatos**, Andres Nötzli, Andrej Spielmann, Christoph Koch, and Victor Kuncak. "Automatic Synthesis of Out-of-core Algorithms". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, USA: ACM, 2013, pp. 133–144. ISBN: 978-1-4503-2037-5. URL: `http://doi.acm.org/10.1145/2463676.2465334`.

MSc Thesis

2012 Angelos Bilas, Michail D Flouris, **Yannis Klonatos**, Thanos Makatos, and Manolis Marazakis. *System and Method for Implementing SSD-Based I/O Caches*. US Patent App. 13/976,271. July 2012.

Yannis Klonatos, Thanos Makatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. "Transparent Online Storage Compression at the Block-Level". In: *Trans. Storage* 8.2 (May 2012), 5:1–5:33. ISSN: 1553-3077. URL: `http://doi.acm.org/10.1145/2180905.2180906`.

2011 **Yannis Klonatos**, Thanos Makatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. "Azor: Using Two-Level Block Selection to Improve SSD-Based I/O Caches". In: *Proceedings of the 2011 Sixth International Conference on Networking, Architecture, and Storage*. NAS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 309–318. ISBN: 978-0-7695-4509-7. URL: `dx.doi.org/10.1109/NAS.2011.50`.

Yannis Klonatos, Manolis Marazakis, and Angelos Bilas. "A scaling analysis of Linux I/O performance". In: *Poster at ACM EuroSys Conference*. 2011, pp. 409–428.

2010 Thanos Makatos, **Yannis Klonatos**, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. "Using Transparent Compression to Improve SSD-based I/O Caches". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 1–14. ISBN: 978-1-60558-577-2. URL: `http://doi.acm.org/10.1145/1755913.1755915`.

Thanos Makatos, **Yannis Klonatos**, Manolis Marazakis, Michail D Flouris, and Angelos Bilas. "ZBD: Using transparent compression at the block level to increase storage space efficiency". In: *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*. IEEE. 2010, pp. 61–70.

Scientific Interests

- Database Management Systems
- Employing compiler techniques for optimizing computer systems
- High performance, adaptive and scalable I/O

- File-system design, optimization and evaluation
- Optimizing storage & server architectures
- Operating Systems design and evaluation
- Solid State Memory Technologies
- Administering Computer Systems
- Parallel and Distrubuted Programming

Speaking Languages

Greek	Mother Tongue
English	Fluent, C1 equivalent, Advanced Certificate in English (University of Cambridge)
French	Intermediate proficiency, B1 equivalent
German	Willing to learn