TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik

Optimizing Relational Query Engine Architecture for Modern Hardware

Timo Kersten

TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik

Optimizing Relational Query Engine Architecture for Modern Hardware

Timo Kersten

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Helmut Seidl

Prüfer der Dissertation:
1. Prof. Dr. Thomas Neumann
2. Prof. Dr. Holger Pirk (Imperial College London)
3. Prof. Dr. Jana Giceva Makreshanska

Die Dissertation wurde am 08.03.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.07.2021 angenommen.

Abstract

In relational database systems, the query engine is a key component to serve the user workload and also extract the highest performance from underlying hardware. This thesis investigates architectures for high-performance query engines which are prepared for compute-intensive workloads and upcoming high-bandwidth storage devices. As foundation, we perform an extensive experimental study to compare the two state-of-the-art engine architectures: vectorization and data-centric code generation. We find data-centric code generation is suited for compute-intensive queries, yet has shortcomings in terms of compilation time, hardware hazards, and software complexity. To solve these issues, we present an architecture which enables very fast compilation without sacrificing peak performance. Further, we address hardware hazards with an optimizer for the pipelines of data-centric code generation which selectively restructures generated code and, thus, mitigates branch and cache misses. Finally, we introduce profiling and debugging techniques to make working with compiling query engines simpler in practice.

Zusammenfassung

In relationalen Datenbanksystemen ist der Abfragebearbeiter eine Schlüsselkomponente, um die Abfragen der Benutzer zu beantworten und gleichzeitig die höchste Leistung aus der zugrunde liegenden Hardware zu extrahieren. In dieser Arbeit werden Architekturen für hochperformante Abfragebearbeiter untersucht, die auf rechenintensive Abfragen und zukünftige Datenspeicher mit hoher Bandbreite ausgelegt sind. Als Grundlage führen wir eine umfangreiche experimentelle Studie durch, um die beiden derzeit besten Abfragebearbeiterarchitekturen zu vergleichen: Vektorisierung und datenzentrische Programmgenerierung. Wir stellen fest, dass die datenzentrische Programmgenerierung für rechenintensive Abfragen geeignet ist, jedoch Defizite in Bezug auf Kompilierungszeit, Hardware-Hazards und Softwarekomplexität aufweist. Zur Lösung dieser Probleme stellen wir eine Architektur vor, die eine sehr schnelle Kompilierung ermöglicht, ohne die Spitzenleistung zu beeinträchtigen. Zudem behandeln wir Hardware-Hazards mit einem Optimierer für die Datenautobahnen der datenzentrierten Programmgenerierung. Dieser restrukturiert selektiv die generierten Programme und schwächt so Effekte durch Fehler der Sprungvorhersage und mangelnde Pufferspeichergröße ab. Schließlich führen wir Leistungsanalyse- und Fehlerfindungstechniken ein, um die Arbeit mit kompilierenden Abfragebearbeitern in der Praxis zu vereinfachen.

ACKNOWLEDGMENTS

While working on this thesis I was fortunate to receive advice, guidance, and challenges from many collaborators. I would especially like to thank

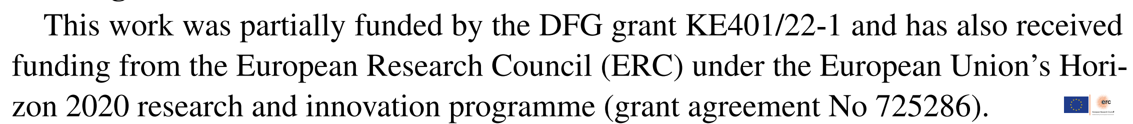
- ⋆ Manuel Then who introduced me to the field of database systems research.
- ⋆ Prof. Alfons Kemper for providing the opportunity to freely research, experiment, and to experience what freedom of research means.
- ⋆ Prof. Thomas Neumann for his help and advice on a large spectrum of topics. It was amazing to see him always lead by example and to be part of his pursuit to build the fastest database system in the world.
- ⋆ Prof. Viktor Leis for being a great mentor, especially on the topics of research opportunities and conveying complex issues in writing and in style.
- ⋆ Prof. Peter Boncz for sharing his knowlege of vectorized query engines.
- ⋆ Alexander Beischl for our collaboration on profilers and visualization.
- ⋆ Prof. Jana Giceva for recognizing the impact of the profiling problem in other domains and her insightful advice.

I would also like to thank my colleagues at TUM, Jan Böttcher, Philipp Fent, Maximilian Bandle, André Kohn, Michael Haubenschild, Moritz Sichert, and Daniel Zügner for the extensive discussions, evaluations, critiques and improvements of good, bad, and crazy ideas. I would also like to thank Prof. Helmut Seidl for chairing my doctoral examination and the thesis committee Prof. Thomas Neumann, Prof. Holger Pirk, and Prof. Jana Giceva.

Finally, I thank my partner, Delia Döring, for her support during this journey.

Funding.

This work was partially funded by the DFG grant KE401/22-1 and has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).



PREFACE

Excerpts of this thesis were published in advance.

Chapter 2 has previously been published in:

Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. "Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask". In: *PVLDB* 11.13 (2018), pp. 2209–2222

Chapter 3 has previously been published in:

Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra". In: *VLDB J.* (2021), pp. 883–905

Chapter 6 has previously been published in:

Timo Kersten and Thomas Neumann. "On Another Level: How to Debug Compiling Query Engines". In: *DBTest@SIGMOD*. 2020, 2:1–2:6

CONTENTS

ACKNOWLEDGMENTS i
PREFACE iii

1 INTRODUCTION 1
1.1 Background . . . 1
1.1.1 High-Performance System Architectures . . . 1
1.1.2 Application Interfaces and System Specializations . . . 2
1.2 Challenges . . . 3
1.2.1 Hardware Trends . . . 3
1.2.2 System Usage Trends . . . 3
1.3 Contributions . . . 4
2 QUERY ENGINES: TO VECTORIZE OR TO COMPILE? 7
2.1 Vectorized and Compiled Queries . . . 9
2.1.1 Vectorizing Algorithms . . . 9
2.1.2 Vectorized Hash Join and Group By . . . 10
2.2 Comparison Methodology . . . 12
2.2.1 Related Work . . . 13
2.2.2 Query Processing Algorithms . . . 13
2.2.3 Workload . . . 13
2.2.4 Experimental Setup . . . 14
2.3 Micro-Architectural Analysis . . . 14
2.3.1 Single-Thread Performance . . . 14
2.3.2 Interpretation and Instruction Cache . . . 17
2.3.3 Vector Size . . . 18
2.3.4 Star Schema Benchmark . . . 18
2.3.5 Tectorwise/Typer versus VectorWise/HyPer . . . 19
2.4 Data-Parallel Execution (SIMD) . . . 20
2.4.1 Data-Parallel Selection . . . 20
2.4.2 Data-Parallel Hash Table Probing . . . 22
2.4.3 Compiler Auto-Vectorization . . . 23
2.4.4 Summary . . . 24
2.5 Intra-Query Parallelization . . . 24
2.5.1 Exchange vs. Morsel-Driven Parallelism . . . 24
2.5.2 Multi-Threaded Execution . . . 25
2.5.3 Out-Of-Memory Experiments . . . 26
2.6 Hardware . . . 27
2.6.1 Intel Skylake X versus AMD Threadripper . . . 27
2.6.2 Knights Landing (Xeon Phi) . . . 28

2.7 Other Factors 30
2.7.1 OLTP and Multi-Language Support 30
2.7.2 Compilation Time 30
2.7.3 Profiling and Debuggability 31
2.7.4 Adaptivity 31
2.7.5 Implementation Issues 31
2.7.6 Summary 32
2.8 Beyond Basic Vectorization and Data-Centric Code Generation 32
2.8.1 Hybrid Models 32
2.8.2 Other Query Processing Models 34
2.9 Conclusions 35
3 FAST CODE GENERATION AND COMPILATION 37
3.1 Tidy Tuples: A Low-Latency Code Generation Framework 40
3.1.1 Background: Compilation Pipeline 41
3.1.2 Layer Overview 42
3.1.3 From Operators to Instructions 43
3.1.4 SQL Values 45
3.1.5 Primitive Types for Code Generation 46
3.1.6 Host Language Integration 47
3.1.7 Control Flow 48
3.2 Umbra Program Representation 50
3.2.1 Umbra IR Structure 50
3.2.2 Physical Program Layout 51
3.2.3 Constants and Dead-Code Removal 52
3.2.4 DBMS-Specific Instructions 53
3.2.5 Comparison to LLVM IR 53
3.3 Flying Start Compiler 53
3.3.1 Background: Adaptive Execution 54
3.3.2 Minimal Compile-Time Design 55
3.3.3 Stack Space Reuse 56
3.3.4 Machine Register Allocation 57
3.3.5 Lazy Address Calculation 58
3.3.6 Fuse Comparison and Branch 59
3.3.7 Implementation of Flying Start 60
3.4 Evaluation 62
3.4.1 Experimental Setup 62
3.4.2 Query Latency: Compile Time + Runtime 63
3.4.3 Compilation Time 65
3.4.4 Runtime Performance Robustness 68
3.4.5 Flying Start Optimizations 69
3.4.6 Implementation Effort 73
3.5 Related Work 73
3.6 Summary 76

4 OPTIMIZING MEMORY ACCESS 79
4.1 Prior Work 80
4.1.1 Hazard Mitigation in Hardware 81
4.1.2 Hazard Mitigation in Software 81
4.1.3 Data-Centric Code Generation 81
4.1.4 Hardware Performance Models 82
4.2 Pipeline Optimizer for Data-Centric Code 83
4.2.1 Problem Formulation 85
4.2.2 Dynamic Programming Approach 86
4.3 Cost Function 86
4.3.1 Processor Model 87
4.3.2 Pipeline Runtime with Hazards 88
4.3.3 Probabilistic Model 90
4.3.4 Multiple Cache Misses and Data Dependencies 91
4.4 Evaluation 94
4.4.1 Experimental Setup 94
4.4.2 Captured Improvements 95
4.4.3 Optimization Optimality 98
4.4.4 Cost Function Accuracy 101
4.5 Summary 105
5 PROFILING COMPILING QUERY ENGINES 107
5.1 Query Engines and Performance Tuning 109
5.1.1 Dataflow systems 109
5.1.2 Code Generation 109
5.1.3 Profiling Tools 110
5.2 Profiling Dataflow Systems 111
5.2.1 Shortcomings of Current Tools 111
5.2.2 The Missing Link: An Example 113
5.3 Abstraction Appropriate Profiling 115
5.3.1 Requirements from an Ideal Profiler 115
5.3.2 Tailored Profiling 116
5.3.3 Benefits of Tailored Profiling 119
5.4 Integration With Umbra 120
5.4.1 Correspondence Tracing 121
5.4.2 Register Tagging 121
5.4.3 Precise Timestamps for Profiling Samples 122
5.5 Evaluation 122
5.5.1 Experimental Setup 122
5.5.2 Use Cases 123
5.5.3 Runtime Overhead 126
5.5.4 Accuracy 127
5.5.5 Implementation Effort 128
5.6 Related Work 129

5.7 Summary 129
6 DEBUGGING COMPILING QUERY ENGINES 131
6.1 Interactive Debugging 132
6.1.1 How Debugging Should Work: Volcano-style Interpreter 133
6.1.2 Debugging Code Generating Engines 135
6.2 Evaluation 138
6.2.1 Multi-level Debugging for Umbra 138
6.2.2 Implementation Effort 139
6.2.3 Runtime Overhead 140
6.3 Related Work 140
6.3.1 Debugging Relational Code Generators 140
6.3.2 Time Travel Debuggers 141
6.4 Future Work 141
6.5 Summary 141
7 CONCLUSIONS AND FUTURE WORK 143
BIBLIOGRAPHY 145

LIST OF FIGURES

Figure 1	Multi-Predicate Example	10
Figure 2	Hash Join Implementations in Typer and Tectorwise	11
Figure 3	Single-thread Performance on TPC-H	15
Figure 4	Memory Stalls	16
Figure 5	Vector Size	18
Figure 6	Scalar vs. SIMD Selection	21
Figure 7	Sparse Selection	21
Figure 8	Scalar vs. SIMD Join Probing	22
Figure 9	Join Probe	23
Figure 10	Compiler Auto-Vectorization	23
Figure 11	Skylake vs. Threadripper	28
Figure 12	Skylake versus Knights Landing	29
Figure 13	Design Space Between Vectorization and Compilation	33
Figure 14	Umbra's Low-Latency Path from Query Plan to Result	37
Figure 15	Best of Both Worlds	39
Figure 16	Compilation Phases of the Compiling System Umbra	41
Figure 17	Architecture for a Low-Latency Code Generation Engine	42
Figure 18	Illustration of an In-Memory Inner Hash Join	44
Figure 19	Internal Structure of an Umbra IR Program	50
Figure 20	Flying Start Optimizations	54
Figure 21	Umbra IR Snippet	57
Figure 22	Foundation of the Flying Start Backend	60
Figure 23	On-the-fly Optimizations in Flying Start	61
Figure 24	Flying Start achieves low query latency over a wide range from tiny to large datasets	63
Figure 25	Umbra with Flying Start achieves low query latency across machine configurations	65
Figure 26	Flying Start compiles large queries quickly	67
Figure 27	Umbra's vs. HyPer's Execution Modes	68
Figure 28	Effect of Optimizations on Compile- and Runtime	70
Figure 29	Effectiveness of LLVM's Optimization Passes	71
Figure 30	Additional Cost and Benefit of Linear Scan Register Allocation	71
Figure 31	The performance of code generated by Flying Start is comparable to Umbra's LLVM backend	72
Figure 32	Effect of Dead Code Elimination on Compile- and Runtime	73
Figure 33	Hazard Cycles on TPC-H	79
Figure 34	Buffers to Relax Operator Fusion	82

Figure 35 Machine Instruction sequence for Relational Operators in a Pipeline . . . 87
Figure 36 Single LLC Miss in Interval Model . . . 88
Figure 37 Hiding of Multiple LLC Misses in Close Proximity . . . 89
Figure 38 Pipeline Execution Repeats to Fit Multiple Iterations into Out-Of-Order Window . . . 90
Figure 39 Memory Accesses as Random Variables . . . 91
Figure 40 Dependent Memory Accesses Within a Pipeline . . . 92
Figure 41 Out-of-order Execution Schedules Dependent Memory Accesses in Waves . . . 92
Figure 42 Query Execution Times with and without Buffering on TPC-H 95
Figure 43 Reduced Cycles per Query on TPC-H. . . . 96
Figure 44 Query Execution Times with and without Buffering on SSB . . 97
Figure 45 Reduced Cycles per Query on SSB . . . 97
Figure 46 Query Execution Times with and without Buffering on TPC-DS 97
Figure 47 Reduced Cycles per Query on TPC-DS . . . 98
Figure 48 Optimization Space of TPC-H, SSB, and TPC-DS Queries . . . 99
Figure 49 Alternative Strategy of Inserting All Possible Buffers vs. Optimizing Buffers . . . 100
Figure 50 Processors Hides Cache Miss Penalty with Out-Of-Order Execution in Short Pipelines . . . 101
Figure 51 Processors Hides Probabilistic Cache Miss Penalty with Out-Of-Order Execution in Short Pipelines . . . 103
Figure 52 Dependent Misses with Multiple Probabilities . . . 104
Figure 53 Layered Organization of Compiling Dataflow Systems and Profiling Results of Tailored Profiling . . . 108
Figure 54 Layers of Intermediate Representation for the Umbra Database System . . . 112
Figure 55 Example Query with Corresponding Dataflow Graph and Generated Code . . . 113
Figure 56 Performance Profile of Generated Program . . . 114
Figure 57 Tailored Profiling Requires Small Extensions . . . 116
Figure 58 Tailored Profiling Applies the Tagging Dictionary to Report the Profiling Results on Higher Abstraction Levels . . . 117
Figure 59 Register Tagging Uses Processor Register to Trace Calls to Shared Functions . . . 118
Figure 60 Tailored Profiling Provides Profiling Reports on Developers' Abstraction Levels. . . . 119
Figure 61 Tailored Profiling Associates each Sample with an Operator and Determines Operator Activity over Time . . . 120
Figure 62 Tailored Profiling Aggregates Samples up to Query Plan Level 123
Figure 63 Alternative Query Plans for the Optimizer Developer's SQL Query . . . 124

Figure 64 Operator Activity Over Time 124
Figure 65 Profile of Memory Access Patterns 125
Figure 66 Performance Overhead of Profiling Approaches 126
Figure 67 Debuggers and the Two Steps of Query Processing in Compiling Relational Engines 132
Figure 68 Example Query with Execution Plan 133
Figure 69 Control Flow of Volcano-Style Query Processing 134
Figure 70 Data-Centric Code Generation Fuses Operators into a Function 135
Figure 71 Multi-Level Debugger Prototype 139

LIST OF TABLES

Table 1	CPU Counters	15
Table 2	Runtime of Typer and Tectorwise on the Star Schema Benchmark	19
Table 3	Production System Performance	19
Table 4	Multi-Threaded Execution	25
Table 5	SSD Results	26
Table 6	Hardware Platforms	27
Table 7	Query Processing Models	34
Table 8	Codegen Primitive Types	46
Table 9	Tidy Tuples, Umbra IR, and Flying Start speed up Umbra's preparation phase	66
Table 10	The Flying Start backend out-performs both HyPer's interpreter- and unoptimized LLVM backends	69
Table 11	Lines of Code of Tidy Tuples and Flying Start	74
Table 12	Microarchitecture Attributes of Used Processors	102
Table 13	Amount of Samples Attributed to Umbra by Tailored Profiling	127
Table 14	Lines of Code of Our Prototype Implementation of Tailored Profiling	128

1 INTRODUCTION

The relational model has stood the test of time and relational database systems play an important role in today's system architectures [8, 10, 184]. One of the merits of the relational model is that it separates applications from the specifics of data management. It creates an interface which lets users specify what they wish to know from the database in a declarative language, i.e., users describe what information they desire, but not how to get it. This separation leaves many degrees of freedom to the database system, which can decide how to extract the requested information from the database. Similarly, users can put information into the database and the system is free to decide how to store it. As a consequence, the separation gives database system implementers the freedom to choose the best strategies for data storage, the best algorithms for query processing, and overall build a query engine which makes best use of the available hardware. Conversely, as the query engine mediates between user requests and the data stored on hardware, the query engine is central to building high-performance database systems. In this thesis, we study the fastest known query engine architectures and distil an architecture that is able to leverage new hardware and is prepared for future needs of computing within relational database systems.

1.1 BACKGROUND

Historically, database systems co-evolved with the available hardware. In the earliest systems, data was stored on spinning disks and the amount of available main-memory within a machine was small compared to the amount of data stored. In this setting, whenever the query engine processes data, the largest fraction of time was spent retrieving data from disks. Consequently, the design of query engines focused on optimizing disk access. Index structures were tailored to minimize the number of disk accesses [12], algorithms focused on avoiding random access [55], and parallelism was encapsulated within operators [54].

1.1.1 High-Performance System Architectures

The last two decades saw a groundbreaking shift away from disk focused engines towards main-memory oriented systems. The price of DRAM kept falling, leading to projections that DRAM would soon become very cheap and abundantly available [134], so that all data can be stored in main memory. Consequently, query engines were rebuilt

for the new bottleneck—main memory—which gave rise to a class of high-performance query engines.

With data stored in DRAM, main memory was not the only bottleneck. Boncz et al. realized that the number of instructions executed by the CPU and the efficiency of data caches within the processor became vital for query engine performance. With the MonetDB/X100 project in 2005, they proposed a vectorized interpreter model, which pipelines chunks of data through relational operators [17]. On the one hand, working on chunks amortizes interpreter overhead and, thus, reduces the amount of necessary instructions. On the other hand, the approach works on a small subset of the data at a time, which makes efficient use of processor caches.

Another line of research sought to implement fast relational operators by generating code for each query and compiling it to machine code [87, 121]. Whereas vectorization of the X100 project was an interpreter-based approach, the idea of code generation was to eliminate all interpretation overhead and thus use processors most efficiently. This idea was demonstrated with C/C++ template expansion in the HIQUE database system in 2010 and was followed in 2011 by the LLVM based data-centric code generation approach of the HyPer project [121]. Data-centric code generation emphasizes keeping data in registers as long as possible in order to eliminate superfluous move instructions and make data access as cheap as possible. Data-centric code generation became the centerpiece of the query engine of the HyPer in-memory database system which served as foundation of a large body of research on in-memory databases: Multi-version concurrency control was adapted to efficiently process transactions and run analytical queries at the same time [126]. The adaptive radix tree showed the way towards fast in-memory secondary indexing [99], data blocks demonstrated how to build in-memory storage [91], and morsel-driven parallelism showed how to to implement relational operators that make best use of many available compute cores [97].

1.1.2 Application Interfaces and System Specializations

While query engines and hardware co-evolved, using the relational model and SQL as interface towards the users was always an option, but by far not the only one. Many data models and query languages were developed to fit specific use-cases. Object-oriented databases extended relational database systems with the ability to store complex objects with types, classes, class hierarchies, and references between objects [72]. Document database systems focused on providing an interface that directly incorporates application data formats such as XML [70, 115] and JSON [25, 106]. Key-value stores reduce the data model and transaction guarantees to key-value pairs in order to make scaling the database system to compute clusters feasible [35, 90]. Meanwhile, array data systems expanded the capabilities of the relational model to store, retrieve, and analyze large-scale scientific data [34, 21, 194]. Further, graph database systems were created to update and analyze graphs, e.g., of online social interactions or road networks [109, 52, 65]. In all of these systems, the user-interfacing query language was adapted to fit the application needs and the underlying query engines were adapted accordingly, e.g.,

to be massively distributable, to cope with dense and sparse data in scientific applications, or to deal with the high interconnectedness of graphs and the ensuing random access requirements of graph analysis algorithms.

1.2 CHALLENGES

1.2.1 Hardware Trends

In the era of main-memory database systems, the primary bottleneck was considered to be the access to DRAM. The systems built under that assumption are very fast at transaction processing [126] and data analysis [121]. The pioneering assumption was that main memory prices will continuously decay, therefore future analytical database systems can afford to store all data in main memory [73, 134, 175]. While the prediction has partly come true and machines with large amounts of main memory are available and used for data storage [45], the growth of data volume outpaces price decay so that more cost efficient solutions are required [98, 123]. Promising hardware alternatives are flash-storage and non-volatile memory. Flash storage in the form of solid state drives (SSDs) currently offers 3 GB/s bandwidth, which can be combined within one machine into 25 GB/s bandwidth [61]. Access latency compared to spinning disks is low at around 100 µs, all at a price of 0.25 $/GB [61]. Non-volatile memory offers even higher bandwidth of 37 GB/s and access latencies of 390 ns at a price of currently 5.2 $/GB [157]. Overall, they offer data access with similar properties to DRAM—very high bandwidth and fast random access—at a fraction of the price [61, 156].

1.2.2 System Usage Trends

Naturally, the main use for database systems always was storing, altering, and retrieving data. Beyond that, modern systems offer many ways to analyze the stored data. For data analysis there there seems to be a recurrent theme with database systems: Novel and experimental data analysis algorithms retrieve data from the database, run their analyses externally, and derive conclusions [160]. Eventually, when the analysis algorithms are used often enough, engineers notice that moving computation to the data is more efficient than moving data to the computation. Therefore, they integrate the analyses into the system's query engine to optimally push computation to data. This process can be observed with integrating window functions into the SQL standard, moving scientific computation to array databases, and integrating concepts for graph-computations such as map-reduce into database systems.

Recent break-through findings in machine learning [96], especially deep learning, have enabled successes in the fields of computer vision [88], speech and hand-writing recognition [63, 46], natural language processing [36], machine translation [30], and many more. The field of machine learning is currently in a big leap forward and the

findings are put to use in many branches of industry and in the consumer market [178, 102, 107]. Since machine learning seems very promising, many users want to apply it to their data, either to train models or to apply already trained models to classify, transcribe, translate, or make predictions from their data. From the perspective of database systems, eventually such machine learning techniques must move closer to the data. That means, query engines will need to integrate machine learning primitives and run computation directly on the data. There are already advances into this direction, e.g., a language of machine learning primitives for big data systems [51], an approach to integrate linear algebra with relational algebra [26], and research into joint optimization of relational algebra and machine learning operations [158, 159]. Overall, the machine learning trend indicates that query engines need to support more computation-intensive workloads in the future [160].

1.3 CONTRIBUTIONS

Most likely, the next generation of high-performance database systems will make use of new storage technologies and support machine learning tasks. Yet, what should a query engine, that can keep up with the large amounts of available bandwidth and future needs of computation look like?

In this thesis we perform an extensive experimental study, described in Chapter 2, which compares the two state-of-the-art query engine architectures: vectorization [17] and data-centric code generation [121]. The study shows, that for today's workloads both architectures have roughly similar performance. We found, in comparison, data-centric code generation is more efficient for compute-intensive queries, more effective at transaction processing, and the architecture is a better fit to integrate foreign programming languages. Consequently, we see data-centric code generation as a promising architecture for future workloads and hardware environments.

However, we also conclude from the study that compilation based approaches have shortcomings. The comparison with a vectorized engine clearly pinpoints the potential for improvement. 1) Inherent to code generation is the issue that time must be spent on compilation for every query, whereas a vectorized engine can start right away. 2) The aggressive operator fusion of data-centric code generation produces concise code, but vectorized processing shows that often a different program structure makes better use of the resources of modern CPUs. 3) Query engines are complex pieces of software and require performance tuning to make best use of the available resources. The extra layer of indirection introduced by code generation makes profiling tedious, error prone, and slow. In contrast, a vectorized engine allows for more direct profiling. 4) In a similar vein, debugging a code generator is more complicated than debugging a vectorized engine. Unmitigated, these issues pose significant drawbacks for a compilation-based query engine. Therefore, this thesis presents solutions to the forementioned shortcomings. All proposals are implemented and evaluated within the next-generation database system Umbra [123].

COMPILATION TIME. From the moment a query arrives at the query engine, several preparation steps are necessary before query evaluation on the data starts. In a compiling engine this involves generating a program and compiling it to executable machine code. Depending on the query complexity and database size the preparations can take even longer than running the query. In Chapter 3, we present an engine architecture—a code generator, an intermediate program representation, and a compilation backend—that allow very fast compilation. Our evaluation shows its preparation speed is on par with interpreter based engines, that means, it is on par with engines that do not generate machine code at all.

PROGRAM STRUCTURE. Data-centric code generation is effective at generating code without superfluous instructions. However, the aggressive operator fusion tends to create long dependency chains, which makes the code prone to cache and branch misses. In Chapter 4, we introduce an optimizer that optimally cuts pipelines—thus cuts dependency chains—, boosts parallel memory access, and removes branch misses. Our evaluation shows that this technique leads to up to 2.6× faster query evaluation and the optimizer takes care to not degrade query performance.

PERFORMANCE PROFILING. Compiling query engines first work a high-level query plan, which they subsequently lower to machine code and finally execute to produce the query result. Performance profiling the generated code is very complex, as interpreting a performance profile requires many steps to find how generated instructions relate to the query plan. In Chapter 5, we introduce Tailored Profiling, a performance profiling approach that automatically maps performance profiles to high-level abstractions. We show that Tailored Profiling presents profiling views helpful to database users and developers, with very little runtime overhead.

DEBUGGING. The two step process of compilation before runtime also creates a difficult debugging environment. When stepping through generated code with a debugger, it is often necessary to know why the current instruction was generated, which operator it implements, etc. Unfortunately, code generation happens in a previous step, which makes all that information inaccessible to the debugger. In Chapter 6, we propose a debugger to step through code generation and code execution simultaneously. We show that it is a powerful tool, as it provides the required context, yet is low-effort to implement.

2 QUERY ENGINES: TO VECTORIZE OR TO COMPILE?

Excerpts of this chapter have been published in [75].
With contributions from Viktor Leis and Peter Boncz.

In most query engines, each relational operator is implemented using Volcano-style iteration [56]. While this model worked well in the past when disk was the primary bottleneck, it is inefficient on modern CPUs for in-memory database management systems (DBMSs). Most modern query engines therefore either use *vectorization* (pioneered by VectorWise [17, 197]) or *data-centric code generation* (pioneered by HyPer [121]). Systems that use vectorization include DB2 BLU [155], columnar SQL Server [92], and Quickstep [142], whereas systems based on data-centric code generation include Apache Spark [4] and Peloton [117].

Like the Volcano-style iteration model, vectorization uses pull-based iteration where each operator has a *next* method that produces result tuples. However, each *next* call fetches a block of tuples instead of just one tuple, which amortizes the iterator call overhead. The actual query processing work is performed by primitives that execute a simple operation on one or more type-specialized columns (e.g., compute hashes for a vector of integers). Together, amortization and type specialization eliminate most of the overhead of traditional engines.

In data-centric code generation, each relational operator implements a push-based interface (*produce* and *consume*). However, instead of directly processing tuples, the produce/consume calls generate code for a given query. They can also be seen as operator methods that get called during a depth-first traversal of the query plan tree, where produce is called on first visit, and consume on last visit, after all children have been processed. The resulting code is specialized for the data types of the query and fuses all operators in a pipeline of non-blocking relational operators into a single (potentially nested) loop. This generated code can then be compiled to efficient machine code (e.g., using the LLVM).

Although both models eliminate the overhead of traditional engines and are highly efficient, they are conceptually different from each other: Vectorization is based on the pull model (root-to-leaf traversal), vector-at-a-time processing, and interpretation. Data-centric code generation uses the push model (leaf-to-root traversal), tuple-at-a-time processing, and up-front compilation. As we discuss in Section 2.8, other designs that mix or combine ideas from data-centric compilation and vectorization have been proposed. In this chapter, we focus on these two specific designs, as they have been highly influential and are in use in multiple widespread systems.

The differences of the two models are fundamental and determine the organization of the DBMS's execution engine source code and its performance characteristics. Because

changing the model requires rewriting large parts of the source code, DBMS designers must decide early on which model to use. Looking at recent DBMS developments like Quickstep [142] and Peloton [117], we find that both choices are popular and plausible: Quickstep is based on vectorization, Peloton uses data-centric code generation.

Given the importance of this choice, it is surprising that there has not yet been a systematic study comparing the two state-of-the-art query processing models. In this chapter, we provide an in-depth experimental comparison of the two models to understand when a database architect should prefer one model over the other.

To compare vectorization and compilation, one could compare the runtime performance of emblematic DBMSs, such as HyPer and VectorWise. The problem is, however, that such full-featured DBMSs differ in many design dimensions beyond the query execution model. For instance, HyPer does not employ sub-byte compression in its columnar storage [91], whereas VectorWise uses more compact compression methods [198]. Related to this choice, HyPer features predicate-pushdown in scans but VectorWise does not. Another important dimension in which both systems differ is parallelism. VectorWise queries spawn threads scheduled by the OS, and controls parallelism using explicit *exchange* operators where the parallelism degree is fixed at query optimization time [7]. HyPer, on the other hand, runs one thread on each core and explicitly schedules query tasks on it on a morsel-driven basis using a NUMA-aware, lock-free queue to distribute work. HyPer and VectorWise also use different query processing algorithms and structures, data type representations, and query optimizers. Such different design choices affect performance and scalability, but are independent of the query execution model.

To isolate the fundamental properties of the execution model from incidental differences, we implemented a compilation-based relational engine and a vectorization-based engine in a single test system (available at [74]). The experiments where we employed data-centric code-generation into C++[1] we call "Typer" and the vectorized engine we call "Tectorwise" (TW). Both implementations use the same algorithms and data structures. This allows an apples-to-apples comparison of both approaches because the only difference between Tectorwise and Typer is the query execution method: vectorized versus data-centric compiled execution.

Our experimental results show that both approaches lead to very efficient execution engines, and the performance differences are generally not very large. Compilation-based engines have an advantage in calculation-heavy queries, whereas vectorized engines are better at hiding cache miss latency, e.g., during hash joins.

After introducing the two models in more detail in Section 2.1 and describing our methodology in Section 2.2, we perform a micro-architectural analysis of in-memory OLAP workloads in Section 2.3. We then examine in Section 2.4 the benefit of data-parallel operations (SIMD), and Section 2.5 discusses intra-query parallelization on multi-core CPUs. In Section 2.6, we investigate different hardware platforms (Intel, AMD, Xeon Phi) to find out which model works better on which hardware. After these

1 HyPer compiles to LLVM IR rather than C++, but this choice only affects compilation time (which we ignore in this chapter anyway), not execution time.

quantitative OLAP performance comparisons, we discuss other factors in Section 2.7, including OLTP workloads and compile time. A discussion of hybrid processing models follows in Section 2.8. We conclude by summarizing our results as a guide for system designers in Section 2.9.

2.1 VECTORIZED AND COMPILED QUERIES

The main principle of vectorized execution is *batched* execution [135] on a columnar data representation: every "work" primitive function that manipulates data does not work on a single data item, but on a vector (an array) of such data items that represents multiple tuples. The idea behind vectorized execution is to amortize the DBMS's interpretation decisions by performing as much as possible inside the data manipulation methods. For example, this work can be to hash 1000s of values, compare 1000s of string pairs, update a 1000 aggregates, or fetch a 1000 values from 1000s of addresses.

Data-centric compilation generates low-level code for a SQL query that fuses all adjacent non-blocking operators of a query pipeline into a single, tight loop. In order to understand the properties of vectorized and compiled code, it is important to understand the structure of each variant's code. Therefore, in this section we present example operator implementations, motivate why they are implemented in this fashion, and discuss some of their properties.

2.1.1 Vectorizing Algorithms

Typer executes queries by running generated code. This means that a developer can create operator implementations in any way they see fit. Consider the example in Figure 1a: a function that selects every row whose color is green and has four tires. There is a loop over all rows and in each iteration, all predicates are evaluated.

Tectorwise implements the same algorithms as Typer, staying as close to it as possible and reasonable (for performance). This is, however, only possible to a certain degree, as every function implemented in vectorized style has two constraints: It can (i) only work on *one* data type[2] and it (ii) must process multiple tuples. In generated code these decisions can both be put into the expression of one `if` statement. This, however, violates (i) which forces Tectorwise to use two functions as shown in Figure 1b. A (not depicted) interpretation logic would start by running the first function to select all elements by color, then the second function to select by number of tires. By processing multiple elements at a time, these functions also satisfy (ii). The dilemma is faced by all operators in Tectorwise and all functions are broken down into primitives that satisfy (i) and (ii). This example uses a column-wise storage format, but row-wise formats are feasible as well. To maximize throughput, database developers tend to highly optimize such func-

2 Technically, it would be possible to create primitives that work on multiple types. However, this is not practical, as the number of combinations grows exponentially.

```
vec<int> sel_eq_row(vec<string> col, vec<int> tir)
  vec<int> res;
  for(int i=0; i<col.size(); i++) // for colors and tires
    if(col[i] == "green" && tir[i] == 4) // compare both
      res.append(i) // add to final result
  return res
```

(a) Integrated: Both predicates checked at once

```
vec<int> sel_eq_string(vec<string> col, string o)
  vec<int> res;
  for(int i=0; i<col.size(); i++) // for colors
    if(col[i] == o) // compare color
      res.append(i) // remember position
  return res

vec<int> sel_eq_int(vec<int> tir, int o, vec<int> s)
  vec<int> res;
  for(i : s) // for remembered position
    if(tir[i] == o) // compare tires
      res.append(i) // add to final result
  return res
```

(b) Vectorized: Each predicate checked in one primitive

Figure 1: Multi-Predicate Example – *The straightforward way to evaluate multiple predicates on one data item is to check all at once (1a). Vectorized code must split the evaluation into one part for each predicate (1b).*

tions. For example, with the help of predicated evaluation (`*res=i; res+=cond`) or SIMD vectorized instruction logic (see Section 2.4.1).

With these constraints in mind, let us examine the details of operator implementations of Tectorwise. We implemented selections as shown above. Expressions are split by arithmetic operators into primitives in a similar fashion. Note that for these simple operators the Tectorwise implementation must already change the structure of the algorithms and deviate from the Typer data access patterns. The resulting materialization of intermediates makes fast caches very important for vectorized engines.

2.1.2 Vectorized Hash Join and Group By

Pseudo code for parts of our hash join implementations are shown in Figure 2. The idea for both, the implementation in Typer and Tectorwise, is to first consume all tuples from one input and place them into a hash table. The entries are stored in row format for better cache locality. Afterwards, for each tuple from the other input, we probe the hash table and yield all found combinations to the parent operator. The corresponding code that Typer generates is depicted in Figure 2a.

Tectorwise cannot proceed in exactly the same manner. Probing a hash table with composite keys is the intricate part here, as each probe operation needs to test equality of all parts of the composite key. Using the former approach would, however, violate

```
query(...)
  // build hash table
  for(i = 0; i < S.size(); i++)
    ht.insert(<S.att1[i], S.att2[i]>, S.att3[i])
  // probe hash table
  for(i = 0; i < R.size(); i++)
    int k1 = R.att1[i]
    string* k2 = R.att2[i]
    int hash = hash(k1, k2)
    for(Entry* e = ht.find(hash); e; e = e->next)
      if(e->key1 == k1 && e->key2 == *k2)
        ... // code of parent operator
```

(a) Code generated for hash join

```
class HashJoin
  Primitives probeHash_, compareKeys_, buildGather_;
  ...
int HashJoin::next()
  ... // consume build side and create hash table
  int n = probe->next()// get tuples from probe side
  // *Interpretation*: compute hashes
  vec<int> hashes = probeHash_.eval(n)
  // find hash candidate matches for hashes
  vec<Entry*> candidates = ht.findCandidates(hashes)
  // matches: int references a position in hashes
  vec<Entry*, int> matches = {}
  // check candidates to find matches
  while(candidates.size() > 0)
    // *Interpretation*
    vec<bool> isEqual = compareKeys_.eval(n, candidates)
    hits, candidates = extractHits(isEqual, candidates)
    matches += hits
  // *Interpretation*: gather from hash table into
  // buffers for next operator
  buildGather_.eval(matches)
  return matches.size()
```

(b) Vectorized code that performs a hash join

Figure 2: Hash Join Implementations in Typer and Tectorwise – *Generated code (Figure 2a) can take any form, e.g., it can combine the equality check of hash table keys. In vectorized code (Figure 2b), this is only possible with one primitive for each check.*

(i). Therefore, the techniques from Section 2.1.1 are applied: The join function first creates hashes from the probe keys. It does this by evaluating the `probeHash` expression. A user of the vectorized hash join must configure the `probeHash` and other expressions that belong to the operator so that when the expressions evaluate, they use data from the operator's children. Here, the `probeHash` expression hashes key columns by invoking one primitive per key column and writes the hashes into an output vector. The join

function then uses this vector of hashes to generate candidate match locations in the hash table. It then inspects all discovered locations and checks for key equality. It performs the equality check by evaluating the `cmpKey` expression. For composite join-keys, this invokes multiple primitives: one for every key column, to avoid violating (i) and (ii). Then, the join function adds the matches to the list of matching tuples, and, in case any candidates have an overflow chain, it uses the overflow entries as new candidates for the next iteration. The algorithm continues until the candidate vector is empty. Afterwards, the join uses `buildGather` to move data from the hash table into buffers for the next operator.

We take a similar approach in the group by operator. Both phases of the aggregation use a hash table that contains group keys and aggregates. The first step for all inbound tuples is to find their group in the hash table. We perform this with the same technique as in the hash join. For those tuples whose group is not found, one must be added. Unfortunately, it is not sufficient to just add one group per group-less tuple as this could lead to groups added multiple times. We therefore shuffle all group-less tuples into partitions of equal keys (proceeding component by component for composite keys), and add one group per partition to the hash table. Once the groups for all incoming tuples are known we run aggregation primitives. Transforming into vectorized form led to an even greater deviation from Typer data access patterns. For the join operator, this leads to more independent data accesses (as discussed in Section 2.3.1). However, aggregation incurs extra work.

Note that in order to implement Tectorwise operators we need to deviate from the Typer implementations. This deviation is not by choice, but due to the limitations (i) and (ii) which vectorization imposes. This yields two different implementations for each operator, but at its core, each operator executes the same algorithm with the same parallelization strategy.

2.2 COMPARISON METHODOLOGY

To isolate the fundamental properties of the execution model from incidental differences found in real-world systems, we implemented a compilation-based engine (Typer) and a vectorization-based engine (Tectorwise) in a single test system (available at [74]). To make experiments directly comparable, both implementations use the same algorithms and data structures. When testing queries, we use the same physical query plans for vectorized and compiled execution. We do not include query parsing, optimization, code generation, and compilation time in our measurements. This testing methodology allows an apples-to-apples comparison of both approaches because the only difference between Tectorwise and Typer is the query execution method: vectorized versus data-centric compiled execution.

2.2.1 Related Work

Vectorization was proposed by Boncz et al. [17] in 2005. It was first used in MonetDB/X100, which evolved into the commercial OLAP system VectorWise, and later adopted by systems like DB2 BLU [155], columnar SQL Server [92], and Quickstep [142]. In 2011, Neumann [121] proposed data-centric code generation using the LLVM compiler framework as the query processing model of HyPer, an in-memory hybrid OLAP and OLTP system. It is also used by Peloton [117] and Spark [4].

To the best of our knowledge, this chapter is the first systemic comparison of vectorization and data-centric compilation. Sompolski et al. [171] compare the two models using a number of microbenchmarks, but do not evaluate end-to-end performance for full queries. More detailed experimental studies are available for OLTP systems. Appuswamy et al. [9] evaluate different OLTP system architectures in a common prototype, and Sirin et al. [168] perform a detailed micro-architectural analysis of existing commercial and open source OLTP systems.

2.2.2 Query Processing Algorithms

We implemented five relational operators both in Tectorwise and Typer: *scan, select, project (map), join, and group by*. The scan operator at its core consists of a (parallel) `for` loop over the scanned relation. Select statements are expressed as `if` branches. Projection is achieved by transforming the expression to the corresponding C code. Unlike production-grade systems, our implementation does not perform overflow checking of arithmetic expressions. Join uses a single hash table[3] with chaining for collision detection. Using 16 (unused) bits of each pointer, the hash table dictionary encodes a small Bloom filter-like structure [97] that improves performance for selective joins (a probe miss usually does not have to traverse the collision list). The group by operator is split into two phases for cache friendly parallelization. A pre-aggregation handles heavy hitters and spills groups into partitions. Afterwards, a final step aggregates the groups in each partition. Using these algorithms in data-centric code is quite straightforward, while vectorization requires adaptations, which we describe in Section 2.1.1.

2.2.3 Workload

In this chapter we focus on OLAP performance, and therefore use the well-known TPC-H benchmark for most experiments. To be able to show detailed statistics for each individual query as opposed to only summary statistics, we chose a representative subset of TPC-H. The selected queries and their performance bottlenecks are listed in the following:

- **Q1:** fixed-point arithmetic, (4 groups) aggregation

3 Although recent research argues for partitioned hash joins [11, 161], single-table joins are still prevalent in production systems and are used by both HyPer and VectorWise.

- **Q6:** selective filters
- **Q3:** join (build: 147 K entries, probe: 3.2 M entries)
- **Q9:** join (build: 320 K entries, probe: 1.5 M entries)
- **Q18:** high-cardinality aggregation (1.5 M groups)

The given cardinalities are for scale factor (SF) 1 and grow linearly with it. Of the remaining 17 queries, most are dominated by join processing and are therefore similar to Q3 and Q9. A smaller number of queries spend most of the time in a high-cardinality aggregation and are therefore similar to Q18. Finally, despite being the only two single-table queries, we show results for both Q1 and Q6 as they behave quite differently. Together, these five queries cover the most important performance challenges of TPC-H and any execution engine that performs well on them will likely be also efficient on the full TPC-H suite [16].

2.2.4 Experimental Setup

Unless otherwise noted, we use a system equipped with an Intel i9-7900X (Skylake X) CPU with 10 cores for our experiments. Detailed specifications for this CPU can be found in the hardware section in Table 6. We use Linux as OS and compile our code with GCC 7.2. The CPU counters were obtained using Linux' perf events API. Throughout this chapter, we normalize CPU counters by the total number of tuples scanned by that query (i.e., the sum of the cardinalities of all tables scanned). This normalization enables intuitive observations across systems (e.g., "Tectorwise executes 41 instructions per tuple more than Typer on query 1") as well as interesting comparisons across other dimensions (e.g., "growing the data size by a factor of 10, causes 0.5 additional cache misses per tuple").

2.3 MICRO-ARCHITECTURAL ANALYSIS

To understand the two query processing paradigms, we perform an in-depth micro-architectural comparison. We initially focus on sequential performance and defer discussing data-parallelism (SIMD) to Section 2.4 and multi-core parallelization to Section 2.5.

2.3.1 Single-Thread Performance

Figure 3 compares the single-threaded performance of the two models for selected TPC-H queries. For some queries (Q1, Q18), Typer is faster and for others (Q3, Q9) Tectorwise is more efficient. The relative performance ranges from Typer being faster by 74% (Q1) to Tectorwise being faster by 32% (Q9). Before we look at the reasons for this, we note that these are not large differences, especially when compared to the performance gap to other systems. For example the difference between HyPer and Post-

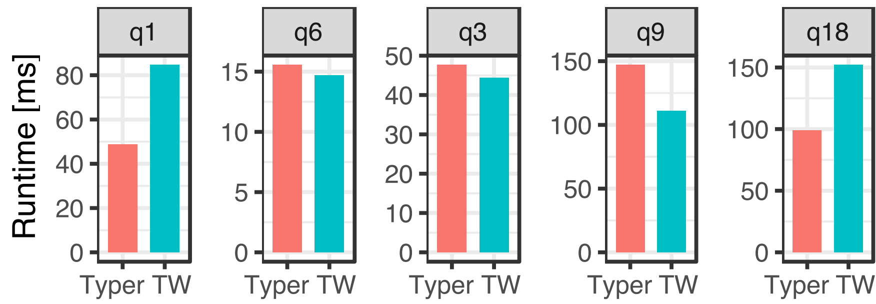


Figure 3: Performance – *TPC-H SF=1, 1 thread*

		Cycles	IPC	Instr.	L1 misses	LLC misses	Branch misses
Q1	Typer	34	2.0	68	0.6	0.57	0.01
Q1	TW	59	2.8	162	2.0	0.57	0.03
Q6	Typer	11	1.8	20	0.3	0.35	0.06
Q6	TW	11	1.4	15	0.2	0.29	0.01
Q3	Typer	25	0.8	21	0.5	0.16	0.27
Q3	TW	24	1.8	42	0.9	0.16	0.08
Q9	Typer	74	0.6	42	1.7	0.46	0.34
Q9	TW	56	1.3	76	2.1	0.47	0.39
Q18	Typer	30	1.6	46	0.8	0.19	0.16
Q18	TW	48	2.1	102	1.9	0.18	0.37

Table 1: CPU Counters – *TPC-H SF=1, 1 thread, normalized by number of tuples processed in that query*

gresSQL is between one and two orders of magnitude [84]. In other words, the performance of both query processing paradigms is quite close— despite the fact that the two models appear different from the point of someone implementing these systems. Nevertheless, neither paradigm is clearly dominated by the other which makes both viable options to implement a processing engine. Therefore, in the following we analyze the performance differences to understand the strengths and weaknesses of the two models.

Table 1 shows some important CPU statistics, from which a number of observations can be made. First, Tectorwise executes significantly more instructions (up to 2.4×) and usually has more L1 data cache misses (up to 3.3×). Tectorwise breaks all operations into simple steps and must materialize intermediate results between these steps, which results in additional instructions and cache accesses. Typer, in contrast, can often keep intermediate results in CPU registers and thus perform the same operations with fewer instructions. Based on these observations, it becomes clear why Typer is significantly faster on Q1. This query is dominated by fixed-point arithmetic operations and a cheap in-cache aggregation. In Tectorwise intermediate results must be materialized, which is similarly expensive as the computation itself. Thus, one key difference between the two

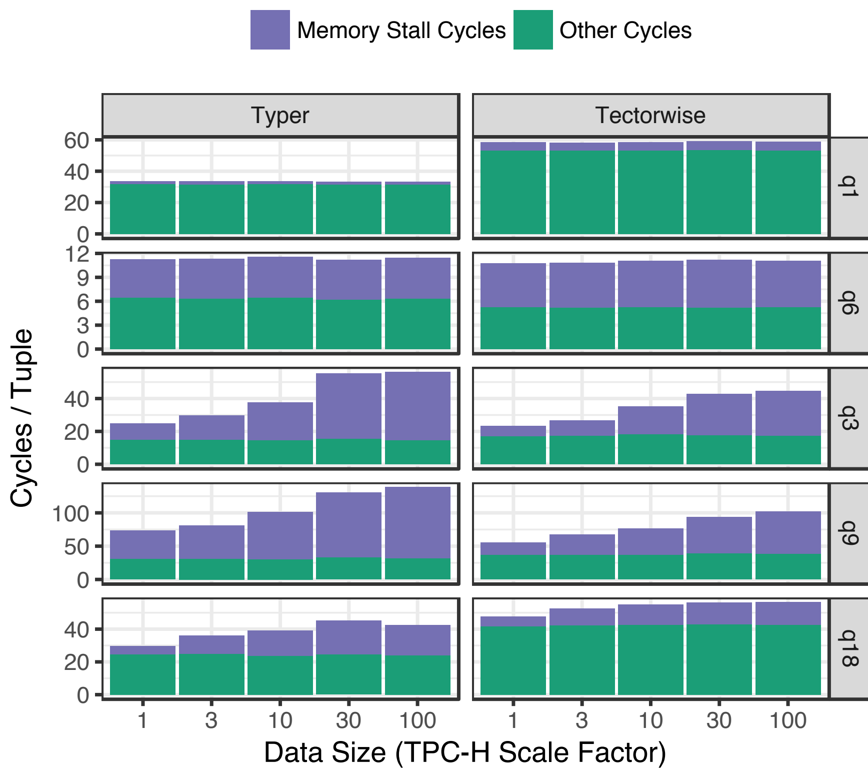


Figure 4: Memory Stalls – *TPC-H, 1 thread*

models is that Typer is more efficient for computational queries that can hold intermediate results in CPU registers and have few cache misses.

We observe furthermore, that for Q3 and Q9, whose performance is determined by the efficiency of hash table probing, Tectorwise is faster than Typer (by 4% and 32%). This might be surprising given the fact that both engines use exactly the same hash table layout and therefore also have an almost identical number of last level cache (LLC) misses. As Figure 4 shows, Tectorwise's join advantage increases up to 40% for larger data (and hash table) sizes. The reason is that vectorization is better at hiding cache miss latency, as observed from the *memory stall* counter that measures the number of cycles during which the CPU is stalled waiting for memory. This counter explains the performance difference. On the one hand, Tectorwise's hash table probing code is only a simple loop. It executes only hash table probes thus the CPU's out-of-order engine can speculate far ahead and generate many outstanding loads. These can even be executed out of order. On the other hand, Typer's code has more complex loops. Each loop can contain code for a scan, selection, hash-table probe, aggregation and more. The out-of-order window of each CPU fills up more quickly with complex loops thus they generate less outstanding loads. In addition every branch miss is more expensive than in a complex loop as more work that is performed under speculative execution is discarded and must be repeated on a miss. Overall, Tectorwise's simpler loops enable better latency hiding.

Another difference between the two executions models is their sensitivity regarding the hash function. After trying different hash functions, we settled on Murmur2 for Tectorwise, and a CRC-based hash function, which combines two 32-bit CRC results into a single 64-bit hash, for Typer. Murmur2 requires twice as many instructions as CRC hashing, but has higher throughput and is therefore slightly faster in Tectorwise, which separates hash computation from probing. For Typer, in contrast, the CRC hash function improves the performance up to 40% on larger scale factors—even though most time is spent waiting for cache misses. The lower latency and smaller number of instructions for CRC significantly improve the speculative, pipelined execution of consecutive loop iterations, thereby enabling more concurrent outstanding loads.[4]

As a note of caution, we remark that one may observe from Table 1 that Tectorwise generally executes more instructions per cycle (IPC) and deduce that Tectorwise performs better. However, this is not necessarily correct. While IPC is a measure of CPU utilization, having a higher IPC is not always better: As can be observed in Q1, Tectorwise's IPC is 40% higher, but it is still 74% slower due to executing almost twice the number of instructions. This means that one has to be cautious when using IPC to compare database systems' performance. It is a valid measure of the amount of free processing resources, but should not be used as the sole proxy for overall query processing performance.

To summarize, looking at the micro-architectural footprint of the two models we found that (1) both are efficient and fairly close in performance, (2) Typer is more efficient for computational queries with few cache misses, and (3) Tectorwise is slightly better at hiding cache miss latency.

2.3.2 Interpretation and Instruction Cache

Systems based on Volcano-style iteration perform expensive virtual function calls and type dispatch for each processed tuple. This is a form of interpretation overhead as it does not contribute to the actual query processing work. Generating machine code for a given query, by definition, avoids interpretation overhead. Vectorized systems like VectorWise are still fundamentally interpretation-based engines and use Volcano-style iteration. In contrast to classical database systems, the interpretation overhead is not incurred for each tuple but is amortized across the eponymous *vector* of tuples. Each primitive is specialized for a particular data type and is called for (e.g., 1,000 values). This amortization is effective: Using a profiler, we determined that across our query set the interpreted part is less than 1.5% of the query runtime (measured at scale factor 10). Thus, the DBMS spends 98.5% of its time in primitives doing query processing work. From Table 1 we observe that vectorized code usually executed more instructions per tuple than compiled code. Since the vast majority of the query execution time is spent within primitives, also the time to execute these extra instructions must be spent within primitives. As primitives know all involved types at compile time, we conclude that

4 Despite using different hash functions, this is still a fair comparison of join performance, as each system uses the more beneficial hash function.

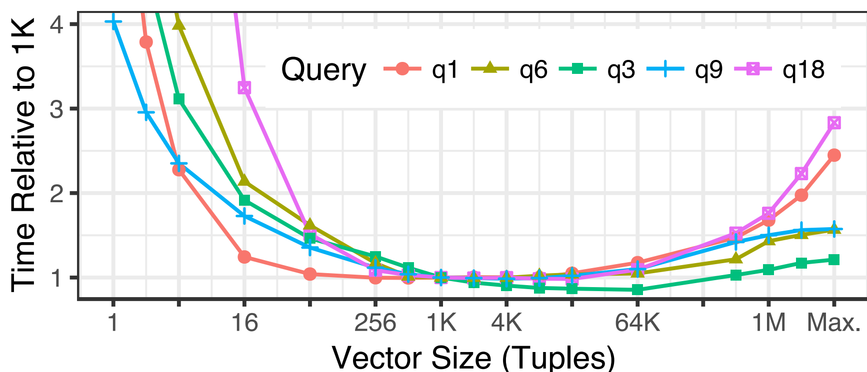


Figure 5: Tectorwise Vector Sizes – *Times are normalized by 1K vector time.*

the extra instructions are not interpretation code that is concerned with interpretation decisions and virtual function calls. It is rather due to the load/store instructions for materializing primitive results into vectors.

Recent work has found that instruction cache misses can be a problem for OLTP workloads [168]. To find out whether this is the case for our two query engines, we measured L1 instruction cache misses for both systems and found that instruction cache misses are negligible, thus not a performance bottleneck for OLAP queries. For all queries measured, the L1 instruction cache (32 KB) was large enough to contain all hot code.

2.3.3 Vector Size

The vector size is an important parameter for any vectorized engine. So far, our Tectorwise experiments used a value of 1,000 tuples, which is also the default in VectorWise. Figure 5 shows normalized query runtimes for vector sizes from 1 to the maximum (i.e., full materialization). We observe that small (<64) and large vector sizes (>64 K) decrease performance significantly. With a vector size of 1, Tectorwise is a Volcano-style interpreter with its large CPU overhead. Large vectors do not fit into the CPU caches and therefore cause cache misses. The other end of the spectrum is to process the query one column at a time; this approach is used in MonetDB [19]. Generally, a vector size of 1,000 seems to be a good setting for all queries. The only exception is Q3, which executes 15% faster using a vector size of 64K.

2.3.4 Star Schema Benchmark

So far, we investigated a carefully selected subset of TPC-H. To show that our findings are more generally applicable, we also implemented the Star Schema Benchmark (SSB), which consists of 4 query templates (with different selections) and which is dominated by hash table probes. We use one thread and scale factor 30 to achieve the runtimes listed in Table 2.

		Cycles	IPC	Instr.	L1 misses	LLC misses	Branch miss	Memory stalls
Q1.1	Typer	28	0.7	21	0.3	0.31	0.69	6.33
Q1.1	TW	12	2.0	23	0.4	0.29	0.05	2.77
Q2.1	Typer	39	0.8	30	1.3	0.12	0.17	18.35
Q2.1	TW	30	1.5	44	1.6	0.13	0.23	7.63
Q3.1	Typer	55	0.7	40	1.1	0.20	0.24	27.95
Q3.1	TW	53	1.3	71	1.7	0.23	0.41	15.68
Q4.1	Typer	78	0.5	39	1.8	0.31	0.38	45.91
Q4.1	TW	59	1.0	61	2.5	0.32	0.63	19.48

Table 2: Runtime of Typer and Tectorwise on the Star Schema Benchmark

These results are quite similar to TPC-H Q3 and Q9 and show once more that Tectorwise requires more instructions but has an advantage for join heavy queries due to better hidden memory stalls. In general, we find that TPC-H subsumes SSB for our purposes and in the name of conciseness, we present our findings using TPC-H in the rest of this chapter.

2.3.5 Tectorwise/Typer versus VectorWise/HyPer

Let us close this section by comparing Actian Vector 5.0 (the current marketing name for VectorWise) and the research (TUM) version of HyPer (a related system is now with Tableau). The results are shown in Table 3 use one thread and TPC-H scale factor 1. The first observation is that HyPer performs similarly to Typer, and Tectorwise's performance is similar to VectorWise. Second, except for Q6[5], either Typer or Tectorwise are

5 HyPer is faster on Q6 than the other systems because it evaluates selections using SIMD instructions directly on compressed columns [91].

	HyPer	VectorWise	Typer	Tectorwise
Q1	53	71	**44**	85
Q6	**10**	21	15	15
Q3	48	50	47	**44**
Q9	124	154	126	**111**
Q18	224	159	**90**	154

Table 3: Production System Performance on TPC-H in comparison to our prototypical implementations.

slightly faster than both production-grade systems. It is unsurprising given that these must handle complex issues like overflow checking (that our prototype ignores).

2.4 DATA-PARALLEL EXECUTION (SIMD)

Let us now turn our attention to data-parallel execution using SIMD operations. There has been extensive research investigating SIMD for database operations [196, 187, 151, 152, 150, 149, 172, 170]. It is not surprising that this research generally assumes a vectorized execution model. The primitives of vectorized engines consist of simple tight loops that can be translated to data-parallel code. Though there has been research on utilizing SIMD in data-centric code [146, 117], this is more challenging since the generated code is more complex. We will therefore use Tectorwise as the platform for evaluating how large the impact of SIMD on in-memory OLAP workloads is. In contrast to most research on SIMD, we use TPC-H and not micro-benchmarks.

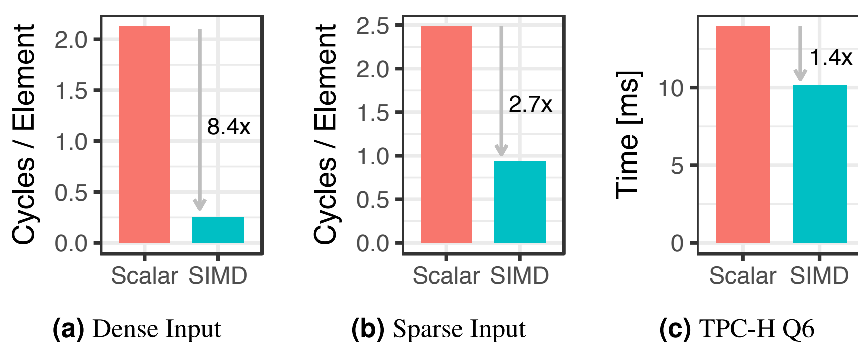
The Skylake X CPU we use for this chapter supports the new AVX-512 instruction set and can execute two 512-bit SIMD operations per cycle—doubling register widths and throughput in comparison with prior microarchitectures. In other words, using AVX-512 one can process 32 values of 32-bit per cycle, while scalar code is limited to 4 values per cycle. Furthermore, in comparison with prior SIMD instruction sets like AVX2, AVX-512 is more powerful (almost all operations support masking and there are new instructions like compress and expand) and orthogonal (almost all operations are available in 8, 16, 32, and 64-bit variants). One would therefore expect significant benefits from using SIMD. In the following, we focus on selection and hash table probing, which are both common and important operations.

2.4.1 Data-Parallel Selection

A vectorized selection primitive produces a selection vector containing the indexes of all matching tuples. Using AVX-512 this can be implemented using SIMD quite easily[6]. The comparison instruction generates a mask that we then pass to a compress store (`COMPRESSSTORE`) instruction. This operation works across SIMD lanes and writes out the positions selected by the mask to memory.

We performed a micro-benchmark for selection, comparing a branch-free scalar x86 implementation with a SIMD variant. In the benchmark, we select all elements from an 8192 element integer array which are smaller than a constant. Results for a best-case scenario, in which all consumed data are 32-bit integers, are present in the L1 cache, and the input is a contiguous vector, are shown in Figure 6a. The observed performance gain for this micro-benchmark is 8.4×. However, as Figure 6c shows, in a realistic query with multiple expensive selections like Q6, we only observe a speedup of

6 With AVX2, the selection primitive is non-trivial and either requires a lookup table [149, 91, 117] or complex permutation logic based on BMI2 [66].



(a) Dense Input **(b)** Sparse Input **(c)** TPC-H Q6

Figure 6: Scalar vs. SIMD Selection in Tectorwise – *(a) 40% selectivity. (b) Secondary selection: Input selection vector selects 40% and selection selects 40%. (c) Runtime of TPC-H Q6, SF=1*

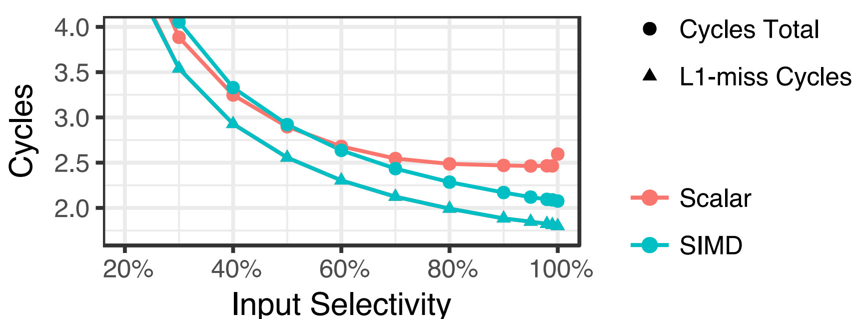
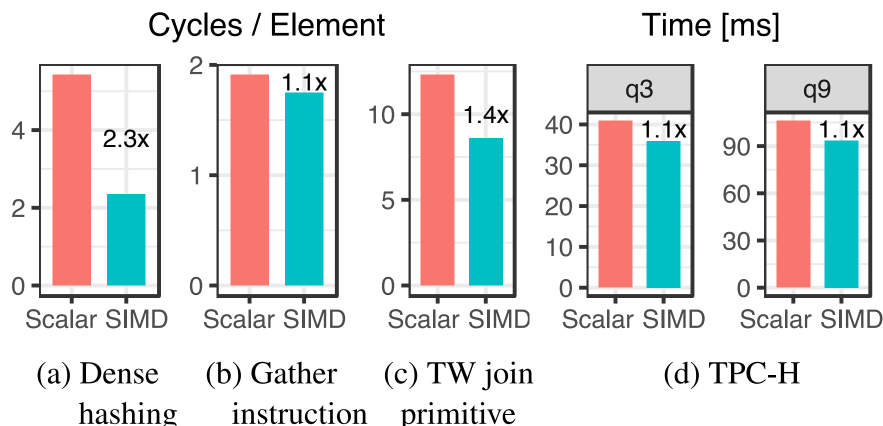


Figure 7: Sparse Selection – *Cost of selection operation with selection vector on input, depending on element size and implementation variant. Output selectivity 40%, data set size 4 GB*

1.4×—even though almost 90% of the processing time is spent in SIMD primitives. Our experiments revealed two effects that account for this discrepancy: *sparse data loading* due to selection vectors and *cache misses* due to varying stride. The remainder of this section discusses these effects.

Sparse data loading occurs in all selection primitives except for the first one. From the second selection primitive on, all primitives receive a selection vector that determines the elements to consider for comparison. These elements must be gathered from non-contiguous memory locations. A comparison of selection primitives with selection vectors (40% selectivity) is shown in Figure 6b. Performance gains in this case range only up to a 2.7× (again for 32-bit types). Considering that the selections in Q6 consist of one initial selection without input selection vector and four subsequent selections that have to consider a selection vector, we can expect the overall speedup to be closer to 3× than to 8×.

The previous benchmarks only considered data sets which reside in L1 cache. For larger data sets, accessing memory can become a limiting factor. Figure 7 shows the interplay of selection performance and input sparsity on a 4 GB data set. Note that the performance drops for selectivities below 100%, while the scalar and SIMD variants are



(a) Dense hashing (b) Gather instruction (c) TW join primitive (d) TPC-H

Figure 8: Scalar vs. SIMD Join Probing – *in microbenchmarks and full queries.*

nearly equal when the is selectivity are below 50%. We also show an estimate of how many cycles on average are spent resolving cache misses. We observe that most of the time is spent waiting for data. Thus the memory subsystem becomes the bottleneck of the selection operation and the positive effect of utilizing SIMD instructions disappears. In the selection cascade of Q6, only the first selection primitive benefits from SIMD and selects 43% of the tuples. This leaves all subsequent selections to operate in a selectivity area where the scalar variant is just as fast.

2.4.2 Data-Parallel Hash Table Probing

We next examine hash join probing, where most of the query processing time is spent in TPC-H. There are two opportunities to apply SIMD: computing the hash values, and the actual lookups in the hash table. For hashing we use Murmur2, which consists of arithmetic operations like integer shifts and multiplications that are available in AVX-512. We can also apply SIMD to lookups into hash tables by using gather, compress store, and masking.

A performance breakdown of components necessary for hash joins is shown in Figure 8. Figure 8(a) shows that for hashing alone a gain of 2.3× is possible. For gather instructions, shown in Figure 8(b), we observe an improvement of 1.1× (in the best case). This is because the memory system of the test machine can perform at most two load operations per cycle—regardless of whether SIMD gather or scalar loads are used. Figure 8(c) shows that when employing gather and other SIMD instructions to the Tectorwise probe primitive, a best-case performance gain of 1.4× can be achieved.

With a SIMD speedup of 2.3× for hashing and 1.4× for probing, one may expect an overall speedup in between. However, as is shown in Figure 8(d) the performance gains almost vanish for TPC-H join queries. This happens even though the majority of the time (55% and 65%) is spent in SIMD-optimized primitives. The reason for this behavior can be found in Figure 9. With a growing working set, gains from SIMD diminish and the execution costs are dominated by memory latency. SIMD is only beneficial

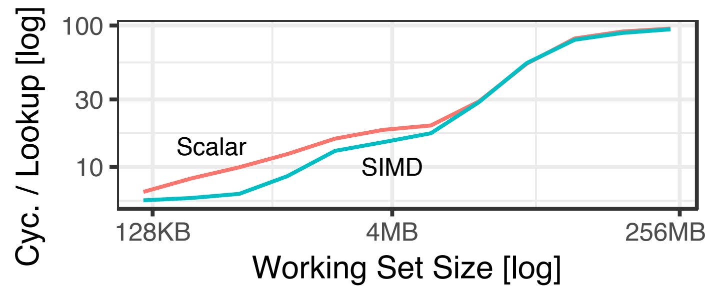


Figure 9: Join Probe – *Interaction of working set size and cost per tuple during Tectorwise hash table lookup*

when all data fits into the cache. We are not the first to observe this phenomenon: Polychroniou et al. [149] found this effect in their study of application of SIMD to database operators.

2.4.3 Compiler Auto-Vectorization

We manually rewrote Tectorwise primitives using SIMD intrinsics. Given that the code of most primitives is quite simple, one may reasonably ask whether compilers can do this job automatically. We tested the GCC 7.2, Clang 5.0, and ICC 18 compilers. Of these, only ICC was able to auto-vectorize a fair amount of primitives (and only with AVX-512). Figure 10 shows how successful ICC was in relevant paths for query processing. Its vectorized variant reduces the observed number of instructions executed per tuple by between 20% to 60%. By inspecting traces of the executed code, we confirmed that automatic vectorization was applied to hashing, selection, and projection primitives. Hash table probing and aggregation, however, were not transformed. We also show a variant with automatic and manual SIMD application combined, which has a benefit for Q3 and Q9.

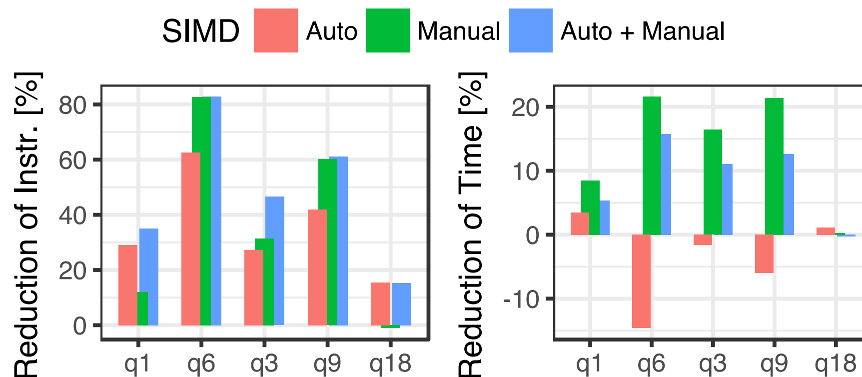


Figure 10: Compiler Auto-Vectorization – *(ICC 18)*

Unfortunately, these automatic SIMD optimizations do not yield any significant improvements in query runtime. Automatic vectorization alone hardly creates any gains but even introduces cases where the optimized code becomes slower. This means that even though primitives can be auto-vectorized, this is not yet a fire-and-forget solution.

2.4.4 Summary

We found with AVX-512 it is often straightforward to translate scalar code to data-parallel code, and observed performance gains of up to 8.4× in micro-benchmarks. However, for the more complicated TPC-H queries, the performance gains are quite small (around 10% for join queries). Fundamentally, this is because most OLAP queries are bound by data access, which does not (yet) benefit much from SIMD, and not by computation, which is the strength of SIMD. Coming back to the comparison between data-centric compilation and vectorization, we therefore argue that SIMD does not shift the balance in favor of vectorization much[7].

2.5 INTRA-QUERY PARALLELIZATION

Given the decade-long trends of stagnating single-threaded performance and growing number of CPU cores—Intel is selling 28 cores (56 hyper-threads) on a single Xeon chip—any modern query engine must make good use of all available cores. In the following, we discuss how to incorporate parallelism into the two query processing models.

2.5.1 Exchange vs. Morsel-Driven Parallelism

The original implementations of VectorWise and HyPer use different approaches. VectorWise uses exchange operators [7]. This classic approach [54] keeps its query processing operators like aggregation and join largely unaware of parallelism. HyPer, on the other hand, uses morsel-driven parallelism, in which joins and aggregations use shared hash-tables and are explicitly aware of parallelism. This allows HyPer to achieve better locality, load-balancing, and thus scalability, than VectorWise [97]. Using the 20 hyper-threads on our 10-core CPU, we measured an average speedup on the five TPC-H queries of 11.7× in HyPer, but only 7.2× in VectorWise. The parallelization framework is, however, orthogonal to the query processing model and we implemented morsel-driven parallelization in both Tectorwise and Typer, as it has been shown to scale better than exchange operators [97].

Morsel-driven parallelism was developed for HyPer [97] and can therefore be implemented quite straightforwardly in Typer: The table scan loop is replaced with a parallel

7 We note that the benefit of SIMD can be larger when data is compressed [91] and on vector-oriented CPUs like Xeon Phi (see Section 2.6).

	Threads	Typer ms	Typer speedup	Tectorwise ms	Tectorwise speedup	Ratio
Q1	1	4426	1.0	7871	1.0	0.56
Q1	10	496	8.9	867	9.1	0.57
Q1	20	466	9.5	708	11.1	0.66
Q6	1	1511	1.0	1443	1.0	1.05
Q6	10	243	6.2	213	6.8	1.14
Q6	20	236	6.4	196	7.4	1.20
Q3	1	9754	1.0	7627	1.0	1.28
Q3	10	1119	8.7	913	8.4	1.23
Q3	20	842	11.6	743	10.3	1.13
Q9	1	28086	1.0	20371	1.0	1.38
Q9	10	3047	9.2	2394	8.5	1.27
Q9	20	2525	11.1	2083	9.8	1.21
Q18	1	13620	1.0	18072	1.0	0.75
Q18	10	2099	6.5	2432	7.4	0.86
Q18	20	1955	7.0	2026	8.9	0.97

Table 4: Multi-Threaded Execution – *TPC-H SF=100 on Skylake (10 cores, 20 hyper-threads)*

loop and shared data structures like hash tables are appropriately synchronized similar to HyPer's implementation [97, 100].

For Tectorwise, it is less obvious how to use morsel-driven parallelism. The runtime system of Tectorwise creates an operator tree and exclusive resources for every worker. To achieve that the workers can work together on one query, every operator can have shared state. For each operator, a single instance of shared state is created. All workers have access to it and use it to communicate. For example, the shared state for a hash join contains the hash-table for the build side and all workers insert tuples into it. In general, the shared state of each operator is used to share results and coordinate work distribution. Additionally, pipeline breaking operators use a barrier to enforce a global order of sub-tasks. The hash join operator uses this barrier to enforce that first all workers consume the build side and insert results into a shared hash table. Only after that, the probe phase of the join can start. With shared state and a barrier, the Tectorwise implementation exhibits the same workload balancing parallelization behavior as Typer.

2.5.2 Multi-Threaded Execution

We executed our TPC-H workload on scale factor 100 (ca. 100 GB of data). Table 4 shows runtimes and speedups in comparison with single-threaded execution. Using the 10 physical cores of our Skylake CPU, we see speedups between 8× and 9× for Q1, Q3,

	Typer ms	Tectorwise ms	Ratio
Q1	923	1184	0.78
Q6	808	773	1.05
Q3	1405	1313	1.07
Q9	3268	2827	1.16
Q18	2747	2795	0.98

Table 5: SSD Results

and Q9 in both systems. Given that modern CPUs reduce clock rates significantly when multiple threads are used these results are close to perfect scalability. For the scan query Q6 the speedup is limited by the available read memory bandwidth, and the large-scale aggregation of Q18 approaches the write bandwidth.

We also conducted these experiments on AWS EC2 machines and found that both systems scale equally well. However, we observe that when we use a larger EC2 instance to speed up query execution, the price per query rises. For example, the geometric mean over our TPC-H queries for a m5.2xlarge instance with 8 vCPUs is 0.0002$ per query (2027 ms runtime). On an instance with 48 cores it is 0.00034$ per query (534 ms runtime). So in this case, running queries 4× faster costs 1.7× more.

Tectorwise and Typer have similar scaling behavior. Nevertheless, the "Ratio" column of Table 4, which is the quotient of the runtimes of both systems, reveals an interesting effect: For all but one query, the performance gap between the two systems becomes smaller when all 20 hyper-threads are used[8]. For the join queries Q3 and Q9, the performance benefit of Tectorwise is cut in half, and Tectorwise comes closer to Typer for Q1 and Q18. This indicates that hyper-threading is effective at hiding some of the downsides of microarchitecturally sub-optimal code.

2.5.3 Out-Of-Memory Experiments

To compare Tectorwise and Typer at maximum speed, all measurements so far were in-memory (i.e., all table data was present in main memory). Large OLAP databases often exceed main memory capacity, which is why we also measured the impact of fetching the data from secondary storage. To do this, we stored the table data in a RAID 5 array of 3 SATA SSDs providing 1.4 GB/s read bandwidth instead of main memory, which has a bandwidth of 55 GB/s. Table 5 shows the runtimes with 20 threads on scale factor 100 when data is read from secondary storage. Comparing these with the in-memory results (cf. Table 4), we can observe that the performance differences between the two query engines are slightly smaller but still noticeable ("Ratio" moves closer to

8 Q6 is memory bound and is the only exception. Typer's branch-free selection implementation consumes more memory bandwidth resulting in 20% lower performance at high thread counts.

	Intel Skylake	AMD Threadripper	Intel Knights Landing
Model	i9-7900X	1950X	Phi 7210
Cores (SMT)	10 (x2)	16 (x2)	64 (x4)
Issue width	4	4	2
SIMD [bit]	2×512	2×128	2×512
Clock rate [GHz]	3.4-4.5	3.4-4.0	1.2-1.5
L1 cache	32 KB	32 KB	64 KB
L2 cache	1 MB	1 MB	1 MB
LLC	14 MB	32 MB	(16 GB)
List price [$]	989	1000	1881
Launch	Q2'17	Q3'17	Q4'16
Mem BW [GB/s]	58	56	68

Table 6: Hardware Platforms – *used in experiments.*

one). Furthermore, as expected, the performance of the scan-dominated Q1 and Q6 are more affected by the slower bandwidth than the performance of the join and aggregation queries. Overall, we find that our in-memory analysis applies to out-of-memory settings with modern I/O devices.

2.6 HARDWARE

In previous experiments, we solely measured on Intel's latest microarchitecture Skylake. To find out whether our results also hold for other hardware platforms, we now also look at AMD with its recent Zen microarchitecture and Intel's Phi product line.

2.6.1 Intel Skylake X versus AMD Threadripper

Table 6 shows the technical specifications for our Intel and AMD CPUs. Intel Skylake and AMD Threadripper cost almost the same, which directly allows comparing performance per dollar. Both systems also posses an almost equal memory bandwidth. However, the AMD Threadripper features 16 compute cores, clocked at maximally 4.0 GHz, while the Intel Skylake clocks at a higher rate of 4.5 GHz but contains only 10 cores. The differences between these processors is not coincidental but rather represents the design choices of the overall CPU product palettes of AMD and Intel. AMD offers more cores per dollar, but has only a quarter of computational SIMD throughput.

In terms of query processing performance our experiments show that both CPU models are roughly on par in absolute performance. Figure 11 shows the performance (in queries / second) for our experimental queries and systems both on Skylake and on

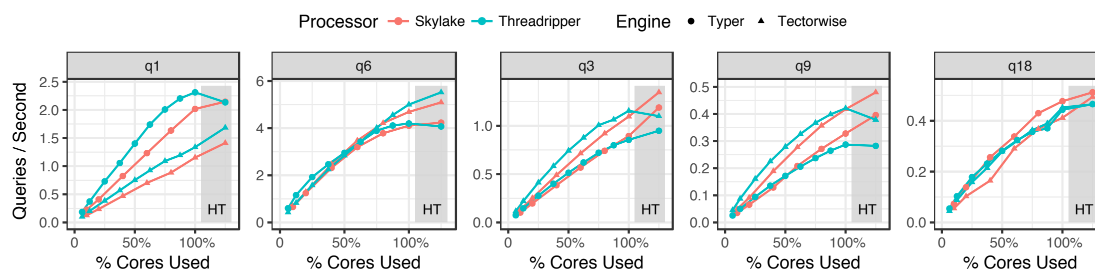


Figure 11: Skylake vs. Threadripper. – *SF=100*

Threadripper. As both processors have a different core counts the graphs are normalized on the x-axis to show which percentage of the available cores was used to achieve the runtime. Notably, the performance of both CPUs is very similar for Q6 and Q18 and the remaining queries are still quite similar (Q1 $<$ 20%, Q3 $<$ 25%, Q9 $<$ 40%). Also the relative performance of Typer and Tectorwise are quite similar. The join queries Q3 and Q9 and the selective scan in Q6 are processed faster by Tectorwise. Typer has an advantage on the computational query Q1. Overall, the performance characteristics two platforms are quite similar and the relative performance between the two hardware platforms is almost the same on both CPUs.

The only significant difference between the two platforms is that, although both platforms offer 2-way Simultaneous Multi-Threading (SMT), Intel's hyper-threading implementation seems to be much better. On the Skylake, we see a performance boost from hyper-threading for all queries. On the AMD system, the benefit of SMT is either very small, and for some queries the use of hyper-threads results in a performance degradation.

2.6.2 Knights Landing (Xeon Phi)

Despite being from two different hardware manufacturers, Skylake and Threadripper are quite similar. This cannot be said of the second generation of Intel's Xeon Phi product line. This microarchitecture is also called *Knights Landing* and is designed as a processor for high-performance computing (HPC). It is an integrated many-core architecture: There are 64 to 72 cores on each chip, but every core is relatively slow compared to Xeon cores. On the plus side, each core is equipped with two 512-bit vector processing units with an aggregate capacity of multiple TFLOPs. That makes it attractive for HPC applications.

From a database systems perspective, Knights Landing seems promising. Main memory can directly be accessed using six DDR4 memory channels (in contrast to GPUs data does not have to copied through PCIe). Each core features a 64 KB L1 cache and a 1 MB L2 cache that is shared with one neighboring core. Additionally, 16 GB of high-bandwidth memory, with a bandwidth of around 300 GB per second, is available. The available memory, number of cores, and the fact that many SIMD resources are available make the Knights Landing processor seem like a perfect OLAP machine.

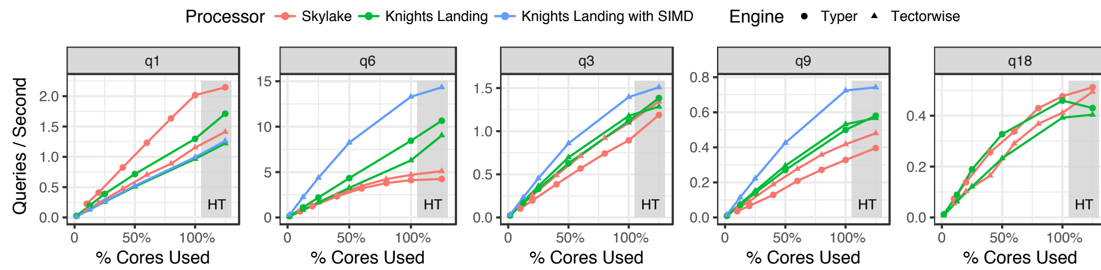


Figure 12: Skylake versus Knights Landing – *SF=100.*

Naturally, we want to explore this machine's qualities and see how Tectorwise and Typer perform in this scenario.

For our experiments on Knights Landing we configured the high-bandwidth memory as a hardware-managed L3 cache and expose all CPUs as one NUMA node (Quadrant Mode). A comparison of query processing performance of Knights Landing against Skylake is shown in Figure 12. Without any changes to the code we observe about the join queries Q3 and Q9 that Knights Landing's execution performance is from 0 to 25% higher than Skylake's. The relative performance of Tectorwise and Typer is similar on both CPUs.

For query Q18 Knights Landing's performance is about 20% lower. On query Q1 it is about 30% lower. Finally, on Q6 Knights Landing is up to 2× faster. Recall that on Skylake query Q6 is bandwidth bound. Thus the extra 2 DDR4 channels of Knights Landing combined with the high-bandwidth memory as cache provide the required resources to get ahead of the Skylake processor. This measurement, however, must be seen in perspective: Each of our measurements is executed repeatedly. The cache of 16 GB, which can hold the entire working set of query Q6, boosts the performance unrealistically. In a real workload the cache would be shared with other queries which would likely evict much of query Q6's data. As a frame of reference one may use our measurement of query Q6 with the hardware configured not to have an L3 cache. In that case Knights Landing's Q6 performance is only 10% higher than the respective performance on Skylake. In a mixed workload one can expect the difference to be between these two. As a summary up to this point, with some queries being slightly faster and others slightly slower than Skylake, Knights Landings's performance seems not that great.

To be fair, this platform is designed for heavy use of SIMD instructions. Therefore, we need to take the measurements with manual SIMD optimizations into account. We observe that Knights Landing is able to execute a join query up to 50% faster than Skylake. On the selection query Q6 even a factor of almost 3× is achieved (although the same remark as for the scalar variant of Q6 applies). However, when taking a step back and looking at the whole performance picture, we also need to take the cost of each processor into account. Unfortunately, Knights Landing comes at almost twice the price of our Intel and AMD processors. Thus when the performance is broken down to execution speed per dollar, the commodity CPUs come out on top.

2.7 OTHER FACTORS

So far, we have focused on OLAP workloads and found only moderate performance differences between the two model—in particular, when properly parallelized. The performance differences are not large enough to make a general recommendation whether to use vectorization or compilation. Therefore, as a practical matter, other factors like OLTP performance or implementation effort, which we discuss in this section, may be of greater importance.

2.7.1 OLTP and Multi-Language Support

The vectorized execution model achieves efficiency when many vectors of values are processed, which is almost always the case in OLAP, but not in OLTP, where a query might only touch a single tuple. For OLTP workloads, vectorization has little benefit over traditional Volcano-style iteration. With compilation, in contrast, it is possible to compile all queries of a stored procedure into a single, efficient machine code fragment. This is a major benefit of compilation for OLTP and HTAP systems. Despite already having a modern vectorized engine (Apollo [92, 93]), the Microsoft SQL Server team felt compelled to additionally integrate the compilation-based engine Hekaton [48].

Compilation can also be highly beneficial for integrating user-defined functions and multiple languages into the same execution environment [32, 136, 165].

2.7.2 Compilation Time

A disadvantage of code generation is the risk of compilation time dominating execution time [185]. This can be an issue in OLTP queries, though in transactional workloads it can be countered by relying on stored procedures, in which case code-generation can be done ahead of time. However, compilation time can also become large if the generated code is large because (optimizing) LLVM compile time is often super-linear to code size. OLAP queries that consist of many operators will generate large amounts of code, but also a small SQL query such as `SELECT * FROM T` can produce a lot of code if table `T` has thousands of columns, as each column leads to some code generation. Real-world data-centric compilation systems take mitigating measures against this. HyPer switches off certain LLVM optimization passes such as register allocation and replaces them by its own more scalable register allocation algorithm, and even contains a LLVM IR interpreter that is used to execute the first morsels of data; if that is enough to answer the query, full LLVM compilation is omitted [84]. This largely obviates this downside of compilation—but comes at the cost of additional system complexity. Spark falls back to interpreted tuple-at-a-time execution if a pipeline generates more than 8 KB Java byte code.

2.7.3 Profiling and Debuggability

A practical advantage of vectorized execution is that detailed profiling is possible without slowing down queries, since getting clock cycle counts for each primitive adds only marginal overhead, as each call to the function works on a thousand values. For data-centric compilation, it is hard to separate the contribution of the individual relational operators to the final execution time of a pipeline, though it could be done using sample-based code profiling, if the system can map back generated code lines to the relational operator in the query plan responsible for it. For this reason it is currently not possible in Spark SQL to know the individual contributions to execution time of relational operators, since the system can only measure performance on a per-pipeline basis.

2.7.4 Adaptivity

Adaptive query execution, for instance to re-order the evaluation order of conjunctive filter predicates or even joins is a technique for improving robustness that can compensate for (inevitable) estimation errors in query optimization. Integrating adaptivity in compiled execution is very hard; the idea of adaptive execution works best in systems that interpret a query—in adaptive systems they can change the way they interpret it during runtime. Vectorized execution is interpreted, and thus amenable for adaptivity. The combination of fine-grained profiling and adaptivity allows VectorWise to make various *micro-adaptive* decisions [154].

We saw that VectorWise was faster than Tectorwise on TPC-H Q1 (see Table 1); this is due to an adaptive optimization in the former, similar to [58], that it not present in the latter. During aggregation, the system partitions the input tuples in multiple selection vectors; one for each group-by key. This task only succeeds if there are few groups in the current vector; if it fails the system exponentially backs off from trying this optimization in further vectors. If it succeeds, by iterating over all elements in a selection vector, i.e. all tuples of one group in the vector, hash-based aggregation is turned into ordered aggregation. Ordered aggregation then performs partial aggregate calculation, keeping e.g. the sum in a register which strongly reduces memory traffic, since updating aggregate values in a hash table for each tuple is no longer required. Rather, the aggregates are just updated once per vector.

2.7.5 Implementation Issues

Both models are non-trivial to implement. In vectorized execution the challenge is to separate the functionality of relational operators in control logic and primitives such that the primitives are responsible for the great majority of computation time (see Section 2.1.1). In compiled query execution, the database system is a compiler and consists of code that generates code, thus it is harder to comprehend and debug; especially if the generated code is quite low-level, such as LLVM IR. To make code generation main-

tainable and extensible, modern compilation systems introduce abstraction layers that simplify code generation [165] and make it portable to multiple backends [136]. For HyPer, some of these abstractions have been discussed in the literature [124], while others have yet to be published.

It is worth mentioning that vectorized execution is at some disadvantage when sort-keys in order by or window functions are composite (consist of multiple columns). Such order-aware operations depend on comparison primitives, but primitives can only be specialized for a single type (in order to avoid code explosion). Therefore, such comparisons must be decomposed in multiple primitives, which requires a (boolean) vector as interface to these multiple primitives. This extra materialization costs performance. Compiled query execution can generate a full sort algorithm specifically specialized to the record format and sort keys at hand.

2.7.6 Summary

As a consequence of their architecture and code structure compilation and vectorization have distinct qualities that are not directly related to OLAP performance:

	OLTP	Language Support	Compilation Time	Profiling	Adaptivity	Implementation
Compilation	✓	✓	(✓)			Indirection
Vectorization			✓	✓	✓	Constraints

On the one hand, compiled queries allow for fast OLTP stored procedures and seamlessly integrating different programming languages. Vectorization, on the other hand, offers very low query compile times, as primitives are precompiled: As a result of this structure, parts of a vectorized query can be swapped adaptively during runtime and profiling is easier. Finally, both systems have their own implementation challenges: Implementing operators with code generation introduces an additional indirection, whereas vectorization comes with a set of constraints on the code, which can be complicated to handle.

2.8 BEYOND BASIC VECTORIZATION AND DATA-CENTRIC CODE GENERATION

2.8.1 Hybrid Models

Vectorization and data-centric code generation are fundamentally different query processing models and the applied techniques are mostly orthogonal. That means that a

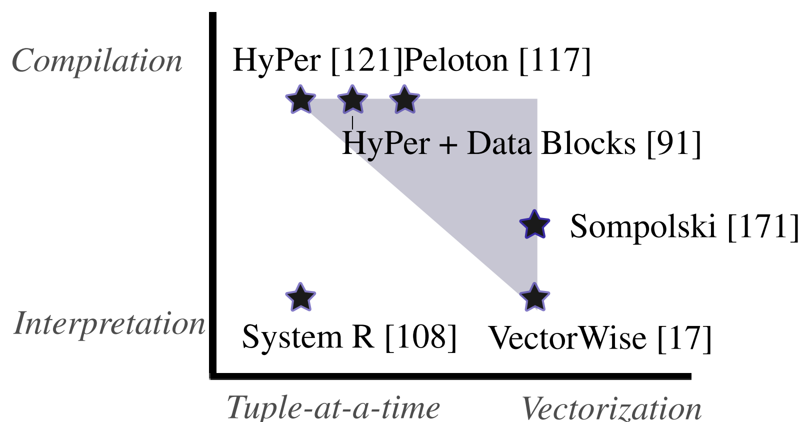


Figure 13: **Design Space Between Vectorization and Compilation** – *hybrid models integrate the advantages of the other approach.*

design space, as visualized in Figure 13, exists between them. Many systems combine ideas from both paradigms in order to achieve the "best of both worlds". We first describe how vectorization can be used to improve the performance of the compilation-based systems HyPer and Peloton, before discussing how compilation can help vectorization.

In contrast to other operators in HyPer, scans of the compressed, columnar *Data Block* format [91] are implemented in template-heavy C++ using vectorization and without generating any code at runtime. Each attribute chunk (e.g., 2^{16} values) of a Data Block may use a different compression format (based on the data in that block). Using the basic data-centric compilation model, a scan would therefore have to generate code for all combinations of accessed attributes and compression formats—yielding exponential code size growth [91]. Besides compilation time, a second benefit of using vectorization-style processing in scans is that it allows utilizing SIMD instructions where it is most beneficial (Section 2.4). Since, Data Blocks is the default storage data format, HyPer (in contrast to Typer) may be considered a hybrid system that uses vectorization for base table selections and decompression, and data-centric code generation for all other operators.

By default, data-centric code generation fuses all operators of the same pipeline into a single code fragment. This is often beneficial for performance, as it avoids writing intermediate results to cache/memory by keeping the attributes of the current row in CPU registers as much as possible. However, there are also cases where it would be better to explicitly break a single pipeline into multiple fragments—for example, in order to better utilize out-of-order execution and prefetching during a hash join. This is the key insight behind Peloton's *relaxed operator fusion* [117] model, which selectively introduces explicit materialization boundaries in the generated code. By batching multiple tuples, Peloton can easily introduce SIMD and software prefetching instructions [117]. Consequently, Peloton's pipelines are shorter and their structure resembles vectorized code (see Figure 13). If the query optimizer's decision about whether to break up a

System	Pipelining	Execution	Year
System R [108]	pull	interpretation	1974
PushPull [120]	push	interpretation	2001
MonetDB [19]	n/a	vectorization	1996
VectorWise [17]	pull	vectorization	2005
Virtuoso [18]	push	vectorization	2013
Hique [87]	n/a	compilation	2010
HyPer [121]	push	compilation	2011
Hekaton [48]	pull	compilation	2014

Table 7: Query Processing Models – *and pioneering systems.*

pipeline is correct (which is non-trivial [101]), Peloton can be faster than both standard models.

The two previous approaches use vectorization to improve an engine that is principally based on compilation. Conversely, compilation can also improve the performance of vectorized systems. Sompolski et al. [171], for example, observed that it is sometimes beneficial to fuse the loops of two (or more) VectorWise-style primitives into a single loop—saving materialization steps. This fusion step would require JIT compilation and result in a hybrid approach, thus moving it towards compilation-based systems in Figure 13. However, to the best of our knowledge, this idea has not (yet) been integrated into any system.

Tupleware is a data management system focused on UDFs, specifically a hybrid between data-centric and vectorized execution, and uses a cost model and UDF-analysis techniques to choose the execution method best suited to the task [32].

Apache Impala uses a form of compiled execution, which, in a different way, is also a hybrid with vectorized execution [186]. Rather than fusing relational operators together, they are kept apart, and interface with each other using vectors representing batches of tuples. The Impala query operators are C++ templates, parameterized by tuple-specific functions (data movement, record access, comparison expressions) in, for example, a join. Impala has default slow ADT implementations for these functions. During compilation, the generic ADT function calls are replaced with generated LLVM IR. The advantages of this approach is that (unit) testing, debugging and profiling can be integrated easily—whereas the disadvantage is that by lack of fusing operators into pipelines makes the Impala code less efficient.

2.8.2 Other Query Processing Models

Vectorization and data-centric compilation are the two state-of-the-art query processing paradigms used by most modern systems, and have largely superseded the traditional pull-based iterator model, which effectively is an interpreter. Nevertheless, there

are also other (more or less common) models. Before discussing the strengths and weaknesses of these alternative approaches, we taxonomize them in Table 7. We classify query processing paradigms regarding (1) how/whether pipelining is implemented, and (2) how execution is performed. Pipelining can be implemented either using the pull (*next*) interface, the push (*produce/consume*) interface, or not at all (i.e., full materialization after each operator). Orthogonally to the pipelining dimension, we use the execution method (interpreted, vectorized, or compilation-based) as the second classification criterion. Thus, in total there are 9 configurations, and, as Table 7 shows, 8 of these have actually been used/proposed.

Since System R, most database systems avoided materializing intermediate results using pull-based iteration. The push model became prominent as a model for compilation, but has also been used in vectorized and interpreted engines. One advantage of the push model is that it enables DAG-structured query plans (as opposed to trees), i.e., an operator may push its output to more than one consumer [120]. Push-based execution also has advantages in distributed query processing with Exchange operators, which is one of the reasons it has been adopted by Virtuoso [18]. One downside of the push model is that it is slightly less flexible in terms of control flow: A merge-sort, for example, has to fully materialize one input relation. Some systems, mostly notably MonetDB [20], do not implement pipelining at all—and fully materialize intermediate results. This simplifies the implementation, but comes at the price of increased main memory bandwidth consumption.

In the last decade, compilation emerged as a viable alternative to interpretation and vectorization. As Table 7 shows, although compilation can be combined with all 3 pipelining approaches, the push model is most widespread as it tends to result in more efficient code [177]. One exception is Hekaton [48], which uses pull-based compilation. An advantage of pull-based compilation is that it automatically avoids exponential code size growth for operators that call consume more than once. With push-based compilation, an operator like full outer join that produces result tuples from two different places in the source code, must avoid inlining the consumer code twice by moving it into a separate function that is called twice.

2.9 CONCLUSIONS

To our surprise, the performance of vectorized and data-centric compiled query execution is quite similar in OLAP workloads. In the following, we summarize some of our main findings:

< *Computation:* Data-centric compiled code is better at computationally-intensive queries, as it is able to keep data in registers and thus needs to execute fewer instructions.

> *Parallel data access:* Vectorized execution is slightly better in generating parallel cache misses, and thus has some advantage in memory-bound queries that access large hash-tables for aggregation or join.

= *SIMD* has lately been one of the prime mechanisms employed by hardware architects to increase CPU performance. In theory, vectorized query execution is in a better position to profit from that trend. In practice, we find that the benefits are small as most operations are dominated by memory access cost.

= *Parallelization:* With find that with morsel-driven parallelism both vectorized and compilation based-engines can scale very well on multi-core CPUs.

= *Hardware platforms:* We performed all experiments on Intel Skylake, Intel Knights Landing, and AMD Ryzen. The effects listed above occur on all of the machines and neither vectorization nor data-centric compilation dominates on any hardware platform.

Besides OLAP performance, other factors also play an important role. Compilation-based engines have advantages in

< *OLTP* as they can create fast stored procedures and

< *language support* as they can seamlessly integrate code written in different languages.

Vectorized engines have advantages in terms of

> *compile time* as primitives are pre-compiled,

> *profiling* as runtime can be attributed to primitives,

> *debugging* as query plan interpreters provide sufficient context, and

> *adaptivity* as execution primitives can be swapped mid-flight.

3 FAST CODE GENERATION AND COMPILATION

Excerpts of this chapter have been published in [76].

Query compilation is a widely adopted approach for relational database systems [4, 37, 140, 186, 60]. Creating machine code for every query removes interpretation overhead and allows the database system to extract the highest performance from the underlying hardware. So far, high processing performance was most relevant in the field of in-memory databases [32, 33, 71, 80, 83, 87, 117, 121, 137, 136, 146]. Yet, the growing bandwidth capabilities of solid state drives and non-volatile memory (also with large bandwidth) make query compilation attractive for a growing field of hardware configurations [123, 156, 61].

Compilation works well for large analytical workloads. However, for some use-cases the extra time spent on compilation—the *latency overhead* of compilation—can be a problem. For example, interactive data exploration tools send many queries to the underlying database system; often even multiple queries for a single user interaction. Any overhead from compilation delays the query response and, especially with a large number of queries per interaction, becomes noticeable to the user and causes them to idly wait. Vogelsgesang et al. reported that for the interactive data exploration tool Tableau some queries, even after careful tuning, still take multiple seconds just in compilation step of the underlying database system Hyper [185]. The Northstar project also encountered the issue. They observed that compilation "has an up-front cost, which can quickly add up" [86] and thus severely deteriorates the interactive user experience.

This chapter presents multiple components for compiling query engines to achieve *low query latency*; that is, to minimize the total time spent for query compilation and execution. Compile time must be addressed in the whole compilation pipeline, thus we address every component (cf., Figure 14). We introduce ① Tidy Tuples, a fast code generation framework, ② Umbra IR, an efficient program representation, and ③ Flying Start, a compiler to quickly generate machine code. All components are integrated

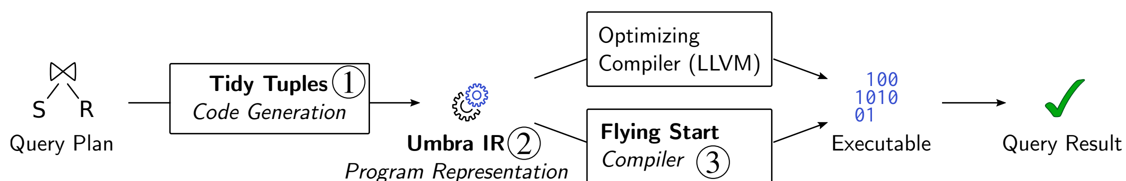


Figure 14: Umbra's low-latency path from query plan to result. – *In this chapter, we explain how Tidy Tuples, Umbra IR, and Flying Start minimize the time each query spends on this path—for short-running and long-running queries alike.*

into the database system Umbra [123] and our experiments show that together, they effectively reduce compilation time and maintain high query execution speed.

The first step toward low query latency is a fast code generator. We present ① the *Tidy Tuples* relational code generator framework. It lowers algebraic operators to Umbra IR in a single pass for low compilation time and in certain cases utilizes pre-compiled code to avoid compilation time all-together. Tidy Tuples is a *latency-streamlined design* that achieves code generation up to three orders of magnitude faster than competitors (e.g., 1000× faster than LB2 [177]) while still providing a clean, type-safe, and easy to understand interface.

The question of how to build a code generator is not yet settled [177, 165, 164], as the generator must handle the complexity of relational operators, many SQL types, NULL values, and much more. To handle complexity, a code generator should adhere to the principles of good software engineering. Tahboub et al. found an elegant way to achieve this. With the LB2 system [177] they built a well-architected query interpreter in Scala. The interpreter is based on the data-centric model, but uses callback functions to structure communication between operators. Employing callback functions is a structural advancement that provides the data-centric model with the clear structure of Volcano-style interpreters. Through extensions in the Scala compiler they are able to transform this interpreter into a code generator so that they get a system with a type safe, easy to read, well-architected code generator.

Unfortunately, the LB2 approach requires very long code generation times, which add to query latency. It fundamentally limits query execution speed to three queries per second. The authors report 299 ms for code generation geometric mean over all TPC-H queries, and that is even before the compiler started generating machine code, so the approach is not viable for low query latencies. We transfer the essence of the LB2 code generator architecture to the systems language C++ and into our Tidy Tuples design. This way, Tidy Tuples obtains a clear structure, yet achieves code generation more than 1000× faster. Additionally, we contribute abstractions on top of the code generator that decompose all issues of code generation into a layered structure.

The next component for low query latency is ② Umbra IR, a *custom intermediate program representation*. It is modelled after LLVM's intermediate representation, but its data structures are optimized for writing and reading speed. Tidy Tuples uses Umbra IR as target for the code generator and source for all compilation backends. This reduces the time to generate programs and to transform them to executables. An alternative, the commonly used intermediate representation from the LLVM compiler framework, is expressive and agile. In the compiler framework, it is used as the common format which all optimization passes edit during compilation. However, we found that its flexibility is counter-productive for query latency. Therefore, with Umbra IR, we trade off the ability to arbitrarily transform programs for optimal writing and reading speed.

Lastly, we introduce ③ the novel *Flying Start* compilation backend which transforms Umbra IR directly into machine-code. Flying Start reduces query latency in two ways: It minimizes time spent for compilation as it generates machine-code very quickly. Further, it reduces the time spent for execution as the speed of the created machine-code is

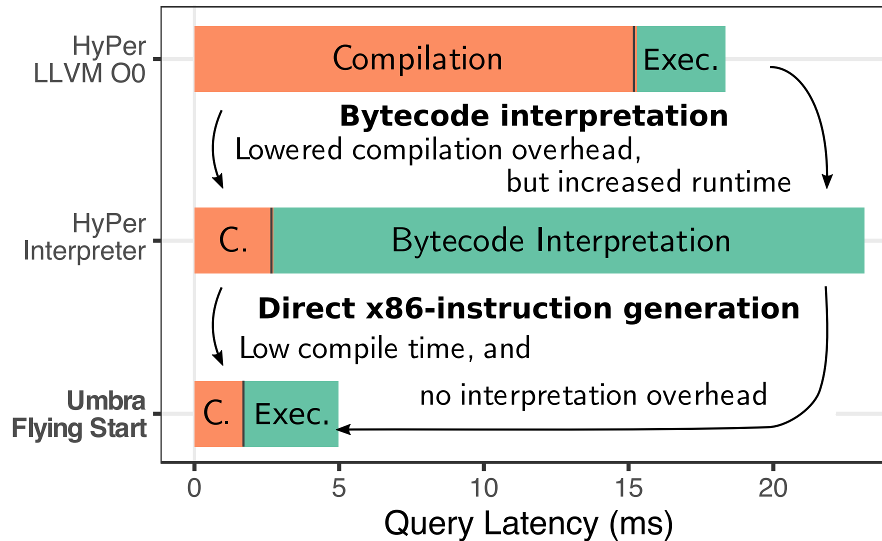


Figure 15: Best of Both Worlds – Umbra's new query engine combines fast compilation, previously reserved for bytecode interpreters, with the fast execution speed of native instructions. For example, in TPC-H query 2 the execution time compared to all other options is greatly reduced. *SF=1, Threads[1]=4*

close to that of thoroughly optimized code. The Flying Start backend is integrated into Umbra through the adaptive execution technique [84]. This allows Umbra to switch dynamically between low-latency compilation with Flying Start and highest-speed query execution by optimizing compilation with the LLVM compiler framework.

Adaptive execution was introduced first to the HyPer query engine. For query execution it has a choice between using intensively optimized code for high-speed execution and two low-latency compilation backends. For low latency, HyPer can either use a bytecode interpreter or the optimizing compiler LLVM with most optimizations turned off (turning optimizations on takes too much compilation time for short-running queries). In the example of TPC-H query 2, HyPer's low-latency choices are the top two in Figure 15. It can either prioritize fast execution, but spend more time in compilation with the LLVM backend, or use the bytecode interpreter for fast compilation at the cost of slower execution. Unfortunately, in cases like this, both options have significant shortcomings: Compilation time with LLVM is not amortized and the bytecode interpretation is so slow that it diminishes the gains from its fast compilation. Ultimately, the query engine is stuck in a performance gap between interpretation and compilation, with no great choice for low query latency.

With the Flying Start backend we show a solution for the low-latency spectrum, i.e., short-running queries. It generates code even faster than HyPer's bytecode interpreter and the resulting execution speed is on par with HyPer's LLVM-generated code. The Flying Start compilation backend, thus, is able to capture the *best of both worlds*: It combines great compilation speed with great execution speed. Effectively, it closes the

1 Umbra and HyPer use a single thread for compilation and multiple threads for query execution.

performance gap between the two execution options and therefore offers much lower query latencies than previous approaches.

Tidy Tuples, Umbra IR, and the Flying Start backend represent the foundation of our new database system *Umbra*. Together, these three components achieve query latencies for short-running queries that previously were only possible using interpretation. Overall, experimental results show that the triad is so effective at reducing latency that Umbra reaches the latency realms of interpretation-based engines like DuckDB and PostgreSQL, all while keeping the execution speed of state-of-the-art compiling systems like HyPer for long-running queries.

The chapter is organized as follows: Section 3.1 explains the code generator and the Tidy Tuples design. Section 3.2 details how our custom intermediate representation aides fast code generation. Section 3.3 outlines the Flying Start compiler and how it achieves low compilation time. The impact of these latency optimizations on the performance of Umbra is evaluated experimentally in Section 3.4. Section 3.5 discusses related work, and Section 3.6 summarizes the main results of this work.

3.1 TIDY TUPLES: A LOW-LATENCY CODE GENERATION FRAMEWORK

The initial component important for low latency is the code generator—the component that lowers relational plans to imperative programs. For maximum speed, we propose to create programs in a single pass over the input. Unfortunately, performance optimizations are often at odds with principles of good software engineering, e.g., separation of concerns, readability, extensibility, and accessibility to newcomers. This also applies to building a SQL database system. Such a system must be able to handle arbitrarily complex SQL queries, handle many SQL types, and cope with the intricacies of NULL values. These requirements are already complex, but paired with the need to optimize for speed one can quickly clash with software engineering principles.

In this section we present our Tidy Tuples design for a relational code generator. It caters to the need for speed, but also provides structure to adhere to principles of good software engineering. Tidy Tuples is a toolbox of complementary components that are organized into layers. It is a solid base to implement relational operators that are easy to read and achieve fast execution.

To introduce the architecture, we first take a short look at the life of a query within a compiling database system in the follwoing Section 3.1.1. Section 3.1.2 starts with an overview of the layers in the toolbox and their contents. In Section 3.1.3, we demonstrate the layers using a short example—peeling off abstractions step by step to provide insight into how the layers fit together. This section shows most clearly how the query plan is conceptually lowered in multiple steps. Finally, we discuss some important details, including the SQLValue abstraction in Section 3.1.4 and the low-level code generator interface in Sections 3.1.5, 3.1.6, and 3.1.7.

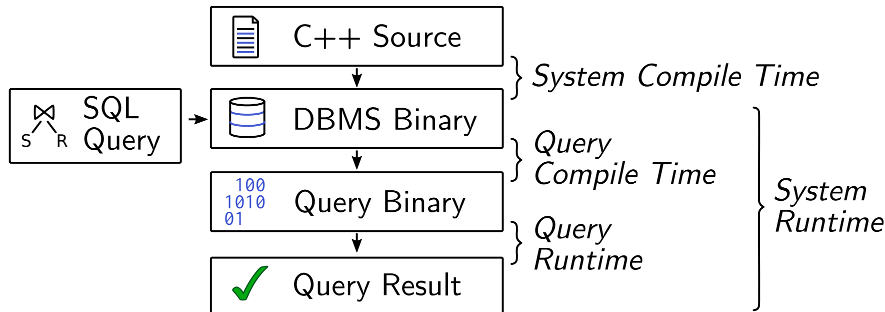


Figure 16: Compilation phases of the compiling system Umbra. – *C++ compilation happens once when the DBMS binary is assembled. Query compilation occurs for every query, thus happens many times during system runtime.*

3.1.1 Background: Compilation Pipeline

Let us first give an overview of the life of a SQL query inside a compiling DBMS, using the system Umbra as an example. A query is parsed to an abstract syntax tree, which is then semantically analyzed and translated to relational algebra. The query optimizer takes the relational algebra tree and creates an optimized physical plan. The plan describes how to process data to obtain the result. All steps described up to here are commonly found in any relational database system. Only the following steps are specific to compiling query execution engines. From the optimized physical plan, the code generator must create a program so that the execution of the program produces the query result. To see an illustration of the process, find the query plan in the top left corner of Figure 17.

Tidy Tuples translates the physical plan operator by operator. It instantiates an operator translator for each algebraic operator which is responsible for generating code that will execute its algebra operator. Conceptually, operator translators get tuples from their child operators and pass control to each other following the *produce/consume* interface [121]. During this translation, every translator appends instructions to a program. Ultimately, all operator translators together create a program that will produce the query result. Umbra represents these programs in a custom intermediate representation called Umbra IR (see Section 3.2).

There are two options for converting a program from Umbra IR into an executable: The low-latency Flying Start backend (see Section 3.3) or the LLVM-based optimizing backend [94]. Both produce machine code which computes the query result when executed. Of all the steps involved in this process, the Tidy Tuples framework focuses on translating algebraic execution plans into IR in a clear, modular, and maintainable fashion. The remainder of this section explains in detail how Tidy Tuples structures the code generator.

For the description of the Tidy Tuples framework, it is important to differentiate between the two compilation phases “system compile time” and “query compile time” (cf., Figure 16). At system compile time the source code of the DBMS is compiled into an executable (DBMS) binary. A user can start that binary to run the system. During

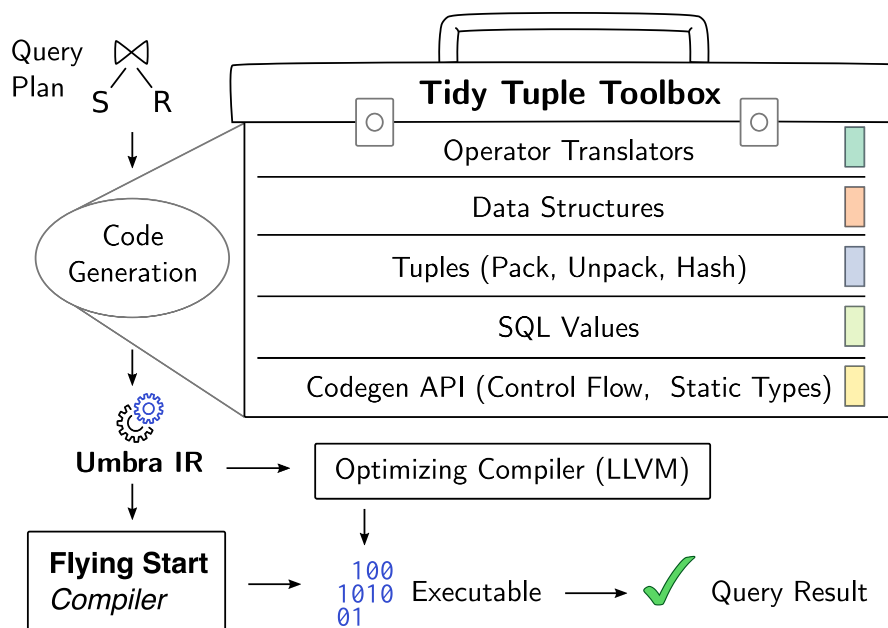


Figure 17: Architecture for a low-latency code generation engine – *In the Tidy Tuples code generation framework each layer offers abstractions to simplify the layers above.*

runtime the system accepts SQL queries, compiles a binary for each query, and runs the query-specific binary to obtain the query result. The distinction between system compile time and query compile time is relevant for the description of the Tidy Tuples compilation framework. For example, Tidy Tuples relies on the C++ type system to ensure correctness of code generated at query compile time. Naturally, correctness checks within the C++ type system already happen at system compile time (thus produce no overhead at query compile time).

3.1.2 Layer Overview

The components of Tidy Tuples are arranged into the five logical layers shown in Figure 17. Each layer acts as a level of abstraction and can use the tools of lower layers to implement its functionality so that conceptually a query plan is lowered through the layers.

- **Operator Translators:** The top-most layer contains algebra operator translators which coordinate in produce-consume style [121].
- **Data Structures:** To handle algorithmic challenges, operator translators use components from the data structure layer, e.g., hash tables (i.e., components that generate code to act on hash tables).
- **Tuples:** The tuples layer provides operations that work on multiple SQL values, e.g., packing tuples into a memory efficient format and hashing.

- **SQL Values:** Operators use the SQL value layer to implement SQL data-type specific parts in which operations on SQL types are performed. These operations include addition, substring search, equality comparison, and many more. Furthermore, the SQL value layer offers tools to operate on SQL values with standard-conform NULL-semantics.

- **Codegen API:** All these layers directly or indirectly use the Codegen layer to append instructions to the output program. Codegen offers operations on low-level types which are close to the hardware, e.g., `Int8`, `UInt64`, `Double`, `Ptr<Int8>`, and also seamlessly integrates C++ types and functions. This is exposed through a statically-typed interface, which ensures that, e.g., the result of `a:Int8 + b:Int8` is again of type `Int8`. Furthermore, the Codegen layer provides constructs to generate control flow.

Overall, these layers are structured from coarse-grained upper layers to fine-grained lower layers. The upper layers perform a lot of work for one operation, e.g., insert all tuples into a hash table, whereas lower layers perform little work for one operation. Thus, operations on lower layers must emit only very few instructions into the program. Conversely, operations in upper layers ultimately emit many instructions. However, this does not mean that the implementation of an upper layer operation must be very lengthy or emits many instructions directly. Through the Tidy Tuples layering scheme, they can use components from lower layers so that the upper layer source code is concise and the intent is expressed directly.

3.1.3 From Operators to Instructions

So far, the overview of the layers gave an abstract description of where tools belong and how they interact. To make this more tangible, let us walk through snippets of the code. The walk-through starts at the top layer, at an operator translator, and then repeatedly zooms in on one element of the implementation at-a-time to reach the next lower layer until it arrives at the Codegen layer. This should give insight into the code structure in each layer, how the layers interact, and how they generate code in a single fast pass.

The walk-through inspects the layers along the example of an in-memory hash-join. At the top-most abstraction level the hash-join operator translator must take each incoming tuple from the build side and insert it into a hash-table. The translator therefore needs to generate code to handle many issues. It must hash the keys from the tuple according to each attribute's SQL type, it needs to find the spot in the hash-table data structure where the tuple belongs, memory must be allocated for storing the tuple, and, finally values must be moved into the allocated spot. Additionally, the source code that implements all this should be well-structured, reader-friendly, and very fast at generating code.

Figure 18 shows the proposed Tidy Tuples implementation, which meets all these requirements. Observe how the hash-join translator, in order to generate code, merely

```
1  # Layer 1: Operator Translators
2  HJTranslator::HJTranslator(CompilationContext& c,
3    algebra::Join& op, Pipeline& p, algebra::IUSet& required){
4    ...
5    hashTable.buildLayout(keys, required /*the payload*/);
6  }
7
8  void HJTranslator::consume(ConsumerScope& scope) {
9    Ptr<Proxy<HashTable>> ht = ...;
10   if (scope.contains(op.left)){ // build side
11     hashTable.insertEntry(ht, scope);
12   } else ... // probe side
13 }
14
15 # Layer 2: Data Structures
16 void HashTable::insertEntry(Ptr<Proxy<HashTable>> ht,
17                             ConsumerScope& scope) {
18   // Resolve the key
19   vector<SQLValue> keys, values;
20   for (IU& k : keyIUs) keys.push_back(scope.deriveValue(k));
21   for (IU& v : payloadIUs) values.push_back(scope.deriveValue(v));
22   UInt64 h = Hash::hash(keys);
23   UInt64 size = keyStore.size(keys) + payloadStore.size(values);
24   // Insert entry for given hash h
25   Ptr<UInt8> entry = Proxy<HashTable>::insert::call(ht, h, size);
26     // Write keys and values into entry
27   keyStore.pack(entry, keys);
28   payloadStore.pack(entry + keyStore.size(), values);
29 }
```

```
30 # Layer 3: Tuples
31 void Storage::pack(
32             Ptr<UInt8> target, vector<SQLValue>& values){
33   unsigned slot = 0;
34   NullIndicator nullIndicator;
35   for (SQLValue& value : values) {//Uses Layer 4: SQLValue
36     Bool isNull = v.null();
37     nullIndicator.store(slot, isNull);
38     {
39       If nullCheck(!isNull); // If from Layer 5: Codegen
40       store(target + layout[slot++].offset, value);
41     }
42   }
43   nullIndicator.store(target + layout.nullOffset);
44 }
45
46 CGType getStorageType(SQLValue::Type t) { ... }
47 void Storage::store(Ptr<Uint8> target, SQLValue v){
48   switch(getStorageType(v.value.type())){
49     case Int64: Ptr<Int64>(target).store(v.value());
50     ...
51   }
52 }
53
54 # Layer 5: Codegen
55 void Ptr<Int64>::store(Int64& v) {
56   // Calls into Layer 6: IR
57   irProgram.createStore64Instr(v.get(), ptr);
58 }
```

Figure 18: Illustration of an in-memory inner hash join – *(Lines 1-13) using a hash-table from the Data Structures Layer (Lines 15-29) which uses the Tuples Layer (Lines 30-52). Eventually, the Tuples Layer uses the Codegen Layer (Lines 54-58) to create a store instruction.*

has to set up a hash-table[2] (Line 5) and insert a tuple (Line 11). All further details are delegated to lower layers. In the next lower layer, the data structure layer, the hash-table insert function assembles the keys and values (Lines 20-21), computes a hash (Line 22), finds the appropriate spot to insert (Line 25), and finally asks the tuple storage component to place the tuple into that spot (Lines 27-28). So, again, the layer decomposes the task and delegates to lower layers. The same mechanic repeats in the Tuples and the SQL Value layer until the Codegen layer is reached. It is the type-safe foundation on which all layers above rest.

Overall, the shown organization into layers results in well-structured source code that separates and orders many concerns. Yet, it requires only a single pass over the physical query plan to generate a program in low-level intermediate representation.

3.1.4 SQL Values

The explanation in the previous section uses the Codegen interface only for a simple store instruction (and some arithmetic). This is one of the simplest operations inside a SQL database system, but clearly, a DBMS needs to support more complex functionality than that. Strings, dates, intervals, JSON, and fixed-point numerics offer many (sometimes) complex functions that need to be integrated into generated code. One option would be to implement this functionality in the Codegen layer and provide layers above that with the complex SQL types they need to work with. However, our design aims to reduce complexity from top to bottom layers and to keep each layer simple. To keep the Codegen layer simple, it only offers *primitive types* plus the means to operate on C++ types. Therefore, we implement the rich semantics of SQL types above the Codegen layer in the SQL Value layer.

The main interface of the SQL Value layer is the `SQLValue` class. A `SQLValue` consists of a NULL indicator, the value, and a SQL type specifier (e.g., Varchar, Integer). Its general interface to invoke operations are the `evaluateBinary` and `evaluateUnary` functions which apply any of the built-in functions. In addition, functions that are frequently used by programmers are offered explicitly, e.g., equality comparison. This interface serves two purposes. First, it bridges from the realm of the (at system compile time) generically typed SQLValue, whose type is determined by the attached type specifier, into the realm of the statically-typed Codegen. Second, it provides a single place that is responsible for the intricacies of SQL values and operations. `SQLValue` handles nullability by also carrying a NULL indicator, and all operations on `SQLValues` handle NULL propagation as dictated by each specific operation. Furthermore, each operation provides overflow checking and implicit type casting if appropriate.

2 Information Unit (IU) is effectively a reference to a column [118].

Type	Available Operations
`(U)Int(8-64)`	`+ - * / % ~ & | ^ << >> ashr rotateL rotateR bswap crc32 == ...`
`Bool`	`lnot && || select == ...`
`Double`	`+ - * / % pow == ...`
`Data128`	`build extract`
`Ptr<T>`	`load store atomicLoad atomicStore atomicXchg atomicCmpXchg refMember`

Table 8: Codegen primitive types – *Type wrappers and operations on them available in code generation API*

3.1.5 Primitive Types for Code Generation

The SQL Values described in the previous section map SQL types to primitive types and construct operations on SQL types from operations on primitive types. Codegen offers a statically-typed interface to work with primitive types and other means to create programs which we show in the following sections.

Most importantly, Codegen offers classes to generate code for primitive types and uses C++ operator overloading to make it convenient to use. The types are modeled after data types that modern CPUs provide and the basic types that are available in C++. Table 8 lists those types and gives an overview of the main methods they provide to generate code.

Any of the operations on the primitive types have a statically-typed interface. For example, the result of a comparison of `Doubles` is a `Bool` and the `Ptr<Int8>.load()` returns a `Int8`. This greatly helps to reduce bugs in code generation and reduces the complexity burden on the programmer as they do not have to keep track of types while implementing algorithms. To see this static type system in action, let us have a look at the implementation of our hash function. It generates code to compute a hash of all given `SQLValues`. As it operates on multiple `SQLValues`, it belongs to the Tuples layer.

```
1  # Tuples layer
2  // Hash.cpp, Hash values
3  UInt64 Hash::hashValues(vector<SQLValue> values) {
4    ... //Concat. all low-level types to 64bit integers
5    //Hash concatenated values into two 32-bit integers
6    UInt64 hash1(6763793487589347598);
7    UInt64 hash2(4593845798347983834);
8    for (UInt64 v : concatenatedValues) {
9      hash1 = hash1.crc32(v); hash2 = hash2.crc32(v);}
10   // Combine the two 32-bit hashes into a 64-bit hash
11   UInt64 hash = hash1 ^ hash2.rotateRight(32);
12   hash *= 11400714819323198485;
13   ... // Hash the C++ types
14   return hash;
15 }
```

Observe how the primitive types from the Codegen interface are used as regular variables (e.g., Line 9 and 12), and the implementation reads as if the hash function directly acted on the values to hash them. This makes the implementation accessible to readers, yet, when executed on, e.g., two 32-bit integers, the following IR code is generated:

```
1  %1 = zext i64 %int1;          Zero extend to 64 bit
2  %2 = zext i64 %int2;
3  %3 = rotr i64 %2, 32;                  Rotate right
4  %v = or i64 %1, %3;           Combine int1 and int2
5  %5 = crc32 i64 6763793487589347598, %v; First crc32
6  %6 = crc32 i64 4593845798347983834, %v; Scnd. crc32
7  %7 = rotr i64 %6, 32;             Shift second part
8  %8 = xor i64 %5, %7;             Combine hash parts
9  %hash = mul i64 %8, 11400714819323198485; Mix parts
```

What also becomes apparent in this example is that even though the implementation of our hash function takes `SQLValues` as input the generated code is without any remainders of these abstractions. It merely consists of the necessary instructions to perform the task, which constitutes very compact code that can be translated and executed efficiently.

3.1.6 Host Language Integration

Previous sections explained how to conceptually lower high-level constructs such as relational algebra operators, data structures, and SQL types to programs in Umbra intermediate representation. This code generation process is already fast, as only a single pass over the query plan is required. It is even faster, though, not to generate code at all. Instead, in some situations, it is possible to call functions implemented in the host language, previously compiled at system compile time, without any runtime performance penalty.

To enable seamless integration between generated and precompiled code, Codegen provides a system of proxies that lets us generate operations on any C++ class. We can access data members and call member functions from generated code. Thus, for every feature to implement, the proxy system offers a choice of whether to write code that generates code or to implement the functionality in C++ and call it from generated code. The advantage of the latter option is reduced code generation time.

The use of this technique is shown, e.g., in Figure 18 Line 25. Instead of generating code to create an entry in a hash-table, manage memory allocation, etc., we call a precompiled C++ function. This reduces code generation time and removes complexity from the code generator.

The proxy system is statically typed like the rest of Codegen and therefore offers a fully typed view of C++ classes. It does not need to be created or maintained manually. We generate proxies completely automatically during C++ compile time for a predefined list of classes and functions.

The proxy system has the valuable property that it reduces query compile time by incorporating precompiled snippets, yet does not sacrifice peak execution performance.

A function call from generated code into C++ is already quite cheap, as no marshaling is required (as, e.g., would be necessary when using the JVM). It does, however, come at a slight cost at runtime because, e.g., register values must be saved, arguments transferred, and the call stack managed. To avoid this call overhead, the proxy system allows that a programmer can mark functions to be inlined. The Flying Start backend will ignore this inlining marker and only profit from lower compilation time. Our optimizing backend, which aims for peak performance, will react to the marker and inline the function at all call sites, thus removing any calling overhead. This mechanism provides an elegant way to implement functionality in C++, use it in generated code, reduce code generation and compilation time, but without any runtime overhead.

3.1.7 Control Flow

In previous sections, we showed how to lower operations from complex to primitive types in an architecture that creates clean code and a Codegen that enables fast code generation. This section shows the last missing piece: How to generate control flow in a type-safe interface directly into static single assignment (SSA) form. This form is the preferred program representation for many compilers, especially for our fast compiler Flying Start, which requires it to calculate value life spans (cf., Section 3.3.3). Generating SSA directly is important for compilation speed, as it removes the necessity to run an extra compiler pass.

The Codegen provides classes for three *control-flow constructs*: `If`, `Loop`, and `Function`. They need to handle two aspects: Basic blocks and PHI nodes. Umbra IR organizes instructions in basic blocks (see Section 3.2). During code generation, there is one current block to which all operations append.

The first aspect is that control-flow constructs need to set the current basic block, so that the following instructions are written to the right location. For example, the `If` first chooses the *then* block and when the *else* block is requested sets it accordingly. When the `If` goes out of scope, it wires all basic blocks together to produce the desired control flow.

Second, Codegen needs to produce static single assignment form. This means that there are no variables, only names for instruction results. As a substitute for multiple variable assignment, PHI nodes are used. A PHI node is an instruction at the beginning of a basic block and has multiple arguments. Depending on which basic block was executed before the PHI node's basic block, it chooses one of its arguments as its value. This is used, for example, to choose values in the presence of control-flow without using multiple variable assignments. So to produce static single assignment form, the control-flow constructs offer facilities to construct PHI nodes when needed.

```
1 # Data Structures layer
2 // Perform a probe and use callback to processhits
3 void ChainingHashTable::probe(
4                     Ptr<Proxy<HashTable>> table,
5                     vector<SQLValue> key,
```

```
6                     FunctionRef<void(...)>& callback){
7   UInt64 hash = Hash::hashValues(key);
8   auto entry = Proxy<HashTable>::lookup::call(table, hash);
9   { // <-- create nested block
10   Loop<Ptr<UInt8>> loop("chain", entry.notNull(), {{entry}});
11   // get SSA handle
12   Ptr<UInt8> iter = loop.getLoopVar<0>();
13   // Compare requested key to the one found in ht
14   vector<SQLValue> found = keyStore.unpack(iter + header);
15   ConsumerScope::testValuesEq(key, found, loop.continueBlock());
16   // Process entry
17   callback(loop.cntBlk(), loop.breakBlk(), iter);
18   loop.continueSequence(); // Go to the next entry
19   Ptr<UInt8> next = Proxy<HashTable>::next::call(table, iter);
20   loop.done(next.notNull(), {next}); // Set next to new loop var
21  } /* <-- close nested block, destruct loop */
22 }
```

The code above demonstrates how both aspects are handled during a lookup in a chaining hash table. Traversing the chain of hash-table entries is implemented with the `Loop` construct. Within a nested block we instantiate an object of class `Loop`, named `loop`. In its constructor, `Loop` creates a new basic block for the loop body and sets the new block as the current block. The constructor arguments are a name for the loop for debugging purposes, an entry criterion, and a list of variables that can be “updated” in the loop.

Loops often need to update values in every iteration, for example, they iteratively follow a pointer or increment a loop counter. In single static assignment form, however, no values can be updated. Instead, the `Loop` class internally uses PHI nodes to pass values to subsequent loop iterations and uses these to present a concept of loop variables to the user (of the `Loop` class). In the example above, the constructor argument `entry` is the initial value for the first loop variable. Inside the loop we access the first loop variable with `getLoopVar`. Behind the scenes, this constructs a PHI node which also manages updated values in later iterations. The value for the subsequent iteration is then set in `loop.done`.

Besides PHI nodes the `Loop` class creates the loop control flow. The constructor generates the loop entry along with the entry criterion, in the example, the criterion was that the `entry` is not null. The `done` function connects the last block to the loop head to form a loop under the condition provided in `loop.done`. On destruction `Loop` create a new basic block for after the loop and thus finalizes the control flow.

We observe that with the help of these control-flow constructs, the code that generates code becomes easier to write and read. Additionally, they allow to directly create static single assignment form in a type-safe manner.

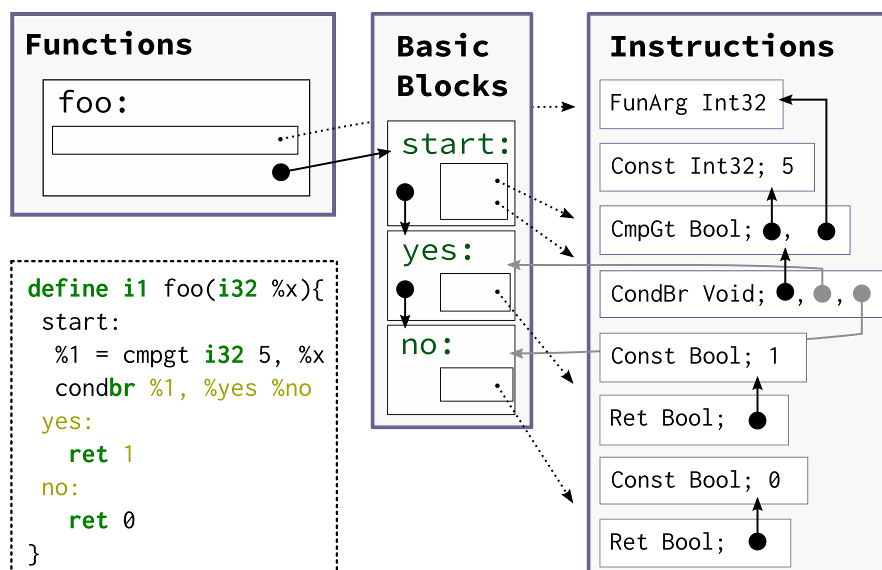


Figure 19: Internal Structure of an Umbra IR Program – *Instructions, basic blocks, and functions live in contiguous memory so that 32-bit integers suffice for addressing.*

3.2 UMBRA PROGRAM REPRESENTATION

A second element important for query latency in the compilation pipeline are the programs the code generator creates. Programs are the main artifact of the compilation pipeline, thus it is important that the code generator is able to *quickly write programs* and the backends can *quickly read* them.

To support this, we designed an intermediate (program) representation that we call Umbra IR. It serves as intermediary between code generator and compilation backends. We took special care that creating programs with Umbra IR is fast. Its data structures are carefully tuned for low memory allocation cost and compactness of representation to efficiently utilize processor caches. The reading speed of Umbra IR is optimized with a low-overhead internal reference format and database-specific instructions. Overall, we chose trade-offs towards low compile-time, yet still perform some optimizations on-the-fly when a program is created.

In the following, we present Umbra IR's internal data layout, the optimizations performed on the IR, and database specific instructions.

3.2.1 Umbra IR Structure

Before going into detail of how Umbra IR contributes to low compilation times this section gives an overview of the logical structure of IR programs. A program in Umbra IR consists of functions, basic blocks, and instructions. Functions contain basic blocks of which one is the entry point of the function—i.e., function execution starts there. The

example in Figure 19 defines the function `foo` with the basic blocks `start:`, `yes:`, and `no:`.

Basic blocks contain sequences of instructions to be executed in the given order. Each basic block must be terminated by a control flow instruction, for example a conditional branch as shown at the end of the `start:` block in Figure 19. The targets of branches are again basic blocks, so the control flow during execution of a program is determined by control flow instructions and the basic blocks they point to.

Umbra IR offers instructions for arithmetic, loading and storing values, comparisons, casts, atomic memory operations, function calls, returns from functions, branches, conditional branches, and switch, similar to optimizing compilers, e.g., LLVM. Putting this all together in the example in Figure 19, execution would begin with the first block, compare the function argument `%x` to 5 and then either branch to block `yes:` or `no:`. From either one of these blocks, execution returns from the function.

3.2.2 Physical Program Layout

To make the creation of and analyses on Umbra IR programs fast we utilize three properties of the code generation pipeline:

- Code generation mostly appends instructions at the end of basic blocks. We do not move instructions.
- Code generation has high locality. We generally first complete one basic block/-function before moving to the next.
- All instructions have the same lifetime as the program.

With the help of these properties we seek to store the program as compactly as possible to make use of caches, but still allow for quick navigation through the program. We also want to minimize the number of memory allocations. A careless implementation can cause thousands of memory allocations during program generation. Naturally, a fast implementation avoids this as allocations require time.

The first ingredient to Umbra IR's compact program representation is a variable length instruction format. All 104 instructions begin with an opcode which identifies the instruction—and determines its lengths—followed by a type identifier that specifies the result type. Each instruction then continues with its specific arguments. The example in Figure 19 shows the program's instructions on the right side. They begin with an opcode and return type followed by a variable number of arguments.

To achieve data locality while reading and writing instructions Umbra IR stores all instructions of a program in a dynamic array (as illustrated by the box around the instructions in Figure 19). This keeps instructions grouped together in memory and appending instructions does not require allocations (most of the time). It also enables us to reference instructions with a 4-Byte offset into the array. That is particularly helpful as it saves space when instructions reference each other, but still allows to follow references with low overhead.

The basic blocks of a program are similarly stored consecutively in memory. A basic block contains a dynamic array of instruction offsets which point into the instruction array and determine which instructions are in the basic block and in which order. This is depicted in Figure 19 by the dotted arrows. Storage for functions is similar. Each function, however, only contains the offset of the first basic block. From there, all other basic blocks are discoverable through the branches at the end of each block.

The shown representation is less flexible than intermediate representations used in optimizing compilers, e.g., LLVM. However, we find that it yields good cache efficiency and accelerates the generation of programs and executables from it.

3.2.3 Constants and Dead-Code Removal

The layout of Umbra IR is optimized for fast program generation and is therefore not well suited for complex restructuring passes. However, there are two important optimizations.

First, the Umbra IR builder applies constant folding to instructions at the moment they are appended to the program and deduplicates constants. This potentially decreases the programs size and reduces the workload of later stages in the compilation pipeline.

Second, a dead code elimination pass removes all instructions whose results are not used by any other instruction and any unreachable blocks. Employing an explicit dead code elimination pass gives an advantage in all layers above the Codegen layer. It removes complexity at places in the code generation where we are not completely certain that there will be a user for the value currently produced. With dead code elimination the code generator does not have to carefully determine all users beforehand which makes the generator simpler. As an example for these complexities consider how Tidy Tuples generates code for this if-then-else construct and how constant folding can help to eliminate the else branch:

```
1  Bool cond = Int32(4) * Int32(5) > Int32(15);
2  If test(cond); // Condition is constant
3    Int64 a = ...;
4  test.elseBlock(); // Else branch is dead
5    Int64 b = ...;
6  test.done();
7  Int64 result = test.phi(a,b);
```

At the time of code generation we know that the else branch will never be taken. Thus, we could try to not even generate a block for it. However, this would mean that all the instructions that would usually belong into that block, in this case the instruction that generates b, could not be placed into the program. All later sections of the code would then have to handle that any of the values may not exist. This approach would introduce additional corner-cases, be prone for errors, and code which generates code in this manner would be hard to understand. We find that a later dead code elimination pass circumvents those problems.

3.2.4 DBMS-Specific Instructions

A benefit of using a custom IR is that we can co-design the instruction set with the database system. Most importantly, instructions can express the intent of operations so execution backends can create efficient code for them. Also, instructions that occur frequently can be represented by compact, specific instructions.

Because many arithmetic operations in SQL require overflow checking, Umbra IR offers *checked arithmetic*. Check arithmetic branches when an overflow occurs or continues otherwise. For example, the following performs a 32-bit integer addition of `%a` and `%b` that branches to the basic block `%overflow` on overflow:

```
1 %c = checkedsadd i32 %a, %b %continue %overflow
```

Such specific instructions remove the need for an extra overflow check and lets backends use the expressed intent to create efficient code.

Umbra IR also combines some instructions in the fashion of inlining to obtain a more compact representation. The `getelementptr` instruction calculates addresses within arrays or structures. Load and store instructions are often combined with address calculation, therefore loads and stores can *inline address calculation*. Other instructions can also benefit from this technique. We introduced an instruction `isNull` which *checks if a value is NULL*. It does not require a second argument and thus also no constant for NULL. For the same reason we introduced instructions for CRC checksums, bit rotation, and the 128-bit data type introduced in Section 3.1.5.

Overall, the benefits of adding database-specific instructions to Umbra IR is (1) that it is easier to generate efficient code in the backends and (2) it yields a more compact program representation.

3.2.5 Comparison to LLVM IR

Compared to HyPer, which uses LLVM's intermediate representation [94], using a custom IR is a different approach. It allows to specifically tune the data layout for low-latency execution and add instructions that more closely express the intent. In terms of semantics, Umbra IR is closely related to LLVM IR. However, LLVM IR is designed to be more generic to support a wide variety of optimization passes. Therefore, LLVM IR has an emphasis on instruction reordering, replacement, and deletion. We observed that this generality entails a performance penalty which we circumvent with Umbra IR.

3.3 FLYING START COMPILER

The goal of the Flying Start compilation backend is to *reduce query latency*, that is, the sum of compile time and query runtime. Flying Start is the *default backend*[3] *for*

3 Umbra's compilation backends all use the same Umbra IR program. Thus adding a new backend does not add complexity to layers above.

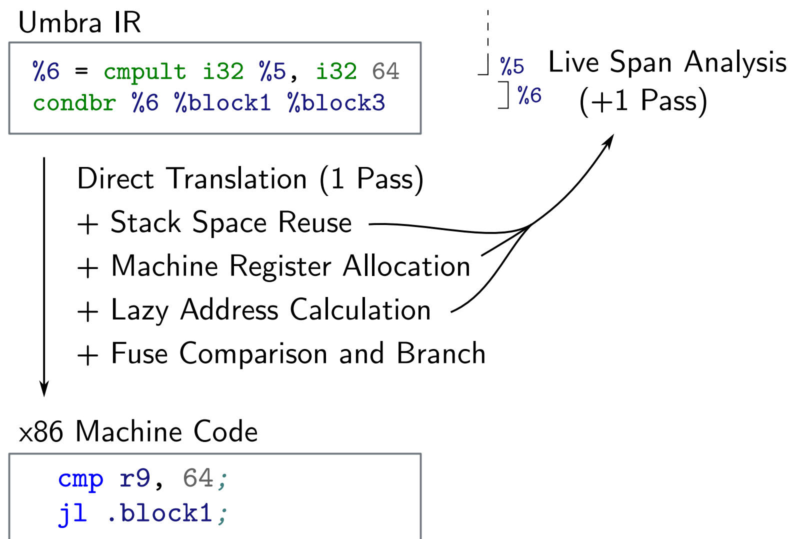


Figure 20: Flying Start optimizations – *are integrated into a single pass over the input program. Allocation optimizations require one preliminary pass to determine value live spans, thus at most two passes are required for translation.*

adaptive execution, therefore compile time should be as low as possible. At the same time, as a secondary goal, it should create fast code, so it achieves the best combination of compile time and runtime.

The best way to make code run fast and remove any interpretation overhead is to *directly generate machine code* (as opposed to bytecode for an interpreter). However, generating optimal machine code can be very time consuming. Our approach is to start out from the most basic machine code generator possible. It maps each Umbra IR instruction to exactly one sequence of x86-instructions. There are no choices or optimizations involved, so this is the fastest way to generate machine code from Umbra IR.

Obviously, the resulting code is completely unoptimized and that impedes the secondary goal, fast query execution. To investigate cheap optimization opportunities, we propose the *optimizations* in Figure 20, which are applied on-the-fly while generating machine code (denoted with "+"). These optimizations explore the design space in the vicinity of the fastest compile time and create different compile-time vs. run-time trade-offs.

The next section gives a short introduction of the adaptive execution technique and details how Flying Start fits into the compilation pipeline. Subsequent sections show the basic translator design and introduce the proposed optimizations step by step.

3.3.1 Background: Adaptive Execution

There are multiple ways to execute an intermediate representation. All have different trade-offs in code generation time and execution time. Generally, interpreters need little

preparation time but execute slower, while optimizing compilers produce fast code, but are slow to generate the code.

Kohn et al. created the adaptive execution method which incorporates multiple execution backends into the HyPer database system [84]. Adaptive execution switches dynamically between execution backends at runtime—even half-way through a query—in order to profit from fast compilation for short-running queries and from fast execution for long-running queries. Figure 15 exemplarily shows two execution backends of HyPer with the trade-offs intrinsic to each backend. HyPer's bytecode interpreter has slow execution, but compilation does not take long. The LLVM backend (with most optimizations turned off) needs some time for code generation, but execution is faster. Additionally, HyPer can employ LLVM with enabled optimizations to generate even faster code (cf. Figure 27).

Umbra also applies the adaptive execution approach. It has an execution backend that uses the LLVM optimizing compiler to produce fast executables. Additionally, we introduce the Flying Start backend for fast compilation.

3.3.2 Minimal Compile-Time Design

Algorithm 1: Basic translation of an add instruction

```
1  Function compile(Program p) is
2      for Function f ∈ p; Block b ∈ f; Instruction i ∈ b do
3          translate(i);
4  Function translate(AddInstruction i) is
5      scratch ← allocScratchRegister();
6      firstArgSlot ← i.firstArg();
7      secondArgSlot ← i.secondArg();
8      result ← allocStackSlotFor(i);
9      emit "copy firstArgSlot into scratch register";
10     emit "add secondArgSlot onto scratch register";
11     emit "copy scratch register value to result";
12     free(scratch)
```

The most basic variant of Flying Start uses a *single pass* over all instructions to generate machine code. Algorithm 1 shows how a program can be compiled with this approach. Each instruction is translated by calling the translator function for its type. Algorithm 1 also shows exemplarily how to translate an Umbra IR add instruction (for an introduction to Umbra IR see Section 3.2). The translator for add emits a *sequence of instructions* that load the inputs from stack, perform the addition, and store the outputs.

To show what this means concretely, let us consider the example Umbra IR snippet of Figure 21. The compile function emits code for instruction after instruction. Eventually, it calls the translate function for the add instruction in Line 4:

```
1 %3 = add i32 %1, i32 %2
```

Obviously, the translate function in Algorithm 1 is only a sketch. To emit concrete machine code for the Umbra IR `add` instruction we must *choose actual machine instructions*. The x86 machine-instruction we want to use for the addition operation is the `add a, b` instruction. It computes the sum of a and b and stores the result in a. This means the instruction overrides the first input operand.

The translate function must take this peculiarity into account. To keep the first operand value available after the add instruction, it must first copy the value to a scratch register and use the copy as first operand. So, to prepare the translation, it first reserves a scratch register and also collects *bookkeeping information* about where the input data resides on the stack and where the result must be stored. Second, it *emits instructions* to copy the first operand from the stack into the scratch register. Then, to perform the addition, and to copy the result onto the stack. This emits the following machine code to perform addition:

```
mov eax, [rsp+firstArgSlot];   Copy arg. into scratch
add eax, [rsp+secondArgSlot];   Add arg. onto scratch
mov [rsp+result], eax;           Copy result on stack
```

Implementing such translators for all Umbra IR instructions yields a program compiler that only requires a single pass over the IR to lower it to machine instructions. The approach has the *lowest compile-time* because it performs the least work possible to translate each instruction. However, it has some drawbacks: All values are stored on the stack, which causes extra memory traffic, it introduces many superfluous copies, and uses more space on the stack than necessary.

In the following, we devise optimizations to address these issues. They allow us to quantify the trade-offs towards better code quality at the expense of longer compile time.

3.3.3 Stack Space Reuse

The first optimization that improves the created machine code is using stack space more efficiently. Flying Start can reuse a stack slot once it knows that the value occupying the slot is never used again. To obtain this information, we arrange the program's basic blocks in reverse post-order and calculate the *live spans* of all values with the linear time algorithm described by Kohn et al. [84, 122]. The live span of a value is the interval from first to last point in the program where the value is live (similar to live intervals of Poletto and Sarkar [148]). Compared to detailed liveness information, e.g., computed by data-flow analysis, live spans are only an approximation. However, live spans can be calculated in linear time and require little memory space per value, which makes them ideal for fast compilation.

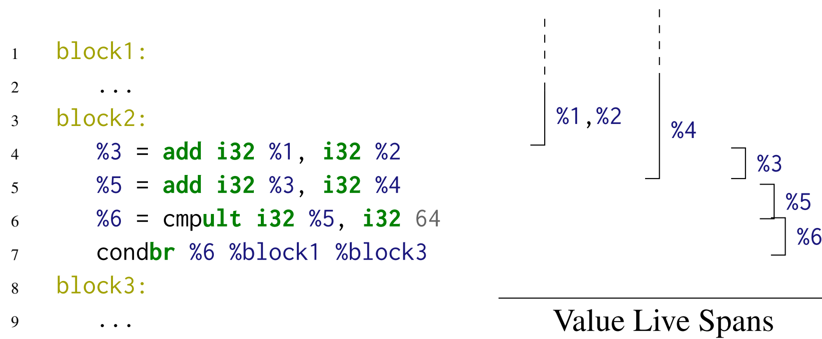


Figure 21: Umbra IR snippet — as running example for Flying Start translation.

Flying Start uses live-span information during compilation to *reuse stack slots* whose values are not used again. For example in Figure 21, the annotated value live spans on the right show that after the addition in Line 4 the stack slot of argument `%1` will not be used again. The computation result `%3` can reuse that slot. In the next line, `%5` can again reuse the slot and so on.

Generally, reusing stack slots can be cheaply implemented by checking after every translation of an instruction whether the arguments have reached the end of their live span. If they have, we return their stack slots to the stack space allocator[4].

Stack space allocation introduces additional translation cost as it requires a pass over the program to determine value live-spans. On the plus side, however, it decreases the memory footprint of the resulting code and thus increases the cache friendliness. More importantly, the analysis enables the next (very profitable) optimization.

3.3.4 Machine Register Allocation

A major issue still is that the compiler generates many *superfluous* `mov` *instructions* to retrieve values from stack and put back results. This behavior is especially unnecessary for values that are passed between instructions within one block, e.g., for the two consecutive additions of Figure 21. For these, the approach of keeping all values on stack generates six instructions of which the majority is unnecessary data movement:

```
mov rax, [rsp+slot1];            Data movement
add rax, [rsp+slot2];         Actual operation
mov [rsp+slot1], rax;            Data movement
mov rax, [rsp+slot1];            Data movement
add rax, [rsp+slot3];         Actual operation
mov [rsp+slot1], rax;            Data movement
```

A way to eliminate data movement to and from stack is to *keep values in machine registers* beyond the boundaries of the translation of a single instruction. Therefore, in

4 To be exact: In some cases we need to keep the value in the stack slot until the end of the loop in which the value is defined.

addition to assigning each value a slot on the stack, we also try to assign a machine register. This reduces the need for data movement instructions as values reside in registers. The two additions, for example, can then be implemented with only two instructions:

```
add r9, r10;              Add second arg. onto first
add r9, r11;         Add third arg. onto prev. result
```

Of course, this example shows the ideal case. In reality, there are much fewer registers available on x86 machines than there are usually values in our Umbra IR programs. In order to make best use of the available registers we adopt a best effort approach. Out of the available 16 registers on the target machine, four registers are scratch registers, one register contains the stack pointer, and the remaining 11 registers store values beyond the translation of a single IR instruction.

One could try to assign these 11 machine registers to values on a first-come first-served basis. Unfortunately, this strategy leads to a shortage of available machine registers for short-lived values and especially inside nested loops. We found it to be more beneficial to assign machine registers to values that either only live within the block they were created in or were created in the most deeply nested loop. This *heuristic* is cheap to compute from the data already at hand and effectively shifts the register usage to the passing of intermediate data and into loops. Consequently, it reduces the number of generated instructions and memory accesses.

3.3.5 Lazy Address Calculation

Besides register allocation, there are two additional minor optimizations. The first concerns the address calculation instruction `getelementptr`. Generated code frequently accesses different elements of one tuple or data structure. For data access Umbra IR programs use pointer arithmetic with the `getelementptr` instruction to compute data locations. This often leads to a chain of multiple address calculation instructions. To extend the running example of Figure 21, `block1` obtains the input data for the example with these address calculations and load instructions:

```
1 block1:
2   %tuple = getelementptr i32 %base, %tid;
3   %ptr1 = getelementptr i32 %tuple, i32 8;
4   %ptr2 = getelementptr i32 %tuple, i32 24;
5   %1 = load i32 %ptr1;
6   %2 = load i32 %ptr2;
7   ...
```

The program first computes a pointer to a tuple, then computes the pointer to the first and second element with separate `getelementptr` instructions. If the compiler would emit machine instructions for each address calculation instruction separately, it would produce an extra `add` instruction and use an extra register. However, the x86 instruction set offers an alternative as it allows to *integrate address calculation* into instruction

operands[5]. To implement the load in Line 5 the compiler can use the `mov` instruction with one register and one memory operand:

```
mov r9, [rdx + offset]
```

This form of integrated addressing can be achieved by delaying address calculation. When translating pointer arithmetic instructions the compiler does not fully resolve them to yield a single pointer value. Instead it keeps the form `[base + offset]` (where `base` is a register[6] and `offset` a constant). This enables the translator to use the composite form in instruction operands.

3.3.6 Fuse Comparison and Branch

The second minor optimization concerns comparisons and branches. Comparisons in Umbra IR result in a Boolean value which conditional branches take as input. This is an elegant construct, but unfortunately, it does not map directly to any machine instructions. In x86, a comparison sets a special flags register and branches take the flags as input. Translating comparison and branch instruction separately would produce extra machine instructions. The compiler would have to retrieve the comparison result from the flags register, only to move it right back into the flags register when translating the next instruction:

```
cmp r9, 64;                                           compare
setlt r13b;  retrieve cmp. result from flags register
cmp r13b, 1;           put decision into flags register
jnz .block1;           branch, depending on flags value
```

To avoid this situation, we must achieve during translation that the comparison and the conditional branch are translated adjacently. Also, during the translation of the comparison the compiler must decide whether to leave the result only in the flags register.

An extra reordering and analysis pass over the input program could enforce adjacency, but the extra pass would come at the expense of compilation time. Instead, within a basic block we defer translation of all, but load, store and control-flow instructions, e.g., we defer comparisons. Instructions are translated at the latest possible time, that is when their results are required by other instructions.

For the above example, the lazy approach first skips the translation of the comparison instruction. On translation of the branch the compiler notices that the input is not yet computed. At this point, it starts translating the input and also passes along the request to put the result into the flags register. Then, in the translation of the comparison it sees the request, checks if there is only one consumer, and puts the result into the flags register. The branch instruction can then directly use the flags register:

5 Similar to address inlining in Umbra IR (Section 3.2.4). Unfortunately, we can not rely on that, because addresses with multiple users can not be inlined.

6 The lifetime of the base register must be extended to cover later uses.

```
1  void translateAddInstruction(IRValue v) {
2    AddInstruction* i = get<BinaryInstruction>(v);
3    // Collect book-keeping info
4    Reg value1 = argumentReg(i->arg[0]);//  First arg.
5    Reg value2 = argumentReg(i->arg[1]);// Second arg.
6    Reg result = resultReg(v);//           Result info
7    // Prepare inputs
8    ScratchReg scratch1(*this);// Acquire scratch reg
9    Operand arg1 = get(value1, scratch1);// To scratch
10   Operand arg2 = get(value2);//  Get as mem. operand
11   // Emit main instruction
12   assembler.emit(X86Inst::Add, arg1, arg2);
13   put(result, arg1);//  Move result to assigned spot
14   //   Destruct Regs and ScratchReg. Yield resources
15 }
```

Figure 22: Foundation of the Flying Start Backend – *This is the core of translation from Umbra IR to x86. The classes* `Reg` *and* `ScratchReg` *perform book-keeping of values and free registers. They determine where inputs are located, where results should be placed, and which temporary registers to use.*

```
cmp r9, 64;                                    compare
jl .block1;          branch, depending on flags value
```

On-demand instruction translation can also pass requests from value users to producers and in this case also guarantees that there is no user of the flags register in between comparison and branch.

3.3.7 Implementation of Flying Start

So far, Algorithm 1 presented the code emitter in a fairly abstract fashion. Our actual implementation in C++, is very similar. Figure 22 shows an implementation of the translation of the Umbra IR addition instruction.

We use the classes `Reg` and `ScratchReg` to keep track of input and result data (Lines 4-6), and to allocate scratch registers (Line 8). To emit machine instructions our implementation uses the asmJIT library [81]. It provides the ability to directly assemble x86 instructions. E.g., in Line 12 the translator emits the add instruction from the running example into a buffer. Appending multiple instructions to this buffer forms the translated program.

All the described optimizations fit very well into this code structure. For example, the register allocation heuristic is hidden in the bookkeeping class `Reg`. Lazy address calculation and fusing comparisons and branches require only small additions.

Figure 23 shows an implementation of the book-keeping functions for instruction translation. Observe how the function `resultReg` decides right at the moment of in-

```
1 Reg resultReg(IRValue v) {// get location for result
2   if (notConst(v)) {
3     if (registersAvailable() &&
4         (onlyLiveInCurrentBlock(v) ||
5          loopIsDeepestNest())) {      // heuristic
6       Location& l = allocateRegister(v);
7       return Reg(RegisterVariable, v, l, this);
8     } else {
9       Location& l = allocateStackSlot(v);
10      return Reg(StackVariable, v, l, this);
11    }
12  } // else ...
13 }
14 Reg argumentReg(IRValue v, LocationHint h = None) {
15  if (notConst(v)) {
16    Location& l = lookupValueLocation(v);
17    // On-demand instruction translation
18    // with hint where to place result,
19    // e.g., in flags register
20    if (!l.assigned) translate(v, h);
21    return Reg(l.type, v, l, this);
22  } // else ...
23 }
24 Reg::~Reg() { // Destructor of Reg
25  if (--location.references == 0)
26    // return register or stack slot to allocator
27    freeResources();
28 }
```

Figure 23: On-the-fly optimizations in Flying Start – *integrate with the book-keeping infrastructure. Register allocation takes place during value placement (*`resultReg`*). Branches and comparisons are fused with hints in deferred instruction translation (*`argumentReg`*). Freeing of resources is managed in the destructor of the book-keeping class* ~Reg.

struction translation where to place computation results. Either the allocation heuristic decides to put the value into a machine register or the result is placed on the stack. Similarly, fusion of comparisons and branches is handled behind the scenes. The function `argumentReg` also handles deferred translation of instructions. When the result of an instruction `%b` is required, e.g., during instruction translation of `%a = add(%b, %c)`, function `argumentReg` checks if `%b` is already computed. If not, the instruction is translated on-demand. At this point, the caller of `argumentReg` can pass a placement hint for the value. E.g., a branch instruction can instruct a compare instruction to place its result in the flags register, skipping a placement on the stack or in another machine register.

Our implementation of Flying Start targets the widely used x86 instruction set. During translation Umbra IR instructions are compiled into semantically equivalent x86 instructions. For other target architectures, e.g., ARM processors, a target specific implementation is necessary. Specifically, the individual translation of Umbra IR instructions to the target instruction set must be adapted. Fortunately, a lot of the infrastructure for translation, such as live span analysis, register allocation, book keeping, and scratch register handling can be reused.

3.4 EVALUATION

This section evaluates the quantifiable properties of Tidy Tuples and Flying Start, confirming these performance hypotheses:

- The design achieves very low overall query latency over all database sizes and across multiple machine configurations (Section 3.4.2).
- Umbra IR speeds up code generation (Section 3.4.3).
- The Flying Start backend dominates multiple state-of-the-art alternatives (Section 3.4.4).
- The optimizations in the Flying Start backend all provide performance benefits (Section 3.4.5).

3.4.1 Experimental Setup

All experiments were run on a machine with a 10-core Intel Skylake X i9-7900X clocked at 3.4 GHz and a turbo boost of 4.5 GHz. The processor provides 20 hyperthreads, an L1-cache of 32 kB for every core and a last-level cache of 14 MB. The machine has 128 GB of DRAM with an aggregate bandwidth of 56 GB/s and uses Ubuntu 19.04 with kernel 5.0.0 as operating system. The TPC-H benchmark serves as workload with scale factors from 0.001 with 10 thousand tuples to scale factor 30 with about 260 million tuples. PostgreSQL was installed with version 11.7 and configured to use up to 20 workers per query. Further, index and bitmap scans are disabled to obtain query

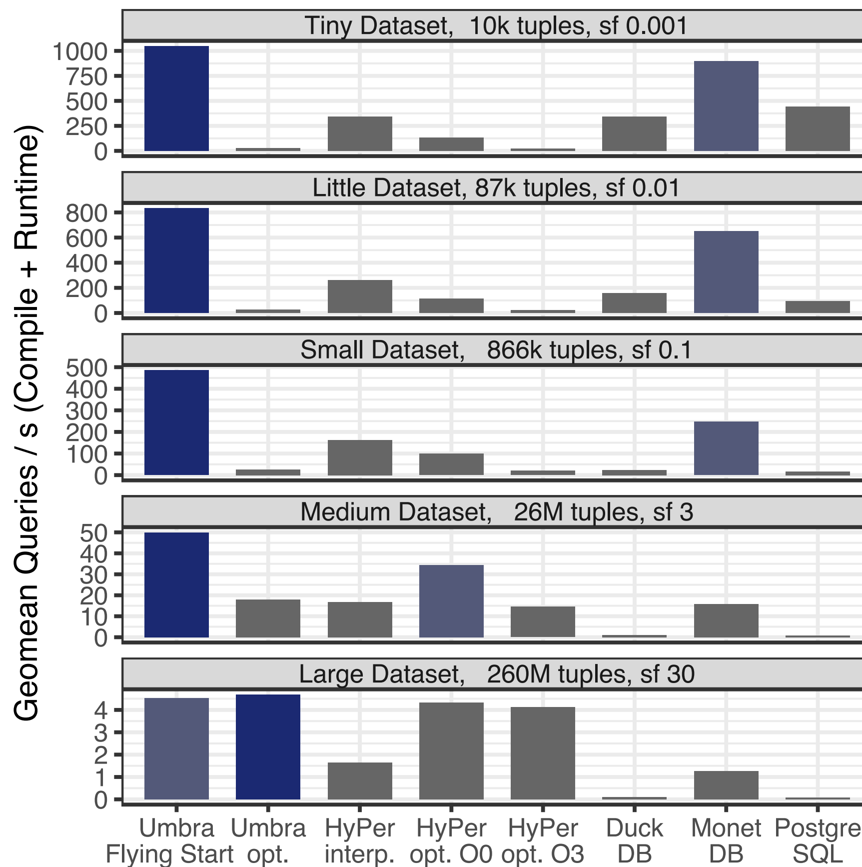


Figure 24: Flying Start achieves low query latency over a wide range from tiny to large datasets. – *Over geometric mean of queries per second over all 22 TPC-H queries Flying Start out-performs even DuckDB, MonetDB, and PostgreSQL, which do not spend any time on code generation and compilation. Threads[7]=20*

plans comparable to Umbra. DuckDB was compiled from commit `aec86f6`; MonetDB was installed in version 11.33.11.

3.4.2 Query Latency: Compile Time + Runtime

The goal of Tidy Tuples and Flying Start is to minimize the query latency of compiling query engines. That is, minimizing compilation overhead while at the same time processing queries as fast as possible. This section we evaluates to what extent that goal is achieved.

Compilation time can be traded for execution time, to a certain degree. Tidy Tuples and Flying Start constitute a specific design point in that trade-off. Whether a chosen trade-off is beneficial depends on the *ratio* of compilation time and execution time within a query. For a given system, compilation time is directly determined by the query.

7 Currently, DuckDB can only use one thread for execution. Nevertheless, it provides an interesting comparison on small datasets.

Execution time depends on the data set size and the amount of resources/threads used for processing. To evaluate the trade-off in a variety of scenarios we use the TPC-H benchmark[8]. It provides representative OLAP queries that cover a range of compilation time characteristics. To influence the execution time we vary the data set size and number of available threads. In combination, these factors cover many scenarios to evaluate the compiletime-runtime trade-off.

The experiments use the database system Umbra, in which we implemented Tidy Tuples and Flying Start. The following state-of-the-art systems serve as a reference to put Umbra's performance into perspective. HyPer serves as a representative of compiling systems. It already uses multiple execution backends to achieve low latency, which makes it a strong competitor. The experiments use PostgreSQL as an instance of classical tuple-at-a-time interpreters with no compilation overhead[9]. Modern interpreter-based based engines are represented by MonetDB and DuckDB, which are built with high-performance vectorized execution engines [19, 153].

The impact of data set size (on the trade-off) is shown in Figure 24. The experiments show that Umbra with Flying Start provides high query throughput over a wide range of data set sizes—consistently out-performing the competitors. Thus, Tidy Tuples and Flying Start effectively minimize query latency. The y-axis of the plot shows the geometric mean of query throughput over all 22 TPC-H queries, a metric for how many queries each system can execute per second. The measured time includes compilation time and execution time. Compilation time is a major factor especially for queries on small data sets, as shown in the top half of Figure 24. On these, there is not a lot of time spent in execution to amortize the time spent on generating code. Yet, Umbra with Flying Start answers queries on small data sets faster than the interpreter-based systems, which spend no time to generate code at all. On larger data sets execution time becomes an important factor. The quality and speed of generated code are relevant here. As shown in the bottom half of Figure 24, Flying Start produces sufficient code quality to out-perform other approaches on data sets with up to hundreds of millions of tuples. Altogether, Umbra processes queries with high speed over all data set sizes, which means the trade-off is beneficial for a large range of scenarios.

The other important factor in the trade-off is the number of threads used for execution. Figure 25 shows Umbra's execution time depending on the number of threads. As a reference point it also shows the fastest competitor, Monet DB. Note, Umbra's execution phase can make use of multiple threads and operators use morsel-driven parallelisation [97]. The compilation phase, with code generation and compilation, uses only a single thread.

In a broad view over all TPC-H queries (Geo.M.), Umbra is able to respond to queries faster than the other systems when using a single thread for execution up to using all

8 At the time of writing, Umbra does not have a high-performance transaction processing implementation. Thus, we can not yet compare on OLTP benchmarks. Umbra's relational operator implementations, however, are prepared to integrate well with transaction processing—similar to HyPer's operators. For example, Umbra does not use precomputed values or dictionary encoding for query processing.

9 We manually decorrelated queries for PostgreSQL for a fair comparison.

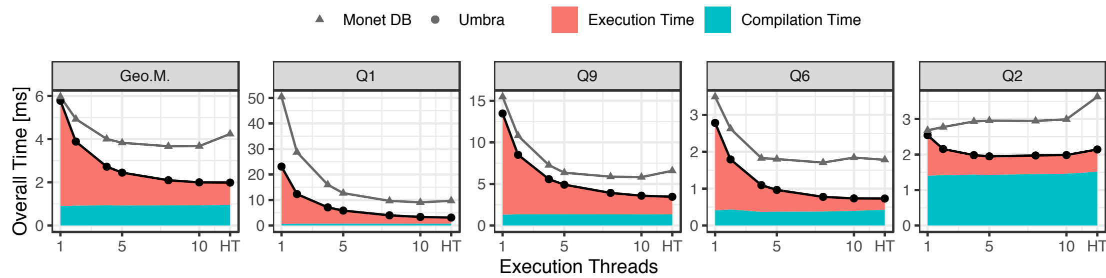


Figure 25: Umbra with Flying Start achieves low query latency across machine configurations. – *In geometric mean (Geo.M.) Umbra answers queries faster than the fastest competitor Monet DB. This is already the case when only one thread is available for query processing and holds true with additional threads. Query latency is low over the full range of long running (Q1, Q9) and short running (Q6,Q2) queries as well as queries with short (Q1,Q6) and long (Q9,Q2) compile time. SF=0.1*

available threads. Figure 25 also shows detailed performance results for queries which are chosen to cover the full range of runtime/compile time characteristics. There are long running (Q1, Q9) and short running (Q6, Q2) queries to examine the interaction of overall query runtime and number of threads. Both groups have a query with low (Q1, Q6) and high (Q9, Q2) compilation time to additionally vary the compile time/runtime ratio within each group and thus cover the whole spectrum. Notably, in call cases code generation and execution provides faster overall query response time than the fastest interpreter-based approach.

Overall, we observe that Umbra's latency optimizations work very well. They allow Umbra to reach far into the low latency realms of query engines that do not compile at all. Furthermore, note that the latency optimizations do not interfere with query execution speed. The combination of Flying Start and the optimizing compiler backend outperform the competitors in all cases. From this, we conclude that our latency optimizations are effective.

3.4.3 Compilation Time

Now that we have seen that the overall design achieves good query latency let us focus on how time is spent in the query compilation phase, i.e., before query execution.

Table 9 shows a breakdown of query processing time for Umbra with Flying Start and its competitors on the little TPC-H data set at scale factor 0.01. Umbra timing is split into the planning phase ("plan"), the code generation phase ("cdg."), machine code generation ("x86"), and query execution ("exec."). The planning phase includes query parsing, semantic analysis, and algebraic optimization. Creation of Umbra IR happens in the code generation phase and the machine code generation phase produces x86 instructions. Similarly for the competitor HyPer, yet instead of generating machine code, it produces bytecode ("bc.") for its interpreter. For the interpreting engines DuckDB,

	Umbra Flying Start					HyPer bytecode Interpreter					DuckDB			MonetDB			PostgreSQL		
#	plan	cdg.	x86	exec.	Σ	plan	cdg.	bc.	exec.	Σ	plan	exec.	Σ	plan	exec.	Σ	plan	exec.	Σ
1	0.19	0.15	0.14	1.71	**2.19**	0.09	0.44	0.30	8.16	8.99	0.17	13.66	13.82	0.60	5.35	5.94	1.14	30.01	31.15
2	0.38	0.31	0.37	0.10	**1.16**	0.53	0.91	0.76	3.06	5.26	1.14	20.69	21.83	0.67	0.51	1.17	1.52	2.28	3.80
3	0.21	0.18	0.19	0.58	**1.15**	0.25	0.57	0.50	2.43	3.75	0.32	7.38	7.70	0.51	0.76	1.27	1.64	11.61	13.25
4	0.16	0.14	0.13	0.52	0.95	0.13	0.49	0.35	4.45	5.43	0.32	14.21	14.53	0.29	0.56	**0.84**	1.18	13.78	14.96
5	0.33	0.22	0.25	0.43	**1.23**	0.49	0.79	0.67	5.40	7.36	1.90	7.17	9.07	0.77	0.88	1.65	2.47	9.63	12.09
6	0.13	0.08	0.07	0.26	**0.54**	0.09	0.29	0.14	0.12	0.65	0.15	1.96	2.11	0.27	1.00	1.27	0.99	18.90	19.89
7	0.31	0.29	0.32	0.50	**1.41**	0.46	0.75	0.62	5.16	7.00	0.80	7.00	7.81	0.55	1.48	2.03	1.33	11.21	12.54
8	0.37	0.30	0.32	0.34	**1.32**	0.63	0.80	0.64	2.94	4.99	2.56	4.82	7.39	1.12	0.84	1.96	1.77	9.71	11.48
9	0.34	0.26	0.28	1.03	**1.91**	0.47	0.73	0.62	7.86	9.68	1.52	20.48	22.00	0.85	1.78	2.63	4.18	16.48	20.66
10	0.27	0.21	0.20	0.70	**1.37**	0.35	0.66	0.47	3.46	4.94	0.55	13.36	13.91	0.74	0.74	1.48	1.99	11.51	13.49
11	0.27	0.24	0.28	0.42	1.21	0.30	0.63	0.53	3.32	4.77	0.54	1.57	2.11	0.57	0.25	**0.82**	1.41	2.47	3.87
12	0.21	0.16	0.16	0.49	**1.03**	0.18	0.56	0.43	3.87	5.04	0.21	2.81	3.01	0.90	0.53	1.43	1.21	13.61	14.82
13	0.16	0.15	0.16	0.68	1.14	0.13	0.51	0.42	6.62	7.67	0.14	5.42	5.56	0.19	0.79	**0.98**	1.25	7.08	8.33
14	0.16	0.14	0.12	0.17	**0.60**	0.16	0.43	0.28	0.28	1.15	0.19	2.02	2.21	0.53	0.41	0.94	1.21	8.93	10.15
15	0.21	0.22	0.21	0.30	**0.94**	0.16	0.54	0.39	2.83	3.92	0.30	2.65	2.95	0.63	0.36	0.99	1.37	15.51	16.88
16	0.29	0.22	0.29	1.14	1.94	0.19	0.61	0.54	4.05	5.39	0.36	1.88	2.24	0.36	0.56	**0.92**	1.56	3.75	5.31
17	0.20	0.21	0.21	0.38	**1.00**	0.28	0.65	0.61	7.09	8.63	0.48	2.67	3.15	0.23	0.87	1.11	1.26	0.32	1.58
18	0.25	0.23	0.25	1.48	2.21	0.25	0.67	0.60	13.14	14.66	0.38	10.92	11.30	0.34	1.84	**2.18**	1.80	26.87	28.68
19	0.42	0.19	0.18	0.72	**1.51**	0.25	0.56	0.35	1.80	2.96	0.33	3.95	4.28	0.93	0.80	1.73	1.51	11.46	12.97
20	0.32	0.23	0.25	0.32	**1.12**	0.36	0.65	0.52	3.09	4.63	0.79	3.15	3.94	0.75	1.05	1.80	1.15	13.69	14.83
21	0.36	0.23	0.26	1.17	**2.02**	0.50	0.71	0.61	8.96	10.78	0.90	27.10	27.99	0.70	1.61	2.31	2.56	10.50	13.06
22	0.24	0.23	0.25	0.32	1.03	0.25	0.68	0.55	3.41	4.89	0.54	5.05	5.58	0.34	0.68	**1.02**	1.23	3.80	5.03
$\mathcal{G}$	0.25	0.20	0.21	0.50	**1.24**	0.26	0.60	0.47	3.33	5.06	0.47	5.72	6.40	0.53	0.84	1.46	1.53	8.50	10.82

Table 9: Tidy Tuples, Umbra IR, and Flying Start speed up Umbra's preparation phase. Umbra is twice as fast as HyPer, thus preparation time is as low as interpreter-based systems. – *The table lists detailed timing in milliseconds for TPC-H queries 1-22 and geometric mean ($\mathcal{G}$). Planning time ("plan") includes query parsing, semantic analysis and algebraic optimization. Umbra and HyPer also list compile time, split into generation of IR ("cdg.") and generation of machine code ("x86"/"bc."). For Umbra and HyPer LLVM compilation is excluded, as its compile times are too long for a data set this small. SF=0.01, Threads=1*

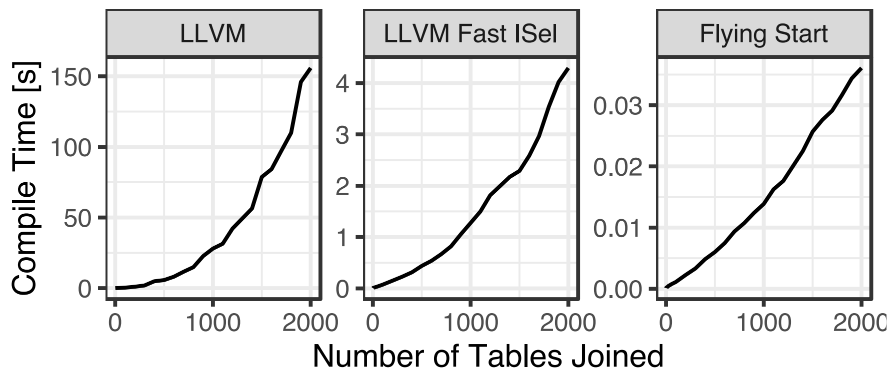


Figure 26: Flying Start compiles large queries quickly. – *LLVM needs considerably longer. Note, that the y-axis scales are orders of magnitude apart. SF=1, Threads=1*

MonetDB, and PostgreSQL the table only distinguishes between plan and execution. Finally, it also lists the time for the sum of all components ("Σ").

For Umbra, we observe that once the query plan is prepared, execution ("exec.") on the little dataset does not take long—it is even shorter than query preparation. All competitors spend more time during execution. The time Umbra spends before execution, to prepare the executable, though, is slightly more than the competitors. On average Umbra takes 0.66 ms to prepare, whereas DuckDB and MonetDB only need 0.47 ms and 0.53 ms respectively. This puts Umbra with Tidy Tuples and Flying Start well within the same order of magnitude of query preparation time as interpreter engines, even though Umbra additionally performs all the steps required for machine code generation.

Compared to HyPer, the most similar system as it also spends time on code generation, we observe that Umbra starts faster. HyPer needs 1.33 ms on average to prepare for a query and Umbra only 0.66 ms. The main difference here is that HyPer generates LLVM's intermediate representation and Umbra uses its Umbra IR representation. The effect clearly shows in the differences of code generation time ("cdg."), where Umbra is more than 2× faster than HyPer. A similar, but smaller, effect is visible during generation of the executable ("x86" and "bc."). Flying Start is faster at x86 generation than HyPer at bytecode generation. We conclude that Umbra IR speeds up code generation and thus serves its purpose well as it effectively reduces Umbra's query latency.

Up to this point, we compared compile times of Umbra with external competitors. An internal alternative to the Flying Start compiler is the LLVM compiler, which Umbra uses adaptively to get optimized code for long-running queries (cf., Section 3.3.1). Figure 26 compares the compilation times on queries with different numbers of joins. In this experiment joins the TPC-H table `nation` multiple times on itself with the predicate `n1.n_name = n2.n_name and n2.n_name = ...`. For a join query with 2000 joins Umbra generates 108000 Umbra IR instructions, of which the vast majority is in a single function. Figure 26 shows that LLVM needs a considerable amount of time to compile such large programs (150 seconds). Even without any optimizations and LLVM's fast instruction selection compilation takes 4 seconds. Flying Start, in comparison, only re-

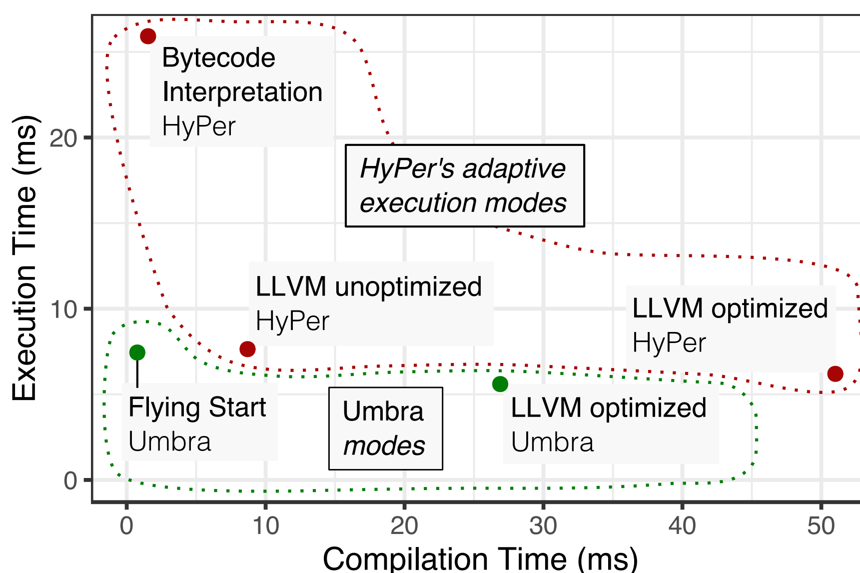


Figure 27: Umbra's vs. HyPer's execution modes. – *Comparison of time taken for compilation and achieved execution time for Umbra's and HyPer's execution modes on TPC-H query 3. SF=1, Threads=20*

quires less than 0.04 seconds to compile the program. Thus, any such query compiled with Flying Start gets a considerable head start to an LLVM-compiled query.

3.4.4 Runtime Performance Robustness

The previous section established that the compilation times of Flying Start are competitive with interpreter engines. Let us now explore the compile time versus execution speed that it offers.

Recall from Section 3.3.1 that Umbra and HyPer both use adaptive execution to run the generated code and to balance compilation time and runtime. The systems use multiple compilation backends that offer different compilation and execution speeds. HyPer switches between the three backends bytecode Interpreter, LLVM unoptimized, and LLVM optimized. Umbra only uses the Flying Start backend and LLVM for thoroughly optimizing machine code.

How all these runtime backends perform is depicted in Figure 27 for the example of TPC-H query 3 at scale factor 1. Among HyPer's execution backends, bytecode interpretation provides the lowest compilation time, albeit with a noticeable execution time penalty. HyPer's next best option is to use the LLVM compiler with almost all optimizations turned off. This yields good execution performance, but comes with a higher compile time[10]. Note, that it is already apparent, that Umbra's Flying Start backend

10 In Figure 27 Umbra's LLVM backend compiles faster than HyPer's. Umbra generates more but shorter functions than HyPer, thus reduces compile-time in LLVM optimization passes with super-linear runtime in function size. This effect does not apply to the Flying Start backend, thus the shown comparison is fair.

Backend Comparison	Compilation	Execution
Umbra		
Flying Start vs. LLVM O3	108× faster	1.2× slower
HyPer		
Interpreter vs. LLVM O3	91× faster	4.1× slower
LLVM O0 vs. LLVM O3	6× faster	1.3× slower

Table 10: The Flying Start backend out-performs both HyPer's interpreter- and unoptimized LLVM backends. – *On geometric mean over all TPC-H queries Flying Start is preferable to HyPer's options. SF=1, Threads=20*

offers a better choice. It is on par with the bytecode interpreter's compilation time and the runtime performance of LLVM (unoptimized) machine code. Hence, Flying Start combines the advantages of HyPer's two low-latency backend options into only one.

When expanding the view from this one example query to all the TPC-H queries, we see a similar picture in the trade-offs in Table 10. In comparison to fully optimizing the machine code with LLVM, on geometric mean over all queries the Flying Start backend offers 108× faster compilation at the low cost of only 1.2× slower execution. This all happens in a single compilation backend. For Umbra's competitor HyPer, this option is split in two: The Hyper interpreter backend provides 91× faster compilation at the cost of 4.1× slower execution. The alternative cheap compilation backend with LLVM offers 6× faster compilation producing code that executes 1.3× slower.

To summarize, HyPer must juggle three execution backends. As shown in Figure 27 each backend provides a different trade-off between compilation time and runtime. The results can be observed in Figure 24, where every backend yields the fastest overall execution speed over a limited range of scenarios. Thus, the system must carefully choose the correct one of three backends, as a wrong choice can gravely impede execution performance. Umbra, on the other hand, only has to choose from two backends. Flying Start combines the best of the bytecode interpreter and the unoptimized LLVM backend. It is as fast in generating code as the interpreter and as fast in execution as the unoptimized LLVM backend. Consequently, it is safe to always begin execution with Flying Start and, if necessary, shift into high gear by using the optimizing compiler. As the difference in execution speed between the backends is only 1.2×, a wrong choice only has a small impact on execution time and the performance cliff in a sense becomes a small performance step.

3.4.5 Flying Start Optimizations

We described in Section 3.3 that the Flying Start backend uses four optimizations to improve the speed of the generated code. We measured the effect of each optimization on compilation and execution time for all TPC-H queries.

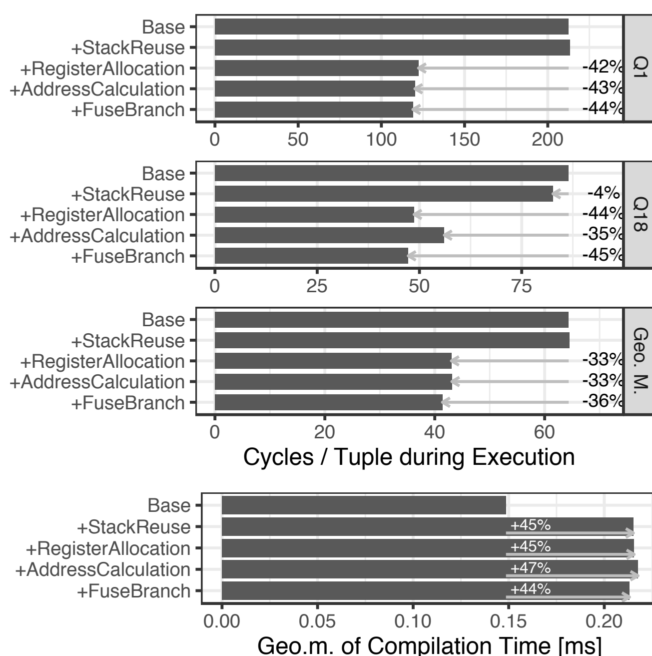


Figure 28: Effect of Optimizations on Compile- and Runtime – *in the Flying Start backend. SF=1, Threads=1*

Figure 28 shows the results for execution time on some exemplary and interesting queries and also of the geometric mean over all 22 queries. Observe that the biggest effect is achieved by register allocation. On average it provides a 32% reduction of execution time. Interestingly, in the Umbra LLVM backend, register allocation also provides the largest performance benefit among the applied optimizations (cf., Figure 29). Switching from fast instruction selection to the default instruction selection enables machine specific optimizations, such as register allocation and instruction scheduling. Further optimizations only have a small effect on the runtime. In other words, the largest optimization potential is covered by Flying Start's register allocation.

Given that register allocation has such a large impact, an interesting idea to improve Flying Start would be to use a better register allocator than the already applied heuristic. An allocation scheme often used in fast compilers is Linear Scan [148]. In a single pass over all lifetime intervals it decides which values live in registers. To compare with our allocation heuristic, we added Linear Scan to the Flying Start backend. Linear Scan produces good allocations; the machine code produced with linear scan leads to 1% faster query execution on TPC-H (cf. Figure 30). However, allocation with Linear Scan takes 14% more compilation time. This presents an interesting trade-off, yet in the interest of low compile time for now we chose not to add Linear Scan to the Flying Start default optimizations.

The experiment shows that some queries profit more from optimizations than others. Query 1 shows the largest gains, as most of its work is in expression evaluation. Thus, keeping intermediate values in registers increases the CPU's instruction throughput. Third, address calculation and comparison-branch fusion provide only a moderate

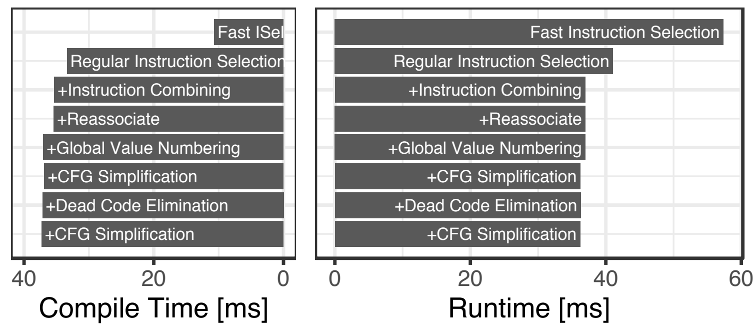


Figure 29: Effectiveness of LLVM's optimization passes – *in Umbra's LLVM backend. Geometric mean over all TPC-H queries. SF=1, Threads=1*

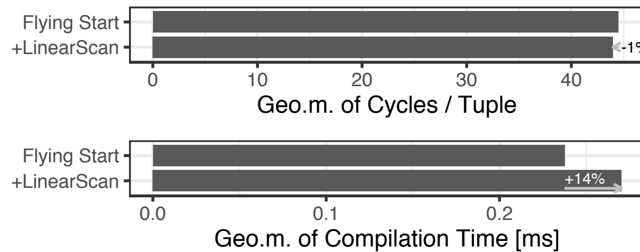


Figure 30: Additional Cost and Benefit of Linear Scan Register Allocation. – *Geometric mean over all TPC-H queries. SF=1, Threads=1*

effect. The benefit is most pronounced on query 18. Overall, we observe that every one of the optimizations increases execution speed.

The quality of the machine code generated by Flying Start is good in comparison to the fully optimized code from LLVM. As Figure 31 shows, performance metrics of Flying Start code for TPC-H queries are well within the same order of magnitude as the corresponding LLVM-generated machine code. Previous experiments already showed that the execution speed of Flying Start code is close to the speed of highly optimized code. This also shows in Figure 31, in which the amount of cycles to execute queries with Flying Start is on median about 1.6× higher than with highly optimized code. Notably though, the number of instructions executed is about 2.3× higher, which means that Flying Start produces some amount of extra instructions. Fortunately, also the number of instructions executed per cycle (IPC) is 1.4× higher. The processor is able to execute more instructions in parallel within each cycle which reduces the negative effect of extra instructions. Branch miss-predictions and last level cache (LLC) misses are about the same for both compilers. The size of the generated code from Flying Start is about 2.4× larger than optimized code. Overall, Flying Start generates some superfluous instructions, yet the hardware is able to partly compensate that. More importantly, Flying Start code triggers the same amount of hardware hazards, i.e., branch-misses and cache-misses, as optimized code, but triggers no additional hardware hazards.

Another optimization that Umbra performs is to eliminate dead (unused) code. Technically, it is an optimization applied during the code generation process, not by Flying Start, yet it effects compile- and runtime performance. Figure 32 shows TPC-H

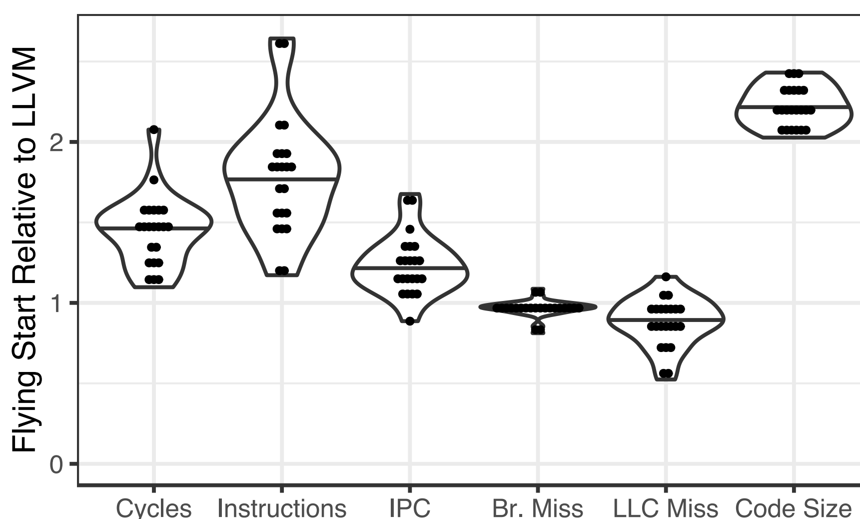


Figure 31: The performance of code generated by Flying Start is comparable to Umbra's LLVM backend. – *These violin plots show the performance of Flying Start machine code relative to LLVM-generated code on the TPC-H queries. SF=1, Threads=1*

compile- and runtime with and without dead code elimination (DCE). As explained in Section 3.2.3, Tidy Tuples uses dead code elimination to simplify structure of the code generation layer. These experiments show that dead code elimination is a rather quick pass compared to the remaining compilation time. Also, as DCE removes about 4% of instructions, it reduces the following compile- and runtime, thus recaptures some of the time spent on the optimization pass.

For all optimizations there is a compilation time price to pay, as shown in the bottom of Figure 28. We note that the only optimization that comes at a measurable cost is the value-lifetime computation which is first used for the stack reuse optimization. On average, it adds 45% to compilation time. Interestingly, all further optimizations more than offset their cost. Any one of these optimizations helps reduce the number of emitted instructions and consequently reduces the time necessary to write machine code.

Given that lifetime computation adds about 45% to compilation time one may also choose to skip it and therefore not employ any of the four optimizations. In Umbra, however, we use it, because it makes the query engine more robust. It prevents that queries with many intermediate values have an unnecessarily large memory footprint. Additionally, it reduces the performance cliff towards the optimizing compiler. To summarize, the optimization in Flying Start increase the execution speed and robustness of the query engine.

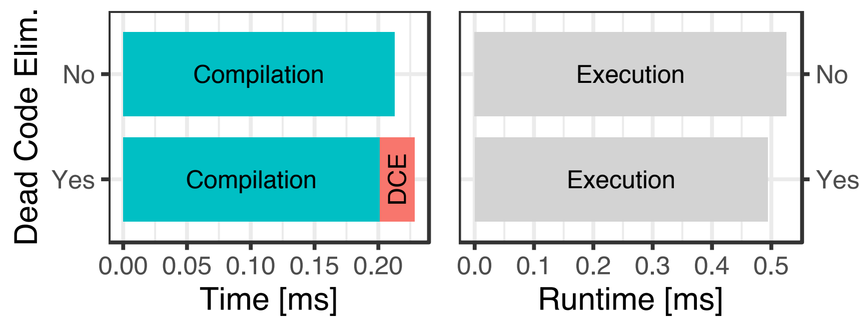


Figure 32: Effect of Dead Code Elimination on Compile- and Runtime – *with Flying Start. Geometric mean over all TPC-H queries. SF=0.01, Threads=1*

3.4.6 Implementation Effort

Building a compiling SQL query engine from scratch is a large undertaking and Tidy Tuples is meant to structure such an effort and serve as a guideline. To give an idea of the size of the code generator in Umbra, Table 11 lists the lines of C++ code required to implement components of Tidy Tuples and Flying Start. Lines with comments and documentation do not count towards the lines of code. We also exclude lines which only contain opening or closing curly braces to account for a peculiarity of the used code style.

The shown components follow the structure of Tidy Tuples as presented in Section 3.1.2. Lines of code are separately counted for C++ header files, C++ implementation files, and unit tests that directly test the functionality of the component (integration tests for the entire system are not listed). Operator translators include table scan, nested-loop join, hash join, multi-way join, group-join, group by, sort, map, select, set operations, expressions, recursive views, and many more. Each of the operators in turn may need to handle multiple variants of the operator. For example, the hash join translator can produce inner, outer, semi, mark [125], and single [125] joins. Thereof all, but the inner join have different right join and left join implementations. Overall, there are many concepts in relational queries and their efficient implementation often requires attention to detail. In our experience with the implementation of Umbra, that detail and the inherent complexity is structured well by the Tidy Tuples design.

3.5 RELATED WORK

There are two state-of-the-art query processing paradigms: vectorization and compilation. Vectorization reduces the overhead of Volcano-style interpreters by performing an operation on many tuples at the same time. It was pioneered in MonetDB [19] and improved upon by MonetDB/X100 [17]. As vectorized engines are interpreters, they can use Volcano-style interpretation and generally have a reputation of being easier to build. Furthermore, because they do not generate machine code, they can potentially

Component	Headers	C++	Tests
Operator translators	2,360	8,347	3,225
→ Hash-join translator	53	597	88
→ Map translator	17	31	55
Data structures	187	399	113
Tuples	172	1,019	2,205
→ Hash	57	320	66
SQL Values	772	6,834	2,283
Codegen	975	1,049	690
Σ Tidy Tuples	4,466	17,648	8,516
Umbra IR	812	2,348	476
Flying Start	399	3,790	1,072
Σ All	5,677	23,786	10,064

Table 11: Lines of Code of Tidy Tuples and Flying Start – *listed seperately for header files, implementation files and unit tests for the respective component. Arrows (→) denote examples from within the previous component; the component line-counts already include the example counts.*

have lower query latency—while being efficient for analytical workloads [75]. However, there are drawbacks with complicated expressions and especially when only few tuples are in a query, as is commonly the case in transaction processing.

Compilation-based engines eliminate interpretation overhead by generating query-specific machine code. An architecture for generating machine code was shown with the HyPer system [121, 124]. This approach was criticized as too low-level [80] and, in the context of LegoBase, an alternative approach was proposed. Instead of generating code from the query plan in one single step, LegoBase gradually lowers it through a cascade of intermediate representations to the effect that each lowering by itself is less complex [165]. Using multiple representations was then criticized as adding unnecessary complexity [177]. A solution was presented by using the idea of the Futamura projection to specialize an interpreter to obtain a code generator. The LB2 system uses Scala language features and compiler extensions to implement this idea and create an interpreter engine as well as a code generator, derived from the same code base. Further research on the structure of relational code generators has shown that, besides HyPer's produce-consume model, Volcano-style communication between operators can also be used for code generators. However, extensive compiler optimizations are required to obtain efficient code from code generators with Volcano-style iterators [163]. An alternative to distinguishing between interpreters and code generators is to use micro-specialization on an interpreter system [192, 193]. Kohn et al. presented the adaptive execution approach for HyPer, which combines an interpreter and a code generator to achieve low latency for cheap queries and fast execution speeds for expensive queries [84].

The work presented in this chapter builds on all of these contributions. Tidy Tuples features a layered architecture of abstractions that conceptually incorporates the gradual lowering of LegoBase, but still achieves code generation in a single step. It also uses a code generator interface, as promoted with LB2 that utilizes the host language's type system. With this code generator interface, the code that performs operator translation closely resembles an interpreter. Unlike LB2, however, we stop short of building an interpreter and always use an explicit code generator. This allows us to tightly control the optimizations that we perform at SQL compile time. An example of these optimizations was shown in the hash function generation in Section 3.1.5 and tuple storage in Section 3.1.3. Also it enables us to immediately create code in static single assignment form so that we can skip an optimization pass at a later stage. In addition, our code generator seamlessly integrates generated code with host language code—a feature that would be hard to realize efficiently between machine code and the Java VM. To achieve low query latencies we propose a lightweight compiler instead of using an interpreter. Further, we advocate to use the produce-consume model (or LB2's callback interface) for code generation to circumvent the optimization effort required to obtain efficient code from code generators with Volcano-style iteration. We show that this approach enables low query latencies that reach into the realm of interpreted and vectorized engines. In addition, it provides the benefit of removing the performance cliff between interpretation and optimizing compilers.

Compilers that are focused on minimal compilation time have been used in other areas before and our approach relies on ideas from the compiler community [147, 148, 40, 145]. Notably, destination driven code generation is an approach that generates machine code directly from the abstract syntax tree (AST) of an input language [40]. It uses one register to transfer intermediate values (in expression evaluation) between neighboring nodes in the AST and thus often achieves that values need not be transferred into memory. During AST traversal every user of a value is visited before the value is calculated and there is exactly one user for every intermediate value (due to the tree structure). The Flying Start backend builds on these ideas, but operates in a different setting. Each value in Umbra IR can have multiple users and the value lifetimes potentially span whole functions. From the view point of one instruction the inputs and their recursive inputs form a DAG instead of a tree. This removes the "one user" property for intermediate values and requires additional analysis for value lifetimes. Further, Umbra IR builds on ideas from the sea-of-nodes programs representation [31]. Umbra IR programs are structures as control-flow graphs where basic blocks are vertices and edges represent control flow. As in the sea-of-nodes representation, the arguments of Umbra IR instructions directly point their defining instructions. Unlike the sea-of-nodes representation, Umbra IR instructions stay attached to their basic blocks, as Tidy Tuples takes care to generate code that does not require a code-motion optimization.

The Chrome browser contains a WebAssembly compiler backend that is also inspired by destination driven code generation. The V8 Liftoff backend aims for low latency in code generation and creates code in only a single pass [62]. WebAssembly uses a stack machine model which takes instruction arguments from a stack and puts results

back onto the stack. This implicitly encodes the lifetime of intermediate values and Liftoff can leverage this information to manage with only a single pass. Liftoff thus depends on the compiler that generates WebAssembly to encode lifetimes. The Flying Start backend cannot do this, as its compiler is executed right ahead of it in the same compilation pipeline. Similarly, Flounder IR is a program representation that relies on the code generator to encode value lifetimes [49]. The proposed design for Flounder IR is to estimate values lifetimes with relational operator lifetimes. For Umbra IR and Flying Start, we observed for TPC-H queries that operator lifetimes overestimate the lifetimes and lead to a shortage of available registers.

LuaJIT is a fast just-in-time compiler for the dynamically typed language Lua. Execution starts with interpreting Lua bytecode [138]. A tracer then finds code sections worth compiling and creates a statically typed IR [139]. This IR, much like the Umbra IR, contains features and instructions that are very specific to Lua. A backend with multiple compiler passes can lower the IR to machine code.

Destination driven code generation, Liftoff, and LuaJIT rely on certain properties of their input programs and so does the Flying Start backend. It profits from the fact that the produce/consume interface generates efficient and short code. Values are typically loaded from memory only once and are then used in multiple places by reference to only a single Umbra IR handle. In addition, constant folding is performed on-the-fly during program generation. The Flying Start backend is tailored to these qualities and makes use of them to save compilation time.

3.6 SUMMARY

This chapter presented the Tidy Tuples architecture, the Umbra IR program representation, and the Flying Start compiler backend to minimize query latency in compiling relational database systems. They optimize the whole execution pipeline from the arrival of a query plan to when the result is ready.

The Flying Start compilation backend showed that very fast machine code generation is possible and the generated code executes queries only slightly slower than highly optimized code. Furthermore, Umbra IR, a customized intermediate representation with optimized data structures helps reduce the time spent for generating code and transferring code into machine instructions. Lastly, Tidy Tuples structure code generators so that complexity is well managed, yet code generation is very fast and thus contributes to lower query latency.

We implemented the proposed optimizations in the database system Umbra. An evaluation found that the optimizations are effective at lowering query latency. The experiments showed that Umbra's compilation latency becomes competitive with systems that do not compile at all, e.g., DuckDB and MonetDB. At the same time, the execution speed of Umbra is on par with state-of-the-art query engines.

To conclude, we advocate the use of a fast compiler that directly generates machine code and in some cases, falls back to an optimizing compiler. This approach reaches the

low-latency realms of interpreter engines and at the same time keeps a high execution speed in larger datasets. Such a query engine can compile very quickly and produces machine code that makes efficient use of the processors. It is thus well equipped to optimally use the large bandwidth provided by main memory and new storage hardware, e.g., SSDs and Persistent Memory. Its low query response time makes it predestined for a burst of many small queries intermixed with large queries—as regularly happens during interactive database use.

4 OPTIMIZING MEMORY ACCESS

Relational database systems have arrived in the age of high-bandwidth processing, where data already resides in main memory or is available from very fast storage technologies such as SSDs [61] and NVRAM [156], whose bandwidths almost match main memory speed. To make best use of the large bandwidth, analytical database query engines generate code for their query plans and rely on compiler technology to remove all overhead and to generate optimized machine code.

A remaining source of inefficiency in this architecture is that data-intensive processing is prone to hardware hazards, such as cache misses due to large working sets and branch misses caused by unpredictable base data. In the analytical TPC-H benchmark, for example, a quite large fraction of cycles is spent resolving hazards (cf. Figure 33) instead of processing data. The problem is known in the database community, which proposed mitigation techniques that can be applied to operators [82, 44, 117] and is also addressed from the side of the computer architecture community. Hardware vendors constantly increase the capabilities of processor-internal branch prediction and out-of-order execution capabilities to reduce branch-misses and hide cache misses [179, 3, 69]. Both, software and hardware based mitigations have merits—and no approach dominates for all use cases—any specific situation warrants specific solutions and a data processing engine must pick wisely to achieve the highest performance.

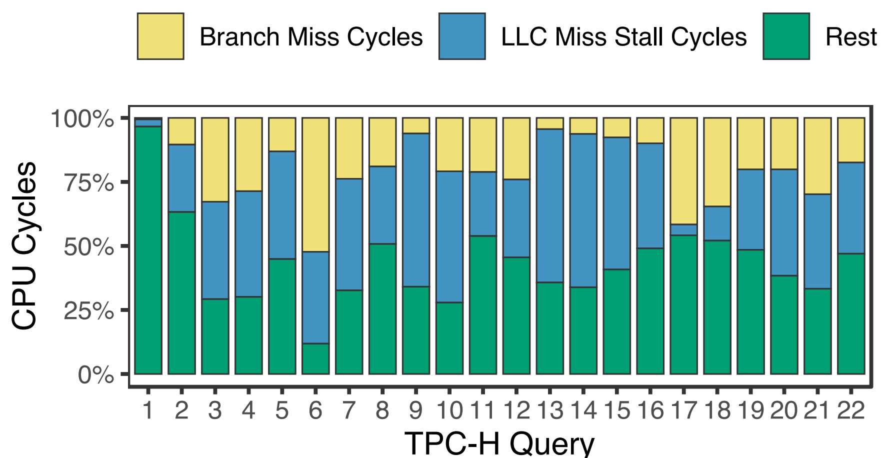


Figure 33: Amount of processing cycles spent on branch and cache misses in TPC-H (at scale factor 10).

Option A, employing software based mitigations, means changing the structure of programs to avoid branch and cache misses. The relaxed operator fusion (ROF) technique does this for compiling RDBMS [117]. It inserts buffers between relational operators, thus creates leeway to construct branch-free code and gives opportunity for a lookahead for prefetch instructions. ROF is an effective mitigation technique, yet must be employed cautiously, as it potentially increases processing cost due to extra data materialization.

Option B, hardware mitigation, consists of the branch predictor and out-of-order (OOO) speculative execution capabilities built into essentially any mainstream server CPU [3, 69]. Out-of-order execution continues processing instructions after a cache miss, potentially finding more instructions with cache misses, which are then also sent to the CPU's memory subsystem. The memory subsystem can process requests in parallel and help the OOO engine hide large parts of the hazardous cache misses. This system is even always on—there is no programmer intervention required. Unfortunately, out-of-order execution is not a panacea. It relies on finite resources which limit the scenarios where OOO is very effective. For example, an Intel Skylake CPU has an out-of-order buffer of 228 instructions. At 4 instructions issued per cycle, this buffer fills up behind an unresolved last-level cache miss in 57 cycles. Assuming DRAM access takes 400 cycles the resulting processor stall is still 343 cycles—a gap not hidden by OOO due to limited resources.

Both mitigation options are effective when certain preconditions are met. However, explicit software based mitigations always compete with hardware based mitigations. Therefore, any decision about software mitigation must weigh cost and gains, and also consider how much hardware built-in mitigation already hides (without any extra cost).

In this chapter, we present an optimization approach to mitigate hardware hazards in data processing pipelines. It leverages ROF and decides where to place buffers in a pipeline in order to prefetch data or remove branch misses. We also introduce a cost function for the optimizer which models the out-of-order execution capabilities, the probabilistic nature of cache misses, and data dependencies between cache misses. It allows the optimizer to make an informed choice of whether to rely on hardware capabilities or to adjust the generated code for better performance. Our evaluation shows that the optimizer's choices achieve a speedup of up to $2.6\times$ for some TPC-H queries and the optimizer never introduces buffers which significantly degrade execution performance.

4.1 PRIOR WORK

To meet the demand and opportunities presented by large amounts of available main memory and bandwidth from fast storage devices, highly efficient relational query evaluation engines were developed [121, 117, 17]. They outperform previous architectures as they eliminate any query interpretation overhead, make good use of available CPU resources, and ultimately keep up with the large bandwidth available to ship data to CPUs.

Nevertheless, a remaining source of inefficiency, thus unused potential, during query evaluation on large datasets, are hardware hazards such as branch and cache misses.

4.1.1 Hazard Mitigation in Hardware

Hazards are problematic for many workloads, also beyond data processing. Therefore, processor designers put mechanisms into their architectures to lessen hazard effects. Most server processors today are built as superscalar out-of-order processors [69, 3] which execute multiple instructions in each cycle and use a derivation of Tomasulo's algorithm [179] to delay instructions with missing inputs and continue execution with others. This helps to hide hazards such as L1 and L2 cache accesses and can help to greatly reduce the penalty for L3 and DRAM accesses if the circumstances are right. Unfortunately, these hardware remedies can not completely hide hardware hazards, which makes software tuning necessary for many data-intensive workloads.

4.1.2 Hazard Mitigation in Software

For database systems there are many proposed techniques to circumvent the effect of hardware hazards. Algorithm specific approaches change broader algorithmic features to cater to hardware specifics, e.g., partitioning joins [11, 166, 112, 113] or B-Trees optimized for prefetching [28]. Yet, there are also more widely applicable techniques that fit multiple use cases. Group prefetching, asynchronous memory access chaining, and interleaved multi-vectorization are general approaches to deal with cache misses, e.g., from data structure access, and with dependent cache misses, e.g., from tree traversals [27, 82, 44]. Predication is a technique to remove branch misses, e.g., from selection predicates, bloom filters, or joins. Further, for general purpose programming the Cimple domain specific language offers a programming model which lets programmers intuitively formulate algorithms that Cimple decomposes into small tasks [78]. Those can be scheduled independently to hide cache misses and remove branch misses. A trait common to these techniques is that to circumvent hazards they require multiple concurrent and mutually independent strands of work, which they use for lookahead, rescheduling, or interleaving.

4.1.3 Data-Centric Code Generation

A prevalent code generation technique for relational operator plans is data-centric code generation [121]. It seeks to generate the most efficient code by keeping data in registers as long as possible so that as many operations as possible can be performed while the data is cheaply available. For the example operator plan in Figure 34, this means when the scan S produces a tuple, as many operations as possible are performed on it. The selection operator processes it, and in case selected it is joined right away with relations Y and X and then processed by the aggregation operator. In data-centric code

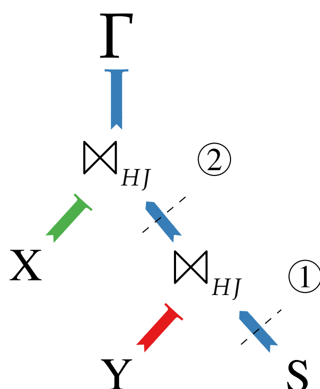


Figure 34: Buffers to Relax Operator Fusion.—*To evaluate an operator tree, data-centric code generation eagerly fuses as many operators as possible into a so-called pipeline. However, strategically refraining from aggressive fusion can produce faster code, as cutting pipelines helps mitigate hardware hazards.*

generation such a set of operators—which subsequently work on a tuple—is called a pipeline. While data-centric code generation is able to create concise code for relational operators, i.e., with very few superfluous moves, eagerly fusing all operators has a downside: The resulting code works on a single tuple at a time, which makes the previously described mitigation techniques for hardware hazards not applicable.

This can be solved by placing small buffers at profitable positions in compiled pipelines [117, 195]. Extra buffers essentially weaken operator fusion, hence Menon et al. named the technique Relaxed Operator Fusion (ROF). In the example of Figure 34 one might insert a buffer at position ①, e.g., to introduce predication to the selection operator and, then, remove branch misses, or to issue prefetch instructions for the hash table of the following join with Y and, thus, reduce cache misses. While ROF is an effective tool to reduce hardware hazards on super-scalar out-of-order CPUs it is yet unclear how an automatic optimizer can decide where to place ROF buffers in a query plan. This chapter answers that question, specifically how to decide whether to place a buffer or to rely on the processor's internal hazard remedies.

4.1.4 Hardware Performance Models

For the proposed optimizer we introduce a cost function to predict pipeline performance on super-scalar out-of-order processors. There are database specific cost models [111] and models for general purpose computing [23], yet those do not incorporate enough detail of processor internals to yield enough information for our needs, i.e., do not model memory level parallelism in detail.

Instead, we build on achievements from the computer architecture community: In an effort to predict the performance of various processor design alternatives Eyerman et al. created the mechanistic processor performance model [43].

In the mechanistic model the processor frontend issues instructions to the execution backend. If the inputs to a given instruction are available, the backend executes it directly. Otherwise, the instruction is deferred and executed when all inputs finally arrive. Since delaying instructions may upend the original program order, the processor keeps track of instruction order in the reorder buffer (ROB). All instructions in flight within the backend have a slot in the reorder buffer. The processor continuously removes the oldest instruction from the ROB, once it is executed, and makes its effect visible to the rest of the system.

In the mechanistic model hardware hazards divide instruction processing into intervals. Processing runs at full speed up to a miss event, e.g., a cache miss. From the miss event on, processing continues until the reorder buffer is full. Processing must then stall until the event is resolved.

For a given instruction stream this model predicts the execution time as the number of instructions in the stream divided by the number of instructions processable per cycle (IPC) plus the sum of all stall cycles from miss events. Interestingly, the sum of stall cycles can be determined well from a given instruction stream. Eyerman et al. investigate how miss events interact with each other and show that in the model a last-level cache miss hides subsequent cache misses which occur within the out-of-order window. They also show this model is able to predict processor performance.

The model was subsequently refined to a model named interval simulation to estimate the runtime of complete programs on hypothetical hardware by simulating caches and branch predictors [50]. Another line of work extends the interval model to the analytical processor performance model, which uses performance profiles of program executions on an existing architecture to predict the performance of hypothetical architectures [174]. Similar to our approach the analytical model takes statistics into account, yet is not able to capture the probabilistic relationship of dependent cache misses.

4.2 PIPELINE OPTIMIZER FOR DATA-CENTRIC CODE

Cache misses occur in many algorithms and slow down processing. To compensate, modern CPUs use a number of facilities.

First, modern CPUs use out-of-order execution, which was pioneered with Tamasulo's algorithm [179]. CPUs that use out-of-order execution still need to execute instructions as if they were executed in the program order which they achieve by keeping track of instructions in an out-of-order buffer of fixed length. Consequently, the number of instructions that can be executed after a waiting instruction is limited by the size of the out-of-order buffer. The typical size of an out-of-order buffer is currently around 200 instructions, e.g., Intel's Skylake architecture uses a 224 instruction buffer [69]. Assuming a processor can execute 4 instructions every cycle, this window will fill after 55 cycles, thus the processor must stall. Therefore, typically an L1 and L2 access can be

hidden completely without stalling, whereas L3 and DRAM accesses cause a stall of 45 and 275 cycles (on the Skylake-X machine used in the evaluation of this chapter).

The second hardware feature that hides the cost of cache misses is that multiple cache misses can be *resolved in parallel*. The amount of parallelism is limited by the number of buffer slots that track cache misses, so called Miss Status Hold Registers (MSHR). AMD's Zen3 architecture uses 24 MSHR per core [3]. Intel's Haswell architecture uses 10 MSHR per core; no official numbers are published for the Skylake architecture, but our experiments in the evaluation indicate that the number has not increased [69]. Combined with out-of-order execution, parallel memory access helps reduce the cost of cache misses: A single DRAM access causes a processor stall for 275 cycles. If the program manages to cause 9 more DRAM accesses within the out-of-order window, 10 memory accesses are in flight at the same time. Once the first memory access is resolved the processor moves the out-of-order window forward until it reaches the next memory access instruction. By that time the memory system just resolved the memory request, so that no further processor stall occurs [43]. Overall, one stall due to memory access can overlap and hide up to 10 (or 24 for AMD) DRAM accesses, reducing the amortized cost for a DRAM access to theoretically 27.5 cycles. Note, however, that by no means does this reduce the cost for every cache miss. Rather, to reach this scenario, the program structure must be tailored to create memory access overlap.

Unfortunately, to make best use of out-of-order capabilities processors would ideally require a long instruction stream. Actual programs, however, don't have long instruction streams, rather, there are jumps after a few instructions. To create a long instruction stream anyway, modern processors *speculate* about the outcome of *branches* and continue execution tentatively [3, 69]. Whenever the actual branch outcome becomes known, the processor compares the actual outcome to the speculated outcome. In case the speculation was wrong, the processor discards the work done after the branch and continues in the other direction. Such an event is called branch-miss, and can be quite costly with about 25 cycles of wasted work.

As query engines also suffer from cache and branch misses, researchers have investigated how to reduce the impact of those hazards [11, 166, 112, 113, 28]. Besides changing the processing algorithms, there are two techniques to reduce the impact of hazards for given query processing algorithms:

- *Prefetch* memory locations to remove cache misses
- *Predicate* decisions to create "branch-free" code

Compiling query engines especially suffer from cache and branch misses, because any interpretation overhead is removed and cache and branch misses are left to take a large chunk of execution time. For example in TPC-H, as shown at the outset of this chapter in Figure 33, for many queries more than 50% of cycles are spent resolving hardware hazards. To apply prefetching and predication to generated code, the technique of relaxed operator fusion (ROF) [117] inserts buffers between operators. Operators can cooperate with these buffers, e.g., to achieve a lookahead for prefetching or to employ predication for a selection operator.

4.2.1 Problem Formulation

Unfortunately, placing buffers between operators is a trade off. On the one hand, there is an opportunity to gain execution speed by removing hardware hazards. On the other hand, materializing data to a buffer introduces extra costs for writing to memory and later on restoring the data, thus creating the risk that the introduced costs outweigh the achieved gains.

In a simple case, deciding on inserting a buffer is the choice whether removing a branch miss that occurs for 10% of tuples pays off, given that inserting a buffer entails writing 16 Bytes and reading them afterward. Here, the possible gain is removing a branch-miss that costs 25 cycles/miss * 0.1 misses/tuple = 2.5 cycles/tuple at the cost of introducing 2 writes to memory at 2 writes per cycles = 1 cycle + 2 reads from memory at 1 read per cycle = 2 cycles. Assuming that no other hazards appear in the pipeline, for this case inserting a buffer is beneficial.

Consider a more complex case where in a pipeline 40% of tuples incur a cache miss and must access DRAM. If we were to place a buffer before the cache miss, we can introduce prefetching and reduce the access penalty by up to a factor of the maximum memory parallelism possible with the processor. By placing a buffer, we can therefore reduce the cost of the memory access, but introduce extra costs for materializing to a buffer and later reading from the buffer. Whether buffering and prefetching is profitable depends on the alternative: The processor's out-of-order capabilities try to overlap cache misses and consequently reduce the individual miss penalties—at no extra cost of materializing to memory. As out-of-order execution is limited by the available resources within the processor its effectiveness depends on the program structure, in this case the pipeline length and whether there are already buffers in the pipeline around the position we consider in this example. Overall, inserting a buffer for prefetching is only profitable when the resulting execution time is lower than what out-of-order execution can already achieve on the program, which in turn depends on the pipeline structure and other buffer placements.

Generally, deciding whether to insert a buffer is complex since any decision logic needs to take into account the cost of materialization and weigh against the possible gains through less branch misses and cache misses. Further, the processor's out-of-order execution capabilities try to achieve the same—at no extra cost. As a result, the decision logic, must also take into account that buffers compete with hardware features. In general, inserting a buffer does not always improve the situation.

In the following we present a strategy to optimize buffer placement. The strategy relies on a cost function that predicts the processing time for pipeline sections and places buffers to cut the pipeline at optimal positions. The employed cost function is presented afterward in Section 4.3.

Algorithm 2: Top down buffer placement

```
1  Function optBuffers(start: int, end: int, operators : array) : (double, int) is
2      minCost ← cost(start, end, operators);
3      minBuff ← −1;
4      for bufPos ∈ [start + 1, end[ do
5          costFront ← opt(start, bufPos, operators).cost;
6          costBack ← optBuffers(bufPos, end, operators).cost;
7          costOfPipelineWithBuffer ← costFront + costBack;
8          if costOfPipelineWithBuffer < minCost then
9              minCost ← costOfPipelineWithBuffer;
10             minBuff ← bufPos;
11     return (minCost, minBuff);
```

4.2.2 Dynamic Programming Approach

As shown, the decision whether to place a buffer depends among other things on where buffers were placed before and after. Thus, a buffer placement cannot be decided locally. Rather, any decision must take previous decisions into account. We propose to solve this optimization problem with the dynamic programming approach shown in Algorithm 2. This algorithm computes an optimal buffer placement plan under the cost function `cost`. The strategy is to first compute the cost of the pipeline without any buffers (Line 2). Then, the algorithm inserts a buffers at the first possible position and determines the cost of the resulting pipeline as the sum of the optimal pipeline before and after the buffer. It then continues with the second buffer position and so on and keeps track of the minimal cost and corresponding buffer placement. The optimal cost of a sub-pipeline is determined by a recursive call to the optimize function. Note, that *optBuffers* computes the cost from start to end (excluding end) and in case end is not the array end the cost includes writing to a buffer. Similarly, when start is not the beginning of the array the cost includes reading from a buffer.

Algorithm 3 shows a non-recursive variant of the optimization algorithm. It computes the cost for the smallest sub-problems and uses those to iteratively construct optimal solutions for larger sub-problems until a solution for the whole problem is found. This algorithm evaluates the cost function $O(n^2)$ times and performs $O(n^3)$ computations of combined pipeline costs.

4.3 COST FUNCTION

The dynamic programming optimizer from Section 4.2 requires a cost function to compute the cost of a pipeline or sub-pipeline. As we want to minimize the amount of cycles spent on processing the pipeline, we propose a cost function that estimates the

Algorithm 3: Bottom up buffer placement

```
   Input   : An array of operators that form a pipeline
   Output  : An optimal buffer placement plan
 1 ops ← operators.size();
 2 costs ← two-dimensional array of size ops × ops
 3 for l ∈ [1, ops] do
 4     for start ∈ [0, ops − l] do
 5         end ← start + l;
 6         minCost = cost(start, end, operators);
 7         minBuff ← −1;
 8         for bufPos ∈ [start + 1, end[ do
 9             costOfPipelineWithBuffer ← costs(start, bufPos).cost + costs(bufPos,
                end).cost;
10             if costOfPipelineWithBuffer < minCost then
11                 minCost ← costOfPipelineWithBuffer;
12                 minBuff ← bufPos;
13         costs(start, end) ← (minCost, minBuff);
```

amount of cycles one tuple spends in a pipeline. It takes into account branch misses, cache misses, dependencies between cache misses, and how speculative out-of-order execution hides branch and cache misses. In the following we present how to construct such a cost function, beginning with a basic case and adding solutions to more complicated issues incrementally.

4.3.1 Processor Model

The cost function estimates the required number of cycles that one tuple spends in a given segment of an operator pipeline. A pipeline segment is a sequence of one or more operators that process the tuple. Consequently, the generated code consists of a sequence of machine instructions that implement the operators, as shown in Figure 35. The foundation of the cost function is that in case there are no hazards the CPU's processing speed is only limited by the issue width, retirement width, and resources available in the processor's backend. Intel's Skylake microarchitecture, for example, can issue and

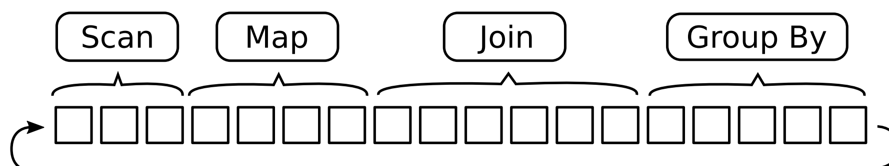


Figure 35: Machine instruction sequence that implements the relational operators in a pipeline.

retire 4 instructions per cycles (IPC). For many instructions relevant for data processing it can even process 4 per cycles. Notable deviations are load instructions with an IPC of 2 and store instructions with an IPC of 1. The "no-hazards" model yields the following cost function:

$$Cost_{no-hazards} = \frac{loads}{IPC_{load}} + \frac{stores}{IPC_{store}} + \frac{other}{IPC_{other}} \quad (1)$$

In this model the processor is assumed to work at maximum speed, so the instruction throughput can be derived from the number of loads, stores, and other instructions divided by their respective IPC (for a given processor).

4.3.2 Pipeline Runtime with Hazards

In reality, processors often incur situations in which they cannot perform at their theoretical maximum speed. A branch-miss for example flushes the execution pipeline and consequently discards work of a few cycles. Cache misses delay the execution of a load instruction. An L1 miss is usually hidden by out-of-order execution. A last-level cache (LLC) miss, however, cannot be hidden and causes execution in the processor to stall. Eyerman et al. observed that the processor executes at full speed until it completely stops at a stall [43]. Once the stall is resolved execution continues with full speed (cf., Figure 36). Thus, stalls divide execution into intervals of full speed and full stop. Eyerman et al. accordingly call this processor model the interval model.

The cost of a single LLC miss in the interval model is determined as follows. When the processor issues a load instruction with a cache miss, this instruction is placed into the front of the out-of-order (OOO) buffer and the memory subsystem fetches the required data from a lower cache or DRAM. This takes latency L_c time. Meanwhile, the processor continues execution with subsequent instructions and also places those into the out-of-order buffer. Once the load instruction with the LLC miss reaches the end of

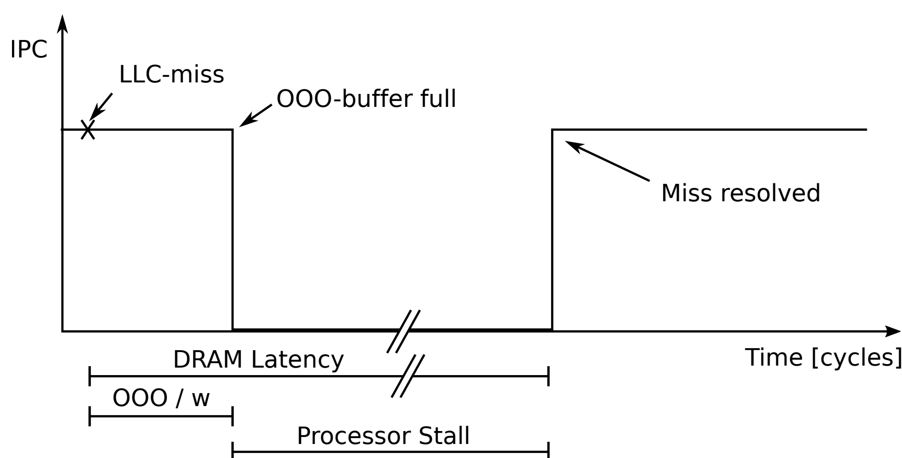


Figure 36: Single LLC miss in interval model [43].

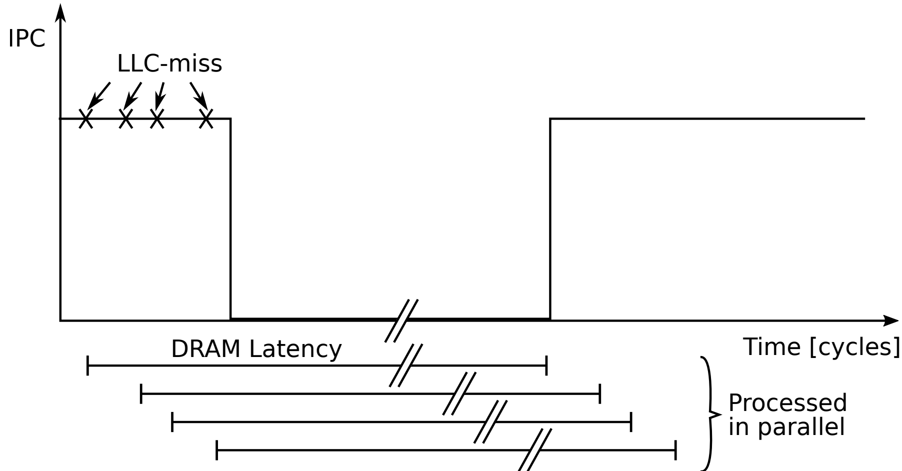


Figure 37: Multiple LLC misses in close proximity are completely hidden in the interval model.

the out-of-order buffer, the processor can not yet retire it, as the data is still outstanding. The processor continues execution until the out-of-order buffer is full. Then processing stalls until data arrives from DRAM and execution continues. Consequently, the cost of a single LLC miss is the memory latency minus the time to fill the out-of-order buffer:

$$C_{single-miss} = L_{DRAM} - \frac{Instr_{OOO}}{IssueWidth} \tag{2}$$

Interestingly, the amortized penalty for multiple subsequent LLC misses can be much lower. When there are multiple LLC misses in close proximity the memory subsystem can service these misses in parallel, as shown in Figure 37. Eyerman et al. found that the penalty from these misses overlap completely. Thus, subsequent misses are totally hidden behind the first miss and the penalty occurs only once. The number of misses that can be hidden is only limited by the number of outstanding memory requests s which the processor can track simultaneously. On current Intel architectures one core can track 10 requests, on AMD architectures one core can track 16 requests [69, 3]. Therefore, the cost for n misses within one out-of-order window is the cost of a single miss for each group of at most s misses:

$$C_{multi-miss} = \left\lceil \frac{n}{s} \right\rceil \cdot C_{single-miss} \tag{3}$$

For a pipeline with a single cache miss this overlapping can reduce the amortized cache miss cost. Initially, we presented a pipeline as a sequence of instructions that process one tuple. Note, however, that once the instruction sequence is finished execution jumps back to the beginning of the sequence to process the next tuple, yielding a very long instruction stream that repeats over and over. Consequently, even when

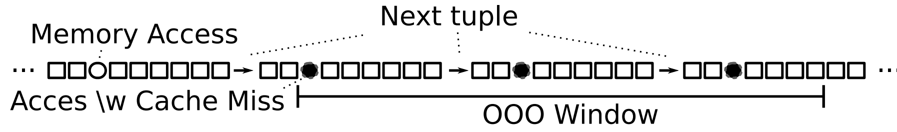


Figure 38: Pipeline execution repeats so that multiple iterations can fit into the processor's out-of-order (OOO) window.

there is only a single cache miss in a pipeline, this miss can overlap with misses from subsequent iterations (cf., Figure 38). In this case the amortized cost for a cache miss is

$$C_{amortized} = \frac{C_{single-miss}}{min(i,s)} \quad (4)$$

where $i = Instrs_{OOO}/Instrs_{Pipeline}$ is the number of pipeline iterations that fit into the out-of-order window.

Eyerman et al. also estimate the cost of a branch miss [43]. They observe that after a mispredicted branch no useful instructions are executed. The situation is only corrected when the branch is resolved. Thus, the cost of a branch miss consists of the time to resolve the branch miss and the time to flush the processor frontend

$$C_{br-m} = C_{resolve} + C_{flush} \quad (5)$$

The time to resolve a branch miss depends on the data dependencies that determine the branch condition and how fast they can be retired from the out-of-order window. Since with our approach we do not want to explore dependencies of each instruction, we assume a fixed branch resolve time for every branch and hence use a constant processor specific branch miss cost C_{br-m}.

With these penalties for cache and branch misses we estimate the cost for a pipeline with a b branch misses and one cache miss as

$$C_{fixed} = C_{no-hazards} + C_{amortized} + b \cdot C_{br-m.} \quad (6)$$

4.3.3 Probabilistic Model

We introduced a cost function that captures how out-of-order execution hides cache misses. Yet, the cost function assumes that a cache miss occurs in every pipeline iteration. This is not a given in all workloads. Rather, one memory access in the pipeline will incur a cache miss only a fraction of times. Therefore, we model a memory access as Bernoulli experiment of which the outcome is a cache miss with probability p and a cache hit with probability $1 - p$.

In case there are multiple memory accesses in a pipeline we assume they are statistically independent. We also assume memory accesses from multiple pipeline iterations to be independent. An independence assumption is reasonable for our use case, as Van den Steen et al. observed that there are two types of cache miss patterns [174]: In the

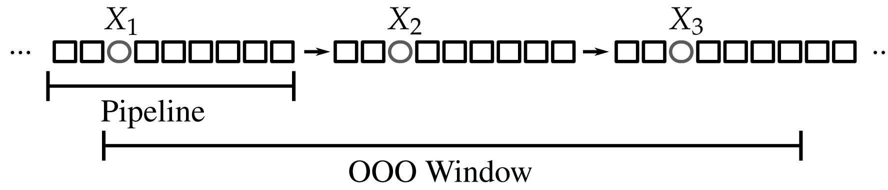


Figure 39: Memory accesses are random variables $X_1, ...X_n$ with cache miss probability p.

first type cache misses occur in bursts when data was freshly mapped to memory and then accessed. In the second type cache misses occur uniformly distributed and independent over all memory accesses when the working set exceeds the cache size. In our data processing case, input data is scanned linearly, which lets hardware prefetchers prevent cache misses, and access to intermediate results, e.g., hash tables for joins, is first produced within a query plan and then accessed in a following step. Should cache misses occur with these accesses then they occur because the working set is larger than the cache, thus individual misses are uniformly distributed and independent.

A probabilistic view yields the following cost model for a pipeline with one memory access with cache miss probability p (cf. Figure 39). Let $n = \lceil OOO/instrs \rceil$ instances of the pipeline fit into the out-of-order window, then n memory accesses fit into the window. We denote the memory accesses with the random variables $X_1, ..., X_n$ which each have cache miss probability p. The processor model states that a cache miss completely hides the following cache misses within the out-of-order window. To get the amortized cost of a cache miss, we consider the case where the memory access X_1 has a cache miss $X_1 = miss$. When X_1 has a cache miss, out-of-order execution continues to evaluate the succeeding pipeline iterations, so that the memory accesses $X_2, ..., X_n$ are executed and possible cache misses are hidden behind the stall caused by the miss of X_1. Therefore, the amortized cost for the miss of X_1 is $C_{single-miss}$ divided by the expected number of misses within the out-of-order window, which is the miss of X_1 plus the expected number of misses of $X_2, ..., X_n$

$$C_{amort}(X_1 = miss) = \frac{C_{single-miss}}{1 + \mathbb{E}(X_2 + ... + X_n)} = \frac{C_{single-miss}}{1 + (n-1)p} \tag{7}$$

We assume the cost of $X_1 = hit$ to be zero, thus the expected amortized cost for one memory access is

$$\begin{aligned} \mathbb{E}(C_{amort}(X_1)) &= P(X_1 = miss) \cdot C_{amort}(X_1 = miss) + 0 \cdot C_{amort}(X_1 = hit) \\ &= p \frac{C_{single-miss}}{1 + (n-1)p} \end{aligned} \tag{8}$$

4.3.4 Multiple Cache Misses and Data Dependencies

The previous section modelled a pipeline with a single memory access that potentially causes a cache miss. Naturally, a generated pipeline can consist of more than one

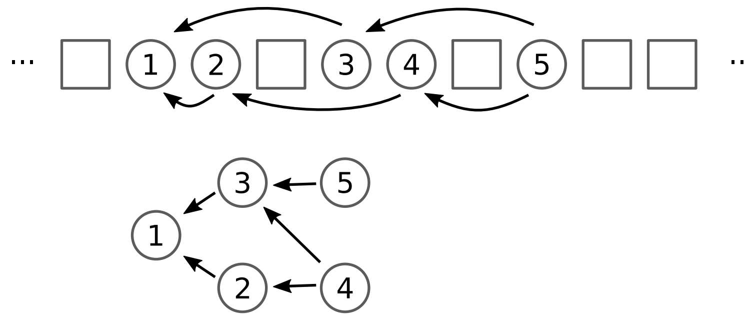


Figure 40: Dependent memory accesses within a pipeline

memory access, e.g., as shown in Figure 40. These accesses can depend on each other, as denoted by the arrows.

Dependencies arise when, e.g., the data loaded with one memory access is used to compute the address for another memory access. For a CPU, even with out-of-order processing, such a data dependency means that dependent memory accesses can not be overlapped, i.e., can not be hidden. That means that a chain of cache misses must be processed one miss at a time. Similarly, a directed acyclic graph (DAG) of cache misses is processed one wave at a time (cf. Figure 41) where cache misses within a wave can be hidden behind the first miss. Albeit, there is a cache miss penalty for every wave.

The wave placement in which all nodes are solid circles assumes that every memory access is a cache miss. However, in the probabilistic model each access has a cache miss probability p, so depending on if accesses actually incur a miss, there are many combinations in which the accesses with their dependencies can be scheduled on the waves. Two example schedules are shown in Figure 41. The dashed nodes in the graph do not incur a cache miss and can therefore be scheduled on the same wave as their input.

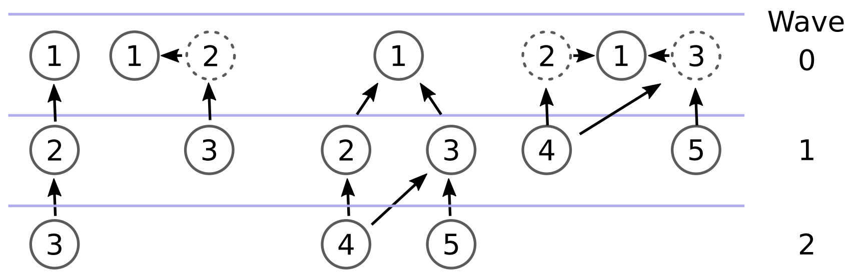


Figure 41: Out-of-order execution schedules dependent memory accesses in waves.

Each wave with at least one cache miss incurs the stall penalty $C_{single-miss}$, which, as shown in the previous section, can be amortized with the cache misses from subsequent pipeline iterations. To simplify slightly, we assume that all cache misses within one wave and one pipeline overlap completely. This lets us compute the amortized cost C_w for a wave w with Equation 8 and the probability p_w that there is at least one cache miss in wave w

$$C_w = p_w \cdot \frac{C_{single-miss}}{1 + (n-1)p_w} \tag{9}$$

where n is the number of pipeline iterations that fit into one out-of-order window. Consequently, the cost for a DAG of memory accesses is

$$C_{DAG} = \sum_w C_w = \sum_w p_w \cdot \frac{C_{single-miss}}{1 + (n-1)p_w} \tag{10}$$

To compute the probability p_w that there is at least one cache miss at wave w for a given DAG we first observe that out-of-order execution issues all memory accesses as soon as all inputs are ready. This schedules all cache misses in the lowest possible wave so that all input cache misses are at lower waves (accesses with cache hits can be at the same wave). For any given arrangement of cache misses this fills all waves with cache misses so that the last memory access node of the DAG is pushed to the last wave after a cache miss (in case there is no single last node we can create an artificial node with cache miss probability $p = 0$ and all nodes not used as input as children). This means, there is at least one cache miss at wave w if the last node is at depth $D > w$. Consequently, the probability p_w that there is at least one miss at wave w is the probability that the root node is at a depth larger than w:

$$p_w = P(D_{root} > w) = 1 - P(D_{root} \leq w) \tag{11}$$

The probability $P(D_i \leq w)$ that the depth D_i of node i is smaller than w can be derived with the inputs of i and by considering both memory access outcomes:

$$\begin{aligned} P(D_i \leq w) = \quad & \overline{p_i} \cdot P(D_1 \leq w \wedge ... \wedge D_k \leq w) \\ & + p_i \cdot P(D_1 \leq w-1 \wedge ... \wedge D_k \leq w-1) \end{aligned} \tag{12}$$

Here, p_i is the cache miss probability of access i and $D_1, ..., D_k$ are the depths of the inputs of access i. Unfortunately, it is computationally expensive to evaluate the joint probability $P(D_1 \leq w \wedge ... \wedge D_k \leq w)$ for a DAG as the D_i are not necessarily independent.

Dependency structure in a tree

Let us first consider the simpler case where the dependence structure is a tree. Here, the depths $D_1, ..., D_k$ are computed from non-overlapping paths and are thus statistically independent. Therefore, the joint probability is

$$P(D_1 \leq w \wedge ... \wedge D_k \leq w) = \prod_{j=1}^{k} P(D_j \leq w) \tag{13}$$

so the depth of a node can be determined with

$$P(D_i \leq w) = \overline{p_i} \cdot \prod_{j=1}^{k} P(D_j \leq w) + \begin{cases} p_i \cdot \prod_{j=1}^{k} P(D_j \leq w-1) & ; w \geq 0 \\ 0 \end{cases} \quad (14)$$

which yields a recursive algorithm to compute the depth probability from the depth probabilities of the inputs and lower depths.

Approximation for dependency DAG

If we use the above approach—which is accurate on trees—to calculate $P(D_i \leq w)$ on a DAG we underestimate the probability. This provides a lower bound. Conversely, we can reduce the DAG to a tree by deleting edges until every node only has one incoming edge. The tree algorithm then overestimates $P(D_i \leq w)$, which is an upper bound. We estimate $P(D_i \leq w)$ for DAGs by taking the average of the lower and upper bound.

4.4 EVALUATION

Previous sections introduced an optimization approach to decide whether to apply operator fusion or to insert buffers between operators. This section analyzes the optimization approach. We implemented the buffer optimizer and buffering operators in the relational database system Umbra, to be able to test effects in a full-fledged system.

First, we evaluate in Section 4.4.2 whether the buffer optimizer places buffers that are able to reduce query runtime on three analytical benchmarks: TPC-H [182], TPC-DS [181], and the star schema benchmark (SSB) [131]. We also investigate whether buffering indeed reduces hardware hazards and whether this is the reason for performance improvement. Second, we check in Section 4.4.3 the quality of the optimizer's decisions as to whether there are cases in which it degrades query performance or misses optimization opportunities. Third, we examine how well the cost function predicts actual processor behavior in Section 4.4.4.

4.4.1 Experimental Setup

All experiments were run on a machine with an Intel i9-7900X CPU with 10 cores at 3.3 GHz and 14 MB of last-level cache. This processor is built with the Skylake architecture, which has 2-fold simultaneous multi-threading and 10 slots in the line fill buffer (LFB) in each core. The benchmarks are TPC-H, TPC-DS, and SSB and are each used at scale factor 10 (unless otherwise noted). We only consider TPC-DS queries without window functions, as those are currently not supported in Umbra. By default all experiments are done with one execution thread.

The pipeline optimizer inputs are retrieved by a preceding run of the query to determine tuple counts for each operator and to sample branch misses and last-level cache

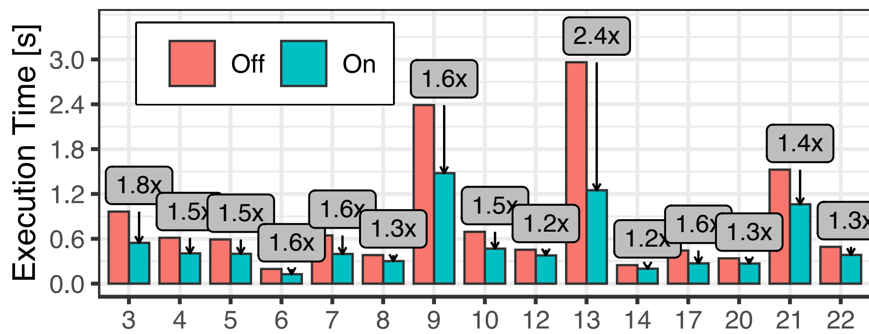


Figure 42: Query execution times with and without buffering for selected TPC-H queries

misses with the processor's internal performance measurement unit (PMU). The profiling samples are then automatically mapped to relational operators and from there used as inputs for the pipeline optimizer. While this strategy is not useful for a production-ready database system, for the purpose of this evaluation it emulates that the pipeline optimizer is integrated into adaptive execution [84]: Ideally, in the first compilation step a query is compiled with the Flying Start backend of Chapter 3 and while the query is running, the PMU collects samples. When adaptive execution decides to recompile the query with more expensive optimizations, the profile is analyzed and used for the pipeline optimizer. Currently, our prototype implementation is not yet able to perform these steps, thus, for this evaluation we emulate the behavior by preceding each query with a performance profiling run.

4.4.2 Captured Improvements

We evaluate the buffer optimizer on a large set of 111 analytical queries from three benchmarks to capture a wide variety of scenarios.

TPC-H Benchmark

Of the 22 TPC-H queries, optimized buffering provides a performance improvement by at least 1.2× for 13 queries. Query runtimes for these 13 queries with and without buffers are shown in Figure 42. Each query is annotated with the execution speed improvement that buffering provides. The remaining queries from the benchmark have a performance improvement of less than 1.2×, yet for all the queries the optimizer chose a buffering plan that is at least as fast as using no buffers at all.

The reason for performance improvement with buffering can be understood from Figure 43. It shows how much fewer cycles per tuple the processor spends to resolve hardware hazards when optimized buffers are enabled. The cycle count is derived from the number of branch misses recorded by the PMU during query execution. The cycle count is estimated by assuming that each branch miss takes around 25 cycles to resolve. The

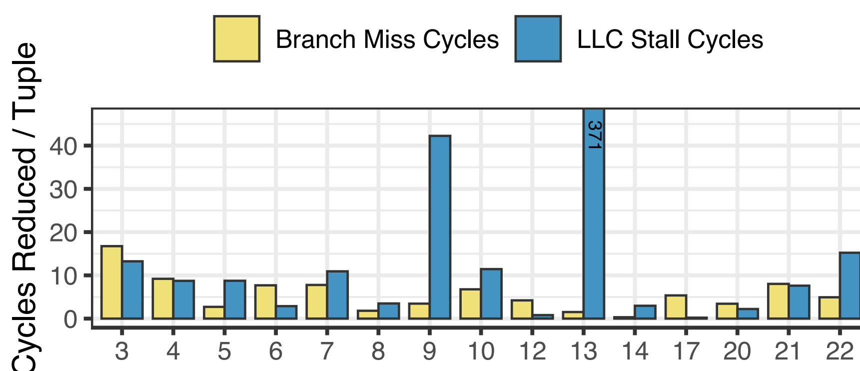


Figure 43: Reduced cycles per query on TPC-H

PMU event `cycle_activity.stalls_l3_miss` directly reports the LLC stall cycles. All event counts are divided by the number of tuples scanned in the query.

Observe, how for the join-heavy queries 3,4, and 9 the stall cycles waiting for memory can be significantly reduced, leading to faster query execution from between $1.4\times$ to $1.8\times$. Further, query 13 performs a group-join on a hash-table with 1.5 million entries and incurs ~ 2 LLC misses per tuple. Buffers are able to greatly reduce this cost by 292 stall cycles per tuple.

A reduction of branch misses also reduces query runtimes. Query 6, for example, only consists of a table scan, selections, and an aggregation. The selection predicates are rather unpredictable for the CPU, thus cause ~ 7 misprediction resolve cycles per scanned tuple. The optimization inserts buffers after these predicates and is able to completely remove any branch miss cycles.

Star Schema Benchmark

The star schema benchmark consists of one large fact table and multiple significantly smaller dimension tables. There are four basic queries 1*, 2*, 3*, and 4* which each join the fact table with one, three, three, and four dimension tables respectively. Each basic query has variations, which with increasing variant number (.*) find less join partners for the fact table. This entails, that hash tables become smaller with increasing variant number.

Figure 44 shows that buffering is beneficial for all but query 23. Figure 45 reveals that for 1* the performance improvement is mainly generated from reducing branch misses. These stem from predicates evaluated on the fact table, which the optimizer cushions with a buffer. Queries 21, 31, 31, and 42 benefit from a reduction in cache miss cost. These occur due to prefetching of hash tables when buffers are placed before the lookup.

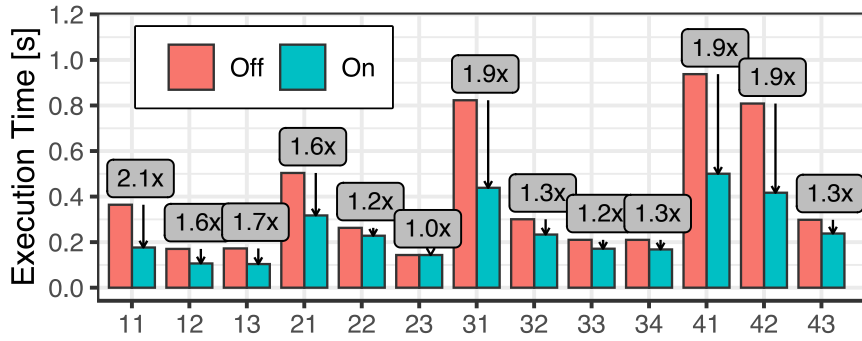


Figure 44: Query execution times with and without buffering for SSB queries

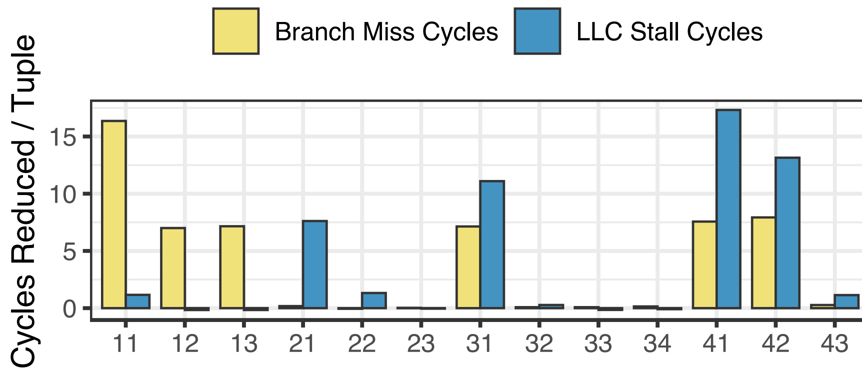


Figure 45: Reduced cycles per query on SSB

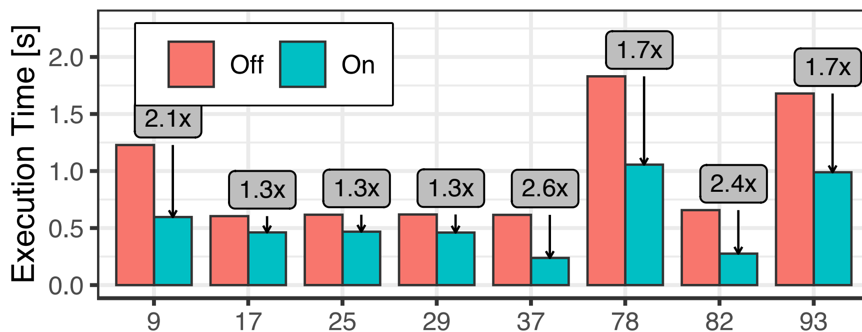


Figure 46: Query execution times with and without buffering for selected TPC-DS queries

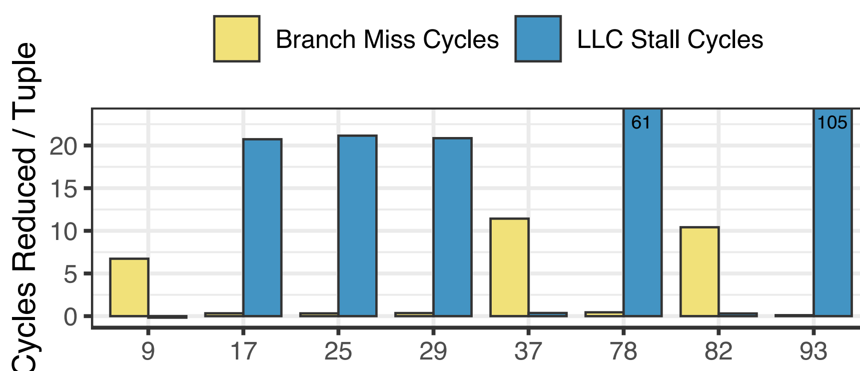


Figure 47: Reduced cycles per query on TPC-DS

TPC-DS Benchmark

TPC-DS is a large benchmark with 99 queries of which Umbra can currently execute 76. We found that only for the 8 queries of Figure 46 does optimized buffering provide a performance improvement of 1.3× or more. The corresponding cycle improvements are shown in Figure 47.

4.4.3 Optimization Optimality

The number of configurations available to the buffer optimizer is quite large, as for a query with n possible buffer positions it chooses a subset of positions to insert buffers. With n positions there are 2^n distinct buffer configurations. In such a large optimization space it is quickly infeasible to execute all configurations.

As alternative, we explore the optimization space starting from three vantage points: All buffers off, all buffers on, and the configuration chosen by the optimizer. We measure the runtime of the base configurations. Then, we generate new configurations by starting from a base configuration and toggle one buffering option, i.e., to insert a buffer when there is none, or remove the buffer otherwise. We also measure these derived configurations. This allows us to explore the surroundings of the base configuration in the optimization space and provides at least a local indication if there are any optimization opportunities that the optimizer did not leverage.

Overall Outcome.

Figure 48 shows the result of these experiments. Each dot represents the measurement of one buffer configuration; the one chosen by the optimizer is marked with a diamond. The vertical axis signifies how much faster the configuration can be executed in comparison to using no buffers at all. In addition, the point density is indicated by an underlying violin plot.

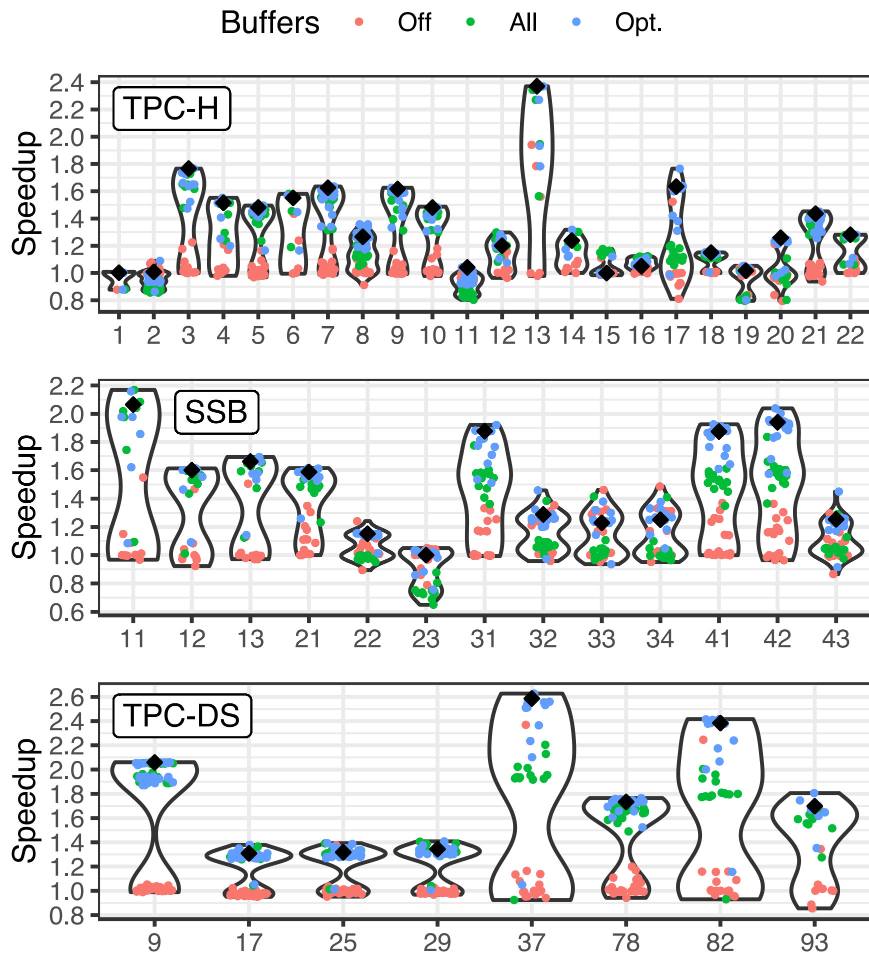


Figure 48: Optimization space of TPC-H, SSB, and TPC-DS queries

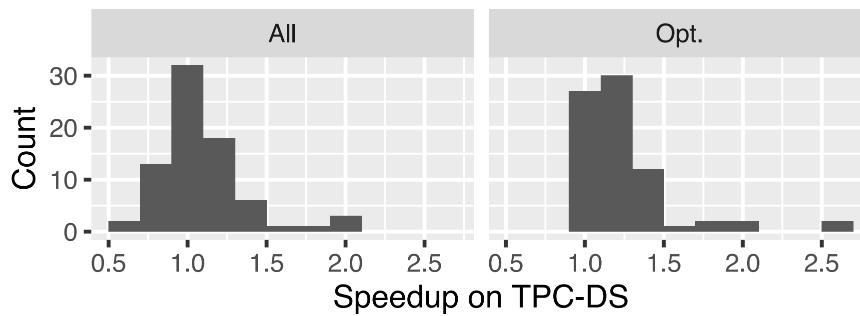


Figure 49: Alternative strategy of inserting all possible buffers vs. optimizing buffers

Importantly, the optimizer is always able to pick a buffer configuration as least as fast as using no buffers at all. That means, processing performance is never degraded by unprofitable buffers. Further, for all of the queries the configuration chosen by the optimizer is very close to the fastest alternative. We conclude that the optimization approach is effective at picking beneficial configurations.

Alternative: All Buffers On.

The results suggest an alternative strategy to using an optimizer: For a number of queries, enabling all buffers is just as fast as the optimizer's configuration. We could just insert buffers whenever possible. In many cases this approach seems competitive or even better than the optimizer's choice. After all, minor inefficiencies caused by some unprofitable buffers are easily hidden by the gains of profitable buffers. However, adding buffers everywhere fails when buffering in general is not very profitable, e.g., in TPC-H queries 1,2,11, and 19. Here, adding all possible buffers lets the query run slower than without any buffers. In other cases, too many buffers merely reduce the overall optimization profit, e.g., in TPC-H query 17 and TPC-DS queries 37 and 82. Figure 49 also shows that inserting all possible buffers is detrimental on a number of TPC-DS queries.

Missed Opportunities

While the optimizer is often able to pick a very good plan, in a few instances it misses opportunities to reduce the query runtime further. Figure 48 shows that for example for SSB queries 3.3 and 3.4 there are faster buffer configurations available. In query 3.4 there exists a configuration that outperforms all other options by only inserting a single buffer S. Curiously, this buffer is placed after the first join were it has no opportunities to reduce branch or cache misses.

For query 3.4 the join order optimizer chooses to first join the lineorder table ($\sim$ 60 M tuples) with the supplier table (creating a hash table with 164 tuples). This join reduces the amount of tuples in the rest of the pipeline from 60 M to $\sim$500k. That means that only 0.8% of the scanned tuples from lineorder reach the buffer S, thus the

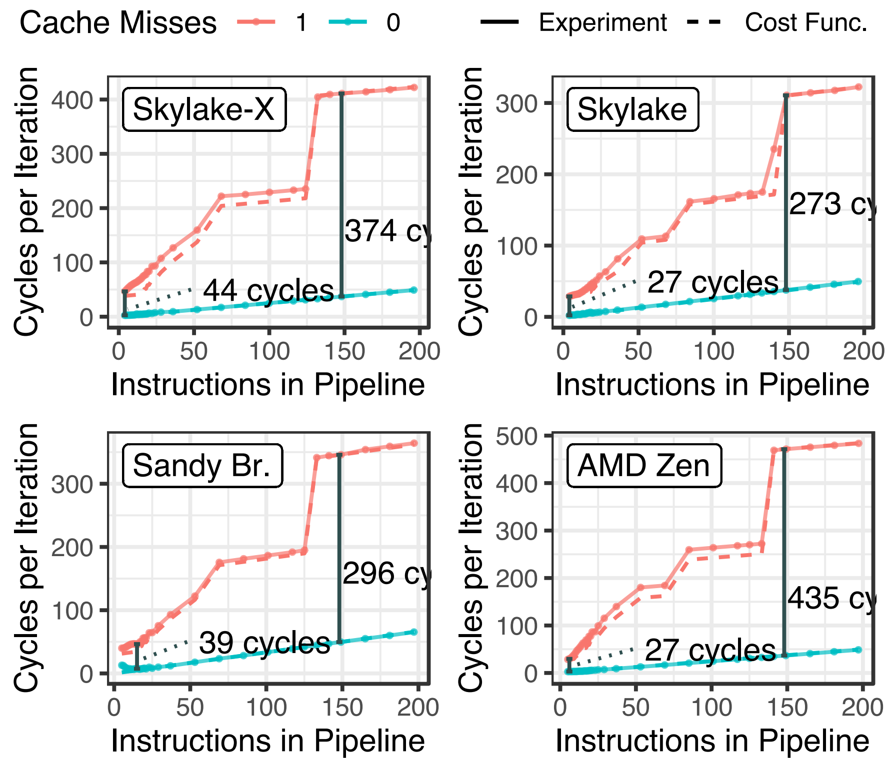


Figure 50: Processors hide cache miss penalties with out-of-order execution in short pipelines.

cost of materializing to the buffer is incurred seldomly. Further, buffer S detaches the first, heavily-used part of the pipeline from the rest. The buffer optimizer does not place any buffers around the first join, so that execution takes 12 cycles and 24 instructions per tuple. With buffer S execution time reduces to 11 cycles and 23 instructions per tuple. Thus with buffer S, which detaches the busy front of the pipeline from the rest, the LLVM compiler is able to generate better machine code, resulting in higher execution speed. The buffer optimizer misses the opportunity as its cost function does not cover optimization effects from the underlying compiler.

4.4.4 Cost Function Accuracy

As an essential part of the optimizer we introduced a cost function to predict the execution time of a pipeline on a modern out-of-order processor. The cost function models cache miss hiding, random behavior, and data dependencies. In this section we evaluate experimentally how the cost function fits real processors.

Cache Miss Hiding

The basic assumption of the cost function is that out-of-order processors can hide multiple cache misses and resolve them while stalling only once. Thus, the stall penalty

Table 12: Microarchitecture attributes of used processors [3, 47, 69]. LLC-miss penalty and number of MSHR for Skylake and Skylake-X was determined experimentally.

Arch.	LLC-miss penalty [cycles]	MSHR	ROB	Integer Registers	observed OOO Window
Skylake-X	374	10	224	180	130
Skylake	273	10	224	180	148
Sandy Bridge	296	10	168	160	130
Zen	435	16	192	168	140

is amortized over multiple cache misses (cf., Equation 4), so that when multiple pipeline iterations fit into the out-of-order window each iteration incurs a reduced (amortized) cache miss cost.

To check whether this is indeed the case on actual processors, we perform the following experiment: We place a load instruction that always incurs a last-level-cache miss into a loop and add padding instructions after the load—we simulate a pipeline as depicted in Figure 38. The load reads from an array that covers 4 million cache lines and every loop iterations reads at a random offset. With the array much larger than the last-level cache the loads almost always get a cache miss. The padding instructions are addition instructions, which processors usually process at full issue width. More padding instructions simulate a longer pipeline, thus fewer pipeline iterations fit into the out-of-order window. Less padding instructions simulate a shorter pipeline, thus more iterations (and cache misses) fit into the window.

Figure 50 shows the result of this experiment for processors with four different microarchitectures. Skylake-X and Skylake represent Intel's current generation architecture, respectively for the server and client market. Sandy Bridge is an older Intel architecture. Zen is a recent architecture from AMD. For each of these architectures we performed the experiment and varied the amount of instructions in the pipeline (x-axis) and measured how long the execution takes (y-axis). Notably, the runtime graph exhibits a staircase pattern. From left to right, the experiment adds instructions to the pipeline. Whenever the amount of instructions exceeds a threshold so that one less pipeline iteration fits into the out-of-order window, one less cache miss is hidden. This makes the amortized cost higher and reflects in a new step in the staircase pattern. We conclude that out-of-order processors are able to hide subsequent cache misses.

As the cost function is built on this assumption and the assumption holds on modern processors, the cost function is able to predict the pipeline execution time accurately (dashed lines in Figure 50). However, the size of the out-of-order window seems to not be limited by the size of the reorder buffer, but by the much smaller, empirically determined values of Table 12. The window size may thus be limited by the number of available integer register names. With this adaptation the cost function is able to predict the processor behavior well.

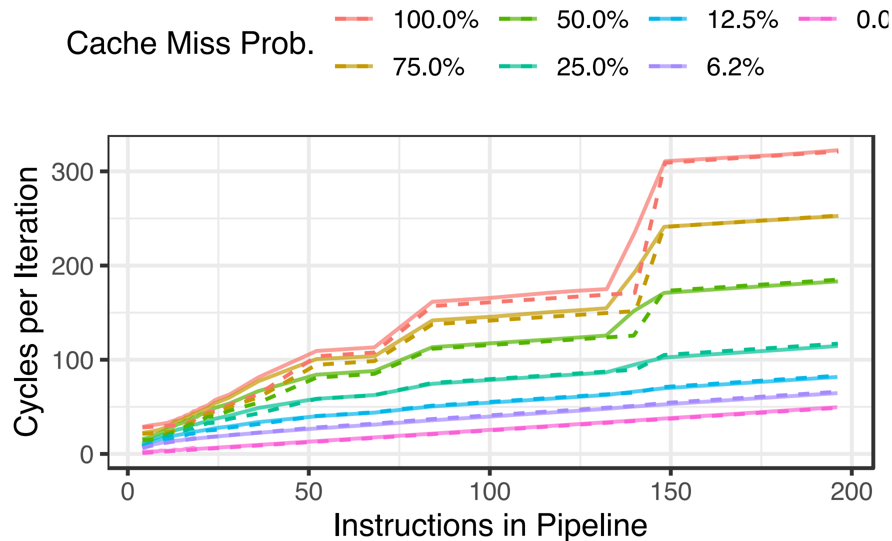


Figure 51: Processors hide cache miss penalties with out-of-order execution in short pipelines, also with probabilistic occurrence.

Figure 50 also shows the runtime for the experiment for the case in which the load instruction always accesses the same offset, thus incurs no cache miss. The difference between the experiment with one and no cache miss is exactly the amortized number of cycles the processor stalls due to the cache miss. Consequently, from this experiment we can determine the cache miss stall penalty (right-hand side) and the amortized stall penalty with maximum cache miss hiding (left-hand side). Observe how for Skylake there is a factor of $10\times$ between these values, as Skylake is limited by having only 10 miss status hold registers (MSHR) per core to keep track of L1 misses. For Zen, there is a factor of $16\times$ between the values. Its limit of 16 MSHR allows for more amortization.

Probabilistic Model

For a given load instruction one pipeline iteration does not necessarily experience a cache miss every time, rather a cache miss only occurs a fraction of the iterations. The cost function takes this into account by considering a cache miss as a random variable with a miss probability p and with an updated cost function that models the expected number of hidden cache misses for amortizing the cost (cf., Equation 8).

To check whether this accurately reflects processor behavior, we extend the previous experiment so that a given fraction f of offsets are not randomly distributed but 0. Loads from offset 0 will not have a cache miss, as it is accessed relatively often. We can thus achieve a cache miss probability p by setting $f = p$. The result of this experiement is shown in Figure 51. The processor is still able to hide multiple cache misses, as evident from the staircase pattern. Also, the cost function is able to predict this behavior fairly

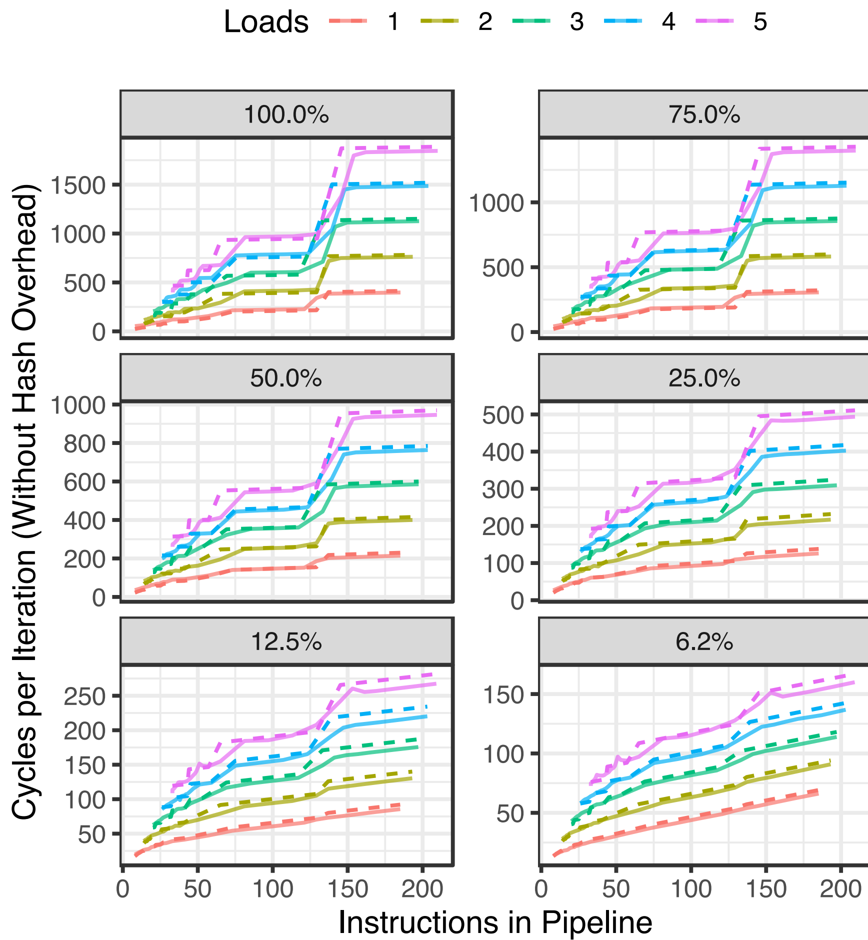


Figure 52: Dependent misses with multiple probabilities

well. We conclude that the probabilistic model (Equation 8) adequately predicts actual processor behavior.

Data Dependencies

The last missing piece for the cost function is the processor behavior when there are multiple load instructions in a pipeline—each with a cache miss probability—and the load instructions have data dependencies amongst each other. The cost function captures this through Equation 10.

To check whether this reflects processor behavior, we again extend the experiment. A pipeline can now contain multiple load instructions, each of which depends on the previous instruction and has a cache miss probability p. The data dependency is created with the following method: Before the experiment, the data array is filled with random numbers. Each load instruction loads one of those random numbers, which is then combined into the offset for the subsequent load instruction—creating a data de-

pendency. For combination we use the crc32 instruction to hash two values into one and afterwards we multiply with the offset to make sure that every intended cache hit (offset = 0) remains a cache hit even after hashing.

The result of this experiment[1] is shown in Figure 52. With cache miss probability $p = 100\%$ a cache miss stall occurs for every load in the dependence chain. However, when multiple pipeline iterations fit into the out-of-order window the processor is able to hide cache misses of the (independent) chains of those iterations, again showing a staircase pattern. In the case of $p = 100\%$ the processor is even hiding cache misses of other chains, but at the same chain depth.

With $p < 100\%$ this is no longer necessarily the case. As explained in Section 4.3.4 the cost function expects that there are cases when cache misses from different levels hide each other. This leads to an overall effect that misses at the start of the chain have a lower amortized cost, as they are likely to hide more other cache misses. Cache misses further down the chain have higher amortized cost, as they are less likely to hide other misses. Overall, the experiement shows that for $p < 100\%$ the actual processor behavior is very close to the dependent cache miss hiding predicted by the cost function. Vice versa, we conclude that the cost function adequately models processor behavior in the presence of probabilistic cache misses and data dependencies.

4.5 SUMMARY

In data-intensive processing hardware hazards are a large source of inefficiency that slow down processing. There are mitigation mechanisms built into every modern processor, and also known software techniques to avoid hazards. Unfortunately, both mechanisms compete and both have their merits—that means both have use cases where they are the better option.

In this chapter, we have shown for the case of compiling relational query engines how to plan and optimize hazard mitigations. Specifically, we presented a cost function to predict processor behavior and the impact of hazards. We also proposed an optimizer which leverages the cost function to place software mitigations when hardware mechanisms on their own are not fully efficient.

Our evaluation shows that the cost function accurately models processor behavior and allows the optimizer to pick effective mitigations. On analytical benchmarks the optimizer is able to let queries run up to 2.6× faster, while producing no significant slow-down for queries where mitigations are unnecessary.

---

1 The cycles per iteration count of Figure 52 does not include extra work due to hashing overhead. For long load chains the experiment experienced more L1 misses due to the extra memory traffic of reading the input offsets for each load. We removed this overhead from the experiment results for clarity.

5 PROFILING COMPILING QUERY ENGINES

With contributions from Alexander Beischl and Jana Giceva.

Compilation in relational query engines splits query evaluation into a compilation and an execution phase, which creates challenges during performance tuning and profiling. This is not only a problem in compiling relational query engines, but in any setting where computation steps are first optimized in an internal logical representation and then translated to executable programs. Internal representations exist in many forms and with different characteristics, yet a common denominator is that computation is expressed as operators which are connected to form a graph to describe the overall dataflow.

Dataflow graphs are a powerful abstraction for a variety of applications and workloads: from more traditional systems like databases and compilers, to more widely adopted computing frameworks for big-data [191, 119, 114], graph- and stream-processing [167, 24], and machine- or deep-learning [1, 141]. It allows developers to express on a high abstraction level the data dependencies between various tasks and map computations to (pipelines of) operators [162].

Although they have a long history, dataflow computing systems are even more relevant today, in particular when deploying computations onto various hardware platforms and accelerators. Using such an expressive high abstraction layer allows the system stack to absorb the complexity of generating efficient code and mapping it onto the available hardware resources, as opposed to burdening the developer. In fact, such compiling and code-generating dataflows are what many believe the only way to address the increasing heterogeneity of the underlying computing resources and allow the domain-expert developers to focus on the important task at hand using a DSL at an abstraction layer they are most comfortable and productive at, without worrying about low-level details [95, 29, 64, 136, 146]. The key to this success is that the background-process involves progressive layering of optimization steps for dataflow graphs that generate lower-level intermediate representations (cf. Figure 53), which eventually lead to a high-performant and efficient binary program.

While this has many advantages, with each optimization layer/step we lose semantic knowledge about the (higher-level abstraction) dataflow so that some critical tasks, like debugging and performance profiling, become intractable. Most profiling tools today primarily operate on a much lower level and report metrics on an assembly instruction- or function- granularity [68, 2, 188, 103]. While for systems experts the task to map the information provided from these profilers to the low-level code they have written is rarely an issue, the problem becomes less trivial when anyone needs to read performance profiling for machine generated code and interpret it in terms of higher-level abstractions.

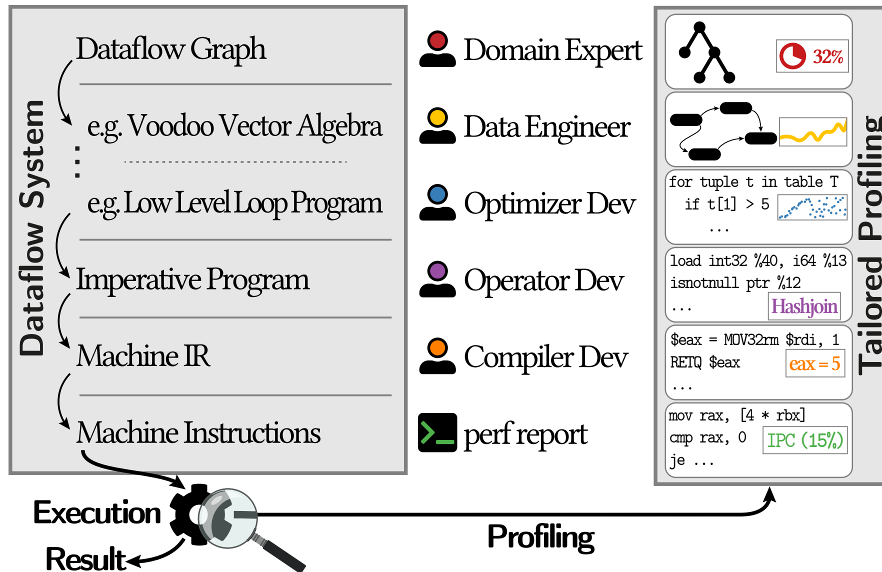


Figure 53: Layered organization of compiling dataflow systems on the left and profiling results of our novel Tailored Profiling approach on the right.

With today's technology, debugging and profiling computer systems is often a non-trivial task with a number of challenges: First, existing software systems are quite complex and involve many components that interact when the dataflow computation is being executed. A source line could be used by multiple components, some of which may be from external libraries, so backtracking which one invoked it can take time. Second, reverse engineering and backtracking an instruction's path through a long list of optimization steps can be a daunting task at best, while almost intractable in general. Third, even for an experienced systems-developer it is difficult to say where exactly the recorded assembly instruction belongs to or what kind of data resides at a given memory address. Fourth, many of the above points become trickier with code generation, because we introduce more layers of indirection.

Yet, high-level performance profiling reports are be very valuable to anyone working on the dataflow graph (or any of the intermediate optimization layers): On the one end of the abstration-layer spectrum, the domain expert (cf. top right of Figure 53) may wish to know how much time is spent on different parts of the dataflow graph, or which operation is the most expensive one. Furthermore, the optimizer developer could potentially identify which particular segment of the pipline of fused operators is the bottleneck and what causes it – perhaps reordering the operations or breaking the pipeline can significantly reduce the runtime. On the other end of the spectrum, the lower-level compiler expert could use insights indicating which part of the pipeline causes control- or data-hazards and tweak the code generation to potentially avoid them in the future.

In this chapter, we present how to performance debug and profile compiling dataflow systems with *Tailored Profiling*—a way that is understandable and brings value to any user working on a selected abstraction layer. To achieve understandable profiling we

begin by analyzing the state of the art to identify the (source of the) problem of the semantic gap between the dataflow graph and it's subsequent transformations in the lower abstraction layers (cf. Section 5.2). We then list the key requirements a dataflow performance profiler should meet and present our high-level design for a generic dataflow systems profiler in Section 5.3. Although at first thought it may seem counter-intuitive, compiling dataflow systems are as much part of the solution as they were part of the problem. In fact, one of our key insights is to extend the compilation steps to also annotate the generated code with meta-data. We can then use this meta-data to bridge the semantic gap and map the profiling results back to the desired abstraction layer. This enables us to post-process the data and present it at a granularity that brings the best insights to the developer. In Section 5.4, we detail the specific steps that were needed to build our prototype as part of the high-performance compiling DBMS Umbra. As appropriate profiling is already challenging, we focused our prototype implementation on single-machine CPU computations. We discuss the benefits of our approach in the context of a few compelling use-cases and evaluate the performance overheads and accuracy of our implementation in Section 5.5.

5.1 QUERY ENGINES AND PERFORMANCE TUNING

5.1.1 Dataflow systems

Dataflow graphs have seen a resurgence, as in recent years a number of influential data processing systems were built that use dataflow graphs at their core [191, 119, 114, 24, 1, 141]. In the following, we call these systems data-flow systems.

Dataflow graphs are a flexible way to compose programs from operators that act on data. Operators are connected to other operators, thus the result of an operator forms the input to further operations. Overall, the construction forms a graph with operators as vertices that pass data along the edges [162].

The popular machine learning framework TensorFlow, for example, uses the dataflow graph abstraction in their user API. Users explicitly construct a graph of machine learning operators that pass along tensors with training data. Similarly, Spark offers the DataSet API to let users combine analytical operators into complex queries.

An advantage of dataflow graphs is that on the user facing side of a system, the graphs create an abstract and composable interface. Furthermore, internally the system can perform high-level optimizations on the graph structure.

5.1.2 Code Generation

Data-flow graphs are used for high-level logical optimization. Systems can automatically restructure the graph to minimize computation time.

To apply dataflow graphs to input data systems must execute the operators in the graph. Operator execution can, e.g., be done by interpretation in which operator implementations are generic and configured according to the dataflow graph to act on the input data.

An alternative—applied to get the highest performance from the underlying hardware—is to generate machine code specifically for each dataflow graph, thus removing any interpretation overhead. Most systems organize machine code generation not in a single step from dataflow graph to machine instructions, but in a layered approach with multiple intermediate representations (IRs) and successive lowerings [95, 29, 64, 136, 191]. An abstract version of such a layered approach is shown in the dataflow system on the left of Figure 53. The top-most graph layer is translated into more concrete intermediate representations, which widely vary depending on the actual system. E.g., Voodoo proposes to use a vector algebra to reason about data partitioning, instruction level and thread parallelism [146]. TVM uses low level loop programs to reason about control flow, but still abstract from concrete hardware [29]. Such IR levels are usually followed by imperative program representations which target specific hardware instructions.

Every IR is designed to support a set of optimizations which reorder and restructure the program to obtain better execution behavior. A particular effect of these optimizations is that when optimizations move instructions, some program parts originally generated by one dataflow graph operator become intertwined with instructions from other operators. This effect is commonly referred to as operator fusion.

5.1.3 Profiling Tools

To analyze the performance characteristics of complex computer systems and find tuning opportunities, developers typically rely on profiling tools. Some commonly used profilers are *Intel VTune Profiler* [68], *Linux perf* [103], *Flame Graphs* [57] and *OProfile* [133]. These tools output the performance profile of the system software for a given workload, and show the utilization of various micro-architectural hardware features. To do that, profilers use the processor's Performance Monitory Units (PMUs) to collect samples of selected hardware events (e.g., stalled CPU cycles, cache- or TLB-misses, memory accesses, etc.) and map them to the assembly instructions that triggered them. Often, to make the output more user-friendly, the profilers will generate a performance report on a source line- or function- granularity. Recently, Intel introduced the Processor Event Based Sampling mode (PEBS) [67], where the processor itself records the samples and writes them into a dedicated buffer in memory, without raising an interrupt. This significantly improves the precision of the profiling samples and reduces the overhead, as the kernel is only involved when the buffer is full. In such cases, the interrupt handler writes out the samples to memory and clears the buffer for further sampling. In the default mode, PEBS just records the instruction pointer (IP) of the executed instruction at the sampling time-point, but one can also configure it to record the full call-stack. For the rest of this chapter, we will use the default mode unless explicitly stated otherwise.

5.2 PROFILING DATAFLOW SYSTEMS

In this chapter we address the challenge of profiling dataflow systems and getting more value out of the process for all parties involved. Dataflow graphs are a powerful abstraction that allows domain experts express the computation they need to perform on data in the form of graph nodes and edges. The rest of the system stack then absorbs the complexity of computing the optimal execution plan and generating high performant code that is optimized for the underlying hardware platform. As discussed in Section 2.2, very often this compilation and optimization process undergoes a series of transformation steps, each optimizing on a metric that brings us closer to generating fast executable binary. Many of these optimization and transformation steps are designed by developers with different fields of expertise. And at each step, any information that identifies the hotspots and bottlenecks in the system (e.g., where and how the time is spent, how the operations interact with one another, how efficiently the different operations use the underlying compute/memory/IO resources, etc.) would be of great use to developers that want to fine-tune the system.

5.2.1 Shortcomings of Current Tools

However, with the current tools this task is not trivial. Even in the simple(r) case where the dataflow runs on a single machine, does not rely on heavy I/O for data exchange, synchronization and communication, and only uses the CPU (and not offload computation to accelerators) the problem of mapping the low-level profiling detail to higher-level concepts and abstraction layers is a challenge. To understand the problem better, we make the following observations.

PROFILERS ONLY AGGREGATE ON LOW-LEVEL IR First, profilers operate only on the lowest level – the hardware-specific executable and its libraries. As a result, the performance profiles they generate can only aggregate the recorded events on an assembly-level or source-line / function call granularity. While this is useful information for a low-level systems engineer working on compilers and code-gen, the data is too raw for anyone working with higher-level constructs and concepts (cf. Figure 54). These developers would then have to reverse-engineer through multiple-layers of code-generation to find where these instructions belong to, in order to create a holistic picture of the performance profile suitable for their job. For example, the developer of a database query optimizer is used to the concepts constituting a query plan – relational operators and pipelines. He is not interested in the costs of each LLVM IR instruction, but the total costs per operator or an operator pipeline. Having that aggregated knowledge could help him reorder the execution of the operators in a pipeline [127], potentially fuse-operators into one [121, 29] or introduce more pipeline breakers [117]. With the present tool-support, he would have to reconstruct these profiles manually. This step can easily become very involved, ineffective and error-prone. Hence, the more complex

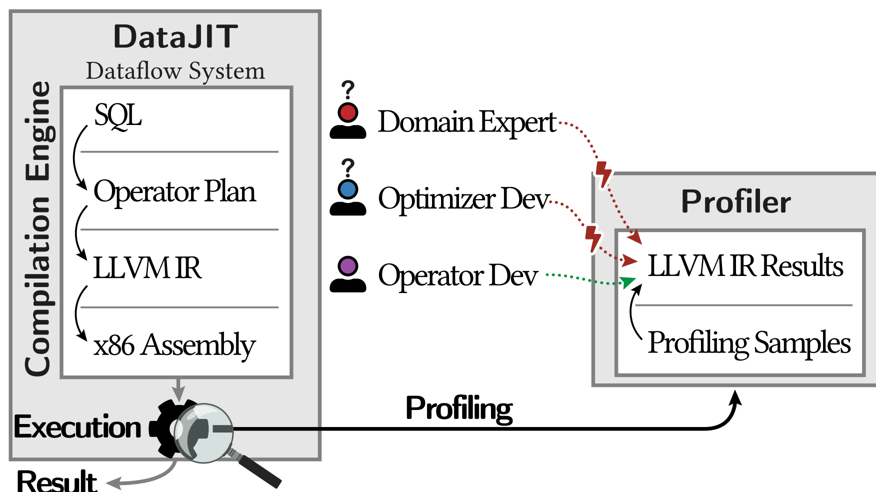
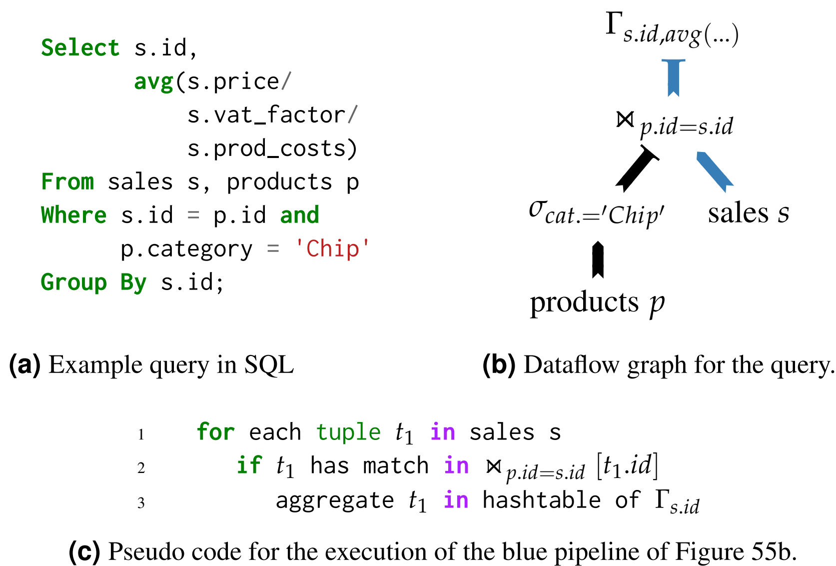


Figure 54: Layers of intermediate representation for the Umbra database system. With today's profilers developers with expertise on different layers must all use profile reports on the lowest IR level.

the system is, the less likely it becomes that someone like the top-level domain expert will be able to identify the most critical component of their dataflow graph.

PROFILING REPORTS OVERALL STATISTICS FOR AN EVENT Second, profilers often fail to leverage the time dimension recorded along with the collected samples. Hence, the developers miss an opportunity to get extra hints on when (in addition to where) the hotspots occur. This is very important for performance tuning pipelines where multiple operations can be active at the same time (e.g., processing data in a pipeline stream). Being able to monitor how efficiently the resources are being used on a time granularity, and correctly attribute them to the exact operation that uses them is not only valuable for the system-stack developers, but also for provisioning resources to different operators at runtime (e.g., for streaming dataflow engines [105]).

MEMORY TRACING IS COSTLY AND DONE BY ANOTHER TOOL Third, low-level profilers are seldomly used to gather memory traces – the set of addresses accessed during the execution of a program. This is unfortunate, as that piece of information can be very valuable to developers. For instance, getting access to which (part of a) data structure was accessed when most of the cache-misses are recorded and by which operation (e.g., that keeps information on the properties of the algorithm's memory access patterns) can help a developer into reorganizing the data in a different data structure, or be more careful about the skew-distribution and data partitioning among the executing threads. Typically, memory tracing is done on a system-level which has two problems: One, it comes with a big performance overhead making it impractical for complex dataflow systems aiming to minimize the job's response time. Two, the output is in a format that maps the frequency of access requests to memory addresses, making it too raw for anyone working on higher abstraction levels. It is again a job left to



(a) Example query in SQL

(b) Dataflow graph for the query.

(c) Pseudo code for the execution of the blue pipeline of Figure 55b.

Figure 55: Example query with corresponding dataflow graph and generated code.

the developer to reconstruct and map how the memory accesses link to the higher-level constructs they operate on.

LACK OF HOLISTIC SOLUTION Lastly, all of the above-identified limitations of existing profilers (and memory tracing tools) are because they operate completely decoupled from the rest of the compilation and optimization process (Figure 54). In fact, the whole focus during the lowering and optimization process is on generating highly optimized code and as a result the system does not keep track of the higher-level concepts. For instance, in the step of lowering a database query plan to LLVM-IR, the codegen produces low-level loops that execute operations and fuses multiple operators together, thereby losing the abstraction concept of operators per-se and the dependency between them. As a result, the profilers cannot re-establish the link because the boundaries of the higher-level constructs are often blurred in the IR of lower layers (e.g., due to operator fusion). This is an important observation as to why profiling dataflows on multiple abstraction layers becomes such a puzzle for any developer.

To make things more clear, let us walk through an example that highlights the different steps needed to identify a potential bottleneck in a code-generating database system.

5.2.2 The Missing Link: An Example

The dataflow system in our example is the relational database system Umbra, which generates machine code to achieve maximum in-memory processing speed. It lowers user requests through a series of optimization layers.

The query in Figure 55a, for example, is first parsed and then internally represented as the dataflow graph in Figure 55b. The dataflow graph is then lowered into imperative

```
1         loopTuples:
2     0%    %localTid = phi [%1, %loopBlocks %2, %contScan]
3   0.1%    %3 = getelementptr int8 %state, i64 320
4   0.1%    %4 = getelementptr int8 %3, i64 262144
5   2.2%    %5 = load int32 %4, %localTid
6   2.3%    %7 = crc32 i64 5961697176435608501, %5
7   1.5%    %8 = crc32 i64 2231409791114444147, %5
8   1.2%    %9 = rotr i64 %8, 32
9   2.3%    %10 = xor i64 %7, %9
10  2.2%    %11 = mul i64 %10, 2685821657736338717
11  1.2%    %12 = shr %11, 16
12  2.4%    %13 = getelementptr int8 %5, i64 %12
13 32.1%    %14 = load int32 %40, i64 %13
14  0.2%    %15 = isnotnull ptr %12
15  0.3%    condbr %15 %loopHashChain %nextTuple
16        loopHashChain:
17  0.1%    %hashEntry = phi [%12, %loopTuples %99, %contProbe]
18  0.2%    %16 = getelementptr int8 %hashEntry, i64 16
19  1.1%    %17 = load int32 %16
20  0.3%    %18 = cmpeq i32 %5, %17
21  0.2%    condbr %18 %else %contProbe
22        else:
23  0.5%    %19 = getelementptr int8 %0, i64 786432
24  2.2%    %20 = load int32 %19, %localTid
25  9.8%    ; ... // load values %22, %24, %26
26  9.5%    %27 = sdiv i32 %22, %24
27  9.6%    %28 = sdiv i32 %27, %26
28  2.9%    %30 = crc32 i64 5961697176435608501, %20
29  2.4%    %31 = crc32 i64 2231409791114444147, %20
30  1.3%    %32 = rotr i64 %31, 32
31  1.4%    %33 = xor i64 %30, %32
32  2.3%    %34 = mul i64 %33, 2685821657736338717
33  1.7%    %35 = and i64 %34, 1023
34  1.9%    ; ... // find entry
35  2.2%    store int32 %20, %37
36  0.2%    %38 = getelementptr int8 %37, %4
37  2.1%    store int32 %28, %38
38           ...
```

Figure 56: Performance profile of the actually generated program in LLVM IR for the marked pipeline of Fig. 55b.

program form. In this case, into LLVM IR, the intermediate representation of the LLVM optimizing compilation framework [94]. LLVM then lowers the IR program down to executable machine code.

Before discussing performance profiles of the generated code, let us shortly inspect the structure of the generated code. The operators of Fig 55b marked in blue form a pipeline of operators that directly pass tuples to each other during execution. Conceptually, the system generates the pseudo-code of Figure 55c, where the scan operator loops over the tuples of the input table (Line 1), passes each tuple to the join operator (Line 2), which in case of a match forwards the tuple to the aggregation operator (Line 3). In reality, however, the system produces the much more detailed LLVM IR shown in Figure 56.

Now, when profiling the example query the profiler will report the results on line- or function-level of the IR program as shown in Figure 56. Each line is annotated with the number of collected samples the profiler attributes to the corresponding source line.

This approximates the execution cost of each instruction. Observe, how this profile view is rather low-level. At first glance, it is apparent that a lot of time is spent on the load instruction in Line 13. However, it takes quite some time and expertise to realize that this instruction implements the directory lookup of the chaining hashtable used in the join operator. Further, it is easy to miss that in total an even higher number of samples belong to the aggregation operator, as those samples are spread out over Lines 23-37. In short, the initial impulse to focus on improving the join operator would miss the fact that the group by operator is the dominating cost factor.

Unfortunately, a report of samples on a function level—as most profilers offer— does not remedy the situation either. Operator fusion tightly couples operators of the whole pipeline into a single function, leaving the function aggregation level too coarse to obtain any useful insights. Additionally, neither the function nor the source level view, though, lend themselves to visualize a time dimension.

5.3 ABSTRACTION APPROPRIATE PROFILING

As shown in the previous section today's profilers present reports mainly on the lowest abstraction level. This covers only a fraction of information needs of the different experts involved in building dataflow systems. Here, we present our profiling approach that caters to everyone involved.

We list the desired features in Section 5.3.1, propose a solution in Section 5.3.2, and present its advantages in Section 5.3.3.

5.3.1 Requirements from an Ideal Profiler

A profiler should report results at a *granularity familiar to the reader* of the report. Specifically, the report should be in terms they already use while interacting with the system. Such terms could be operators from the dataflow graph or vectors, loops, etc. from lower optimization layers.

While these terms can be quite high-level the profiler should *not hide details* due to aggregation. Information that is available in profiling samples, e.g., timestamps, accessed memory addresses etc. should still be presented to the reader.

Beyond the right format, a profiling report should also accurately reflect the actual behavior of the executed computation. That means, first, *association* of samples with high-level components must be *correct*. Second, the sampling *frequency* must be *high* enough to not miss any behavior, e.g., due to aliasing effects. Third, the performance *overhead* of sampling should be *low*, so that the behavior of the profiled process can be observed undisturbed.

In the next section, we present a profiler that meets these demands. Our solution relies on hardware profiling support to supply accurate, low-overhead samples with instruction

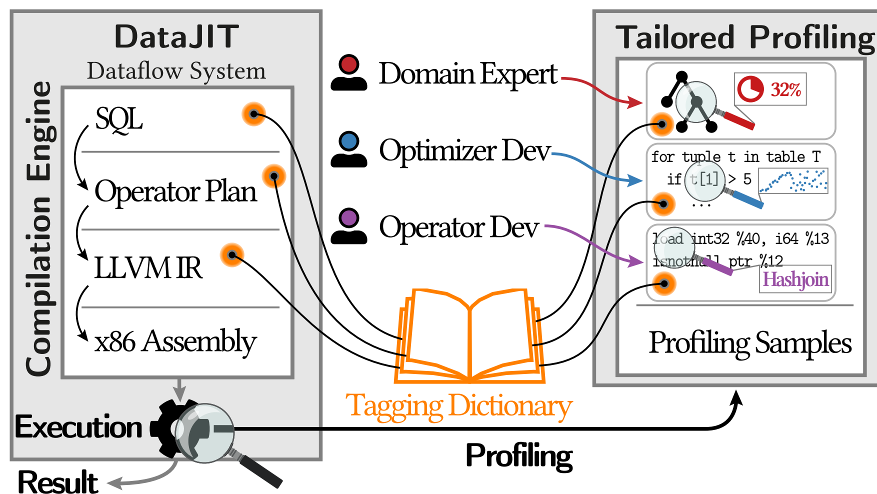


Figure 57: Tailored Profiling requires small extensions to collect a tagging dictionary during code generation and to enable Register Tagging. With this, it can generate high-level performance reports for all parties involved.

pointers and timestamps, and requires that each low-level component can be mapped to exactly one component on the next-higher abstraction level (cf. Section 5.3.2).

5.3.2 Tailored Profiling

With Tailored Profiling we bridge the semantic gap between the low-level results traditional profilers produce and the developers' need for reports on higher abstraction levels. Tailored Profiling supports all requirements listed in Section 5.3.1 and requires no conceptual changes of the dataflow system.

Solution overview

Tailored Profiling solves traditional profilers' shortcomings by tracking the lineage of the low-level IR code generation across the many compilation steps to enable linking the profiling samples to higher abstraction levels. Our approach, illustrated in Figure 57, achieves this by: 1) tracing the links between concepts of the different abstraction levels throughout the lowering process and 2) storing the links in an Tagging Dictionary. After profiling, 3) a post-processing phase uses the dictionary to annotate the collected samples with abstraction information and 4) produce a profile meeting the needs of the selected developer.

Tagging Dictionary

The Tagging Dictionary is populated during the lowering of the dataflow graph and consists of multiple logs, one for each lowering step as illustrated in Figure 58. Each log is filled during its respective lowering phase, and contains an entry for each lower-level

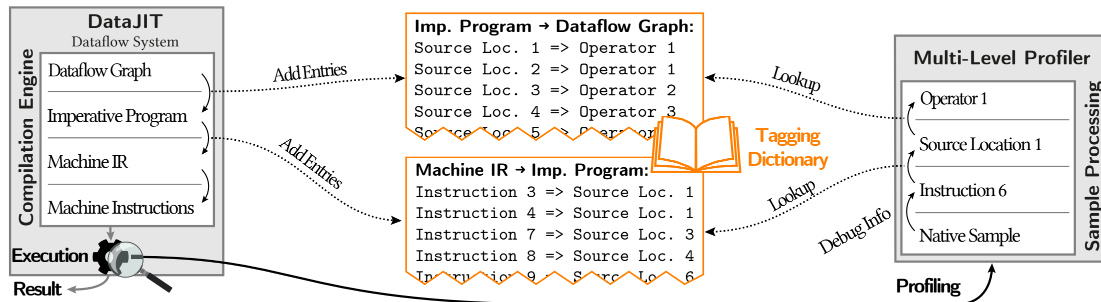


Figure 58: Tailored Profiling applies the Tagging Dictionary to report the profiling results on higher abstraction levels. During the compilation of the query the Tagging Dictionary is created. After the execution the profiler uses the Tagging Dictionary to map the native samples to higher abstraction levels.

component that links it to the corresponding higher-level component. To capture the links, the system's compilation engine keeps track of the currently active component on the higher-level and adds an entry to the Tagging Dictionary whenever a lower-level component is created.

Looking back at the example of Figure 55: Here, the higher-level components are operators in the dataflow graph. Lower-level components are lines of imperative source code, e.g., as shown in Figure 55c. When the dataflow engine lowers the scan operator, the operator generates the for loop of Line 1. Because the engine tracks this process, it notes in the tagging dictionary that source line 1 belongs to the scan operator. During further translation, the engine also tracks that Line 2 belongs to the join operator and Line 3 belongs to group by.

Challenges with Shared Source Locations

The Tagging Dictionary implicitly makes the assumption that every lower-level component is constructed and used by exactly one higher-level component, i.e., every source location in the generated code belongs to exactly one operator in the dataflow graph. With this assumption we can map every profiling sample to one source location and, thus, to exactly one operator.

The assumption is true for most of the generated code, however, it is possible that two operators share a source location. This happens, for example, in Umbra's join operator. It calls a pre-compiled function to insert entries into a hashtable. Two instances of the join operator thus share all source locations of the pre-compiled function. Yet, any given profiling sample must be attributed to only one of the two operators, so we need to disambiguate at shared source locations. This can either be achieved with call-stack-sampling or our novel *Register Tagging* approach.

CALL-STACK INSPECTION. The default approach on how a profiler can disambiguate shared code locations is by using call-stack sampling that records the entire call-stack with each sample. Having the call-stack stored in the Tagging Dictionary

```
...
prevValue = setTag(op1); // set op1 as currently active
insert(); // call shared code location
setTag(prevValue); // reset to previously active op
...
```

Figure 59: Register Tagging uses a processor register to trace the component that calls the shared function. The register is reserved for exclusive use by Register Tagging.

can then help us identify the higher-level component for each function that executes the shared code location. One drawback of the approach is that call-stack sampling suffers either from high performance overhead or is limited to a low sampling frequency (cf. Section 5.5). The positive aspects of it are that it can be applied without any alteration of the generated code and when hardware support for register value sampling is not available.

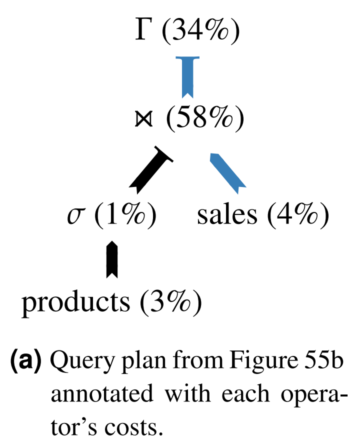
REGISTER TAGGING. As an alternative, we propose a novel light-weight approach that we refer to as Register Tagging. The key idea is to disambiguate the source location by storing an operator tag in a machine register. During sampling, the profiler checks the disambiguation tag when necessary, as modern x86 processors have the ability to record register values along with a profiling sample.

Linking back to our example, where two joins share the function `insert`, just before the first join calls the common function, the Register Tagging would generate code that moves a unique identifier for the first join into the tag register (shown in Figure 59). Note, that on setting the tag in Line 2 we remember the previous value of the tag in order to reset the value after the function call (Line 4). Register Tagging also instructs the compiler to not use the tag register for any other purposes, to avoid overriding the value. Ultimately, when a profiling sample is taken from the `insert` function, the value of the tag register is also captured so that we can uniquely identify the caller.

One set-back for Register Tagging, is that it relies on hardware profiling support to capture register values (and hence would not work for dataflow systems that run in managed runtime like JVM) and is also invasive with respect to the code generation engine. But, the amount of changes to the code generation process are quite small and in return we get a much lower overhead compared to call-stack sampling without compromising accuracy.

Generating Tailored Profiling Reports

Applying Tailored Profiling, the profiler aggregates the samples at the abstraction level meeting the developer's needs. Therefore, the profiler processes the samples and maps them to the needed higher abstraction levels in a bottom-up approach using the Tagging Dictionary as illustrated in Figure 58. For example to map a sample containing `Instruction7` to the dataflow graph the profiler proceeds as as follows: At first the profiler looks up the entry of `Instruction7` in the Tagging Dictionary to map it to



(a) Query plan from Figure 55b annotated with each operator's costs.

```
1        |loopTuples:(tablescan 2.4% hash join 45.7%) |
2        |  ...                                        | hash join

13 32.1% |  %14 = load int32 %40, i64 %13              | hash join
14  0.2% |  %15 = isnotnull ptr %12                    | hash join
15  0.3% |  condbr %15 %loopHashChain %nextTuple       | hash join
16       |loopHashChain:              (hash join 1.9%) |
17  0.1% |  %hashEntry = phi [%12, %loopTuples...]     | hash join
18  0.2% |  %16 = getelementptr int8 %hashEntry, ...   | hash join
19  1.1% |  %17 = load int32 %16                       | hash join
20  0.3% |  %18 = cmpeq i32 %5, %17                    | hash join
21  0.2% |  condbr %18 %else %contProbe                | hash join
22       |else:                       (group by 50.0%) |
23  0.5% |  %19 = getelementptr int8 %0, i64 786432    | group by
24  2.2% |  %20 = load int32 %19, %localTid            | group by
25  9.8% |  ; ... // load values %22, %24, %26         | group by
26  9.5% |  %27 = sdiv i32 %22, %24                    | group by
27  9.6% |  %28 = sdiv i32 %27, %26                    | group by
28            ...
```

(b) Excerpt of the performance profile from Figure 56 extended using the data from the Tagging Dictionary. Note, the percentages are based only on the samples of the blue pipeline.

Figure 60: Tailored Profiling provides the profiling reports on developers' abstraction levels.

its imperative program component, which is `Source Location 3`. Then, the profiler can lookup the dataflow graph operator of `Source Location 3` in the corresponding log of the Tagging Dictionary to map the sample to `Operator 2`. For samples containing shared instructions, the profiler first retrieves the unique calling instruction either from Register Tagging or the call-stack-sampling. Then, the profiler can just apply the Tagging Dictionary for the call instruction to map the sample to the needed higher abstraction level as described in the example.

5.3.3 Benefits of Tailored Profiling

To show the advantages and practical impact of our approach, let us revisit the example from Section 5.2.2 and apply Tailored Profiling this time.

For domain experts the profiler maps the samples to the dataflow graph, in this example the query plan, and aggregates them per operator as shown in Figure 60a. The domain expert can then inspect the annotated query plan to learn about the costs of each operator, derive decisions to reconfigure the database system and fine tune SQL queries.

For the optimizer developer the operator plan is also a familiar abstraction. They can compare the profiling results of different query plans for the same query to evaluate the cardinality estimates of the optimizer and refine the query plan optimizations.

The operator developer—even though they are familiar with the low-level results of the IR program—still benefits from Tailored Profiling. The profiler enriches the profiling results as shown in Figure 60b. It annotates each instruction with its operator and

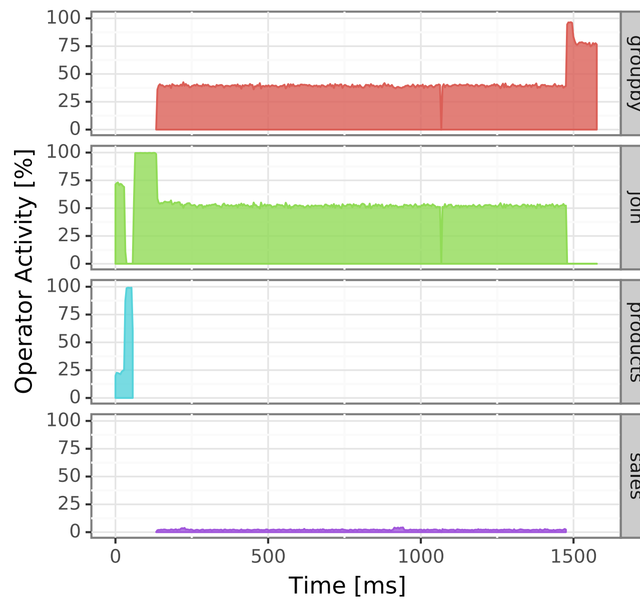


Figure 61: Tailored Profiling associates each sample with an operator and thus determines operator activity over the query runtime.

aggregates the costs of each operator on different granularities, e.g., on basic blocks and functions. Thereby, the costs of each operator is provided as a frame of reference to avoid missing expensive operations distributed across multiple instructions.

Aggregating to appropriate levels enables an additional, cross-cutting feature. The components from each level provide an ideal base to visualize the performance profile over time. For example, the profiler can show operator activity over time, as shown in Figure 61. The operator developer can inspect this to learn about the interaction between operators and detect temporal hotspots. Then they can use the profiler to narrow down on the next lower abstraction level, i.e., limit the results to the time interval of the hotspot. With visualization over time developers can pinpoint bottlenecks that would otherwise be hidden in aggregation.

5.4 INTEGRATION WITH UMBRA

We implemented Tailored Profiling in the compiling data-flow system *Umbra* to demonstrate its feasibility and advantages. In this section we discuss implementation details of our prototype.

Umbra is a high-performance relational database system, which compiles queries with data-centric code generation based on the produce & consume model [121, 123, 117, 191]. Umbra's query engine is implemented in C++ and lowers dataflow graphs through LLVM IR to machine instructions. Thus, the engine runs queries by executing native instructions, which allows the profiler to directly use hardware features, such as PEBS, to collect samples.

5.4.1 Correspondence Tracing

As introduced in Section 5.3.2, Tailored Profiling requires to keep track of high-level constructs in the compilation phase. In case of Umbra, the compilation phase lowers a relational algebra operator tree to an imperative program in LLVM IR. This translation step is implemented in the produce & consume model. Keeping track of high-level constructs (i.e., relational algebra operators) is integrated into the translation step.

In produce & consume, each operator is responsible to generate the code that implements the operator functionality. When operators are composed into an operator tree, e.g., as shown in Figure 55b, they need to pass tuples amongst each other. This happens through the interface of `produce` and `consume` functions. An operator can ask its input operator to produce tuples, i.e., generate code to produce tuples, by calling the input's `produce` function. The input generates code to prepare a tuple, then passes the tuple to the operator's `consume` function.

Within this code generation process, we always keep track of the operator that currently generates code. The active operator only changes when either produce or consume is called, thus on entry of either function, we set the active operator to the called operator and reset to the previous value on exit. Then, whenever an instruction is generated in the LLVM IR program, we add an entry to the Tagging Dictionary to associate the instruction with the active operator.

Even through the described procedure seems to require many changes to the code generator, this is not necessarily the case. In Umbra, for example, produce, consume, and instruction generation are all funnelled each through a single code location, which we use to update the active operator and the Tagging Dictionary.

5.4.2 Register Tagging

Umbra applies Register Tagging to attribute samples of shared code locations to their correct operator. The system therefore guards each call to a shared code location with inline assembly instructions that execute the tagging.

Let us pick up the example from Figure 59 to show how it works. Umbra includes the `insert` into the generated code of an operator by generating a function call instruction. Register Tagging is applied by adding inline assembly instructions implementing `setTag` before and after the call instruction. These inline assembly instructions extract the register's previous value and write the tag into the register.

The system ensures only Register Tagging alters the used register by removing it from allocation in the compilers. Umbra itself is compiled with `gcc` and the system uses LLVM compiler framework [94] to lower the generated code from LLVM IR to native instructions. For `gcc` the system reserves the register using the `ffixed` flag and we have modified the LLVM compiler framework to exclude it as well. Only the inline assembly instructions of Register Tagging can therefore access the register.

5.4.3 Precise Timestamps for Profiling Samples

Tailored Profiling requires profiling samples with a reliable timestamp to report results with a time dimension. Umbra therefore uses the Linux kernel's perf API [104] to record profiling samples with PEBS.

However, the sample timestamps provided by the Linux kernel seem to not represent the sampling time point correctly which we observed in initial experiments. As workaround, we directly extract the processor's internal timestamp counter (TSC) from the samples from the PMU [67]. The TSC has cycle-grained resolution and is already collected in PEBS samples of processors since Skylake, though currently dropped by the Linux kernel during sample formatting. We therefore modified the Linux kernel with a workaround to include the TSC in the formatted samples and convert it to *ns* using a kernel module [15].

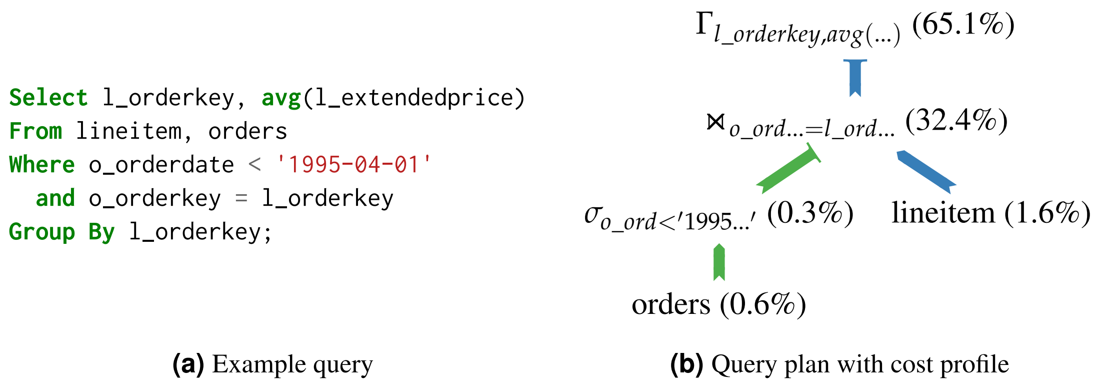
5.5 EVALUATION

In this section we evaluate the advantages of Tailored Profiling as well as its accuracy and runtime overhead.

Tailored Profiling's major feature is to produce profiling reports at the right abstraction level for the developer, which is hard to quantify and very subjective. Thus, instead of success metrics, we show the value of Tailored Profiling with use cases for different users. Afterwards, we evaluate its accuracy and the induced overhead in Sections 5.5.3 to 5.5.4.

5.5.1 Experimental Setup

We use the TPC-H benchmark [182] with a scale factor of 1 (dataset size 1 GB) for the use-cases, and scale factor 10 (dataset size 10 GB) to measure performance and accuracy. We execute all queries single-threaded with Umbra. The use-cases are executed on a machine with an Intel Core i7-7700K running at 4.2 GHz (turbo boost of 4.5 GHz), 32 GB DRAM and Ubuntu 19.10. The performance experiment test machine has an Intel Core i9-9900X with 3.5 GHz (turbo boost of 4.4 GHz), 64 GB DRAM and Ubuntu 20.04. We use Linux perf version 5.2 [103] to profile with PEBS, disable sample throttling and transfer the samples to Tailored Profiling with `perf script`. To profile costs and operator activity, we use the event `INST_RETIRED.PREC_DIST` and record a sample every 5000 events. For memory access patterns, we capture a sample every 1000 loads of the `MEM_INST_RETIRED.ALL_LOADS` event.



(a) Example query

(b) Query plan with cost profile

Figure 62: Tailored Profiling can aggregate samples up to query plan level—a concept database users are familiar with.

5.5.2 Use Cases

To show the utility of Tailored Profiling, let us present a number of use cases where Tailored Profiling is employed on different abstraction levels and by engineers in different roles.

Domain Expert

In the first use case, a user of Umbra investigates why the query from Figure 62a runs slower than expected.

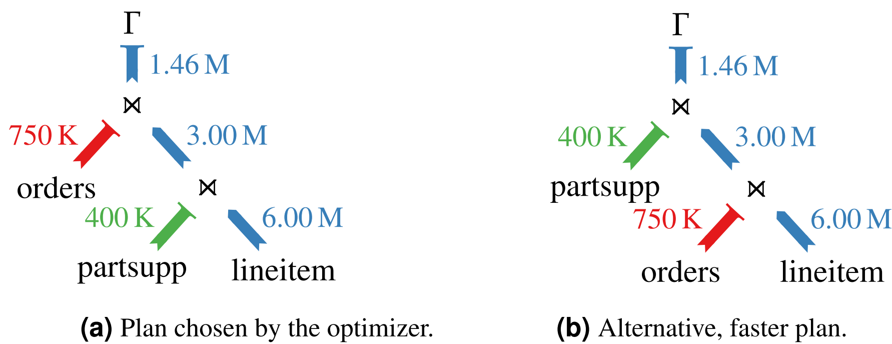
At a familiar abstraction level, Tailored Profiling enables the user to view how much compute time each operator takes, as shown in Figure 62b. Here, they can quickly grasp the overall execution plan for the query. The report reveals that 65% of the runtime are spent in the aggregation operator and 32% in the join operator.

To speed up the query, the user can now make an informed decision on whether to, e.g., introduce index structures to reduce the cost of the join computation. Alternatively, they may decide to take computational shortcuts and add a sampling operator in order to reduce the number of tuples that reach the aggregation operator.

Note that most database systems have a feature that seemingly offers the same view. The explain analyze command counts how many tuples each operator processes and visualizes the statistics in an operator tree. However, even though the tuple count is a decent approximation, our sampling approach captures the actual time spent in each operator.

Optimizer Developer

As a second use case, we inspect the work of an expert in the Umbraquery optimizer. They investigate the performance of a query with the two alternative plans, as shown in Figure 63. Both plans have identical intermediate result size, so with the standard cost function the optimizer could choose either plan. Choosing the left one (Figure 63a)



(a) Plan chosen by the optimizer. **(b)** Alternative, faster plan.

Figure 63: Alternative query plans for the optimizer developer's SQL query from Section 5.5.2.

seems like a good option as the query plan first probes the smaller hashtable (expecting fewer cache-misses) which will consequently reduce the number of tuples that also probe the (more expensive) larger hashtable. Yet, this results in a slower runtime than the alternative.

As this is counter-intuitive, the developer wants to identify the cause and refine the cost function. The developer thus applies Tailored Profiling to inspect the operator activity over time in the probing pipeline (cf. Figure 64). The report confirms that the alternative plan is faster. Moreover, starting at 70 ms in the alternative plan the join on orders becomes dominant while in the original plan becoming negligible. After this hint, further investigation reveals that lineitem is scanned in the order of the join attribute, which leads to a situation where first the join on orders finds a match for all tuples and passes them to the next operator. Then, starting at 70 ms, the join on orders eliminates all tuples so the hashtable for partsupp is not probed at all, yielding an overall behavior that is easy to predict by branch predictors, which is especially beneficial for

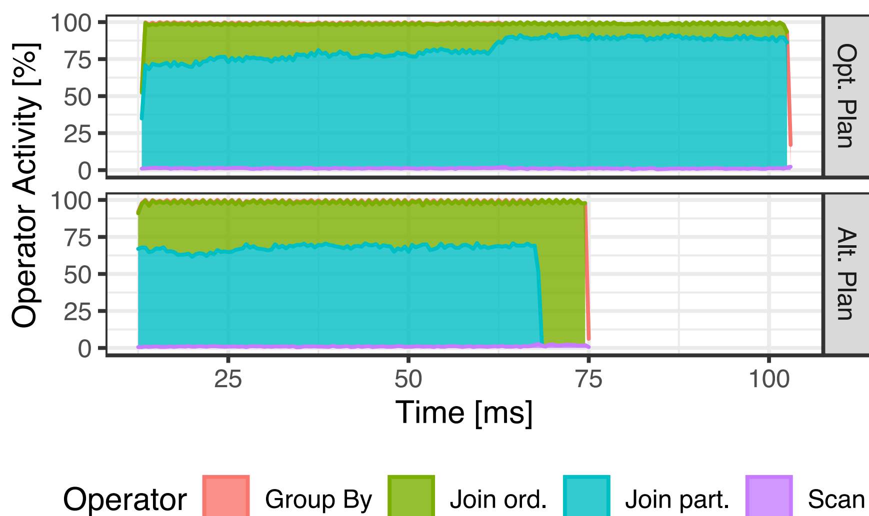


Figure 64: Operator activity over time for the plans of Figure 63.

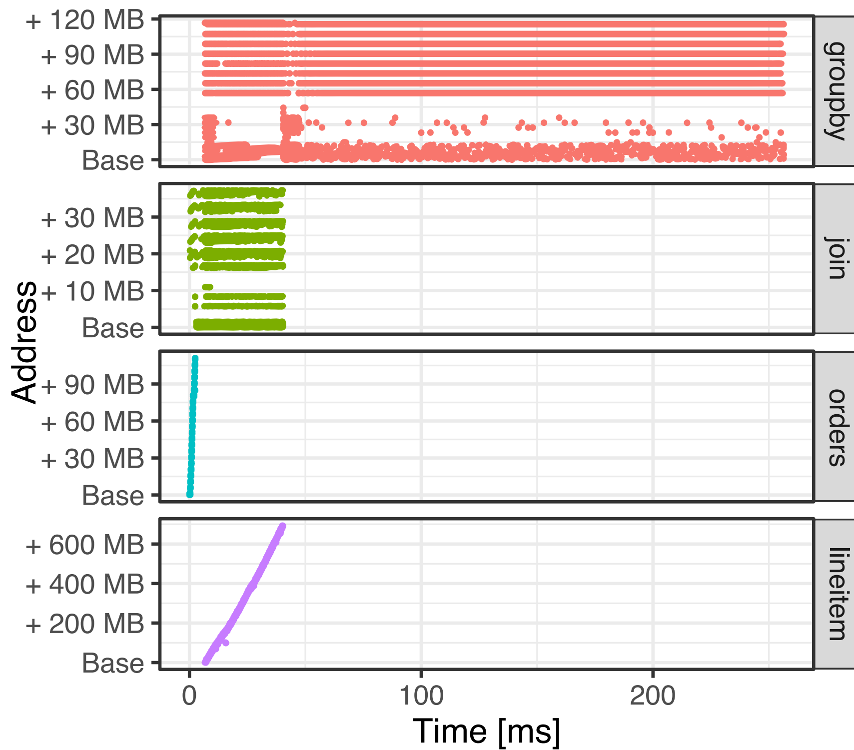


Figure 65: Profile of memory access patterns for the operators of Figure 62b. Each point denotes a sample with time (from query start) and accessed address (offset from lowest address the operator accesses).

hardware with out-of-order execution capabilities. The optimizer developer can now decide whether to extend to cost function with such data-layout and hardware-specific properties.

Operator Developer

In the first use case, we have seen how a user of the database system can get a higher-level overview on the performance of the query (recall Figure 62). An operator developer, who is responsible for implementing efficient operators, needs a more detailed view of the internals. Very often they are interested in the data access patterns, which can play a big role on the actual performance of the algorithm.

Tailored Profiling makes use of hardware sampling support to also record the addresses with every memory access. With the Tagging Dictionary the instruction that initiated the memory access can be associated with an operator, and as a result we can get an accurate memory access profile for each operator (Figure 65). The operator developer can inspect the memory profile and compare it to their expectations. In this example, the table scans on orders and lineitem show a linear data access pattern over time, which is ideal for hardware prefetchers etc. The join and group by operators access memory in a more widespread fashion, as a result of using hashtables in their implementation.

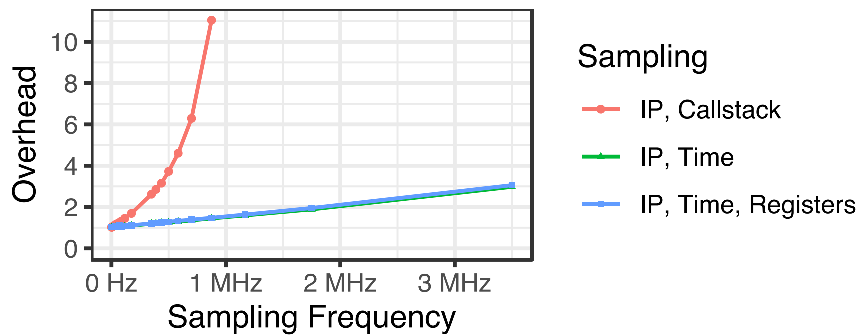


Figure 66: Performance overhead of the profiling approaches for TPC-H query 16.

This can be used as a starting point for further investigation, e.g., into a memory access profile with cache-miss information, or for considering alternative operator algorithms.

5.5.3 Runtime Overhead

Our approach to Tailored Profiling incurs three sources of runtime overhead:

First, while profiling, the hardware sampling mechanism stores samples in a memory buffer, which occasionally must be flushed by the operating system. Figure 66 shows how the sampling overhead increases with sampling frequency. At our default setting of taking a sample every 5000 cycles (0.7 MHz) the overhead is 35%.

Second, the amount of information included in the samples potentially increases the overhead. Figure 66 also shows the overhead for additionally sampling register values, as required for Register Tagging. When sampling every 5000 cycles the overhead grows to 38%. Call-stack sampling—the alternative to Register Tagging—incurs an overhead of 529%. In comparison, the overhead of Register Tagging is moderate.

Third, reserving one register for Register Tagging slows down query execution, as the compiler generates worse code. On average over all 22 TPC-H queries, we observed an overhead of 2.8%.

Further overhead from profiling occurs in form of storage space required for the recorded samples and the Tagging Dictionary. Samples with IP, timestamp, and register values require 54 Bytes (when adding call-stack information 265 Bytes). Thus, at a sampling frequency of 0.7 MHz we need to store 77 MB per second. Each entry in the Tagging Dictionary is a triple of operator, pipeline, and source line, which we represent with 24 Bytes. With one triple per LLVM IR instruction—of which there are on average ∼1320 in a TPC-H query—the dictionary requires ∼30 kB.

Overall, we observe that the induced overhead is rather low, as we never encountered any interference of profiling overhead with query execution and the performance profiles are very plausible.

Attribution	Amount of Samples
Umbra	98.0%
→ Operators	95.4%
→ Kernel Tasks	2.6%
No attribution	2.0%

Table 13: Amount of samples attributed to Umbra by Tailored Profiling over all TPC-H queries.

5.5.4 Accuracy

To validate the correctness of Tailored Profiling reports, we check the accuracy of our approach and evaluate the accuracy of the samples recorded by PEBS.

To test the accuracy of the profiling reports, we profiled all 22 TPC-H queries with Tailored Profiling and report the amount of samples covered by the Tagging Dictionary's mapping in Table 13. The experiment shows that our approach can attribute 98% of the samples to Umbra's higher abstraction levels and the Kernel (e.g., for memory allocations). Further investigation reveals that the remaining 2% belong to other system libraries, for which we did not apply Register Tagging.

An astute reader may have already observed that the Tailored Profiling can only attribute samples correctly when the sampled instruction pointer is accurate. We cross-checked the sampled instruction pointers with Register Tagging by applying the tagging not only for shared code locations, but also for all instructions in generated code. Our test over all TPC-H queries yields no mismatches, thus, the instruction pointer matches the Register Tagging for all samples. Furthermore, we evaluate the sample accuracy empirically by profiling the query execution for different profiling events. We cross-checked whether the instruction pointers in the samples occur at instructions that could plausibly cause the sampled event, e.g., samples for load-misses always point to loads and branch-misses contained either the branching instruction or the preceding compare causing the misprediction.

Finally, we evaluate the accuracy of the sampled timestamps for Tailored Profiling's time dimension. For this, we profiled the query execution taking a sample every 5000 cycles and check the TSCs of consecutive samples. In our experiment, the TSC values reflect the sampling distance (max. deviation $\sim$40 cycles) and adapt accordingly when we vary it. Ultimately, Tailored Profiling's timing information depends on the accuracy provided by the hardware. In our experience, for TSC based time appears to provide a precise resolution reflecting the samples' recording time.

Overall, our validation yields very small inaccuracies and confirms the reliability of Tailored Profiling's reports and time dimension.

Component	Lines Added	Lines Before
Umbra Code Gen.	56	$\sim 22,000$
Tailored Profiling	1,686	0
$\rightarrow$ Sample Processing	1,176	0
$\rightarrow$ Visualization	510	0
Σ	1,742	$\sim 22,000$

Table 14: Lines of code of our prototype implementation of Tailored Profiling.

5.5.5 Implementation Effort

Integrating our approach is lightweight and requires only small additions to the dataflow system as shown in Table 14. Tailored Profiling leverages existing profilers to record samples and processes the profiling samples with the Tagging Dictionary to map them to higher abstraction levels.

We, thus, need to add the Tagging Dictionary mechanism and Register Tagging into the dataflow system and populate the Tagging Dictionary during the lowering process, as shown Figure 57. Integrating the dictionary into Umbra required only 50 lines of code, while the Register Tagging needed 6 lines. The main implementation effort went into mapping the profiling samples to higher abstraction levels, followed by creating the visualizations of the developer tailored views. Modifying the kernel for samples with TSC timestamps needed just 1 line of code, and reserving a register in the LLVM compiler framework took only 2 lines.

PORTABILITY. Porting our approach to a different compiling dataflow system requires minor effort: adding the Tagging Dictionary mechanism into the system, creating a dictionary log for each lowering step and, depending on the runtime environment, either integrating Register Tagging or using call-stack sampling. The most critical part would be that the reports created by Tailored Profiling will need to be adapted to the system's abstraction levels.

CONFIGURATION TRADE-OFF. Depending on the dataflow system's runtime environment and requirements, one can either rely on using callstack sampling or Register Tagging. Some dataflow systems that run on managed runtimes (e.g., Spark on JVM) can primarily rely on callstack sampling, while others can decide on the trade-off between profiling resolution and reserving machine registers.

To make that decision we need to consider the number of lowering steps that the system employs without a unique mapping between the higher- and lower-level components. For each of those lowering steps, Register Tagging requires one exclusive register for disambiguation, which comes with a performance overhead. Thus, we need to make the trade-off between reserving more registers or switching to call-stack sampling.

5.6 RELATED WORK

Most research on profiling and work on profilers focuses on software that is compiled ahead of time (i.e., without compilation at runtime), for example vTune, Hotspot, Linux perf, HPCToolkit etc. [68, 2, 188, 103]. Their frame of reference for a performance profile is software source code. Consequently, they present profiles in terms of assembly, source lines, and function calls. Hotspot and vTune also offer an interactive view to zoom in on function specific profiles or time intervals. Users can also choose which hardware events to record and view. Further, there are profilers built to analyse specific events. The Intel PIN tool, for example, monitors memory bandwidth utilization and Noll et al. presented a profiler to visualize memory access patterns [79, 128, 110].

Meanwhile, hardware vendors constantly improve the selection of events available for profiling, increase the accuracy, and reduce the overhead [69]. How these improvements translate into practice is constantly investigated [130, 38, 5, 129, 42]. Overall, there are a large variety of events available to profilers for which the TMAM method offers a guideline for categorization and interpretation [190, 169].

These profiling tools are already useful and will most likely even improve further in the future. For a large number of dataflow systems [191, 119, 114, 24, 1, 141], which increasingly use compilation techniques to achieve high performance [95, 29, 64, 136] having the power of profiling tools is essential for informed tuning decisions. Previous approaches to the problem include manual analysis of profile components to attribute samples to operators [128], replaying execution in a simulator [180], tracking memory allocations to map samples to data-structures [144], and call-stack sampling within the Java virtual machine [176]. All of these approaches, however, fall short of providing a universal operator mapping that works for any abstraction level and can at the same time be sampled with low overhead and sufficient frequency to show behavior over time.

5.7 SUMMARY

For profiling, extensive hardware support is available with a large selection of events and data to record. Such input is essential to make well-informed decisions when profiling dataflow systems. However, we showed that current profiling technology is not able to adequately present performance profiles to fit the needs of everyone involved in building and using dataflow systems.

As a solution, we advocate to use Tailored Profiling, an approach that tracks high-level query concepts through dataflow systems' compilation pipeline and leverages the tracked information to post-process profiling samples. Tailored Profiling raises the abstraction level of profiling reports to the concepts readers of those reports are familiar with and, thus, can interpret reports easier and more accurately. Interestingly, we found that raising the abstraction level clears the way to adding more information to profiling reports while keeping them understandable. For example, including a time dimension

for the operator activity or visualizing data access patterns is possible with Tailored Profiling.

FUTURE WORK Our research into Tailored Profiling focused on analyzing software running on a single CPU. Interesting further avenues of research are cases in which the software is not running, e.g., because it is blocked waiting for I/O or waiting for a lock to be released. Further, integrating samples from multiple servers or compute accelerators may pose challenges in unification and visualization. Another issue is, that in our implementation users of the profiler still need to be well versed in hardware behavior and choose appropriate analyses. Even though Tailored Profiling offers predefined views, it might be interesting to create an automatic user guidance, akin to TMAM [190], which determines problems and bottlenecks automatically per operator and over time.

6 DEBUGGING COMPILING QUERY ENGINES

Excerpts of this chapter have been published in [77].

With the advent of in-memory databases, high-bandwidth solid state drives, and recently also persistent memory [61], high-performance relational query execution engines compile machine code for query execution. This approach creates optimal code for each query and thus makes best use of available computing resources [75]. Consequently, code generating execution engines are able to make the most of the large available bandwidth.

Query execution in a compiling query engine is done in a two-step process (cf. Figure 67). First, the engine generates code for the query plan. Second, the machine's processors execute this code to compute the query result [121]. For the developer of a compiling engine this two-step process can become a challenge. When, during development, they find their computation results are wrong, they need *debugging tools* to efficiently triangulate the cause of the fault.

Conventional debuggers support the search of errors by allowing the developer to stop the execution at any point. The developer can then inspect the program state, view the value of variables, explore data structures, and examine the call-stack to decide whether the observed behavior is as expected or already affected by an error. To make this process efficient, the debugger should show the developer a full view of the program state in the source language and the format that the developer wrote it. In other words, the debugger should present the state in terms the developer is familiar with.

In a compiling query engine, however, this integrated experience is not possible with a regular debugger. A compiling engine splits the query execution into the two phases shown in Figure 67: *Compile time*, which generates code for a query plan and compiles it to machine instructions, and *runtime*, which runs the machine instructions to produce the query result. To debug this two-level setup, most toolchains already offer the means to step through either the code generator or the runtime code. However, the *link* between the generated code and the source code that generated it, is *missing*. Without the link the developer is missing most of the query context.

Currently, there are two limitations that cause this disconnect: First, current debuggers are not built for this kind of debugging. GDB, for example, supports only to *stop at one position* in the machine code and map that position to one source location. There is currently no support to handle a second source location that generated the first source location. Second, as generating code and running it is a two-step process, there is a *lifetime* issue between multiple source locations. When the debugger stops a multi-level program, it can map the current program state to a source line on the first level. Mapping also to a second source line that generated the first source line is difficult, because

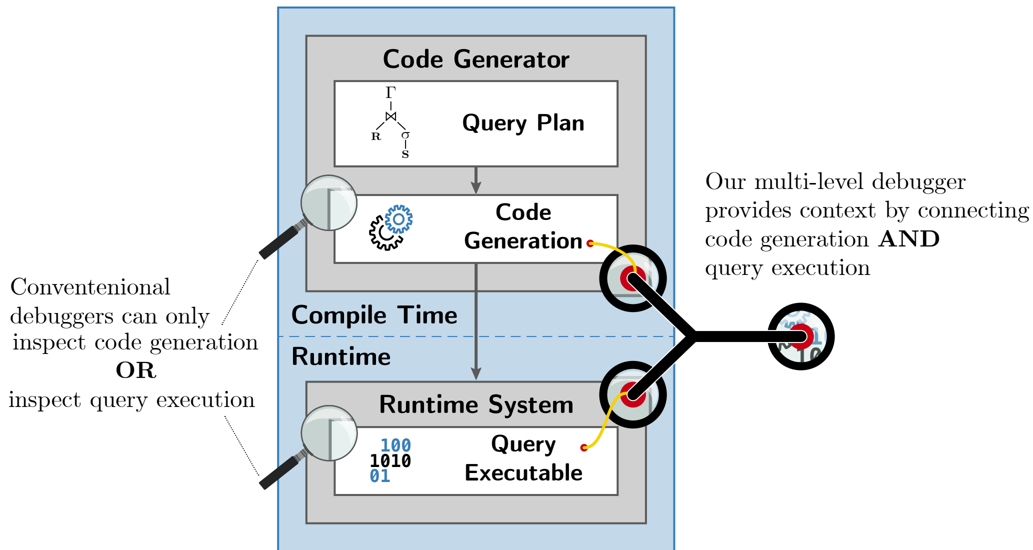


Figure 67: Compiling relational engines process queries in two steps: *Code generation and execution. Conventional debuggers can only attach to one step, so that debugging execution misses lots of context information. Our multi-level debugger provides this context.*

the second source line was executed much earlier in time. That means that the current program state does correspond to the first source line, but not to the second. Therefore, the debugger can't use the program state to inspect the call-stack and variables for the second source line. To this day, we are not aware of any debuggers that fully bridge this gap.

In this chapter, we present how to build a *multi-level debugger* that can reconnect an arbitrary number of source levels and fully inspect the program state at any level. This allows us to provide the required context at any point in the program and thus significantly boost developer productivity. Our solution is built in large parts from *existing* debugging technology, so implementing it for any mature compiler-debugger toolchain is only a *small development effort*. We propose to use a time-travelling debugger to bridge between generated code and generating code and to use unique markers during code generation to reliably perform the connection.

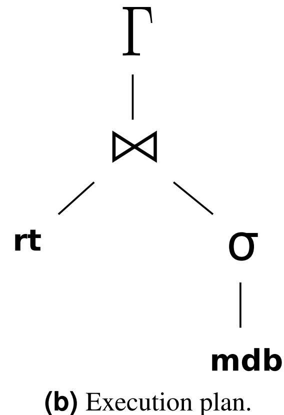
We show that our approach is feasible by implementing it for the Umbra database system [123]. During the development of Umbra's query engine the multi-level debugger setup has proven immensely useful.

6.1 INTERACTIVE DEBUGGING

Locating the root cause of a failure in a relational query engine works much the same as in any other large code base. A developer first tries to isolate the smallest scenario

```sql
1 select count(*)
2 from
3   RotatingTomatoes rt,
4   MovieDatabase mdb
5 where
6   rt.name = mdb.name and
7   rt.rating = mdb.rating and
8   mdb.reviews > 10;
```

(a) Query – How many movies receive the same rating in both sources?



(b) Execution plan.

Figure 68: Example query with execution plan.

that exhibits the *erroneous behavior*. Then, they create and refine hypotheses about the cause of the failure and accept or reject them based on observations. This way they trace back from the observably wrong behavior to the root cause. To execute that process efficiently the developer requires debugging tools that can stop the program at a location and observe variables, data-structures, the call-stack, etc.

To show how this process can be applied to a relational query engine and to introduce our proposed tooling, we use a *running example*: Assume that there are two sources of movie ratings, Rotating Tomatoes and the Movie Database. Our example query in Figure 68a counts how many movies receive the same rating in both sources. Also, it only considers movies with more than 10 reviews in the Movie Database. Unfortunately, our example database system returns a wrong result for this query. It returns $count = 0$, even though through inspection of the data set we found a movie that fulfills the criteria.

As a first step to find the fault, we check whether the database frontend works correctly. We find that it produces the reasonable execution plan shown in Figure 68b. Therefore, we decide to search for the error in the execution of that plan (as opposed to in the creation of the plan).

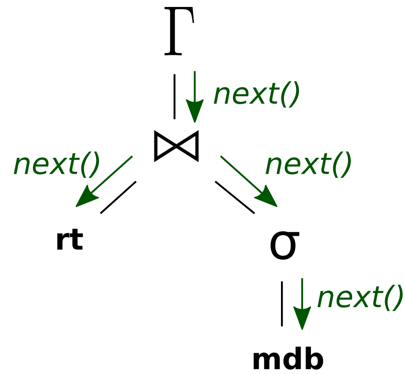
In the remainder of this section, we discuss the process and information required for a debugging workflow to find such errors. First, we examine how to debug an execution engine that is built as a Volcano-style interpreter. Here we show how debugging an execution engine should work and which context should be available. Second, we contrast that workflow with debugging an execution engine built with code generation. We show that context information is lost between compile-time and runtime and propose a solution to reconstruct it for debugging purposes.

6.1.1 How Debugging Should Work: Volcano-style Interpreter

Conventional debuggers are already well suited to debug query execution engines that are built as Volcano-style interpreters. In this section we show how the debugging workflow works and the context information that is available.

```
1 Tuple JoinOperator::next()
2  hashTable.buildFrom(leftChild)
3  # Probe with tuples from right side
4  while(right = rightChild.next())
5   for(left in hashTable.find(right))
6    yield left.concat(right)
```

(a) The join operator in a Volcano-style interpreter retrieves tuples from left and right child, passes matches to parent.



(b) Tuple passing between operators by `next()` calls.

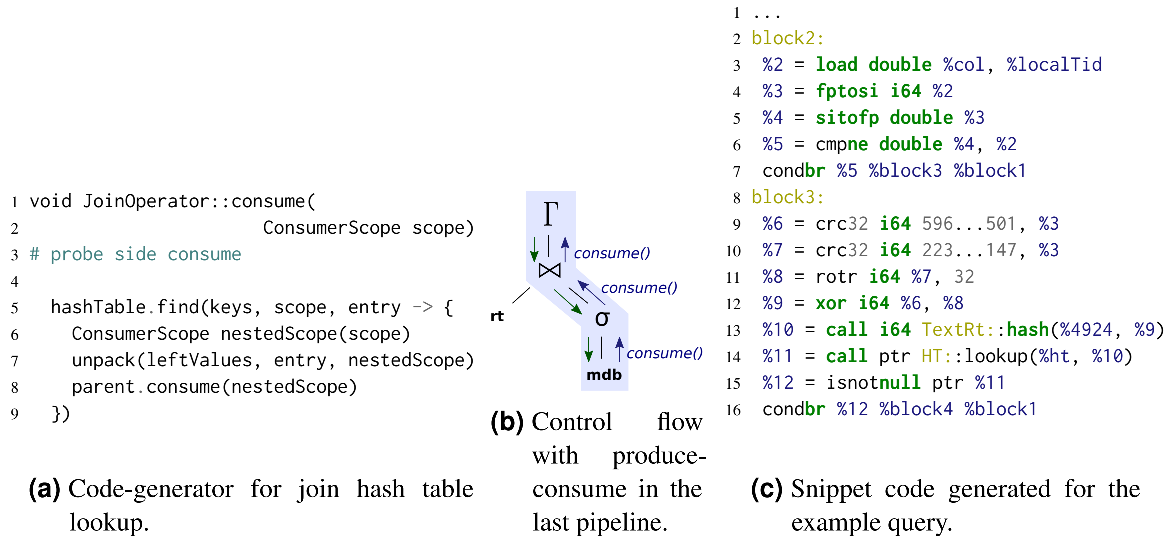
Figure 69: Control flow of Volcano-style query processing and implementation of the hash join operator.

In a Volcano-style interpreter, the execution plan is represented in an object-oriented fashion as *tree of operators* [56]. These operators execute the query plan and are, thus, also well suited for conventional debugging. Figure 69b shows such an operator tree for the example plan. The execution of the plan is coordinated through a small *iterator interface* between operators. Each operator calls `next()` on its child operators to receive the next tuple. When the call returns, the operator performs its own work and passes the tuple on. In this manner, tuples are passed between operators until the query result is computed.

This happens, for example, in the `next()` implementation of the hash join operator in Figure 69a. First, the operator builds a hash table for all the tuples from its left child operator. Second, the operator iterates all the tuples from the right child. For each, it searches matching tuples in the hash table and passes any matches on to the parent operator.

When a developer searches for an error, e.g., for the query plan in Figure 69b, they can *attach a debugger* to the database executable. They can set a breakpoint, e.g. in the hash join operator, so that the debugger stops right in the operator code. By using the debugger's stepping features they can follow a single tuple through multiple operators. At any point when the debugger stops at a breakpoint the developer is able to *inspect variables and data structures* that the operator uses for query processing. This lets the developer check whether the actual query execution still matches their expectation. Furthermore, the debugger allows to unwind the call-stack and thus not only inspect the current operator, but also operators higher up in the query plan. That context helps to understand the current step and allows the developer to decide whether the current program state is still ok or already affected by the error.

Recall that in the example query an error causes *count* to be zero. A good starting point for debugging might be to set a breakpoint in the hash join in Line 5. Once the debugger stops there, we could *inspect* the `hashTable` and check whether it contains any tuples. If so, we can also look at some of those tuples and check if the placed data



(a) Code-generator for join hash table lookup.

(b) Control flow with produce-consume in the last pipeline.

(c) Snippet code generated for the example query.

Figure 70: Code generation with produce-consume fuses all operators of a pipeline into one function.

is ok. In case it is, we could then decide to step into the hash table's `find` function and investigate further. A debugging workflow as just described is already well supported by conventional debuggers, e.g., gdb, lldb, Visual Studio debugger, etc.

6.1.2 Debugging Code Generating Engines

In contrast to Volcano-style interpreters, compilation-based engines execute a query plan in a two-step process. This results in high query execution speed, but also entails that the previously described debugging workflow—stepping through the operators—is not possible.

Background: Code Generation and Execution

In the first step—called *compile time*—the execution engine *generates code* for the query plan and compiles it to machine code. In Umbra we generate a custom intermediate representation, modelled after LLVM IR, that we call Umbra IR and which we will use in our example. The architecture we use for code generation is the produce-consume method [121]: To generate code for an operator tree the topmost operator calls `produce()` on its child operators. The response from the child operator is that eventually it calls back the operator's `consume()` function and passes an input tuple. Here, the operator has access to the input tuple and generates code to process it. After the code generation, the code is passed to a compiler to produce natively executable machine code.

When again applying this to the example of a hash join we get the implementation of Figure 70a. The `consume()` function gets a scope in which it can find all the values that previous operators produced. It uses these to generate a hash table lookup with the

join keys. In this example, the `hashTable` takes care of generating code for hashing the keys, lookup, etc. All matching hash table entries are then passed to the lambda function in Lines 6 to 8. Our implementation then takes the values from the found entry, puts them into a nested scope, and passes them on to the next operator. In contrast to interpretation based engines, we traverse the operator tree during code generation and, instead of directly processing tuples, we produce code to process tuples.

As the second step—called *(query) runtime*—the generated code is *executed* to process tuples and compute the query result. In this step all the effort invested at compile time pays off through high-speed execution of native code.

Missing Debug Context at Query Runtime

Due to this two-part process, the debugging situation in a code generating query engine is very different. We can use a conventional debugger to place breakpoints in the generated code, e.g., in Umbra we can place breakpoints in Umbra IR, *step through the IR program* and inspect the values. Figure 70c shows an excerpt of code that the hash join operator generates for the example query. Here, we could set a breakpoint Line 3 where data is loaded from memory. By stepping to the next line with the debugger, we can then trace the execution and print values, but we can *only guess* which operator generated the instructions and what the values mean.

Generally, this method can work for an *expert developer* who knows the code generator very well and is familiar with what the generated code usually looks like and which patterns usually occur. In that case, stepping through Umbra IR only helps to find the most obvious programming mistakes. However, if the fault is caused by a more complex interaction of operators, the IR quickly becomes a *confusing* place. The experience of debugging Umbra IR that is generated from a query plan is very much similar to stepping through x86-assembly that was generated from C++, but without any debug information to link the assembly to C++ source lines. Furthermore, newcomers to a code generating project lack the experience to read and understand the rather low-level intermediate representation and it can represent a *high entry barrier*.

Reconstructing Context

What this situation calls for is that the developer gets context information about the operator that generated the code and which specific purpose it serves. All that context is available in the code generation phase. Unfortunately, due to the two-step process, the *context information* is at query runtime *no longer available* to a conventional debugger. In this situation we believe that the debugging experience can be greatly improved by providing developers with the necessary context when debugging generated code. Ideally, the same information about context, operators, and variables as when debugging a Volcano-style interpreter should be available.

To make this possible and supply the necessary context at query runtime, we propose to build a *multi-level debugger*. The required components are a time-travelling debugger and unique identifiers that map instructions in the generated code back to the code

generation. The time-travelling debugger allows us to record the program execution and to replay code generation as often as necessary. With unique identifiers we can navigate to the generation of specific instructions in the replay. That is, we can stop the replay when, e.g., instruction 5 (Line 6) is appended to the program.

To put it all together, during query runtime we can set breakpoints and step through generated code with a conventional debugger. If at any line of generated code we need to understand which operator generated it and why, we use the time-traveling debugger to replay the recording of the code-generation process up to exactly where the line is generated. This *reconstructs* the exact program state during code generation and we can inspect it with all the usual debugging tools so that we can explore *all the required context*.

Debugging the Example Query

We can use this approach to debug the example query: We set a breakpoint in the generated code (as previously) in Line 3 of Figure 70c. The debugger breaks at that position and we start stepping line by line to reach the bottom part of the snippet where the instructions seem related to a hash table lookup. Unfortunately, execution does not reach to that point. The conditional branch in Line 7 always branches to `%block1` and thus never continues to `%block3`. At this point, we need to decide whether that behavior is ok, but we don't understand why the branch is there and what the comparison in Line 6 should accomplish.

In order to get the missing context information we start an additional debugger session with the time-travelling debugger and replay to the point where Line 6 is generated. By unwinding the call-stack from there, we observe that the join operator is currently performing a hash-table lookup (Line 5 in Figure 70a). Next, we go down in the call-stack and learn that the hash table currently collects the join keys to compute a hash from them. But why are there two floating-point conversions in the code? We inspect the join keys that are just being hashed and learn that they are of type string and double! A short check of the other side of the join reveals that those join keys are of type string and integer.

Going down one more stack frame into the function that collects keys for hashing, we learn that the rating value is casted from double to integer in order to compute the hash for the equality comparison. The cast implementation performs one cast to integer and another cast back to double. Only when the round-trip cast gives the same value as the original double value for rating there can be a join partner from the left side. Otherwise, the double value is outside the domain of integers. From the current position on the call-stack we also learn that Line 6 is generated for the comparison of round-trip casted value to the original value and that the comparison result is stored in a variable named `outsideDomainIndicator`. We can then use the time-travelling debugger session to step forward and follow the uses of `outsideDomainIndicator`. This way, we learn that the hash table uses it to skip the lookup. The current code performs a lookup when the

value is outside the domain, however, it should be the other way around. This is easily fixed, e.g., by negating the indicator.

We have seen in this process, that generated code can be rather low-level and piecing together what it should accomplish can be like solving a puzzle. Thus, the ability to connect the generated code back to the code generator is an invaluable tool to debug complicated cases.

6.2 EVALUATION

In this section we check the following hypotheses:

- Creating a multi-level debugger using time-travelling debuggers is feasible.
- The effort to implement such a solution is low.
- The runtime overhead of the time-travelling debugger is acceptable for database system development.

6.2.1 Multi-level Debugging for Umbra

We implemented the proposed solution for Umbra [123], our code-generating database system. Umbra is written in C++ and generates Umbra IR as intermediate representation. We use the LLVM compiler framework to generate optimized machine code from Umbra IR. Our existing infrastructure uses LLVM's debug information mechanisms to attach debug information to the machine code. This already enables us to use a debugger, e.g., the GNU debugger gdb, to stop the program at query runtime, step through the generated code, and print variables from Umbra IR.

To extend this setup for multi-level debugging, we employ Mozilla's RR debugger [132]. RR is a deterministic time-travelling debugger based on gdb. It can record a program execution, in this case how Umbra processes a query, and replays it any number of times exactly as during the recording. During a replay it offers all features of gdb, e.g., breakpoints, printing and stepping. We chose RR because it is readily available and light-weight, but other time-travelling debuggers may also be used for this.

As RR is based on gdb we extended RR through gdb's Python interface. We implemented a `goto-instruction` command. It takes one instruction identifier (from the generated code) as argument and replays execution to the point where the instruction is generated. The command's core is a temporary conditional breakpoint:

```
1 gdb.execute("tb IRProgram.cpp:972 if ip == " + instructionId)
```

This sets a breakpoint at the source location where instructions are appended to Umbra's intermediate representation. The condition on the breakpoint ensures that the debugger only stops when the requested instruction is generated (otherwise it would stop at every instruction).

```
dump3.uir
269            %4892 = add i64 %4846, %4378 # { "instructionId": 4892}
270            %4906 = select i64 %4878, %4892, %4846 # { "instructionId": 4906}
271            %4924 = builddata128 d128 %4828 %4906 # { "instructionId": 4924}
272            %4938 = getelementptr int8 %4571, i64 1048576 # { "instructionId": 4938}
273            %4960 = load double %4938, %localTid # { "instructionId": 4960}
274            %4982 = getelementptr int8 %state, i64 320 # { "instructionId": 4982}
275            %5004 = fptosi i32 %4960 # { "instructionId": 5004}
276            %5014 = sitofp double %5004 # { "instructionId": 5014}
277            %5024 = cmpne double %5014, %4960 # { "instructionId": 5024}
278            %5038 = not bool %5024 # { "instructionId": 5038}
>279            condbr %5038 %then0 %cont8 # { "instructionId": 5048}
280
281          then0:
282            %5066 = zext i64 %5004 # { "instructionId": 5066}
283            %5076 = crc32 i64 5961697176435608501, %5066 # { "instructionId": 5076}
284            %5090 = crc32 i64 2231409791114444147, %5066 # { "instructionId": 5090}
multi-thre Thread 0x7ffff34b3f In: _3_groupby_join_tablescan_movi* L279  PC: 0x7ffff7fbd824
(gdb)
```

```
cts/codegen/algebra/loleop/HashTable.cpp
295                  // Validate the key
296                  auto existingKey = storage.extractKey(result.valueCache);
297                  ConsumerContext::testValuesIs(key, existingKey, collates, htProbe.continueBB)
298
299                  // And pass to callback
300                  callback(result);
301               });
302            };
303
304            if (outsideDomainIndicator.isSet()) {
>305               If checkIndicator(outsideDomainIndicator.lnot());
306               probeLogic();
307            } else {
308               probeLogic();
309            }
310         }
extended-r Thread 18286.18286 In: umbra::codegen::KeyValueHashTable::probeIm* L305  PC: 0x5555574fb717
(rr)
```

Figure 71: GDB on the left, stepping through generated code. RR on the right, providing context from the code generator

With this tooling we can run two debug sessions side-by-side as shown in the screenshot in Figure 71. In the left panel, we use gdb to step through the execution of generated code. In the right panel, we used the `goto-instruction` command to navigate to the source code that generated the code in the left panel. Note that the full context of the generated code is available in the time-travelling session on the right. In the shown example it is possible to unwind the call-stack and reach the implementation of the hash-join operator. It is also possible to go down in the stack to lower abstraction layers of the code generator and observe which instruction is generated. At all positions in the code we can print variables and inspect data structures. Additionally, the debugger offers the ability to step forwards and backwards through the code generation process. This implementation shows that the concept is feasible and in our experience it proved useful for the development of Umbra.

6.2.2 Implementation Effort

To estimate the effort to build a multi-level debugger, let us note that implementing the core functionality—the `goto-instruction` command—only takes 11 lines of Python code. It already gives users the ability to replay to the generation of a specific instruction and provides all the necessary context.

Additionally, we added a convenience feature that, after replaying to the point where one instruction was generated, unwinds the call-stack to the first operator translator. That code location gives a quick overview of where the translation process currently stands and the developer finds operator objects there to inspect.

We can also control RR from other programs to show context information. An http interface exposes the `goto-instruction` command so that it can be triggered from outside RR. For example it can be controlled from a gdb session that debugs the query runtime or from a text editor where a developer inspects the generated code.

When also accounting for these additional features, the Python plugin to RR has 74 lines of code. After the initial investigation and development of the core ideas for the multi-level debugger, the implementation took less than a week of work. Given this short time and how short the implementation is, we conclude that the overall effort to build such a tool is rather low.

6.2.3 Runtime Overhead

In the default setup our multi-level debugger uses RR to record the whole process of query execution. It records query parsing, optimization, code generation, compilation, and execution. In order to be able to replay that exact behavior, RR must record also all the data that is loaded from disk, thus write a copy of it into a recording file. Obviously, in the context of a database system this can amount to large volumes of data, which ultimately impacts the recording and replaying speed.

For example running TPC-H query 1 at scale factor 1 with Umbra generates 467 MB of recorded data. The runtime without recording is 1 second, wheres with recording it is 10 seconds, so the introduced overhead is a factor of 10x slow-down. This large overhead may be acceptable for some tricky debugging cases, where the full power of RR's deterministic replay features are actually helpful.

However, we find that if we want to use the multi-level debugger as a fast-paced tool that is quick to provide feedback to developers it is sufficient to only create a recording of the code generation process with RR. Afterward we start a new debugging session with the conventional debugger gdb to step through the execution quickly and use the previous recording to provide context. That approach generates a smaller recording of 254 MB and only takes 2 seconds. In the majority of cases we found the latter technique to be adequate, as the two program runs perform the exact same operations. Thus the RR recording supplies accurate information at a low runtime overhead.

6.3 RELATED WORK

6.3.1 Debugging Relational Code Generators

In a comparison of interpreting and compiling query engines we observed that a major difference between the approaches is that interpreters are debuggable with conventional tools [75].

Kohn et al. propose as an approach to debug compiling query engines to collect information about the call-stack at compile time [85]. This information can then be used in a purpose-built debugger. It executes the generated code with a virtual machine, can step through it, and uses the collected call-stack information to provide context from compile time. Consequently, the available context is limited to the collected information. It is not possible to inspect variables and data structures from compile time. In contrast, our approach is more comprehensive, as all context that is available at compile time is also available for debugging. It is even possible to step through compile time code while debugging runtime code. Furthermore, our approach is easier to build as it reuses available tools and when those are improved it is directly reflected in our debugger.

Another approach to debugging compiling query engines is presented by Tahboub et al. with the LB2 system [177]. Although LB2's primary goal is not ease of debugging, it offers an elegant way around the two-phase problem of code generators. LB2 uses

extensions to the Scala compiler to instantiate a (somewhat) Volcano-style interpreter and also a code generator for relational queries from the same source base. That means that implementing an operator once yields an interpreter and a compiler. Conveniently, debugging can thus be performed mostly in the interpreter with conventional debuggers. This approach is an excellent idea, however, the reported code generation times of LB2 are on average 300× longer than those we observe in Umbra. Thus, for reasons of practicality, we stay with our C++ and LLVM based approach instead of switching to Scala, and make use of our multi-level debugger.

6.3.2 Time Travel Debuggers

Recording program execution and replaying the exact execution deterministically was an active research field for at least two decades. The proposed record and replay techniques are powerful, yet the user must consider certain trade-offs between available techniques. Engblom provides a comprehensive overview and classification [41].

One group of techniques works in user-space and replays execution at the machine level, thus they are simple to deploy. PinPlay [143], iDNA [14], UndoDB [183] and TotalView ReplayEngine [53] use binary instrumentation to track data coming in from outside the bounds of recording. RR [132], on the other hand, intercepts system calls to record their effects and traps certain non-deterministic instructions. As this approach does not account for inter-thread data races, RR forces execution to use only a single thread at each point in time, thus slowing execution of highly parallel programs.

Other approaches to record and replay include extending language runtime environments [6], frameworks [22] or libraries [59], OS Kernel support [173, 89, 13], and replayable virtual machines [39, 189]. However, these solutions are too intrusive or heavy-weight for a multi-level debugger.

6.4 FUTURE WORK

Our current multi-level debugger implementation with RR already serves us well, yet certain aspects can still be improved. Using RR snapshots for the `goto-instruction` command may enable us to jump to instruction creation instead of replaying from the start. Furthermore, to reduce the overhead of RR, especially when handling large databases, it may be interesting to move code generation to a separate process and only record that.

6.5 SUMMARY

We showed that debugging compiling query engines with the currently available tools can be a lengthy and involved process. We identified that the main issue of debugging

code generators is that at query runtime essential context information from query compile time is missing. This makes debugging a relational code generator a daunting task for newcomers and experts alike.

As a solution, we proposed to build a multi-level debugger that supplies the necessary context. It facilitates a more efficient debugging process and also can also serve as an exploratory tool for beginners. We showed how to build a multi-level debugger from existing technology with low engineering effort and proved its feasibility as Umbra's debugger.

7 CONCLUSIONS AND FUTURE WORK

For the space of high-performance analytical relational query engines, we have shown that on today's hardware and for today's workloads engine-architectures based on vectorization and based on data-centric code generation are roughly on par from a performance perspective. As we believe compilation based approaches are more versatile and better equipped for future computation-intensive workloads, we contributed solutions to the design's shortcomings. Tidy Tuples and the Flying Start compilation backend have shown that very fast code generation and compilation is possible, almost hiding compilation overhead completely. With the pipeline optimizer we demonstrated how automatic, hardware and data-specific tuning can significantly improve the structure of generated programs and, thus, use hardware resources even more efficiently. Finally, Tailored Profiling and multi-level debugging point the way to more productive user and developer tools and increase the utility of data-centric code generation for production systems and software engineering teams.

The presented findings and solutions also pave the way for further interesting avenues of research. The Flying Start compiler has shown that fast compilation is possible, yet our implementation is tailored to the x86 instruction set. When porting to the ARM instruction set, a large fraction of the infrastructure, such as register allocation etc., can be reused. However, implementing instruction translation manually is still an engineering effort. An interesting avenue of inquiry would be to automatically generate a fast compiler from a semantic description of the input language and the target instruction set. This could significantly reduce the maintenance effort.

The buffer optimizer uses a cost model for super-scalar out-of-order processors to weigh optimization alternatives in data-centric pipelines. It might be interesting to apply this cost function to optimizations in other data processing languages, e.g., to the Voodoo vector algebra [146]. Similar to data-centric pipelines it could benefit from precise hazard prediction.

In multi-level debugging we proposed to use time-travelling debuggers to replay code generation and execution simultaneously. The recording unit of the time travelling debugger can, however, become overwhelmed by queries on large amounts of data. It may instead be worthwhile to investigate ways for selective recording or coordinating recording with database internal snapshots to enable cooperative fast forward to keep the debugging process lightweight and fast.

Finally, this theses does not cover how to integrate micro-adaptivity [154] to compiling engines. There are, however, interesting adaptivity mechanisms in the NoisePage database system, which allow runtime adaptation of the initial query plan [116]. An interesting idea might be to combine their proposed probabilistic counters and adaptivity strategies with Flying Start's ability to quickly generate code for alternative execution

plans or update existing programs with variations more suitable to the current data characteristics.

BIBLIOGRAPHY

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *OSDI*. 2016, pp. 265–283.

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. "HPCToolkit: Tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 6 (2010), pp. 685–701.

[3] Advanced Micro Devices. *Software Optimization Guide for AMD Family 19h Processors*. `https://www.amd.com/system/files/TechDocs/56665.zip`. Nov. 2020.

[4] Sameer Agarwal, Davies Liu, and Reynold Xin. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop*. "`https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html`". 2016.

[5] Soramichi Akiyama and Takahiro Hirofuchi. "Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis". In: *ROSS@HPDC*. 2017, 3:1–3:8.

[6] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, and John M. Vlissides. "A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications". In: *IPDPS*. 2001, p. 23.

[7] Kamil Anikiej. "Multi-core parallelization of vectorized query execution". `http://homepages.cwi.nl/~boncz/msc/2010-KamilAnikijej.pdf`. MA thesis. University of Warsaw and VU University Amsterdam, 2010.

[8] Panagiotis Antonopoulos et al. "Azure SQL Database Always Encrypted". In: *SIGMOD*. 2020, pp. 1511–1525.

[9] Raja Appuswamy, Angelos Anadiotis, Danica Porobic, Mustafa Iman, and Anastasia Ailamaki. "Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads". In: *PVLDB* 2 (2017), pp. 121–134.

[10] David F. Bacon et al. "Spanner: Becoming a SQL System". In: *SIGMOD*. 2017, pp. 331–343.

[11] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware". In: *ICDE*. 2013, pp. 362–373.

[12] Rudolf Bayer and Edward M. McCreight. "Organization and Maintenance of Large Ordered Indices". In: *Acta Informatica* (1972), pp. 173–189.

[13] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. "Deterministic Process Groups in dOS". In: *OSDI*. 2010, pp. 177–191.

[14] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. "Framework for instruction-level tracing and analysis of program executions". In: *VEE*. 2006, pp. 154–163.

[15] Trail of Bits. *A tsc_freq_khz Driver for Everyone*. `https://github.com/trailofbits/tsc_freq_khz`. June 2019.

[16] Peter Boncz, Thomas Neumann, and Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark". In: *TPCTC*. 2013.

[17] Peter Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *CIDR*. 2005.

[18] Peter A. Boncz, Orri Erling, and Minh-Duc Pham. "Advances in Large-Scale RDF Data Management". In: *Linked Open Data - Creating Knowledge Out of Interlinked Data - Results of the LOD2 Project*. 2014, pp. 21–44.

[19] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. "Breaking the memory wall in MonetDB". In: *Commun. ACM* 12 (2008), pp. 77–85.

[20] Peter A. Boncz, Wilko Quak, and Martin L. Kersten. "Monet And Its Geographic Extensions: A Novel Approach to High Performance GIS Processing". In: *EDBT*. 1996, pp. 147–166.

[21] Paul G. Brown. "Overview of sciDB: large scale array storage, processing and analysis". In: *SIGMOD*. 2010, pp. 963–968.

[22] Brian Burg, Richard Bailey, A. J. Ko, and Michael D. Ernst. "Interactive record/replay for web application debugging". In: *UIST*. 2013, pp. 473–484.

[23] Victoria Caparrós Cabezas and Markus Püschel. "Extending the roofline model: Bottleneck analysis with microarchitectural constraints". In: *IISWC*. 2014, pp. 222–231.

[24] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 4 (2015), pp. 28–38.

[25] Craig Chasseur, Yinan Li, and Jignesh M. Patel. "Enabling JSON Document Stores in Relational Systems". In: *WebDB*. 2013, pp. 1–6.

[26] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. "Towards Linear Algebra over Normalized Data". In: *PVLDB* 11 (2017), pp. 1214–1225.

[27] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. "Improving Hash Join Performance through Prefetching". In: *ICDE*. 2004, pp. 116–127.

[28] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. “Fractal prefetching B$\pm$Trees: optimizing both cache and disk performance”. In: *SIGMOD*. 2002, pp. 157–168.

[29] Tianqi Chen et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: *OSDI*. 2018, pp. 578–594.

[30] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *EMNLP*. 2014, pp. 1724–1734.

[31] Cliff Click. “Global Code Motion / Global Value Mumbering”. In: *SIGPLAN*. 1995, pp. 246–257.

[32] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. “An Architecture for Compiling UDF-centric Workflows”. In: *PVLDB* 12 (2015), pp. 1466–1477.

[33] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. “Tupleware: "Big" Data, Big Analytics, Small Clusters”. In: *CIDR*. 2015.

[34] Philippe Cudré-Mauroux et al. “A Demonstration of SciDB: A Science-Oriented DBMS”. In: *VLDB* 2 (2009), pp. 1534–1537.

[35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. “Dynamo: amazon’s highly available key-value store”. In: *SOSP*. 2007, pp. 205–220.

[36] “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups”. In: *IEEE Signal Process. Mag.* 6 (2012), pp. 82–97.

[37] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. “Hekaton: SQL server’s memory-optimized OLTP engine”. In: *SIGMOD*. 2013, pp. 1243–1254.

[38] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. “Reliable and efficient performance monitoring in linux”. In: *SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. 2016, pp. 396–408.

[39] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. “ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay”. In: *OSDI*. 2002.

[40] R Kent Dybvig, Robert Hieb, and Tom Butler. *Destination-driven code generation*. Tech. rep. Indiana University Computer Science Department, 1990.

[41] Jakob Engblom. “A review of reverse debugging”. In: *S4D*. 2012, pp. 1–6.

[42] Stéphane Eranian. *Linux perf_events updates*. Scalable Tools Workshop 19. 2019.

[43] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. “A mechanistic performance model for superscalar out-of-order processors”. In: *ACM Trans. Comput. Syst.* 2 (2009), 3:1–3:37.

[44] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. “Interleaved Multi-Vectorizing”. In: *PVLDB* 3 (2019), pp. 226–238.

[45] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. “SAP HANA Database: Data Management for Modern Business Applications”. In: *SIGMOD Rec.* 4 (Jan. 2012), pp. 45–51.

[46] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. “An Application of Recurrent Neural Networks to Discriminative Keyword Spotting”. In: *ICANN*. 2007, pp. 220–229.

[47] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs; An optimization guide for assembly programmers and compiler makers*. `https://www.agner.org/optimize/microarchitecture.pdf`. Sept. 2020.

[48] Craig Freedman, Erik Ismert, and Per-Åke Larson. “Compilation in the Microsoft SQL Server Hekaton Engine”. In: *IEEE Data Eng. Bull.* 1 (2014), pp. 22–30.

[49] Henning Funke, Jan Mühlig, and Jens Teubner. “Efficient generation of machine code for query compilers”. In: *DaMoN*. 2020, 6:1–6:7.

[50] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. “Interval simulation: Raising the level of abstraction in architectural simulation”. In: *HPCA*. 2010, pp. 1–12.

[51] Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. “SystemML: Declarative machine learning on MapReduce”. In: *ICDE*. 2011, pp. 231–242.

[52] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. “GraphX: Graph Processing in a Distributed Dataflow Framework”. In: *OSDI*. 2014, pp. 599–613.

[53] Chris Gottbrath. “Reverse debugging with the TotalView debugger”. In: *Cray User Group Conference*. 2008, pp. 5–8.

[54] Goetz Graefe. “Encapsulation of Parallelism in the Volcano Query Processing System”. In: *SIGMOD*. 1990, pp. 102–111.

[55] Goetz Graefe. “Query Evaluation Techniques for Large Databases”. In: *ACM Comput. Surv.* 2 (1993), pp. 73–170.

[56] Goetz Graefe and William J. McKenna. “The Volcano Optimizer Generator: Extensibility and Efficient Search”. In: *ICDE*. 1993, pp. 209–218.

[57] Brendan D. Gregg. *Flame Graphs*. `http://www.brendangregg.com/flamegraphs.html`. Oct. 2019.

[58] Tim Gubner and Peter Boncz. "Exploring Query Compilation Strategies for JIT, Vectorization and SIMD". In: *IMDM*. 2017.

[59] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. "R2: An Application-Level Kernel for Record and Replay". In: *OSDI*. 2008, pp. 193–208.

[60] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. "Amazon Redshift and the Case for Simpler Data Warehouses". In: *SIGMOD*. 2015, pp. 1917–1923.

[61] Gabriel Haas, Michael Haubenschild, and Viktor Leis. "Exploiting Directly-Attached NVMe Arrays in DBMS". In: *CIDR*. 2020.

[62] Clemens Hammacher. `https://v8.dev/blog/liftoff`. 2018. URL: `https://v8.dev/blog/liftoff` (visited on 07/09/2019).

[63] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 8 (1997), pp. 1735–1780.

[64] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. "Green-Marl: a DSL for easy and efficient graph analysis". In: *ASPLOS*. 2012, pp. 349–362.

[65] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. "PGX.D: a fast distributed graph processing engine". In: *SC*. 2015, 58:1–58:12.

[66] `https://stackoverflow.com/questions/36932240/avx2-what-is-the-most-efficient-way-to-pack-left-based-on-a-mask`. 2016. URL: `https://stackoverflow.com/questions/36932240/avx2-what-is-the-most-efficient-way-to-pack-left-based-on-a-mask`.

[67] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals*. `https://software.intel.com/en-us/articles/intel-sdm`. May 2020.

[68] Intel. *Intel VTune Profiler*. `https://software.intel.com/en-us/vtune`. 2020.

[69] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. `https://software.intel.com/content/dam/develop/public/us/en/documents/64-ia-32-architectures-optimization-manual.pdf`. May 2020.

[70] H. V. Jagadish et al. "TIMBER: A native XML database". In: *VLDB J.* 4 (2002), pp. 274–291.

[71] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. "Just-In-Time Data Virtualization: Lightweight Data Management with ViDa". In: *CIDR*. 2015.

[72] Alfons Kemper and Guido Moerkotte. "Access Support Relations: An Indexing Method for Object Bases". In: *Inf. Syst.* 2 (1992), pp. 117–145.

[73] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *ICDE*. 2011, pp. 195–206.

[74] Timo Kersten. `https://github.com/TimoKersten/db-engine-paradigms`. 2018. URL: `https://github.com/TimoKersten/db-engine-paradigms` (visited on 07/01/2018).

[75] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. "Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask". In: *PVLDB* 13 (2018), pp. 2209–2222.

[76] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra". In: *VLDB J.* (2021), pp. 883–905.

[77] Timo Kersten and Thomas Neumann. "On Another Level: How to Debug Compiling Query Engines". In: *DBTest@SIGMOD*. 2020, 2:1–2:6.

[78] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman P. Amarasinghe. "Cimple: instruction and memory level parallelism: a DSL for uncovering ILP and MLP". In: *PACT*. 2018, 30:1–30:16.

[79] Andi Kleen. *pmu tools*. `https://github.com/andikleen/pmu-tools`. May 2020.

[80] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. "Building Efficient Query Engines in a High-Level Language". In: *PVLDB* 10 (2014), pp. 853–864.

[81] Petr Kobalicek. `https://github.com/asmjit/asmjit`. 2014. URL: `https://github.com/asmjit/asmjit` (visited on 07/23/2019).

[82] Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. "Asynchronous Memory Access Chaining". In: *PVLDB* 4 (2015), pp. 252–263.

[83] 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". In: *VLDB J.* 2 (2014), pp. 253–278.

[84] Andre Kohn, Viktor Leis, and Thomas Neumann. "Adaptive Execution of Compiled Queries". In: *ICDE*. 2018.

[85] Andre Kohn, Viktor Leis, and Thomas Neumann. "Making Compiling Query Engines Practical". In: *IEEE Trans. Knowl. Data Eng.* (2019).

[86] Tim Kraska. "Northstar: An Interactive Data Science System". In: *PVLDB* 12 (2018), pp. 2150–2164.

[87] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. "Generating code for holistic query evaluation". In: *ICDE*. 2010, pp. 613–624.

[88] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *NeurIPS*. 2012, pp. 1106–1114.

[89] Oren Laadan, Nicolas Viennot, and Jason Nieh. "Transparent, lightweight application execution replay on commodity multiprocessor operating systems". In: *SIGMETRICS*. 2010, pp. 155–166.

[90] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *SIGOPS* 2 (2010), pp. 35–40.

[91] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. "Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation". In: *SIGMOD*. 2016, pp. 311–326.

[92] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. "SQL Server column store indexes". In: *SIGMOD*. 2011, pp. 1177–1184.

[93] Per-Åke Larson et al. "Enhancements to SQL Server column stores". In: *SIGMOD*. 2013, pp. 1159–1168.

[94] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *CGO*. Mar. 2004, pp. 75–88.

[95] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: A Compiler Infrastructure for the End of Moore's Law". In: *CoRR* (2020).

[96] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 11 (1998), pp. 2278–2324.

[97] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age". In: *SIGMOD*. 2014, pp. 743–754.

[98] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. "LeanStore: In-Memory Data Management beyond Main Memory". In: *ICDE*. 2018, pp. 185–196.

[99] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". In: *ICDE*. 2013, pp. 38–49.

[100] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. "Efficient Processing of Window Functions in Analytical SQL Queries". In: *PVLDB* 10 (2015), pp. 1058–1069.

[101] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Query Optimization Through The Looking Glass, and What We Found Running the Join Order Benchmark". In: *VLDB J.* (2018).

[102] Edo Liberty et al. "Elastic Machine Learning Algorithms in Amazon SageMaker". In: *SIGMOD*. 2020, pp. 731–737.

[103] Linux. *Linux perf*. `https://github.com/torvalds/linux/tree/master/tools/perf`. Apr. 2020.

[104] Linux. *perf_event_open(2)*. `http://man7.org/linux/man-pages/man2/perf_event_open.2.html`. Feb. 2020.

[105] Xunyun Liu and Rajkumar Buyya. "Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions". In: *ACM Comput. Surv.* 3 (2020), 50:1–50:41.

[106] Zhen Hua Liu, Beda Christoph Hammerschmidt, and Douglas Mcmahon. "JSON data management: supporting schema-less development in RDBMS". In: *SIGMOD*. 2014, pp. 1247–1258.

[107] Irene Lopatovska and Harriet Williams. "Personification of the Amazon Alexa: BFF or a Mindless Companion". In: *CHIIR*. 2018, pp. 265–268.

[108] Raymond A. Lorie. "XRM - An Extended (N-ary) Relational Memory". In: *IBM Research Report* (1974).

[109] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning in the Cloud". In: *PVLDB* 8 (2012), pp. 716–727.

[110] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *SIGPLAN*. 2005, pp. 190–200.

[111] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. "Generic Database Cost Models for Hierarchical Memory Systems". In: *PVLDB*. 2002, pp. 191–202.

[112] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. "Optimizing Main-Memory Join on Modern Hardware". In: *IEEE Trans. Knowl. Data Eng.* 4 (2002), pp. 709–730.

[113] Stefan Manegold, Peter A. Boncz, and Niels Nes. "Cache-Conscious Radix-Decluster Projections". In: *PVLDB*. 2004, pp. 684–695.

[114] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. "Differential Dataflow". In: *CIDR*. 2013.

[115] Wolfgang Meier. "eXist: An Open Source Native XML Database". In: *NODe*. 2002, pp. 169–183.

[116] Prashanth Menon, Amadou Ngom, and Andrew Pavlo Lin Ma Todd C. Mowry. "Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling". In: *PVLDB* 2 (2020), pp. 101–113.

[117] Prashanth Menon, Andrew Pavlo, and Todd Mowry. "Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last". In: *PVLDB* 1 (2017), pp. 1–13.

[118] Guido Moerkotte. *Building Query Compilers*. `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`.

[119] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: a timely dataflow system". In: *SOSP*. 2013, pp. 439–455.

[120] Thomas Neumann. "Efficient generation and execution of DAG-structured query graphs". PhD thesis. University of Mannheim, 2005.

[121] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *PVLDB* 9 (2011), pp. 539–550.

[122] Thomas Neumann. *Linear Time Liveness Analysis*. 2020. URL: `http://databasearchitects.blogspot.com/2020/04/linear-time-liveness-analysis.html` (visited on 08/12/2020).

[123] Thomas Neumann and Michael J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance". In: *CIDR*. 2020.

[124] Thomas Neumann and Viktor Leis. "Compiling Database Queries into Machine Code". In: *IEEE Data Eng. Bull.* 1 (2014), pp. 3–11.

[125] Thomas Neumann, Viktor Leis, and Alfons Kemper. "The Complete Story of Joins (in HyPer)". In: *BTW*. 2017, pp. 31–50.

[126] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems". In: *SIGMOD*. 2015, pp. 677–689.

[127] Thomas Neumann and Bernhard Radke. "Adaptive Optimization of Very Large Join Queries". In: *SIGMOD*. 2018, pp. 677–692.

[128] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. "Analyzing memory accesses with modern processors". In: *DaMoN*. 2020, 1:1–1:9.

[129] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. "On the Applicability of PEBS based Online Memory Access Tracking for Heterogeneous Memory Management at Scale". In: *MCHPC@SC*. 2018, pp. 50–57.

[130] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. "Establishing a Base of Trust with Performance Counters for Enterprise Workloads". In: *USENIX ATC*. 2015, pp. 541–548.

[131] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. "The Star Schema Benchmark and Augmented Fact Table Indexing". In: *TPCTC*. 2009, pp. 237–252.

[132] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. "Engineering Record and Replay for Deployability". In: *USENIX ATC*. July 2017, pp. 377–389.

[133] *OProfile*. `https://oprofile.sourceforge.io/`. 2019.

[134] John K. Ousterhout et al. "The case for RAMClouds: scalable high-performance storage entirely in DRAM". In: *SIGOPS* 4 (2009), pp. 92–105.

[135] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures". In: *ICDE*. 2001, pp. 567–574.

[136] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. "Weld: A Common Runtime for High Performance Data Analysis". In: *CIDR*. 2017.

[137] Shoumik Palkar et al. "Evaluating End-to-End Optimization for Data Analytics Applications in Weld". In: *PVLDB* 9 (2018), pp. 1002–1015.

[138] Mike Pall. `http://wiki.luajit.org/Optimizations`. 2012. URL: `http://wiki.luajit.org/Optimizations` (visited on 07/28/2019).

[139] Mike Pall. `http://wiki.luajit.org/SSA-IR-2.0`. 2013. URL: `http://wiki.luajit.org/SSA-IR-2.0` (visited on 07/28/2019).

[140] Drew Paroski. *Code Generation: The Inner Sanctum Of Database Performance*. "`http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html`". 2016.

[141] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *NeurIPS*. 2019, pp. 8024–8035.

[142] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Hakan Memisoglu, Navneet Potti, Saket Saurabh, Marc Spehlmann, and Zuyu Zhang. "Quickstep: A Data Platform Based on the Scaling-In Approach". In: *PVLDB* 6 (2018), pp. 663–676.

[143] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. "PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs". In: *CGO*. 2010, pp. 2–11.

[144] Aleksey Pesterev, Nickolai Zeldovich, and Robert Tappan Morris. "Locating cache performance bottlenecks using data profiling". In: *EuroSys*. 2010, pp. 335–348.

[145] Holger Pirk, Jana Giceva, and Peter R. Pietzuch. "Thriving in the No Man's Land between Compilers and Databases". In: *CIDR*. 2019.

[146] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. “Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware”. In: *PVLDB* 14 (2016), pp. 1707–1718.

[147] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. “tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation”. In: *SIGPLAN*. 1997, pp. 109–121.

[148] Massimiliano Poletto and Vivek Sarkar. “Linear scan register allocation”. In: *ACM Trans. Program. Lang. Syst.* (1999), pp. 895–913.

[149] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. “Rethinking SIMD Vectorization for In-Memory Databases”. In: *SIGMOD*. 2005, pp. 1493–1508.

[150] Orestis Polychroniou and Kenneth A. Ross. “Efficient Lightweight Compression Alongside Fast Scans”. In: *DaMoN*. 2015.

[151] Orestis Polychroniou and Kenneth A. Ross. “High throughput heavy hitter aggregation for modern SIMD processors”. In: *DaMoN*. 2013.

[152] Orestis Polychroniou and Kenneth A. Ross. “Vectorized Bloom filters for advanced SIMD processors”. In: *DaMoN*. 2014.

[153] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. In: *SIGMOD*. 2019, pp. 1981–1984.

[154] Bogdan Raducanu, Peter Boncz, and Marcin Zukowski. “Micro adaptivity in Vectorwise”. In: *SIGMOD*. 2013, pp. 1231–1242.

[155] Vijayshankar Raman et al. “DB2 with BLU Acceleration: So Much More than Just a Column Store”. In: *PVLDB* 11 (2013), pp. 1080–1091.

[156] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. “Managing Non-Volatile Memory in Database Systems”. In: *SIGMOD*. 2018, pp. 1541–1555.

[157] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. “Persistent Memory I/O Primitives”. In: *DaMoN*. 2019, 12:1–12:7.

[158] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. “Learning Linear Regression Models over Factorized Joins”. In: *SIGMOD*. 2016, pp. 3–18.

[159] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. “A Layered Aggregate Engine for Analytics Workloads”. In: *SIGMOD*. 2019, pp. 1642–1659.

[160] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. “Learning Models over Relational Data: A Brief Tutorial”. In: *SUM*. 2019, pp. 423–432.

[161] Stefan Schuh, Xiao Chen, and Jens Dittrich. “An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory”. In: *SIGMOD*. 2016, pp. 1961–1976.

[162] Malte Schwarzkopf. *The Remarkable Utility of Dataflow Computing*. 2020. URL: `https://www.sigops.org/2020/the-remarkable-utility-of-dataflow-computing/`.

[163] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. "Push versus pull-based loop fusion in query engines". In: *J. Funct. Program.* (2018), e10.

[164] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. "Building Efficient Query Engines in a High-Level Language". In: *ACM Trans. Database Syst.* 1 (2018).

[165] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. "How to Architect a Query Compiler". In: *SIGMOD*. 2016, pp. 1907–1922.

[166] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. "Cache Conscious Algorithms for Relational Query Processing". In: *PVLDB*. 1994, pp. 510–521.

[167] Julian Shun and Guy E. Blelloch. "Ligra: a lightweight graph processing framework for shared memory". In: *SIGPLAN*. 2013, pp. 135–146.

[168] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. "Micro-architectural Analysis of In-memory OLTP". In: *SIGMOD*. 2016, pp. 387–402.

[169] Utku Sirin, Ahmad Yasin, and Anastasia Ailamaki. "A methodology for OLTP micro-architectural analysis". In: *DaMoN*. 2017, 1:1–1:10.

[170] Evangelia A. Sitaridi, Orestis Polychroniou, and Kenneth A. Ross. "SIMD-accelerated regular expression matching". In: *DaMoN*. 2016.

[171] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. "Vectorization vs. compilation in query execution". In: *DaMoN*. 2011, pp. 33–40.

[172] Dongxiao Song and Shimin Chen. "Exploiting SIMD for complex numerical predicates". In: *ICDE*. 2016, pp. 143–149.

[173] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging". In: *USENIX*. 2004, pp. 29–44.

[174] Sam Van den Steen, Stijn Eyerman, Sander De Pestel, Moncef Mechri, Trevor E. Carlson, David Black-Schaffer, Erik Hagersten, and Lieven Eeckhout. "Analytical Processor Performance and Power Modeling Using Micro-Architecture Independent Characteristics". In: *IEEE Trans. Computers* 12 (2016), pp. 3537–3551.

[175] Michael Stonebraker and Ariel Weisberg. "The VoltDB Main Memory DBMS". In: *IEEE Data Eng. Bull.* 2 (2013), pp. 21–27.

[176] Christian Stuart. "Profiling Compiled SQL Query Pipelines in Apache Spark". MA thesis. Universiteit van Amsterdam, Jan. 2020.

[177] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. "How to Architect a Query Compiler, Revisited". In: *SIGMOD*. 2018, pp. 307–322.

[178] AzureML Team. "AzureML: Anatomy of a machine learning service". In: *PAPIs*. 2015, pp. 1–13.

[179] Robert M Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of research and Development* 1 (1967), pp. 25–33.

[180] Pinar Tözün, Brian Gold, and Anastasia Ailamaki. "OLTP in wonderland: where do cache misses come from in major OLTP components?" In: *DaMoN*. 2013.

[181] Transaction Processing Performance Council (TPC). *TPC BENCHMARK^{TM} DS – Standard Specification Version 2.13.0*. Apr. 2020.

[182] Transaction Processing Performance Council (TPC). *TPC BENCHMARK^{TM} H – Standard Specification Revision 2.18.0*. 2018.

[183] *UndoDB*. `https://undo.io`. Accessed: 2020-02-18.

[184] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases". In: *SIGMOD*. 2017, pp. 1041–1052.

[185] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. "Get Real: How Benchmarks Fail to Represent the Real World". In: *DBTest*. 2018.

[186] Skye Wanderman-Milne and Nong Li. "Runtime Code Generation in Cloudera Impala". In: *IEEE Data Eng. Bull.* 1 (2014), pp. 31–37.

[187] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. "SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units". In: *PVLDB* 1 (2009), pp. 385–394.

[188] Milian Wolff. *Hotspot - the Linux perf GUI for performance analysis*. `https://github.com/KDAB/hotspot`. Apr. 2020.

[189] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissmann. "ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay". In: *Workshop on Modeling, Benchmarking and Simulation, June, 2007*. 2007.

[190] Ahmad Yasin, Jawad Haj-Yahya, Yosi Ben-Asher, and Avi Mendelson. "A Metric-Guided Method for Discovering Impactful Features and Architectural Insights for Skylake-Based Processors". In: *TACO* 4 (2019), pp. 1–25.

[191] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. "Apache spark: a unified engine for big data processing". In: *Communications of the ACM* 11 (2016), pp. 56–65.

[192] Rui Zhang, Saumya Debray, and Richard T. Snodgrass. "Micro-specialization: dynamic code specialization of database management systems". In: *CGO*. 2012, pp. 63–73.

[193] Rui Zhang, Richard T. Snodgrass, and Saumya Debray. "Micro-Specialization in DBMSes". In: *ICDE*. 2012, pp. 690–701.

[194] Ying Zhang, Martin L. Kersten, and Stefan Manegold. "SciQL: array data processing inside an RDBMS". In: *SIGMOD*. 2013, pp. 1049–1052.

[195] Jingren Zhou and Kenneth A. Ross. "Buffering Accesses to Memory-Resident Index Structures". In: *PVLDB*. 2003, pp. 405–416.

[196] Jingren Zhou and Kenneth A. Ross. "Implementing database operations using SIMD instructions". In: *SIGMOD*. 2002, pp. 145–156.

[197] Marcin Zukowski. "Balancing Vectorized Query Execution with Bandwidth-Optimized Storage". PhD thesis. University of Amsterdam, 2009.

[198] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. "Super-Scalar RAM-CPU Cache Compression". In: *ICDE*. 2006.