Adaptively Generating Heterogeneous Execution Strategies using the VOILA Framework

Tim Gubner

The research reported in this thesis was carried out within the Database Architectures group at Centrum Wiskunde & Informatica (CWI), the National Research Institute for Mathematics and Computer Science in the Netherlands.

SIKS Dissertation Series No. 2024-26
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

VRIJE UNIVERSITEIT

Adaptively Generating Heterogeneous Execution Strategies using the Voila Framework

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. J.J.G. Geurts,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op vrijdag 4 oktober 2024 om 13.45 uur
in een bijeenkomst van de universiteit,
De Boelelaan 1105

door

Tim Karl-Heinz Gubner

geboren te Plauen, Duitsland

promotor:	prof.dr. P. Boncz
copromotor:	prof.dr. S. Manegold

promotiecommissie:	prof.dr.ir. H.E. Bal
	prof.dr. T. Neumann
	prof.dr. G. Alonso
	dr. A. Katsifodimos
	dr. P. Tözün

To my parents, Ines and Karl-Heinz,
my wife, Marina,
and our son, Emil.

Contents

Acknowledgements

First, I am deeply indebted to my direct supervisor, Peter Boncz. Thanks to Peter, I had the freedom to pursue the concepts mentioned in this thesis, along with several others (e.g. on benchmarking [RHGM18]). I especially want to express my gratitude to Peter for the fantastic conversations and insights on both macro and micro-optimizations that, usually, helped maximize (CPU) performance. His ongoing feedback was invaluable; it really motivated me to enhance my work significantly.

Special thanks to Martin Kersten, who unfortunately passed away during my Ph.D. Besides our interesting brainstorms on sometimes somewhat exotic ideas, he often advised to challenge the consensus (his work often showed that as well). This generic but, nevertheless, great advice stuck with me, as the following chapters will show on multiple occasions.

Thanks to Stefan Manegold for helping me during the early days of my Ph.D. as well as later on. Especially, I am very thankful for his feedback.

Many thanks to Thomas Neumann, Viktor Leis and Guido Moerkotte for their early yet incredibly helpful feedback on this work (at the ICDE PhD Symposium in Paris).

Thanks should also go to the committee members, Henri Bal, Thomas Neumann, Gustavo Alonso, Asterios Katsifodimos and Pinar Tözün, for their time and effort reviewing this thesis and taking part in the defense.

I am grateful to my office mate, Pedro Holanda – truly the best office mate anyone could ask for! We had an outstanding time together, from "practicing" the hits of the 90s for Casablanca, or "killing rats" as some people might call it, to contemplating possibly world-changing ideas such as the publicly shared wallet (Commie-Coin) or brakes for databases.

I would like to extend my sincere thanks to Mark Raasveldt and Diego Tomé. Together with Pedro, we four had lots of fun and some great years together.

Many thanks to Niels and Arjen for quickly fixing IT issues and, more importantly, installing new and (sometimes) exotic hardware. Furthermore, I had the pleasure of working with Azim, Daniël, Dean, Eyal, Gabor, Hannes, Hassan, Ilaria, Laurens, Leonardo, Madelon, Matheus, Nantia, Pedro F., Sjoerd among other members of the Database Architectures group at CWI, MonetDB Solutions and DuckDB Labs.

I am extremely grateful to my wife, Marina, who has kept my spirits and motivation high, especially in the process of getting my first big (non-workshop) paper published (which came with almost a handful of rejections).

Furthermore, I also want to thank my German friends who helped me keep my sanity, Dominik, Erik, Isabell, Marcus, Martin H., Martin K., Steve as well as Susann.

Finally, I would also like to thank my cat, Foppie, for emotional support and providing a great excuse from finishing this document.

CHAPTER 1

Introduction

Since the beginning of computers, we have been using machines to analyze data to find answers to seemingly important questions. The challenge of developing a program to compute these answers efficiently was usually solved by programmers. Solving many different questions for different architectures, and maintaining them, must have been incredibly tedious (i.e. a nightmare).

DBMS. A couple of years later emerged the Database Management System (DBMS). DBMSs are systems that abstract data storage and data access. For users of DBMSs, this brings several advantages. For example, data can be stored in optimized format(s) that better facilitates certain access patterns, or index structures can be added to allow faster reads and updates of certain cells. A DBMS can ship multiple such methods, while also providing certain guarantees, most notably data consistency. Thus, the user does not have to re-implement these methods but can use a DBMS instead.

Query Execution. One important aspect of DBMSs is their ability to evaluate queries (*query execution*). DBMSs, often allow queries formulated in high-level languages and do not require the user to write a rather complicated program to extract answers from data in a specific layout.

To evaluate queries, a DBMS typically compiles the high-level program into a low-level program. In particular, we focus on relational DBMS which translate SQL to a directed acyclic graph with specific operators based on relational algebra (query execution plan, or plan).

Especially for complex queries on a large and often increasing amount of data, performance of query execution became, and still is, an important factor for the overall performance. Therefore, in modern DBMSs, query execution tries to squeeze the most out of current hardware, and current methods (vectorized [BZN05] and data-centric [Neu11] execution) are efficient, i.e. roughly as fast as naive implementations written by hand.

1.1 Research Questions and Contributions

Query execution techniques (e.g. vectorized [BZN05] or data-centric [Neu11] execution) are rather generic, typically for implementation and maintenance reasons. This raises the question, of whether performance can be further improved with optimizations specific to the combination of data, query and environment, so-called *instance-specific optimizations*. Such situations can create opportunities for further optimizations. For example, a join with a tiny inner relation, of say one tuple, can be rewritten into a filter which, in case of a hash join, consequently removes memory-access and hash probing overhead. In practice, such a join might not be detectable before executing the query (i.e. at query optimization time) as we need to verify the upper bound on the cardinality of the inner relation, and cardinality estimation is often inaccurate by orders of magnitude [LGM+15]. Thus, our upper bound is likely "off" by a margin that makes our optimization not worth the effort (e.g. if we allow such joins for inner relations with one row, but the upper bound reports $\leq$ 100 rows). Similarly, other techniques can be applied as well to "cut corners". In particular, we wonder how a human, given a total understanding of the environment (query, data distributions as well as software and hardware setup), would implement a query that requires minimal runtime. We, therefore, posed the first question:

> **Research Question 1:** *How far can query plans be optimized to the specific instance?*

Research Question 1 is investigated by the Chapters 3 and 4.

Chapter 3 explores how a relatively simple query (TPC-H Q1) can be optimized further given the specific instance of the problem (low number of groups in group-by, limited ranges of values, almost all rows surviving the filter, relatively arithmetic-intensive).

Chapter 4 discusses compressing hash tables, an often used data structure in query engines, and exploiting compression during lookup operations (key equality checks). Therefore, tuning query execution to specific data distributions.

Part of tailoring implementations specifically to the environment concerns hardware. Modern hardware has become (and is becoming) increasingly heterogeneous. This trend happens in multiple dimensions: Frequently, systems feature CPUs as well as GPUs (e.g. smartphone, laptop, desktop, cloud) and different CPU architectures became important (e.g. ARM for smartphones, laptops and servers, RISC-V), but also on the same CPU architectures there exist features that can significantly accelerate certain operations (SIMD, encryption, compression). Query engines are, typically, rather static constructs optimized for a handful of systems (or less). However, in a world dominated by increasingly heterogeneous hardware, they become increasingly unable to exploit all resources of the underlying system. While query engines can be optimized for more systems, this will likely add new code paths that need to be tested and maintained. Additionally, code needs to be added that chooses the best path which, in practice, also needs to guarantee robust performance. With a high degree of heterogeneity, both, maintaining and choosing code paths, will become practically unfeasible. Therefore, we pose the second question:

> **Research Question 2:** *How can query engines exploit increasingly heterogeneous modern hardware?*

Research Question 2 is investigated in the Chapters 5 to 8.

Chapters 5 and 6, we propose using the domain-specific language VOILA to synthesize many different implementations (flavors) of a given query. Besides the flexibility, VOILA also can serve as an extensible abstraction between high-level relational operators (physical operators) and low-level machine code. With the emergence of new accelerators (e.g. triggered by specific intrinsics), new code generation rules or constructs can be added to VOILA. However, the VOILA framework also facilitates the automated discovery of "good" flavors, instead of implementing flavors *by hand*.

The following chapter (Chapter 7) investigates how current query execution paradigms (Data-Centric and Vectorized Execution, and variations thereof) perform on

various hardware architectures. If there exists a significant performance diversity on current/modern hardware, code generation is required to synthesize many flavors, because it is typically not known which will perform best. Thus, static query engines, which focus on one flavor (typically either Vectorized or Data-Centric), might be unable to perform optimally.

Chapter 8 presents Excalibur, a framework that automatically explores the design space micro-adaptively *while the query is running*. Thus, tuning the query to the specific instance.

1.2 Thesis Outline & Publications

This thesis reflects my journey from manually navigating the design space through hand-implemented optimizations, to identifying a technique for automating this process, culminating in the creation of an adaptive virtual machine that exploits the design space to minimize query response times. Apart from the manual exploration, this thesis follows the ideas and plan outlined at the ICDE PhD Symposium [Gub18]:

- The proposal describes that, first, the design space should be abstracted using a domain-specific language. Afterward, this language can be exploited to adaptively choose new implementation flavors (i.e. generate and actually run them):
  - Tim Gubner. Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware. IEEE International Conference on Data Engineering (ICDE) – PhD Symposium, 2018

Outline & Publications. The first chapter (*Background*) discusses background necessary for understanding this thesis.

The following two chapters are case studies that showcase the potential specific points in the design space have to offer:

- Analytical queries often perform arithmetic, e.g. to compute the tax rates. Chapter 3 (*Compact Types & In-Register Aggregation*) explores shrinking the width of data types to increase the SIMD throughput of arithmetic operations. Additionally, it highlights that group-by and aggregation can be implemented more efficiently, for a low number of groups. The research in this chapter is based on:

  - Tim Gubner and Peter Boncz. Exploring Query Execution Strategies for JIT, Vectorization and SIMD. Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS), 2017

- The performance of analytical queries is often limited by the memory-access costs caused by group-by, join operations, as well as operations on strings. Chapter 4 (*Compressed Hash Tables & Soviet Strings*) explores how compression of intermediate data structures, i.e. hash tables and strings, can reduce, both, memory-access cost and memory footprint. The research is based on two publications. The first won the award for the best research paper. Later, our work was selected by SIGMOD Record, leading to the second publication:

  - Tim Gubner, Viktor Leis and Peter Boncz. Efficient Query Processing with Optimistically Compressed Hash Tables & Strings in the USSR. IEEE International Conference on Data Engineering (ICDE), 2020

  - Tim Gubner, Viktor Leis and Peter Boncz. Optimistically Compressed Hash Tables & Strings in the USSR. SIGMOD Record, 2021

The previous case studies relied on manual, and arguably repetitive, engineering efforts. Therefore, the following two chapters will discuss using a domain-specific language to (1) abstract query execution and (2) generate specific implementations (flavors) from that language. Utilizing a domain-specific language, we can explore the design space more efficiently, by synthesizing different implementations from one description, but are, also, more future-proof as details are abstracted away:

- Chapter 5 (*Encapsulating the Essence in VOILA*) addresses abstracting query execution details using a domain-specific language (VOILA), and is based on:

  - Tim Gubner and Peter Boncz. Charting the Design Space of Query Execution using VOILA. International Conference on Very Large Data Bases (VLDB), 2021[1]

- In the following chapter (*Synthesizing Engines from VOILA*), VOILA is used to explore the design space. This chapter is also based on:

  - Tim Gubner and Peter Boncz. Charting the Design Space of Query Execution using VOILA. International Conference on Very Large Data Bases (VLDB), 2021[1]

[1] Research was extended and split into two chapters.

The experimental evaluation in the two previous chapters was limited to one to two machines (sets of hardware). However, in practice, the hardware environment is often uncontrollable (from the software developer's side).

- Engines typically focus on one specific flavor. In Chapter 7, we explore how different CPUs and hardware architectures affect the performance of certain flavors:
  - Tim Gubner and Peter Boncz. Highlighting the Performance Diversity of Analytical Queries using VOILA. Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS), 2021

The previous chapters discussed how specific design points for query execution systems can be generated in an automated manner. However, they required (a) knowing which point (flavor) to generate, and (b) incurred significant compilation time as they generated source code (plain text) which was later compiled. Both make design space exploration rather time-consuming and impractical for database systems:

- The following chapter (*Adaptive Virtual Machines*) demonstrates how the design space flexibility can be dynamically exploited at query runtime. This chapter is based on:
  - Tim Gubner and Peter Boncz. Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA. International Conference on Very Large Data Bases (VLDB), 2023
- In the last chapter (*Conclusion & Future Work*), we present the insights and conclusions gained and provide directions for future research.

Publications not part of this thesis. My journey led to numerous publications. Unfortunately, some publications do not contribute enough to the core of this thesisbut, nevertheless, deserve to be mentioned:

- While GPUs offer high memory-bandwidth and high computational power, real-life performance is often limited by data transfers from/to GPU. The work investigates how (a) data can be transferred more efficiently by using compression and how (b) a complete query (TPC-H Q1) can benefit from execution on CPU and GPU, in parallel.

  – Diego Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg and Peter Boncz. Optimizing Group-By And Aggregation using GPU-CPU Co-Processing. Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS), 2018

- The bandwidth limitations of transfers from/to GPUs often disqualify GPU as accelerators for certain tasks during query execution. To benefit from GPUs, such a task needs limited input and output. The work highlights how selective joins can be improved by using early pruning with Bloom filters and how Bloom filters can efficiently be (dynamically) offloaded to the GPU.

  – Tim Gubner, Diego Tomé, Harald Lang and Peter Boncz. Fluid Co-processing: GPU Bloom-filters for CPU Joins. Workshop on Data Management on New Hardware (DaMoN), 2019

CHAPTER 2

Background

2.1 Increasingly Heterogeneous Hardware

Recently, hardware has become more heterogeneous. We observed a divergence from the previously mainly generic X86-CPU-based hardware (one instruction set, one development target) to a broader platform with multiple devices, each one "good" at rather specific tasks (multiple targets with multiple instruction sets, optimized for different purposes).

In this chapter, we describe the current hardware trends and how they impact software design. First, we discuss trends in hardware architecture. Afterward, we dive into trends in CPU architecture, followed by trends in CPU features. Finally, we summarize the findings and their impact on software design.

2.1.1 Brief Introduction to Trends in CPU Architectures

Faster Clock Frequencies. Since the early beginnings of CPUs, it has been possible to increase performance by increasing clock frequencies while fitting more transistors on the chip. Applications could conveniently benefit from these improve-

ments without requiring changes, i.e. the era of "free lunch". Unfortunately, this era did not last forever.

More Cores. When clock frequencies started to stagnate, processor manufacturers started adding more cores to the chip. This was easily possible, as transistor sizes were decreasing further, thus providing enough area for additional cores. Unfortunately, benefiting from this trend, the multi-core trend, required changes on the software-side (efficiently utilizing multiple threads). As time has shown, also this trend has come to an end.

Application-Specific Accelerators. When it was not (easily) possible anymore to add more cores (due to limitations imposed by heat emission), processors started to include specific accelerators. Usually, such accelerators exist for very demanding applications with wide use, e.g. graphics, encryption or artificial intelligence. This is the trend du jour. For applications, this means that the underlying hardware is becoming more heterogeneous. To provide optimal performance, application developers will have to find good tasks for each accelerator, in the hope to elevate application-specific bottlenecks. Likely, this will require rethinking traditional software and larger, potentially architectural, changes.

2.1.2 Trends in CPU Architectures.

For decades, the market for processors for laptops, desktops, and servers was dominated by CPUs with the X86 architecture which made X86 the de facto standard for software development. X86-based CPUs are typically developed by Intel and AMD. But there exist also implementations by other entities, like e.g. the Zhaoxin KaiXian [Alc20]. However, recent pushes into ARM- and RISC-V-based processors seem to threaten the X86's hegemony:

ARM. First, ARM-based architectures started to dominate the market for smartphones. Over the years, ARM tried to enter the many-core server market (e.g. with the Cavium ThunderX [DG16]). With the AWS Graviton 1 and 2 [Ama21a], ARM started to enter the cloud market. Meanwhile, ARM entered the laptop market (with mostly high-end smartphone chip sets, e.g. Chromebook-branded laptops [Var21, Sam12]). More recently, Apple switched to ARM-based laptops (Apple M1 [App21]). Consequently, ARM-based architectures captured a higher share of the laptop market. Recent high-performance ARM-based chips, like the Graviton 2 [Ama21a] and 3 [Ama23], tend to threaten X86's hegemony in terms of performance as well as power usage. A noteworthy development is the emergence of data center CPUs by NVIDIA with the Grace CPU [NVI]. Even more interestingly,

the Grace can be combined with a GPU to a "Grace Hopper Superchip" [NVI]. Notably, the push towards ARM is not limited to the Western hemisphere, as Alibaba [Shi21] and Huawei [Hua19, XCZ+21] presented their own ARM-based CPUs.

RISC-V. Besides pushes into ARM-based architectures, the European Processor Initiative (EPI) [euR24] and Alibaba [ris21] are developing/developed RISC-V-based processors. Most notably, the EPI tries to establish a European processor for high-performance computing that is relatively easily extensible with application-specific accelerators [EPI24].

ARM and RISC-V vs. X86. Both, ARM and RISC-V, share one major advantage: It is possible to license a ready-made chip design, extend with application-specific optimizations (e.g. accelerators), send the design to a fab and receive a chip optimized for certain applications (sufficient funds assumed). That is something that was previously very challenging to achieve and required a certain scale (e.g. Microsoft and Sony repeatedly managed to obtain somewhat customized AMD chips for their gaming consoles [Son, Shi13, LR20, Mic]). However, with this newly obtained freedom, many other companies with "deep pockets" can relatively easily create their own semi-custom chips. If this trend continues, we will likely see many slightly different ARM or RISC-V chips optimized for very specific tasks.

Conclusion. The rule of X86 in the laptop, desktop, and server market is under attack by ARM and, potentially, RISC-V. One major advantage ARM and RISC-V have is an easy path to a customized chip design for specific applications (designs can be licensed). Should this trend persist, it is probable that CPU architectures will become increasingly heterogeneous. While it is possible to write portable code, optimal performance is not guaranteed. Specifically for performance-critical software, this is a major challenge, as it needs to run "well" on all possible architectures.

2.1.3 Trends in CPU Features

So far, we have observed that both, system architectures and CPU architectures, seem to drift towards semi-customization with application-specific optimizations. This trend can even be observed within a single CPU architecture. Here, we analyze recently introduced CPU features.

X86. Over the years, new instructions were added to X86. Typically, these are very specific optimizations that can be harnessed to significantly accelerated certain tasks, as e.g. indicated in Table 2.1. These optimizations instructions reach from encryption to inference on neural networks.

Table 2.1: Selected CPU features able to accelerate certain tasks.

Instruction Set	Instruction	Interesting Use-Case(s)
AES		Acceleration for AES encryption
		Near-cryptographic hash function for strings [Gör13]
POPCNT	`popcnt`	Number of one bits in word
BMI1	`tzcnt`	Count trailing zero bits
	`bextr`	Extract contiguous bits
		Selection pushdown [LLC23]
SSE4.2	`crc32`	Check sum (CRC32)
		String hash [KLK+18]
SSE4.2	`pcmpestri`	String comparisons
AVX512-F	`vpcompress` & `vexpandp`	Buffering [LPK+20, FZW19])
	`vpcompress`	Selection vector creation [KLK+18]
	`vperm`	In-register lookup tables (Chapter 3)
AVX512-CD	`vpconflict`	Duplicate detection & conflict avoidance (Chapter 3)
		RLE compression [UPD+18]
AVX512-VBMI/VBMI2		Extended support for smaller bit-widths (Chapter 3)
AVX512-VNNI		Acceleration for neural networks
	`vp4dpwssd`	Multiple dot products
AVX512-VP2INTERSECT	`vp2intersectq`	Pair-wise intersection
GFNI		Galois Field, affine transformations
		Error correction, cryptography [gfn20]
Accelerator		**Description**
Advanced Matrix Extensions (AMX)		Operations on matrices (e.g. machine learning) [Intd]
In-Memory Analytics Accelerator (IAA)		Compression, filtering, encryption [Int23]
Data Streaming Accelerator (DSA)		Data generation, comparisons & copying [Inta]

Cloud. The recent advent of cloud computing, or more generally multi-tenant computing, also had its influence on CPU features.

Especially, avoiding data leaks between concurrently running tenants is challenging. With SGX [Intb], Intel introduced Enclaves, i.e. private scratch memory for applications, when a context switch happens, the private memory is encrypted and written into the main memory. Since this private memory has a limited size [EHZH+22], the application needs to ensure that all sensible data will fit.

Machine Learning. Similar the Cloud, the recent rise of Machine Learning left its footprints on CPU features. Typical ML tasks are compute- and memory-intensive and often involve matrix multiplications with floating-point numbers (e.g. used in neural networks). Relatively recent CPU features such as AMX and AVX512-VNNI appear to be crafted to accelerate machine learning workloads [Intd, Intc].

AWS Graviton 2. As a CPU typically only found in the cloud, the AWS Graviton 2 offers very cloud-specific features, such as "always-on fully encrypted DDR4 memory" [Ama20] and hardware-accelerated cloud management (Nitro) [Ama19]. Transparently encrypted memory is useful when the machine is moved between tenants, such that the next/other tenant(s) cannot access the previous tenant's,

possibly important, data, thus mitigating the risk of data leaks, but without requiring important data to fit into a fixed-size enclave (as SGX). Besides, the Graviton also features acceleration for data compression [Ama19].

2.1.4 Conclusion

From the current trends, it appears that hardware is becoming increasingly heterogeneous. New processors can use a different architecture (e.g. ARM) extended with accelerators for very specific tasks. For software, this heterogeneity poses a major challenge as different accelerators can require major changes on the software-side (e.g. generating GPU code inside a database system). Complex software systems (such as database systems) cannot cost-efficiently support many different accelerators, as they might require very different implementations. Thus, if we do not solve this challenge, complex software might be unable to fully exploit the current hardware, and more-so benefit from future hardware improvements.

It is not clear how we can exploit specific accelerators cost-efficiently, without having to rewrite our software for each accelerator.

2.2 Abstraction using Virtual Machines

A virtual machine (VM) is a conceptual machine that allows the execution of programs formulated in some instruction set. Arguably, this definition is very general. Typical examples are VMs used for:

- Virtualization, emulating a different environment within a host, e.g. VirtualBox, QEMU, Xen, Hyper-V
- Executing programs (instruction set = language, virtual machine = interpreter), e.g. JVM, LuaJIT, PyPy
- Compiling programs (instruction set = language, virtual machine = compiler), e.g. LLVM

VMs to Execute Programs. In this thesis, we focus on the aspect of using VMs to execute programs. In the most basic instance, such a VM is just an interpreter for the language (e.g. interpreting the language instruction by instruction; or function by function). Depending on the language, interpretation can have significant overhead compared to statically compiled code (interpretation overhead). Besides this disadvantage, VM-based execution minimizes the required compilation time (compared to static compilation) and allows optimizing specific parts of the program.

JIT-Compilation. For instance, VMs often generate compiled code for certain parts of the program (Just-In-Time-Compilation, or JIT-Compilation). Typically, traces are gathered during interpretation and later optimized code for (hot) traces is generated, so-called trace-based JIT-compilation (used by HotpathVM [GPF06], TraceMonkey [GES+09], LuaJIT [Pal09], SPUR [BBF+10], PyPy [BCFR09]). More generically, VMs can optimize frequently executed code (e.g. the HotSpot VM for Java [PVC01]). When certain conditions are met, JIT-generated code has the chance to beat statically compiled code (e.g. if object types are known, expensive virtual method calls can be removed). Besides, machine-specific code can be JIT-ted which can be advantageous when the target machine is not known beforehand. In practice, code often has to be compiled for the target machine that is the least common denominator. Optimizations for newer features (AVX-512, AES ...) require special extensions as well as detection at runtime (e.g. via the `cpuid` instruction).

2.2.1 Notable Examples

In the following, we highlight examples of VMs. We start with the Java Virtual Machine, followed by the Android ecosystem, LLVM and LuaJIT.

Java and the Java Virtual Machine (JVM). The unique selling point of the Java ecosystem has been the mantra of portable code: write once, run anywhere. Thus, not requiring code modifications to run on other machines (including esoteric architectures) as long as there is a JVM for that platform[1].

From the user's perspective, Java code is first compiled into byte code. This byte code is portable and can be executed on any host with a JVM. Over the years, many JVMs have been created (e.g. HotSpot, Graal, IBM) and each allows different optimizations at runtime for different architectures.

Android (Dalvik and Android RunTime). Closely related, to Java and the JVM, is the Android platform, which is often used on mobile devices (smartphones, smartwatches, tablets). Android programs are commonly written in Java, or other JVM languages, and compiled to JVM byte code. Before execution of the program, it is then, on Android, compiled to Dalvik byte code (DEX [and20]).

Android started with a VM that interpreted Dalvik byte code, the Dalvik VM [Fru14]. Later, a JIT-compiler [Fru14] was added. Eventually, due to suboptimal perfor-

[1] Similar to portable C code that can be compiled, and later executed, wherever there is a C compiler. However, C code requires an additional compilation step.

mance, Dalvik got replaced by the Android RunTime (ART) that compiles byte code into native instructions [Fru14].

LLVM started as a toolkit for multi-stage optimization [Lat02]. But nowadays, LLVM is a "full-blown" compiler framework [LA04]. It comes with multiple compiler front-ends (e.g. for C and C++) that generate an Intermediate Representation (LLVM IR; virtual instructions). Based on LLVM IR, many optimizations can be applied until either a back-end emits machine code (e.g. X86, ARM, WebAssembly) or the IR is interpreted directly (as e.g. Kohn et al. [KLN18] did). The power of the LLVM framework lies in the shared representation (LLVM IR) that allows sharing optimization passes between compilers. Thus, LLVM eases the effort required to develop new compilers with good end performance.

LuaJIT is a VM for executing Lua code. Contrary to the regular Lua VM (`lua.org`), LuaJIT [Pal09] features a lightweight trace-based JIT-compiler with additional optimizations. Interestingly, LuaJIT runs on many architectures (e.g. X86, ARM, PPC, MIPS) while being "widely considered [...] one of the fastest dynamic language implementations" [Pal].

2.2.2 Conclusion

VMs define programs in the means of emulating some machine's instruction set. As instructions are executed, VMs can apply optimizations that are target-specific or specific to the instance (e.g. less frequently executed code requires fewer optimizations with similar end-performance). Thus, VMs provide the platform to modify the physical execution while the program is running ("mid-flight").

2.3 Analytical RDBMS

Since the early days of computers, applications that manage data, so-called Database Management Systems (DBMSs), became popular. Nowadays, a plethora of such systems exist, reaching from spreadsheet software to fully featured systems that provide certain guarantees.

2.3.1 Relational Database Management Systems (RDBMSs)

Since the term DBMS is quite general, we focus on a subset. The, so called, Relational Database Management Systems (RDBMSs), are DBMSs using the relational model [Cod83]. In the relational model, data is presented in tuples, which are grouped into relations.

RDBMSs store and provide access to tables (which are relations). Queries in RDBMSs are typically decomposed into operations on such relations with set/bag semantics (relational algebra).

Nowadays, users interact with an RDBMS using a (arguably, more or less) standardized language SQL, which provides means to define schemata, modify and query data. SQL is a high-level declarative language. Thus, evaluating queries requires lowering SQL into a lower-level representation, which is usually some form of relational algebra (query plan). Typically, there are many implementations of a SQL query, often also with highly different runtimes. Therefore, an RDBMS commonly tries to find a cost-minimal implementation via some cost model. After finding an implementation, the query plan is evaluated (query execution) and should[2] eventually return the answer (of the query).

Transactional Workload. Typically, RDBMSs were used to handle transactional workloads (i.e. many data modifications using rather simple queries). For this purpose, RDBMS usually include certain index data structures that accelerate access to certain tuples. Thanks to the rather simple queries, queries often significantly benefit from indexes and, thus, efficient query execution (physical evaluation of a query, after optimization of the query plan) is less relevant.

2.3.2 Analytical RDBMS

Various users of DBMSs tend to analyze the stored data. Such queries are typically rather complex, read-intensive and long-running. Data modifications tend to happen relatively rarely and in bulk.

With the increasing volume of data, DBMSs emerged that are optimized for these, analytical, workloads. Notable examples include MonetDB [IGN+12], Vector/Vectorwise [BZN05], DB2 BLU [RAB+13] as well as C-Store [SAB+05]. Compared to the "old DBMSs", these systems differ in the way they store data, handle data modifications and evaluate queries.

Data Storage. Since these queries (i.e. analytical queries) are read-intensive and often read large amounts of data, data tends to be stored compressed and in a columnar layout. Compression tends to increase the net read bandwidth and allows storing more data closer to the CPU [ZHNB06].

[2] Arguably, "will" is the better word. But, assuming SQL (with common table expression and window functions) is Turing complete [Pos11], non-terminating queries are possible.

Columnar layout stores each column in separate memory regions. A columnar layout provides multiple advantages beneficial to data analytics: (1) When not all columns are required, unneeded columns do not need to be read, as opposed to a row-wise layout which requires reading the whole row[3]. (2) Columns contain data of similar shape (similar type, but can also have similar ranges) and, thus, benefit compression, but also enable efficient bulk operations (via SIMD).

Data Modifications. Since data is typically stored columnar and compressed, modifying it is rather slow (e.g. might require re-compression of larger chunks). Often, systems optimized store newly modified data out-of-place which is, then, periodically integrated into the compressed main storage. Options for out-of-place storage of modifications include e.g. a specialized write store, as used by C-Store [SAB+05], or delta structures, like the Positional Delta Tree [HZN+10] in case of Vectorwise.

Query Execution. The larger the amount of data analyzed, the more important the efficiency of evaluating queries becomes. Analytical systems tend to focus on query execution paradigms that are as efficient as possible. Query execution paradigms are discussed in the following section (Section 2.2.4).

2.3.3 Conclusion

Database Management Systems (DBMSs) have become the de facto standard of storing, modifying and analyzing data. Especially, Relational DBMSs are widely popular.

Performance, often, depends on the workload. For analytical workloads, which have specific characteristics (read-intensive and complex queries, bulk updates), specialized systems emerged. These specialized systems often outperform generic systems by a significant margin. By being optimized for analytical workloads, these systems differ from generic systems in the way they store data, handle updates and execute queries.

Note that there also exist rather generic systems that perform well on analytical workloads. Most notable examples are Hyper [Neu11] and Umbra [NF20].

[3] Attributes of rows could be skipped, but this would introduce many small skips, which does not interact well with prefetching (be it hardware prefetcher(s) or fetching blocks from disk. Another major point is that typically a whole block has to be read from storage. Thus, such small skips are unlikely to reduce data read from storage.

Table 2.2: Summary of basic query execution paradigms.

Name	Execution	Materialization	Interpretation Overhead
Iterator	tuple-at-a-time	1 row	high
Data-Centric Compilation	tuple-at-a-time	≤ 1 row	none
Columnar Execution	column-at-a-time	full column	minimal
Vectorized Execution	vector-at-a-time	column chunk	low

2.4 Query Execution Paradigms

In this section, we explain basic query execution tactics commonly used in RDBMSs. A brief overview can be found in Table 2.2. We first explain the simple iterator-based execution model, followed by data-centric compilation, columnar execution and vectorized execution.

2.4.1 Iterator-based Execution

Query evaluation can be implemented using iterators, which is e.g. described by Graefe [Gra94]. Each operator implements a `open-next-close` interface which resembles an iterator, the same for expressions. To evaluate the operator tree, `next` is called on the top-most operator (the root). Depending on the implementation of each operator, it may call `next` of its children to evaluate the corresponding subtree.

Materialization Overhead. Typically, `next` only returns one row (later, we show that more rows can be returned as well). Thus, only one row need to be materialized in the pipeline at a time (per operator). Consequently, the memory footprint of iterators is rather low (compared to Columnar and Vectorized Execution).

Interpretation Overhead. Typically, each `next` call either involves calling a virtual method (e.g. in C++; comes with two indirect memory accesses) or calling a function pointer (e.g. in C; involves one indirect memory access) and potentially also requires loading machine code into instruction cache (e.g. L1d). For evaluating expressions, this typically requires at least one call to a virtual method/function pointer per expression. Thus, interpretation overhead is high.

Notable Systems. Iterator-based Execution is used by well-established systems such as Postgres [SR86], MySQL [BZN05] and SQLServer [LCH+11].

2.4.2 Data-Centric Compilation

Data-Centric Compilation [Neu11] is based on the idea to compile a query fragment (a pipeline, starting from a source until results are materialized in a sink) into

a single loop. Inside that loop, one row is processed at a time. The row's values (attributes) are stored inside regular variables. Expressions then become expressions on these values. This is the first step, followed by machine code generation. The idea is that after the compilation step, each variable will fit a CPU register and expressions become native CPU instructions.

Materialization Overhead. Since attributes fit CPU registers, the intra-pipeline materialization overhead is minimal. However, attributes can be spilled to slower memories (done by the compiler as part of register allocation, typically happens when more variables are used than there are available registers).

Interpretation Overhead. Since the whole pipeline is compiled into machine code, the interpretation overhead is non-existent.

Notable Systems. Data-Centric Compilation is mainly used in Hyper [Neu11] and Umbra [NF20].

2.4.3 Columnar Execution

A different approach is Columnar Execution, which executes the query column-at-a-time. In other words, expressions are computed by fully reading the input columns and fully producing the output column(s). This offers essentially two advantages: (1) Interpretation can happen per operation, which now processes whole columns. Therefore, interpretation overhead is low. (2) Processing can happen in tight, often data-independent, loops, which is not only CPU-efficient (can use SIMD, CPUs can speculate ahead and fill processing pipeline with load, stores and other operations) but also allows efficiently loading/storing data from/to main memory (data parallel access).

Materialization Overhead. The obvious disadvantage is that the full columns need to be materialized, for each expression. Consequently, the memory footprint can be high and, if data does not fit into main memory anymore, trigger spooling to disk.

Interpretation Overhead. Since interpretation only needs to be done once per expression, interpretation overhead is very low (assuming corresponding tables have many rows).

Notable Systems. The most notable example for Columnar Execution is MonetDB [Bon02, IGN+12].

2.4.4 Vectorized Execution

While columnar execution has its advantages, materialization overhead is high. Vectorized Execution [BZN05] introduces limited materialization into Columnar Execution. Instead of evaluating the full table (or full columns thereof), Vectorized Execution partitions the table into chunks. Each chunk consists of columnar vectors. Inside that chunk, Evaluation can then be done "column-at-a-time", or rather "column-vector-at-a-time".

Moreover, Vectorized Execution provides some additional benefits. Additional optimizations can be triggered via micro-adaptivity [RBZ13], which dynamically choose alternative execution tactics (per expression), or by cheap checks once per vector.

Materialization Overhead. Compared to Columnar Execution, materialization overhead is typically lower, but obviously higher than Data-Centric Compilation as columnar vectors need to be materialized.

Interpretation Overhead. Interpretation Overhead is higher than for Columnar Execution, as expression evaluation needs to call a function pointer/virtual method for each chunk. Typically, chunk sizes are in the 1-2k rows, rendering the interpretation overhead rather insignificant.

Notable Systems. Notable systems that utilize Vectorized Execution are Vector/Vectorwise [BZN05], DB2 BLU [RAB+13], Photon [RBŁ21] and DuckDB [dud].

2.4.5 Relative Performance

Iterator vs. Vectorized. The initial paper on Vectorized Execution [BZN05] highlighted that Vectorized Execution outperforms Iterator-based Execution by "between one and two orders of magnitude" [BZN05].

Data-Centric vs. Vectorized. There has been a study on the relative performance between Data-Centric Compilation and Vectorized Execution, by Kersten et al.[KLK+18]). The study established that:

1. Data-Centric Compilation provides better performance at computationally intensive workloads.
2. Vectorized Execution excels at data-parallel workloads.
3. Vectorized Execution features low compilation time (pre-compiled primitives), accurate profiling and adaptivity.

4. Data-Centric Compilation shines at stored procedures and language integration, as these can be compiled into functions.

In Chapter 7, we show that point 2 depends on the actual hardware setup. Thus, slightly weakening the findings by Kersten et al. In practice, the hardware setup is not always controllable. As a consequence, neither of the two paradigms can be optimal (given the uncontrollable setup).

2.4.6 Conclusion

There are multiple options to evaluate queries. We described Iterator, Data-Centric Compilation, Columnar Execution, Vectorized Execution. Each has their advantages and disadvantages in interpretation, materialization overhead as well as other performance characteristics that depending on the hardware "at hand". Thus, if one specific paradigm has to be chosen:

"There are no solutions. There are only trade-offs." (Thomas Sowell)

2.5 Domain-Specific Languages

Since there are many possible languages to encode programs, we focus on languages tailored to analytical queries, which are: Plans, Comprehensions, Vector Models and Low-Level Imperative languages. Figure 2-1 classifies the most common works.

2.5.1 Plans

Plans are frequently used in data management systems. Most commonly, they either describe logical or physical plans in relational algebra. Recent works, using the concept of low-level plan operators (LOLEPOPs) [HCL+90, Loh88], break queries and operators into primitive operations and are conceptually very similar to VOILA, which we propose in Chapter 5. In a query engine, LOLEPOPs might not be so low-level (e.g. describe a hash join via `FindMatch` and `GatherPayload`), or require a more complex environment to function (program instead of directed acyclic graph). More high-level LOLEPOPs tend to lead to higher (re-)implementation effort, as the operator needs to be implemented for every flavor. Very low-level LOLEPOPs would be similar to VOILA which requires state management/update. This also holds for LOLEPOP-based representations such as Hawk [BKF+18].

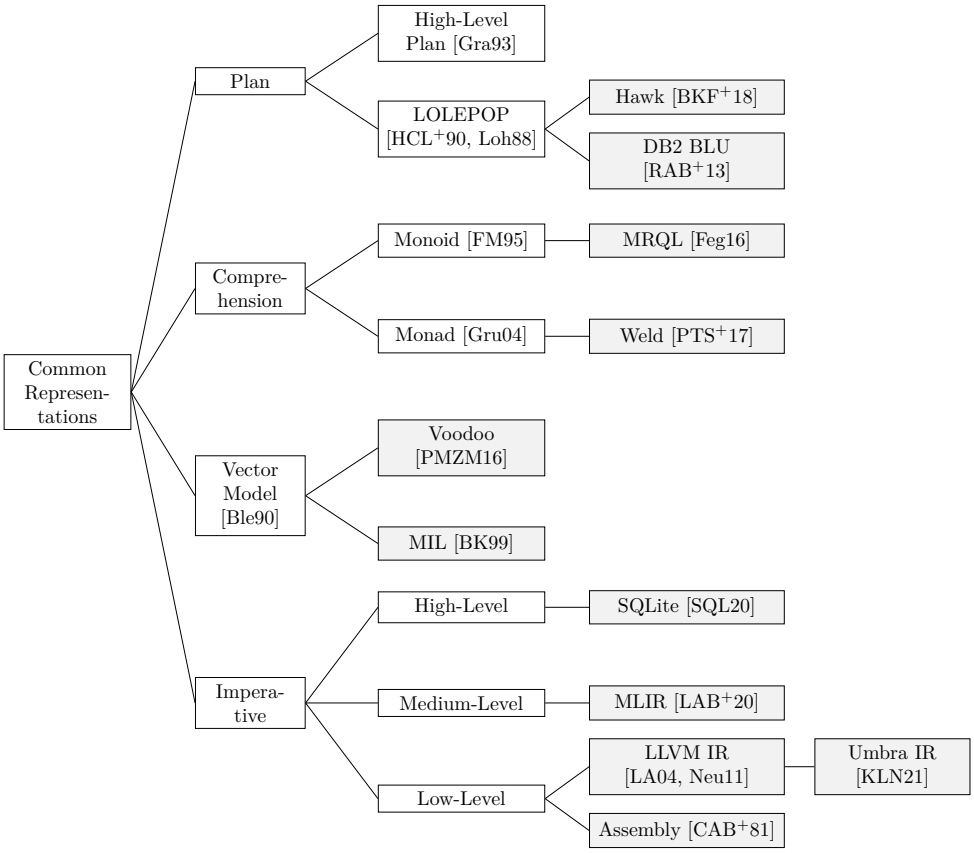


Figure 2-1: Classification of domain-specific languages for analytical queries. Specific examples are marked in gray. Admittedly, the separation of imperative languages into high-, medium- and low-level is almost arbitrary because the borders are not clearly definable and comparison is relative to the choice of examples.

2.5.2 Comprehensions

Comprehensions describe enumerations as the composition of scalar operations. Well-known classes are Monad [Gru04] (e.g. Weld [PTS+17]) and Monoid [Feg16] comprehensions. In general, comprehensions heavily rely on scalar operations and, therefore, lose information about data-parallelism, e.g. branches are introduced. This requires re-discovery of data-parallelism, when e.g. SIMD or GPUs are supposed to be used. Like many high-level languages, Weld [PTS+17] allows the creation of temporary collections (arrays, lists, etc.) and, therefore, requires deforestation [Wad88] to eliminate unnecessary intermediate data structures. Deforestation is a very hard optimization problem and not fully solvable in a reasonable time. VOILA avoids creating such intermediates through a more complex program.

2.5.3 Vector Models

Vector Models describe queries as the application of certain primitive functions onto vectors of data. Notable examples are MIL and VOODOO. MIL [BK99] (or now MAL) has been the foundation of query execution in MonetDB [IGN+12]. It defines operations in a column-at-a-time fashion. However, non-trivial plan operators, such as hash join or hash group-by, commonly translate to complex primitive expressions in MIL. For example, a join in MonetDB translates into multiple binary `JOIN` primitives. For design space exploration, this would require re-implementing many different joins. Instead, VOILA decomposes complex operators into sequences of statements and expressions, e.g. a hash join will end up as a sequence of hash table lookup, hash table insert, gather, etc. VOODOO [PMZM16] has no notation of hash tables (hash join, hash group-by) and "deliberately omits control-statements' [PMZM16], instead VOILA embraces both.

2.5.4 Low-level Imperative Languages

Low-level Imperative Languages typically break complex operations (e.g. hash join) into smaller very specific instructions. To allow fast execution (of generated) programs, their instructions tend to be close to the actual hardware. Due to their performance, low-level languages are frequently used as compilation targets. Most notably, SystemR [CAB+81] generates/generated assembly code, as well as HyPer [Neu11] and Umbra [KLN21] which both generate LLVM IR [LA04]. Umbra, however, also allows a "Flying Start" [KLN21, GBE+23] by directly emitting assembly. Compared to LLVM IR [LA04], VOILA is much less low-level. For instance, VOILA supports multiple execution strategies (tuple-at-a-time, vector-at-a-time). LLVM's auto-vectorization could come somewhat "close". However, not all algorithms are vectorizable, e.g. a selection might introduce a branch and, hence, break possible auto-vectorization for the whole operator/pipeline. Similar low-level languages to LLVM IR are assembly or C. But, these also require re-discovering data-parallelism via auto-vectorization.

2.5.5 Conclusion

We presented an overview of languages able to abstract the physical implementation of queries. There are, however, a few trade-offs. We focus on a brief general discussion.

Abstraction. High-level languages are typically good at abstracting many specific details and, thus, tend to provide more freedom over *how* a program is ex-

ecuted. This can allow many (high-level) optimizations (e.g. the combination of two functions can be simplified into a compound function, for which a specific CPU instruction exists).

Deforestation. Higher-level languages typically bring complex data structures with them (e.g. lists, or lists of lists). Often, they represent intermediates (e.g. result of a function). To achieve high performance, these intermediate data structures should be avoided, whenever possible. Complex intermediates can be removed automatically (Deforestation [Wad88]). However, since it is computationally expensive (exponential time complexity [FW88]), in practice, intermediates cannot always be removed. In such cases, performance tends to suffer (e.g. such intermediates might require costly sequential pointer chasing).

Low-Level Languages, typically, encode how a program is executed. Thus, the use of specific data structures is left to the program author. High-level optimizations are typically not possible without code changes.

Essentially, we would wish for *one* language expressive enough to support all queries, and allow all optimizations/transformations on all levels (high-level, low-level and everything in between). Unfortunately, it is unclear whether such a language exists. If it does, it would probably be rather domain-specific.

CHAPTER 3

Compact Types & In-Register Aggregation

3.1 Introduction

To explore the design space for instance-based optimizations, it appeared reasonable to start with a simple query and minimize the overall runtime for that specific query for a given (set of) hardware. Here, we try to answer the question:

How would a human implement the fastest version of TPC-H Q1, knowing data distributions?.

3.2 Compact Data Types

The database schema typically influences the data representation chosen for query execution. For example, in Q1 (Listing 3.1), we notice that column `l_tax` is of SQL type decimal(15,2). The more efficient, and often used, way of implementing decimal(x,y) types in SQL is to represent them as integers $x.10^y$, where the system remembers y for each decimal expression. As such, we can deduce that `l_tax` will fit into a 64-bit integer ($log_2(10^{15}) < 64$).

Listing 3.1: TPC-H Query 1 (with DELTA=90)

```
SELECT
  l_returnflag,
  l_linestatus,
  SUM(l_quantity),
  SUM(l_extendedprice),
  SUM(l_extendedprice * (1 - l_discount)),
  SUM(l_extendedprice * (1 - l_discount) *
    (1 + l_tax)),
  AVG(l_quantity),
  AVG(l_extendedprice),
  AVG(l_discount),
  COUNT(*)
FROM
  lineitem
WHERE
  l_shipdate <= date '1998-09-02'
GROUP BY
  l_returnflag, l_linestatus
ORDER BY
  l_returnflag, l_linestatus;
```

In terms of real data population, though, the actual value domains are much more restricted:

- `l_tax` only contains values (0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8), which represented as integers (value range $[0, 80]$) fit a single byte. Similarly, `l_discount` only contains values 0.0 and 0.09, so the integer range is even more restricted: $[0, 9]$.
- `l_returnflag` and `l_linestatus` contain single-character strings that range respectively between `'A'-'R'` and `'F'-'O'` (in Unicode, the code ranges are $[65, 82]$ and $[70, 79]$). It is possible to represent both as bytes.
- `l_extendedprice` varies between $[901.00, 104949.50]$, thus its integer domain [90100, 10494950] fits a 32-bit integer (it actually needs 24 bits).
- the expression `(1+l_tax)` produces values in the domain $1.0, 1.8]$, which in the integer representation of `decimal(15,2)` becomes $[100, 180]$. This still fits a single unsigned byte.
- `(1-l_discount)*(1+l_tax)` produces values in the domain $[0.91, 1.8]$, which in the integer representation of `decimal(15,2)` becomes $[91, 180]$. This still fits a single unsigned byte. Arguably, to guarantee no loss of precision and exact answers, in a multiplication the return type could be `decimal(30,4)`[1], which would lead to $[9100, 18000]$, which still fits a two-byte short.
- following the latter, `l_extendedprice*(1-l_discount)` produces integer values in the domain $[8199100, 1049495000]$, fitting a 32-bit integer.

[1]Please note that this is not specified by the SQL standard. An alternate policy could be to deliver the maximum precision of both * operands, i.e. `decimal(15,2)` again, or `decimal(30,2)`.

- `l_extendedprice*(1-l_discount)*(1-l_tax)` produces integer values in the domain [819910000,188909100000], for which a 64-bit integer must be used.

We can see that the two additions and subtractions in Q1 can be done using single-byte logic. The three different multiplications (that remain after common-subexpression elimination) should be done with 16-bit, 32-bit and 64-bit precision, respectively. The often-followed policy of using the implementation type implied by the SQL type (i.e. 64-bit integer due to decimal(15,2)) wastes SIMD effort. We can calculate this as doing two (add/sub) plus three (mult) hence 5 operations on 64-bit = 320 bits, whereas doing two on 8-bit, one on 16-bit, one on 32-bit and one 64-bits leads to 128-bit (3x less "SIMD-work").

With that in mind, a natural question is: how can the query compiler know about these column domains? Obviously, there are statistics that analytical database systems already keep, for instance ZoneMaps (also called MinMax) indexes specifically provide this information, not only for whole table columns, but also for specific stretches.

A more critical variant of the question is how a system can guarantee tight [Min,Max] bounds if the system can be updated. A single (inserted) large value could hence force all computation onto the widest possible integer type, which seems a serious performance vulnerability given that real data (unlike TPC-H) is noisy.

This chapter does not aim to provide a definite answer to that question, rather aims to instill research towards achieving systems that both keep most of their data tightly represented, yet can handle updates. We do observe that columnar compressed systems typically handle updates out-of-place, in C-store this is called Read-Store and Write-Store [SAB+05]. VectorWise similarly performs updates in a differential structure called Positional Delta Tree (PDT [HZN+10]). HyPer stores its DataBlocks [LMF+16] for cold data in compressed form, whereas hot data is stored uncompressed. Until now, these systems move data periodically towards the compressed format based on data hot-ness. However, it is envisionable to do this not solely based on data temperature, but to also let data distributions play a role. That is, one could let outliers stick around in the Write-Store longer, or move them to an uncompressed block (HyPer terminology) such as not to disrupt the tight [Min,Max] ranges that can be kept on the compressed Read-Store.

One can alternatively see execution on compact data types as a case of "compressed execution" [AMF06]. The compression schemes of VectorWise use "patching" [ZHNB06], which means that the compression scheme stores outliers in a sepa-

rate location, and they are decompressed in a separate "patching phase". One could envision systems where execution on a range of tuples is split up in disjunct subsets of tuples that within each subset have a homogeneous (compressed) representation – which is very similar to patching. This way, one can JIT-generate specific code for each such specific subset.

Guarding against Overflow in Aggregates. Finally, we turn our attention to the task of calculating the count() and sum() aggregates (avg() is a combination of these two), specifically focusing on the problem of overflow in sum(). A database query processor is required to produce correct answers. It might occur that due to enormous data sizes (very many, large values) a sum() result can no longer be represented by the system. Rather than giving a wrong answer in such a case, the system should give a runtime error: it should check for numeric overflow.

Checking for overflow is an unsexy database engine topic that has received scant attention in literature. Many database systems are implemented in C or C++ and these programming languages do not expose the overflow flags that modern CPUs have for their scalar instructions. This means that overflow-checking involves rather convoluted extra if-then-else tests, where the if-branch must perform various comparisons. This represents extra work that needs to be executed for every addition in the sum(). Given the simple work a sum() performs, this can impose significant overhead.

Interestingly, the use of the LLVM environment by HyPer gives that system access to CPU overflow flags, as these are exposed by LLVM. This significantly reduces the checking overhead, but there still is a necessity to perform an if-then based on the overflow flag, as an error needs to be raised in case of overflow. An optimized method of guarding for overflow is to continue execution but just OR the overflow flag results of all operations together. Only at the end of the query (or morsel) an error is raised if the flag got set. This further reduces overhead, but due to the OR work some overhead still remains.

In the previous sections, we discussed how SIMD-friendly systems might be very aware of the domains of the columns involved in (sum) expressions, to choose compact representations for them. Such systems could often move from overflow-checking to *overflow prevention*. In fact, for the simple additions, subtractions, and multiplications we discussed before, the knowledge about the domains not only allowed us to choose compact data types, it also allows skipping overflow checking altogether (as we know it cannot occur). Please note that SIMD instruction

sets, unlike scalar CPU instruction sets, do not provide support for overflow flags, generically, so overflow prevention is crucial for SIMD.

However, for (sum) aggregates, we cannot so easily avoid overflow checking. If we knew a limit on the amount of tuples that will be aggregated, we could obtain bounds on the outcome of the aggregate function by multiplying this limit with the Max and Min (if negative) statistic on the aggregated expression. Often, a query pipeline can derive a particular limit on the amount of tuples that pass through it. For instance, in HyPer, execution is morsel-driven, so the outer-loop will only be taken as many times as the morsel size. Furthermore, the full table (partition) size is also such a bound, however in case of joins and its worst case of Cartesian product, multiple such bounds would need to be multiplied to get a reliable limit.

Alternatively, the system might impose a maximum-tuple-bound at a particular extreme value that will never occur in practice (e.g. one US trillion or 2^{40} tuples passing through a single thread – it would take hours and the system could, in fact, raise a runtime error if it would pass in reality). If we take that approach to Q1, we can see that under the logic of bounding the max-tuples-per-thread to 2^{40}:

- `sum(l_discount)`, `sum(l_quantity)`, `sum(l_extendedprice)` will fit a 64-bit integer.
- `sum(l_extendedprice*(1-l_discount)*(1+l_tax))`, and `sum(l_extendedprice*(1-l_discount))` will need a 128-bit integer.

Of course, if the decimal type of the summed expression has a lot of precision and the actual values stem from a wide range, then 128-bit integers might still not be enough in terms of this limit. Even though systems often do not support any decimal larger than what fits in a 128-bit integer, queries with these characteristics should not simply fail because, in practice often, the result will fit (after all, we are computing a worst-case bound). For those cases, a system trying to apply overflow prevention in aggregates does need to have a back-up solution that in fact performs the (slower) overflow checking, using CPU overflow flags or comparisons. One can argue whether operating on 128-bit integers is efficient, anyway. These data types are supported by C/C++ compilers (`__int128_t`). However, modern CPUs do not support them natively, so every simple calculation must be mapped to multiple 64-bit calculations that are combined. Moreover, there is no SIMD support for such 128-bit calculations at all.

In all, systems that support overflow prevention as an optimization for aggregate calculations are still faced with significant cost for these primitives. This is one of

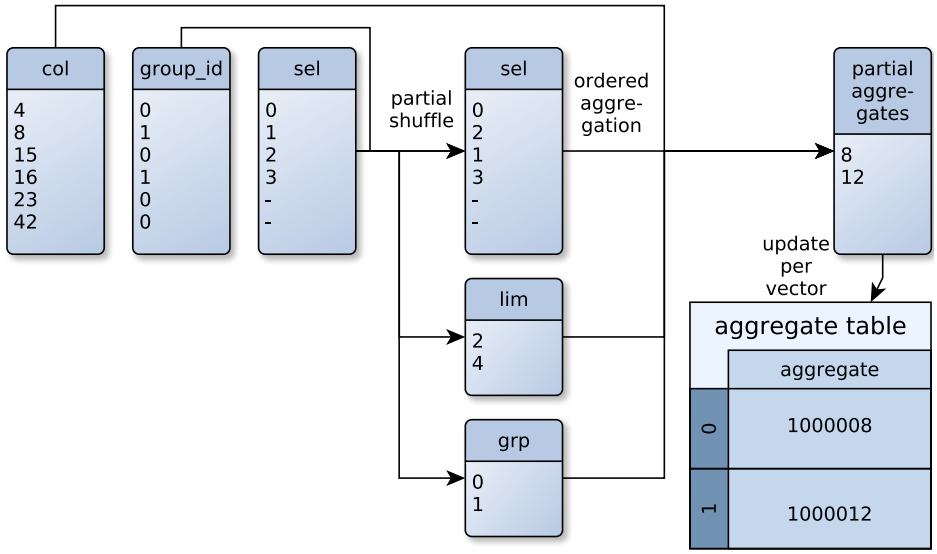


Figure 3-1: In-register aggregation with one column (*col*), index into aggregate table (*group_id*) and selection vector (*sel*)

the reasons that HyPer could claim such a performance advantage over VectorWise in Q1 in [Neu11]. The "in-register" aggregation that we will introduce in the sequel reduces memory access (by aggregating in a register instead) but has as additional advantage that overflow prevention can be applied much more aggressively. In fact, it allows doing almost all sum() calculation on the 64-bit granularity which, in turn, *is* SIMD-friendly.

3.3 In-Register (Group-By &) Aggregation

In the context of Q1, with few columns, the simple implementation with a thread-private aggregate table is used. As described earlier, finding the aggregate slot (computing the group-ID) can be done very fast using the perfect identity hash optimization (otherwise, a hash-table could be used). When SIMD-izing such aggregation, special care is needed to avoid anomalies like lost updates in the aggregation table. This could happen because when multiple aggregates are updated within a SIMD register, it is not guaranteed that these (partial) aggregates belong to different groups. Thus, a naive *scatter* to the aggregation table might lead to lost updates. This issue can be addressed in multiple ways: (1) the system might prevent conflicts, (2) conflicts are detected and resolved (in register) before the aggregation table is updated, or (3) the layout of the aggregation table is modified in a way to allow for conflict-free updates.

Contrary to array-based aggregation, we propose in-register aggregation which prevents conflicts, and reduces the amount of load/stores (scatter/gathers). It is based on the idea that it is favorable to virtually reorder all active vectors, via modification of the selection vector, such that all tuples belonging to the same group appear consecutively. Afterward, the (virtual) order can be exploited to aggregate more efficiently.

Figure 3-1 depicts an example. The algorithm consists of two phases: First, a – *partial shuffle* – is performed, which uses the *group_id*s and the selection vector to build a new permutation of the selection vector in which all the groups are clustered.

The algorithm consists of two steps: First, we store the per-group consecutive positions. We use two arrays. The first array maps the group-ID to a position in the second array (indirection to minimize the active working set). The second array stores the consecutive positions in slots. Whenever a new group appears, we allocate vector-size many slots in the second array (edge case: all tuples in a stride belong to one group). Note that this allocation of slots can fail in case of too many groups. Afterward, we build the new selection vector by iterating over the groups and copying the positions. In addition, this step also computes the group boundaries (*lim*) and memorizes in which order the groups appear (*grp*) which will be needed in the next phase.

Using a lot of care, we were able to create an initial fast scalar version of this shuffle that operates at just 6 cycles/tuple on Sandy Bridge. It works similar to the simplified code in Listing 3.2.

Additionally, it is possible to SIMD-ize *partial shuffle* using AVX-512. In our implementation we exploit the conflict detection instruction (`vpconflictd`) which, in the 32-bit version, takes a 16 (32-bit) integers as its input and produces 16 (32-bit) masks. For one element i it checks whether the elements from 0th to $(i-1)$th (previous elements in the SIMD lane) are equal to the element i. If this is the case for an element k, then the k-th bit of mask i will be 1; otherwise it will be 0. If done on the SIMD register full of group-IDs, it is possible to determine how many tuples end up in the same group before a given tuple (number of 1 bits in mask, i.e. population count [2]). Together with a per-group offset, this produces unique indexes, which are used to scatter out the positions from the selection vector. The per-group offset is the current location where new positions into a group will be written, and is essen-

[2]Sadly, in AVX-512 there is no instruction that calculates the population count in parallel. Hence, we implemented a fast population count using an in-register lookup table with 32-bit granularity, which in our specific case leads to 4 lookups done via `vpermd`.

Listing 3.2: Partial Shuffle in simplified version without low-level performance improvements. It produces the inputs for the subsequent ordered/in-register aggregation (`lim`, `grp` and `sel`) by virtually reordering the rows such that groups are clustered together.

```
int partial_shuffle(int* lim, u64* grp, int* sel, u64* group_ids,
    int* in_sel, int in_num, u16* grppos, u16* selbuf) {
  u16* buf_ins=&selbuf[0];
  u16* buf_end = &selbuf[GROUP_BUF_SIZE];
  u16 buf[MAX_ACTIVE_GROUPS];
  u16* max_gid = &buf[0];

  // 1. pre-group
  if (sel) {
    for (int i=0; i<in_num; i++) {
      int k = in_sel[i];

      auto gid = group_ids[k];
      auto dstpos = &grppos[gid];
      if (!(*dstpos)) {
        // allocate new group (rather unlikely)
        if (buf_ins >= buf_end) {
          // too many groups (very unlikely)
          return -1;
        }
        *dstpos = buf_ins;
        buf_ins += MAX_VSIZE;
        *max_gid = gid;
        max_gid++;
      }
      **dstpos = k;
      (*dstpos)++;
    }
  } else {
    // similar, with 'k' = 'i'
  }

  // 2. build output
  int num_groups = 0;
  int num_tuples = 0;

  for (auto curr=buf; curr < max_gid; curr++) {
    auto gid = *curr;

    // copy ordered groups
    auto pos = grppos[gid];
    auto num = (pos - selbuf) % MAX_VSIZE; /* Optimized into bit-wise op */
    auto start = pos - num;
    for (int i=0; i<num; i++) {
      sel[i] = start[i];
    }
    grppos[gid] = NULL;
    sel += num;
    num_tuples += num;

    // update resulting group boundaries
    lim[num_groups] = num_tuples;
    grp[num_groups] = gid;
    num_groups++;
  }

  return num_groups;
}
```

Listing 3.3: Ordered Aggregation: When number of distinct groups is low and data is clustered on the group ids, we can save memory accesses by first accumulating partial aggregates in registers and, eventually, write them back into the table. However, this requires to compute the group boundaries (`group_ids`, `lim` and `num_groups`) first, which can be done once for multiple aggregates (with the same keys)

```
void ordaggr_sum(i128* aggr_col, u64* group_ids, int* lim, i32* values,
    int num_groups) {
  int k = 0;
  for (int g=0; g<num_groups; g++) {
    // locally pre-aggregate in register
    i64 sum = 0; // pre-aggregate typically fits into smaller type
    for (; k<lim[g]; k++) {
      sum += values[k];
    }
    aggr_col[group_ids[g]] += sum; // update table
} }
```

tially an array of 16-bit indexes indexed by the group-ID. Afterward, the per-group offsets have to be updated by scattering the (unique) indexes back. Note that in case of conflicts, the highest SIMD lane [Int16] wins, which – here – is the highest index. After the previous steps have been done for the whole selection vector, the new selection vector will be built. This, per-group, copies the stored positions into the resulting selection vector. On Knights Landing our SIMD-ized *partial shuffle* was able to operate at 9 cycles/tuple whereas the initial scalar version runs at 13 cycles/tuple.

The second phase calculates the aggregates using ordered aggregation (Listing 3.3). This aggregation algorithm exploits the fact that groups now appear in order. Hence, it just aggregates the column values until the group boundary is detected, then it updates the final aggregates in the table using the partial aggregate. The depicted *partial aggregates* array does not exist because the ordered aggregation directly updates the aggregate in the aggregate table. Note that ordered aggregation can relatively easily be implemented using SIMD.

This method avoids read/write conflicts which would otherwise occur on a per-tuple basis. Further, the type of the partial aggregate can be restricted because we know that the partial sum is computed maximally for the whole vector. The vector-size, say 1024, is a much tighter bound than 2^{40}. This means all partial aggregates in Q1 fit in a 64-bit register. In case of Q1, this removes almost all expensive 128-bit arithmetic from the hot-path. The 128-bit additions still have to be done, but only once per group, per vector.

We finally note that "in-register" aggregation is an adaptive way of handling aggregates. The partial shuffle is very fast, but can fail, if there are too many distinct

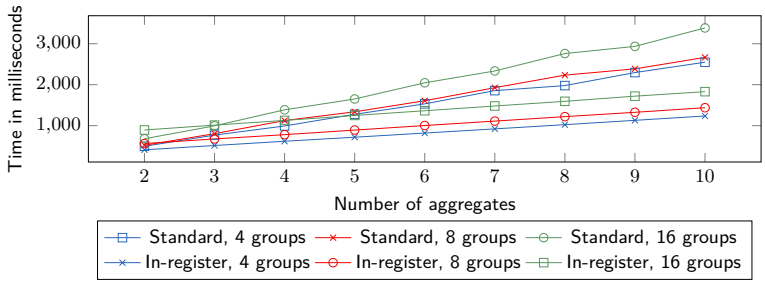


Figure 3-2: Standard vs. in-register aggregation, 128 bit aggregates

group values in the vector (more than 64). In that case, normal vectorized aggregation primitives are used, and we use exponential back-off to re-try with in-register aggregation much later. Such *micro-adaptive* behavior is easy to integrate in vectorized query processors [RBZ13].

3.4 Evaluation

For most experiments we used a dual socket machine with two Intel Xeon E5-2650 v2 (Sandy Bridge) and 256 GiB main memory running Fedora 24 with Linux kernel version 4.7.9 and GCC 6.3.1. To reduce interference with NUMA effects, the test process was bound to processor 0 (on NUMA node 0) whereas all memory allocations were bound to NUMA node 0.

3.4.1 Standard vs. In-Register Aggregation

To find out in which situations the in-register aggregation excels, we compare it against the standard array-based aggregation on the – above-mentioned – Sandy Bridge machine. The setup of the experiment is as follows: We used a vectorized execution pipeline. For each column, we load a number of vectors and then aggregate each column into a sum of each column's values for each group. Each vector consists of up to 1024 values and the size of the input is 10^8 tuples. We generate a uniformly distributed *group-ID* (key) column. These group-IDs are spread across the aggregate table in row-wise layout with a spreading factor of 1024. All columns including the group-ID are 64-bit integers. The aggregates are 128-bit integers, and it is assumed that partial aggregates fit into 64-bit integers.

Figure 3-2 compares standard aggregation with in-register aggregation in the block-at-a-time processing model, i.e. for n aggregates n aggregation primitives have to be called. It can be seen that for a low number of aggregates both aggregation

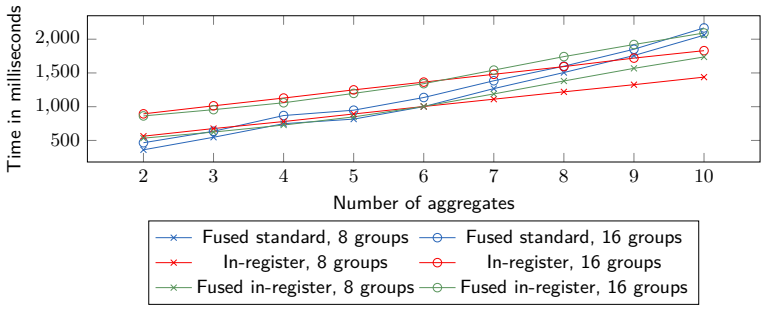


Figure 3-3: Fused Standard vs. in-register aggregation, 128-bit aggregates

strategies perform almost equal while with increasing number of aggregates in the query, the standard aggregation shows a worse performance mainly due to (1) with more aggregates the cost of the partial shuffle is better amortized (2) Standard aggregation touching the same cache lines containing the aggregates in the aggregate table many times and (3) load/store conflicts when only a few entries (groups) in the aggregate table are thrashed.

Problem (2) can be mitigated by fusing the aggregation primitives together through merging them into one loop that updates multiple aggregates (essentially "Loop Fusion", note that this typically requires JIT compilation, see [SZB11], as the combination of aggregates is only known at query time). In Figure 3-3, the fused standard aggregation is compared to the fused and non-fused in-register aggregation. In our case, Loop Fusion improves the standard aggregation's performance in comparison to in-register aggregation but problems (1) and (3) (in case of few distinct group values) still hold, which is the reason in-register aggregation out-performs the standard aggregation for more than 6 (8 groups) resp. 8 (16 groups) aggregates. Interestingly, fusing the in-register ordered aggregation primitives provides hardly any benefit (≤ 6 aggregates) and can even be detrimental (> 6 aggregates).

3.4.2 Q1 Flavors

Based on the promising results of Section 3.4.1 we implemented multiple versions of Q1 to compare their response times, single-threaded. Starting from 3 base implementations, we derived different flavors: A vectorized X100-alike implementation which processes a block-at-a-time and prevents overflows, a HyPer-alike which processes a tuple-at-a-time and checks for overflows and a handwritten AVX-512 version which processes a block of 16 tuples at a time and prevents overflow. Flavors include different methods for overflow detection and prevention, different aggregation techniques, varying aggregate table layout, as well as, different data representations.

p

Table 3.1: Q1 Flavors

Flavor name	X100	HyPer	Data types	Overflow	Layout	Aggregation	Comments
X100 Full NSM Standard	✓	-	Full	Prevent	NSM	Standard	
X100 Full DSM Standard	✓	-	Full	Prevent	DSM	Standard	
X100 Full NSM Standard Fused	✓	-	Full	Prevent	NSM	Standard & fused	
X100 Full NSM In-Reg	✓	-	Full	Prevent	NSM	In-register	
X100 Compact NSM Standard	✓	-	Compact	Prevent	NSM	Standard	
X100 Compact DSM Standard	✓	-	Compact	Prevent	DSM	Standard	
X100 Compact NSM Standard Fused	✓	-	Compact	Prevent	NSM	Standard & fused	
X100 Compact NSM In-Reg	✓	-	Compact	Prevent	NSM	In-register	
X100 Compact NSM In-Reg AVX-512	✓	-	Compact	Prevent	NSM	In-register	Optimized for AVX-512
HyPer Full	-	✓	Full	Detect (flag)	NSM	Standard	
HyPer Full OverflowBranch	-	✓	Full	Detect (branch)	NSM	Standard	
HyPer Full NoOverflow	-	✓	Full	Prevent	NSM	Standard	
HyPer Compact	-	✓	Compact	Detect (flag)	NSM	Standard	
HyPer Compact OverflowBranch	-	✓	Compact	Detect (branch)	NSM	Standard	
HyPer Compact NoOverflow	-	✓	Compact	Prevent	NSM	Standard	
Weld	-	-	Full	Prevent	NSM	Standard	
Handwritten AVX-512	-	-	Full	Prevent	NSM	Standard (SIMD)	handwritten in AVX-512
Handwritten AVX-512 Only64BitAggr	-	-	Full	Prevent	NSM	Standard (SIMD)	handwritten in AVX-512 all aggregates in 64-bit arithmetic

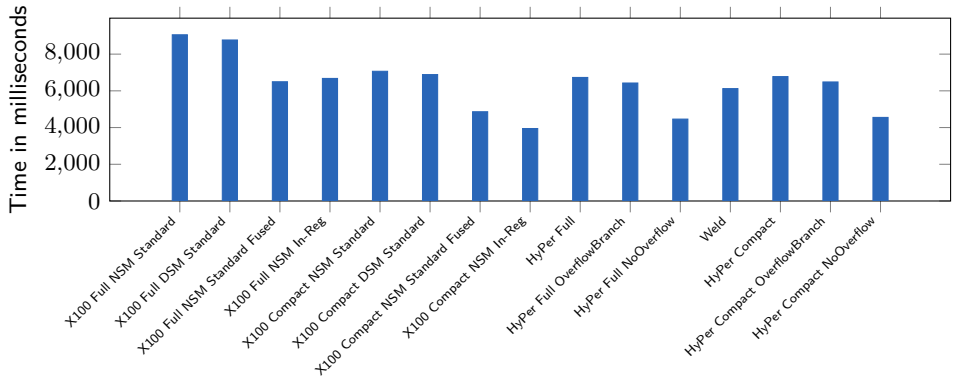


Figure 3-4: Different Q1 implementations on Sandy Bridge using scale factor 100.

We tested these flavors on two machines. One is the above-mentioned Sandy Bridge machine. Figure 3-4 visualizes the response times each flavor achieved. Refer to Table 3.1 for a description of each flavor. It can be seen that the vectorized approach together with in-register aggregation, compact data types and overflow prevention (*X100 Compact NSM In-Reg*) can outperform the other approaches. As visualized, this approach also beats the HyPer-alike implementations with and *without* overflow detection. Generally speaking, fusing the aggregate calculation into one primitive improves the response time because the aggregates which are accessed concurrently are often in the same cache-line. Further, standard aggregation can be beaten by in-register aggregation in Q1. NSM appears to be the better choice, as aggregates are closer together in memory as compared to DSM and compact data types tend to speed up vectorized processing whereas, in Q1, they slow down HyPer-alike implementations.

Additionally, we evaluated our Q1 flavors on an Intel Xeon Phi 7210 (Knights Landing) with 110 GB of main memory running Ubuntu 16.04 LTS using Linux Kernel 3.10.0 and GCC 5.3.1. The main memory is split into different NUMA regions: Four regions à 24 GB represent the normal (DRAM) main memory, whereas the other four NUMA regions represent the accessible High-Bandwidth Memory (HBM). We limited the scale factor to 75 because scale factor 100 would exceed the local (DRAM) main memory capacity of a single NUMA node and would have caused interference with High Bandwidth Memory and/or cross-node NUMA traffic.

Figure 3-5 plots each flavor's response time. In general, it shows a similar picture as with the Sandy Bridge machine with one exception being the handwritten AVX-512 implementation(s) which are the fastest of the flavors tested. Moreover, it can be seen that other implementations can be optimized using AVX-512, i.e. wider

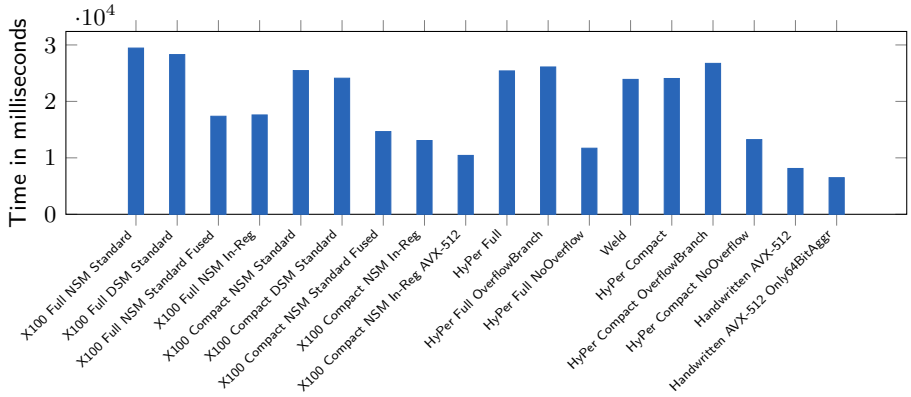


Figure 3-5: Different Q1 implementations on Knights Landing using scale factor 75.

SIMD and more complex operations are available as instructions. Additionally, it can be said that the HyPer implementation performs very slowly, which is caused by overflow detection through GCC's builtins, whereas the implementation without overflow detection performs better than the average.

3.5 Conclusion

In this chapter, we discussed how to minimize the overall runtime of a specific query (TPC-H Q1) by exploiting the data distributions (range of values & upper bound on cardinality). Therefore, we proposed two rather generic methods to improve Q1 and similar queries: Compact Data Types and In-Register Aggregation.

Compact Data Types. Fitting data types to the actual data, instead of using the user-provided schema, can lead to significantly faster arithmetic. In our experiments, this is caused by the better utilizing of SIMD registers. 4× smaller data types (by transforming 32-bit integers to 8-bit integers) allow processing 4× number of values, keeping the number of cycles constant and assuming appropriate SIMD instructions exist. Not only can thinner data types lead to faster arithmetic, but they can also significantly reduce the footprint of data structures stored in memory (we use a more extreme variation of this idea in the following chapter). Note that shrinking data types can be regarded as lightweight compressed execution that, not only, brings significant benefits, but is also relatively easily added to existing DBMSs.

In-Register Aggregation. In certain scenarios, a group-by/aggregation produces only a relatively low number of groups (a "handful"). In such cases, a hash-based group-by can commonly use a perfect hash function and, consequently, avoid rela-

tively costly hash bucket traversal and key checks. The aggregates are then computed for each bucket (basically updating values in a potentially large array). For a "handful" of groups, however, this is suboptimal as this computation (a) introduces repeated read-write conflicts (in the underlying hardware), (b) introduces, generally speaking, excessive memory-access (even though most accesses are relatively cheap in a row-wise layout, thanks to caches [ZNB08]) and (c) relies on updating values in the data type calibrated on the worst-case (i.e. updates are expensive). Therefore, we introduce in-register aggregation, a technique that exploits the previously known ordered aggregation to quickly aggregate values using accumulator variables (typically stored in 1-2 CPU registers, depending on the data type). Note that the accumulator variable can have a thinner type than the aggregate, as it only needs to accumulate one vector full of values (typically 1k). The underlying idea is to re-order the values such that the groups appear ordered. We described such an algorithm that re-orders the groups cheaply, only requiring a few CPU cycles. We have shown that this technique leads to further gains and, thus, outperforms the naive aggregation in TPC-H Q1.

Hardware-Dependent Performance. The combination of In-Register Aggregation and Compact Data Types performed well (i.e. better than either the vectorized and data-centric implementation) on the *Sandy Bridge* and *Knights Landing*. However, on the Knights Landing machine, handwritten AVX-512 code outperformed. If we can further assume the 128-bit `SUM` aggregates fit into 64-bit, the AVX-512 implementation can be further improved (explored more deeply in the next chapter as Optimistic Aggregates in Section 4.3.1).

Summary. The presented optimizations are useful in instances of TPC-H Q1 (and similar queries), but their performance depends on data distribution (number of groups and range of values). While initially this became the fastest single-threaded implementation of Q1, it was eventually "dethroned" by Nowakiewicz et al. [NBH+18], who further improved the group-by/aggregation by exploiting the existence of only 4 groups in Q1 (using 4 SIMD comparisons, one per group, instead of the more expensive, but more generic, In-Register Aggregation).

CHAPTER 4

Compressed Hash Tables & Soviet Strings

4.1 Introduction

Given a physical execution plan, there is a plethora of different implementations to execute the plan – the design space. One interesting dimension is the use of compression. Typically, compression is only used for (base table) scans to elevate the disk bottleneck (and sometimes even the memory wall [ZHNB06]). However, more interesting is, so-called, *Compressed Execution*, the ability to operate on (at least partially) compressed representations, inside the query processing pipeline. This has, at least partially, been explored [AMF06, Łus11, RAB+13]. Still very efficient – and practically useful – new approaches can be found, which we discuss in this chapter:

We focus on compressing (and operating on compressed) hash tables – a frequently used data structure for query processing – as illustrated in Figure 4-1. We discuss Domain-Guided Prefix Suppression, which bit-packs values. Then, we explore Optimistic Splitting which separates parts of values, to optimize towards "hot" values.

Strings often occur in real-life data sets [VHF+18]. Whenever strings are present, they tend to lead to storage overhead and inefficient processing (e.g. comparisons

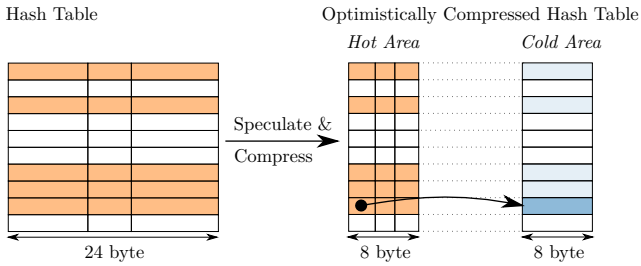


Figure 4-1: Optimistically Compressed Hash Table, which is split into a thin hot area and a cold area for exceptions

have to compare all characters of a string). To alleviate these two disadvantages, we introduce the dynamic string dictionaries (Unique Strings Self-aligned Region) which encodes frequent strings, optimistically and on-the-fly.

4.2 Domain-Guided Prefix Suppression

Domain-Guided Prefix Suppression reduces memory consumption by eliminating the unnecessary prefix bits of each attribute. This enables us to cheaply compress rows without affecting the implementation of the hash table itself, which makes it easy to integrate our technique into existing database systems. In particular, while our system (Vectorwise) uses a single-table hash join, Domain-Guided Prefix Suppression would also be applicable (and highly beneficial) for systems that use partitioning joins [BTAÖ13, SCD16]. Domain-Guided Prefix Suppression also allows comparisons of compressed values without requiring decompression. In the rest of this section, we describe Domain-Guided Prefix Suppression in detail using Figure 4-2 as an illustration.

4.2.1 Domain Derivation

in a query plan, an in-flight column can originate directly from a table scan or from a computation. If a value originates from a table scan, we determine its domain based on the scanned blocks. For each block, we utilize per-column minimum and maximum information (called ZoneMaps or Min/Max indices). This information is typically *not* stored inside the block itself, as this would require scanning the block (potentially fetching it from disk) before this information can be extracted. Instead, the meta-data is stored "out-of-band" (e.g. in a row-group header, file footer or inside the catalog). By knowing the range of blocks that will be scanned, the domain can be calculated by computing the total minimum/maximum over the range.

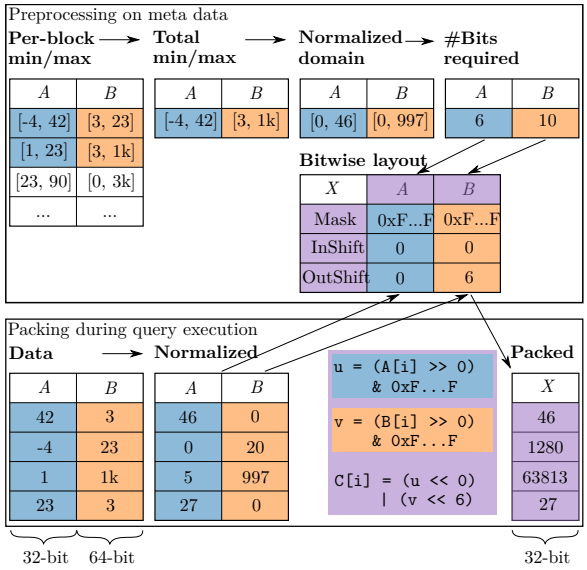


Figure 4-2: Domain-Guided Prefix Suppression

On the other hand, if a value stems from a computation, the domain minimum and maximum can be derived bottom up according to the functions used, based on the minimum/maximum bounds on its inputs under assumption of the worst case. Consider, for example, the addition of two integers $a \in [a_{\min}, a_{\max}]$ and $b \in [b_{\min}, b_{\max}]$ resulting in $r \in [r_{\min}, r_{\max}]$. To calculate $r_{\min}$ and $r_{\max}$ we have to assume the worst-case that means the smallest (r_{min}), respective highest ($r_{\max}$), result of the addition. In case of an addition, this boils down to $r_{\min} = a_{\min} + b_{\min}$ and $r_{\max} = a_{\max} + b_{\max}$.

Depending on these domain bounds, an addition of two 32-bit integer expressions could still fit in a 32-bit result, or less likely, would have to be extended to 64-bit. This analysis of minimum/maximum bounds often can allow implementations to ignore overflow handling, as the choice of data types *prevents* overflow, rather than having to *check* for it. For aggregation functions such as `SUM`, overflow avoidance is more challenging, but in Section 4.3.1, we discuss Optimistic Splitting, which allows doing most calculations on small data types, also reducing the cache footprint of aggregates.

4.2.2 Prefix Suppression

Using the derived domain bounds, we can represent values compactly without losing any information by dropping the common prefix bits. To further reduce the number

of bits and enable the compression of negative values, we first subtract the domain minimum from each value. Consequently, each bit-packed value is a positive offset to the domain minimum. We also pack multiple columns together such that the packed result fits a machine word. This is done by concatenating all compressed bit-strings, and (if necessary) chunk the result into multiple machine words. Each chunk of the result constitutes a compressed column, which can be stored just like a regular uncompressed column.

Figure 4-2 shows an example where column A contains values ranging from $d_{min} = -4$ to $d_{\max} = 42$. First, we transform values from that domain into non-negative integers by subtracting the domain minimum (-4). These values can then be represented using $\lceil log_2(d_{\max} - d_{\min} + 1) \rceil = 6$ bits. Afterward, we "glue" values from multiple columns together (here: A and B) together.

4.2.3 Compression and Decompression

Like many modern column-oriented systems [ABH+13], Vectorwise is based on vectorized *primitives* that process cache-resident vectors (=arrays of single column values). These primitives process items from multiple inputs in a data-parallel (SIMD-friendly) fashion in a tight loop. Consequently, modern compilers automatically translate such code into SIMD instructions for the specified target architecture (e.g. AVX-512). In our vectorized hash table implementation, `pack` primitives compress and "glue" multiple inputs together to produce one intermediate result. Later, this intermediate result is then stored inside the hash table. With all the inputs and the one output being cache-resident vectors, the compression itself happens in-cache. For bit-packing, our `pack` primitives look similar to the following pseudocode, which packs parts of `n` 32-bit and 16-bit integers into 32-bit integers:

```
void pack2_i32_i16_to_i32(i32* res, int n,
    i32* col1, i32 b1, int ishl1, int oshr1, i32 m1,
    i16* col2, i16 b2, int ishl2, int oshr2, i32 m2) {
  for (int i=0; i<n; i++) {
    // Select portion of input and cast to result's type
    i32 c1 = ((col1[i] - b1) >> ishl1) & m1;
    i32 c2 = ((col2[i] - b2) >> ishl2) & m2;
    // Move to output positions
    res[i] = (c1 << oshr1) | (c2 << oshr2);
} }
```

After bit-packing, we scatter the intermediate results into its final positions in the hash table. For improved cache-locality, the hash table is stored in row-wise layout (NSM) [ZNB08]. Therefore, we need to convert columnar vectors (DSM) into tuples (NSM). An interesting observation is that in NSM two subsequent column values are

precisely one row width apart. Hence, for each attribute, we calculate an attribute multiplier `stride := row_width / attr_width` which projects the row width onto the current attribute's width. When scattering attributes from vectors into `i`th NSM record, we calculate its final position `i * stride` and copy each attribute to its respective position.

When decompressing values, we fetch up to 4 columns from the hash table and directly decompress them. For decompressing a vector of `n` packed 16-bit integers from 32-bit and 16-bit integers at positions `idx` in the hash table, this leads to the following pseudocode (2-column example):

```
void unpack2_i32_i16_to_i16(i16* res, int n, int* idx, i16 b,
    i32* col1, int ishr1, int oshl1, i16 m1, int s1,
    i16* col2, int ishr2, int oshl2, i16 m2, int s2) {
  for (int i=0; i<n; i++) {
    // DSM (columnar) position -> NSM (row) position
    int idx1 = idx[i] * s1;
    int idx2 = idx[i] * s2;
    // Extract relevant bits from NSM record
    i16 c1 = (col1[idx1] >> ishr1) & m1;
    i16 c2 = (col2[idx2] >> ishr2) & m2;
    // Stitch back together
    res[i] = (c1 << oshl1) | (c2 << oshl2) + b;
} }
```

Notably, compression and decompression operate in a non-intuitive fashion: Both process m inputs and produce *one* output. This particular approach has two advantages: (a) In contrast to approaches with multiple outputs, it allows decompressing specific columns without enforcing decompression of neighboring cells. This allows an efficient mix of key checks on compressed data together with key checks on bit-packed non-integer data, most notably strings. (b) We concatenate bit-strings directly in registers, as opposed to approaches that partially compress/decompress, which require multiple rounds of reading/writing from/to output vectors to concatenate partial output vectors into the final output.

We implemented our primitives similarly to the ones shown above and improved them further with "micro-adaptive" optimizations [RBZ13]. (1) Even though the implementation of `pack` supports selective processing, it can dynamically decide to process its inputs fully instead whenever $\geq 25\%$ of the tuples in a batch are still active (selected); which favors SIMD. (2) In case all base values are zero, we avoid the integer subtraction, in case of `pack`, or addition, in case of `unpack`. This further reduces the number of required operations if normalization is not required.

4.2.4 Operating on Compressed Keys

Domain-Guided Prefix Suppression also allows comparing compressed values themselves (without having to decompress). Assume the key value A is stored in the hash table and probe key B is compared to A. Normally, one would just fetch the key A from the table and then compare it to B. In combination with compression, fetching A also requires decompressing A. We argue it is better to first bring B into the same representation as A, i.e., compressing B, and then directly compare the compressed values. This is especially true if keys A and B consist of multiple columns. For instance, a group-by on two columns can often be mapped into a single-integer compressed key, reducing the computational work of hash aggregation (e.g. perform a single comparison, using fewer branches).

4.2.5 Generating Pre-Compiled Kernels

To integrate the on-the-fly compression, decompression, and checking routines into Vectorwise, we needed to generate pre-compiled kernels for each type combination. This means that we would have to generate kernels for up to $n+1$ types (n inputs, one output). For $n = 8$ inputs, one output and 10 distinct types, we count 10^{8+1} kernels, leading to a heavily inflated binary file and—potentially—high lookup costs when resolving single kernels.

To reduce the number of generated kernels, we (a) restrict the number of inputs to ≤ 4. In addition, we (b) restrict the types we pack into to 32-, 64- and 128-bit unsigned integers and (c) impose an order on the inputs (ordered by bit-width). All three restrictions limit the number of kernels that need to be generated to 3,000 pack kernels and 340 kernels that do decompression such as unpack-fetch or key checks. These kernels are generated using templating.

4.2.6 Tackling the Packing Problem

Prefix Suppression combines multiple prefix-suppressed codes from different attributes. However, in the vectorized execution model, we rely on pre-compiled primitives that only allow a fixed number of inputs (here $n = 4$), to avoid a combinatorial explosion in the number of functions needed. We must also choose the output data types of which the hash table record is made up (32 or 64-bit integers and rarely 128-bit integers) and appropriately spread the compressed input columns in non-overlapping fashion over these base data types. In effect, we have to generate a *packing plan*, consisting of pack functions that (a) respects the maximum n on pack input columns, (b) minimizes the total hash table record width in row-wise

```
function GreedyPack(b_w, C)
    Q ← c ∈ C ORDER BY b(c) DESC
    n_b ← Σ_{c∈C} b(c)                          ▷ Length of compressed bit-string
    U ← n_b mod b_w                             ▷ Number of unused bits
    loop                                        ▷ Each round creates one output word w
        L ← b_w                                 ▷ Unused bits at end of word
        w ← new_word(b_w)
        Q' ← ∅
        while Q ≠ ∅ do                          ▷ Tries to fit columns into word
            c ← pop(Q)
            if L ≥ b(c) then                    ▷ c fits into word w
                append(w, c)
                L ← L − b(c)
            else
                push(Q', c)                     ▷ Delay
        Q ← Q'
        if Q = ∅ then return                    ▷ Done
        if L ≤ U then                           ▷ Free bit budget, leave bits free
            U ← U − L
        else                                    ▷ Slice first/biggest column
            c ← pop(Q)
            (c, c') ← slice(c, L)               ▷ First L bits go into c, remainder into c'
            append(w, c)                        ▷ Split c at L bits into c and c'
            push(Q, c')                         ▷ Process remainder later
            Q ← Q ORDER BY b(c) DESC
```

Figure 4-3: Greedy packing algorithm with word size b_w, set of columns C and $b(c)$ compressed size of column $c \in C$ in bits

(NSM) layout as well as (c) minimizes the number of slices an input column is cut into.

For joins, we separate the packing problem into two sub-problems, one for packing the hash table key-columns only, and the second for packing all other columns. For aggregates, we only pack the key-columns. The other columns are aggregate results, and they are left in their uncompressed layout. The reason is that packing and unpacking needed for every update to an aggregate result would slow down aggregations dramatically. In the next section, however, we describe a technique, called *Optimistic Splitting*, to shrink aggregates into smaller data types and, therefore, reduce the active working set, as well as CPU effort.

We pack the key-columns together, such that e.g. the TPC-H join on `PARTSUPP` will pack `PS_PARTKEY` and `PS_SUPPKEY` into one word, so we can execute the join as if there were just one column: this halves both hashing and comparison work. The algorithm is invoked twice: once packing into 32-bit words and once packing into 64-bit words. We use the 64-bit solution if this yields less hash table columns than the 32-bit solution, or otherwise, if the 64-bit solution produces a NSM record of the same size.

Greedy Packing Algorithm. The algorithm (Figure 4-3) packs a set of columns, and first orders them in a queue Q on their bit-width (the bits needed after Domain-Guided Prefix Suppression). The sum of these bit-widths generally is less than a full multiple of the output word bit-size; let the amount of unused bits be U. The main round of the algorithm iteratively pops the largest column off Q and checks if it fits the current output word. If not, it puts the column in the initially empty queue Q'. Otherwise, it maps this column onto the current output word, hence reducing the amount of still unused bits L in this output word (L is initialized to the output word bit-width at the start of each round). When no column fits anymore and $Q = \emptyset$, we reset $Q = Q'$ and $Q' = \emptyset$ to move to a next round (output word). If at the end $L \leq U$ (there is free bit budget), then we simply decrement U by L and leave these bits free. Otherwise, the first popped column in the next round will be sliced: putting its highest unprocessed L bits into the previous output word and starting the round with the rest of the column. The algorithm continues its rounds until all columns (or slices thereof) are mapped to bit ranges in the output words.

4.3 Optimistic Splitting

The goal of *Optimistic Splitting* is to exploit skewed access frequencies by separating the common case from exceptional situations. We physically split the hash table into two areas: The frequently accessed *hot area* and the *cold area*, which is accessed rarely. This approach does not necessarily save space. However, it shrinks the active working set, leading to lower memory access cost. Also, it converts operations on the final, widest, data type into operations on a potentially smaller data type. Specifically, if 128-bit operations become 64-bit or 32-bit; this can speed up computation noticeably. As we show in the following, Optimistic Splitting is especially important for data that is difficult to compress, such as aggregates and strings.

4.3.1 Optimistic Aggregates

Aggregates are difficult to compress with Domain-Guided Prefix Suppression as it is not possible to obtain tight bounds for aggregation results (for example `SUM`s). The reason is that one has to be pessimistic when deriving domain bounds to prevent integer overflows: Assuming a `SUM` of at most 2^{48} integers from, say an 18-bit domain, would overflow 64-bit and thus need a 128-bit aggregate. If this type is used for the aggregate, on each addition in the sum this large 128-bit integer will be read, updated, and written back.

Table 4.1: Optimistic Aggregates

Aggregate	Common case	Exception
`SUM`	Small integer	Overflow counter
`MIN`	Small upper bound	Minimum
`MAX`	Small lower bound	Maximum
`COUNT`	Similar to `SUM`	
`AVG`	Rewritten into $\frac{\texttt{SUM}}{\texttt{COUNT}}$	

Using a 64-bit integer for the aggregate, on the other hand, would (a) reduce reads and writes by a factor 2 and (b) provide faster updates. Without sacrificing correctness, Optimistic Splitting allows one to do just that in the common case (i.e., when no overflow occurs): The 128-bit aggregate result is split into a frequently accessed 64-bit sum and another, rarely accessed 64-bit overflow/carry field, which is stored separately. In pseudocode, this looks as follows:

```
void opsum(u64* common, u64* except, int group, i32 value) {
  common[group] += value; // 64-bit unsigned addition
  // Overflow handling
  bool overflow = common[group] < (u64)value;
  bool positive = value >= 0;
  if (!(overflow ^ positive)) { // Rare: handle overflows
    if (positive) except[group]++;
    else          except[group]--;
} }
```

Note that this is a generic implementation that handles positive as well as negative `values`. In combination with domain bounds (MinMax information) it is possible to prove the absence of negative or positive values, which leads to simplified logic and improved performance. Our later micro-benchmarks show that this is an optimization that often out-performs the full 128-bit `SUM`.

We can generalize Optimistic Splitting to other aggregates, as Table 4.1 illustrates.. We use the *associativity* of aggregates to provide a fast path for large aggregates and a smaller working set. `MIN` can be implemented using an upper bound (s) inside the hash table and storing the full minimum e as an exception ($s \geq e$). When calculating the aggregate, one would first check against s and discard values that cannot become the new minimum. For the remaining values, one has to check against the full minimum and potentially update the full minimum e as well as the upper bound s. Similar is the implementation of `MAX` whereas the other aggregate functions, `COUNT` and `AVG`, can be implemented similar to `SUM`. However, in case of `COUNT` one can more aggressively reduce the common case to a 16-bit integer and after $2^{16} - 1$ iterations update both, the small optimistic counter and the exception.

4.3.2 Other Applications

Optimistic Splitting is a very general idea that, we believe, can be applied in many different use cases. It only requires that the entries of a hash table have different access patterns, and can be decomposed in some form.

Besides the aggregate decomposition approach described above, a second use case in the hash aggregation are functionally dependent group by keys. These are not touched by lookups, and can therefore be placed in the cold area.

Optimistic Splitting is also applicable to certain hash joins. For selective joins (i.e., where most probes are misses), only the keys need to be stored in the hot area, whereas the payload columns, which often occupy much more space than the keys, can be moved to the cold area.

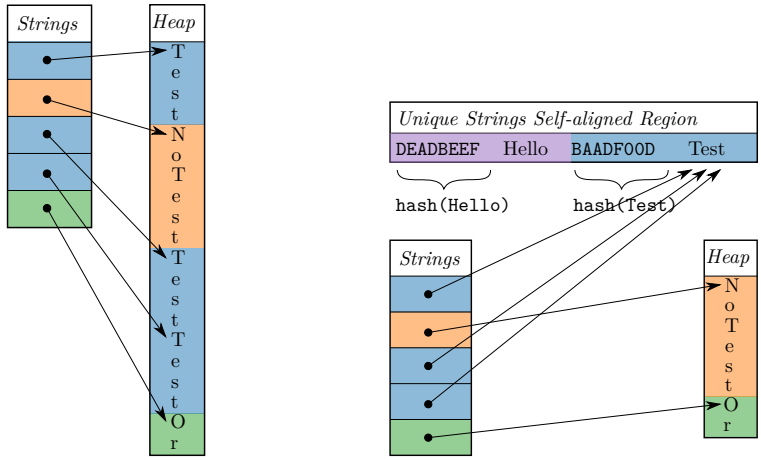
Even the fact that the `next` pointer in chaining-based hash tables is often `NULL` can be exploited. A few bits in the hot area can indicate whether a `next` bucket is absent or is nearby. The full next pointer can reside in the cold area.

Finally, a weakness of the usage of global MinMax information, is that outliers can destroy the tight MinMax bounds that would capture most of the value distribution. Alternatively, one could derive MinMax bounds from a table sample. The hot area in hash tables would hold the "Sample-Guided" Prefix Suppressed values extended with an exception bit. The full uncompressed columns would be present in the exception area, but only be accessed for the outlier values.

These examples show that Optimistic Splitting is widely applicable. We also think that some implementation techniques could be further developed. For instance, for aggregates with few groups (and certainly global aggregates, without grouping), vectorized systems could keep more aggressive overflow bounds that guarantee that a batch of aggregate updates cannot overflow the partial aggregate. This way, overflow checking could be done once per vector, rather than for every tuple. We defer investigation of these ideas to future work.

4.4 USSR: A Dynamic String Dictionary

Strings are prevalent in many real-world data sets [MRF14, JMH+16, VHF+18] and present additional challenges for query performance. In contrast to integers, any individual string generally does not fit into a single CPU register and requires multiple instructions for each primitive operation (e.g. comparison). Strings are



(a) Typical Heap-based strings (b) USSR encodes frequent values

Figure 4-4: String implementations

also often larger than integers, which negatively affects memory footprint and cache locality. Furthermore, neither Domain-Guided Prefix Suppression nor Optimistic Splitting can directly be applied to strings. This section presents a dynamic data-structure called *Unique Strings Self-aligned Region*, which saves memory and enables processing strings at almost the same speed as integers.

4.4.1 The Problems with Global Dictionaries

To improve the performance of strings, some main-memory database systems—most notably SAP HANA [FCP+12]—represent strings using per-column dictionaries where codes respect the value order. Using these dictionaries, string comparisons and hashing operations can be directly performed on the dictionary keys, which are fixed-size integers, rather than variable-length strings. Unfortunately, global dictionaries have significant downsides, which have precluded their general adoption. First, because random access to the dictionaries is common, the dictionaries must fully reside in main memory. For systems that must manage data sets larger than main memory (e.g. analytical column stores), this is a major problem. Furthermore, systems that support parallel and distributed execution, including those designed or optimized for the cloud, face the problem that bulk-loading or updating tables in parallel would require continuous synchronization to maintain a consistent global dictionary. Another downside is that dictionaries incur significant overhead for inserts, updates, and deletes—in effect they are a mandatory secondary index on every string column. If, for instance, new values appear, extending the dictionary such that one additional bit is needed to represent a code, updates will no longer

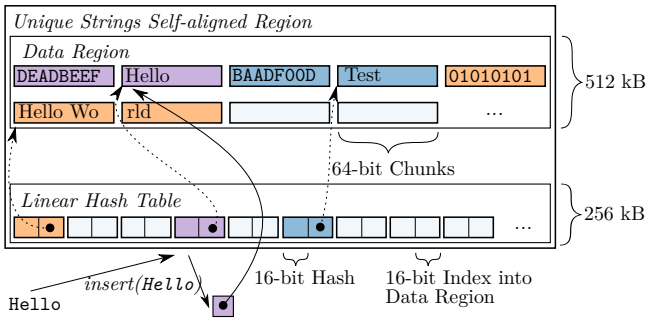


Figure 4-5: Unique Strings Self-aligned Region (USSR) data structure details

fit in previously encoded data. Deletes of no longer used strings leave holes in the code space that need to be garbage collected and inserts in sorted dictionaries often require re-coding—which involves fully rewriting all encoded columns periodically.

Given these problems with global dictionaries, most database systems therefore limit themselves to per-block dictionaries (e.g. one dictionary for every 10,000 strings). With this approach, dictionaries are a local feature mainly used for compression rather than a global data structure. Per-block dictionaries are often almost as space-effective as per-column dictionaries, without sharing their in-memory limitations and update overheads. For query processing, however, the advantage of per-block dictionaries is limited. While some systems evaluate pushed-down selections directly on the dictionary [LMF+16], all other operations require decompression and therefore do not benefit from the dictionary. The reason is that the dictionary is only available to the table scan operators. Materializing operators like hash join and aggregation therefore typically allocate memory on the heap for every string, as is illustrated in Figure 4-4a. Needless to say, this is very inefficient, yet for some reason dealing with strings is only a sparsely researched topic.

4.4.2 Unique Strings Self-aligned Region (USSR)

The Unique Strings Self-aligned Region is a query-wide data structure that contains the common strings of a particular query. In contrast to the heap, all strings within the USSR are known to be unique, which enables fast operations on these strings. To make it cache resident and efficient, the USSR has a limited size. Once it is full, strings need to be allocated on the heap as usual. Figure 4-4b shows an example with a mix of USSR-backed and heap-backed strings. By removing duplicates in this opportunistic fashion, the USSR reduces the number of heap allocations and therefore minimizes peak memory consumption.

By default, both heap-backed and USSR-backed strings are represented as normal pointers, which means that query engine operators can treat all strings uniformly without any code modifications. This allows to retro-fit this idea easily into already existing engines. However, by exploiting the dictionary-like nature and artful implementation of the USSR, the following additional optimizations become possible for USSR-based strings: (a) String comparisons are almost as fast as integer comparisons. (b) Hashes are pre-calculated and stored within the USSR, speeding up hash-based operators like join and group by. (c) Since the size of the USSR is limited, frequent strings can also be represented using small integer offsets, which can be exploited e.g. in Optimistic Splitting.

To summarize, the USSR is a lightweight, dynamic, and opportunistic string dictionary. It does not require changes to the storage level but is implemented in the query processor, and speeds up queries with low to medium string cardinalities, which is where global string dictionaries excel.

4.4.3 Data Structure Details

Our USSR implementation limits its capacity to 768 kB: it consists of a *hash table* (256 kB) and a *data region* (512 kB). Figure 4-5 serves as an illustration of the USSR.

The 512 kB data region starts at a *self-aligned* memory address (i.e., the pointer has 0s in its lowest 19 bits). If one allocates 1 MB of data, there is always a self-aligned address in its first half for the data region; and there is always either 256 kB space before or after the data region for the hash table. The self-aligned memory address guarantees that all pointers inside the data region start with the same 45 bits prefix. This allows to very efficiently test whether a string pointer points inside the USSR (by applying a mask).

The data region stores the string data and materializes the string's hash value just before it. These numbers are stored aligned, so the data region effectively consists of 64k slots of 8 bytes where a string can start. Given that each string takes at least two slots (one for the hash and one for the string) the USSR can contain maximally 32k strings.

When inserting a string, the USSR needs to check whether that string is already stored, and if so, return its address rather than insert a new string. To do this in low $O(1)$, there is a fast linear probing hash table, consisting of 64k 4-byte buckets. Each bucket consists of a 16-bit hash extract and a 16-bit *slot number* that points

into the data region to the start of the string. The lowest 16-bits of the string hash are used for locating the bucket, and the next 16-bits are the extract used to quickly identify collisions. The load factor is always below 50% (64k buckets for at most 32k strings).

4.4.4 Insertion

The purpose of the USSR is to accelerate operations on frequent strings. In the extreme, all strings could be part of the USSR. However, due to its limited size, the USSR can only fit a sample. The sampling happens during insertion into the data structure. Failure during insertion might happen because (a) the string is rejected based on our sampling strategy or (b) a probing sequence of longer than 3 in the linear hash table is detected (due to the low load factor, this is highly infrequent, yet keeps negative lookups fast).

Our sampling strategy gives priority to *string constants* that occur in the query text; these are inserted first. Subsequently, scans will insert strings until the USSR is full. We argue that the fact that a string column is dictionary-compressed, indicates that strings stem from a domain with a small cardinality. Therefore, these strings are good candidates for insertion into the USSR.

Vectorwise stores and buffers data in compressed form and decompresses column slices on the fly in the table scan operator. When reading a new dictionary-compressed block, the scan needs to set up an in-memory array with string pointers. Strings are represented as pointers in-flight in a query and decompression means looking up dictionary codes into this array. Rather than pointing into the dictionary inside the buffered block, when setting up this array, the scan inserts all dictionary strings into the USSR, so (most of) these pointers will point into the USSR instead. Insertion may fail, in which case the pointers still point into the block.

The sampling strategy further tries to optimize usage of the limited data region, by failing inserts of long strings that occupy $> \min(F, \max(2, \lfloor \frac{F}{64} \rfloor))$ 8-byte slots, where F is the free space in the data region (in slots). The idea is that it is better to accept more small strings than a few large strings, in case space fills up.

4.4.5 Accelerating Hashing & Comparisons

The USSR can be used to speed up hash computations. After testing whether a given string resides in the USSR using a bit-wise `and` operation, one can directly access the pre-computed hash value, which physically precedes the string:

```
inline u64 hash(char* s) {
  if (((u64)s & USSR_MASK) != ussr_prefix)
    return strhash(s); // compute hash
  return ((u64*) (s))[-1]; // exploit pre-computed hash
}
```

The USSR also speeds up string comparisons when both compared strings reside in it. We exploit the fact that all strings within the USSR are unique. Hence, if the pointers are equal, the strings themselves are:

```
inline bool equal(char* s, char* t) {
  if ((((u64)s & USSR_MASK) != ussr_prefix) |
        (((u64)t & USSR_MASK) != ussr_prefix))
    return strcmp(s, t)==0; // regular string comparison
  return s==t; // pointer equality is enough in USSR
}
```

4.4.6 Optimistic Splitting & the USSR

Optimistic Splitting and the USSR are complementary. The idea is to store USSR-backed strings, as small integers, compactly in the hot area and heap-backed strings in the cold area. Specifically, rather than storing string pointers in the hot area, we store slot numbers, pointing into the USSR. As mentioned earlier, these slot numbers are limited to 2^{16}, so they can be represented as unsigned 16-bit integers.

During packing, we represent exceptions using the invalid slot number 0 in the hot area of the hash table, and store the full 64-bit pointer in the exception area. Whenever a string needs to be unpacked, we first access the hot area and unpack the slot number. For non-zero slot numbers, we can directly reconstruct the pointer of the string (`base address of USSR data region + slot*8`). However, we can further accelerate equality comparisons on strings by first comparing the slot numbers and, only if they are 0, comparing the full strings. A USSR encoded string `p` can be translated into a slot number quickly using `(p » 3) & 65535`.

4.5 Evaluation

In this section, we provide an experimental evaluation of our contributions to show that our techniques improve performance as well as memory footprint.

Implementation. For this evaluation, we integrated Domain-Guided Prefix Suppression, Optimistic Splitting, and the USSR into Vectorwise. Besides generating all the necessary function kernels, we had to extend the domain derivation mechanism

and implement our greedy packing algorithm. In addition, we modified the existing hash table implementation, extended the hash join operator to take advantage of compressed key and payload columns, as well as the hash aggregation (group by) operator to support Optimistic Aggregates.

Content. We first evaluate the end-to-end performance on the TPC-H benchmark. We then present a high-level comparison on a real BI workload from Tableau Public [VHF+18]. Next, we selected queries from the workload, provide a detailed breakdown and explain the impact of our techniques. Afterward we move to micro-benchmarks, analyze and discuss the impact of the USSR on string-intensive queries. Then we evaluate the hash probe performance over varying hash table sizes and the influence of different domains on hash table performance. Afterward, we present and discuss the performance of our compression kernels, followed by an evaluation of Optimistic `SUM` aggregates.

Experimental Setup. All experiments were performed on a dual-socket Intel Xeon Gold 6126 with 12 physical cores and 19.25 MB L3 cache each. The system is equipped with 384 GB of main memory. All results stem from hot runs using single-threaded execution.

4.5.1 TPC-H Benchmark

We evaluated the impact of Domain-Guided Prefix Suppression, Optimistic Splitting and the USSR on the widely used TPC-H benchmark with scale factor 100. We executed all 22 queries on our modified Vectorwise with and without our optimizations. We measured hash table memory footprint, as well as query response time. First, we present and discuss the performance regarding memory footprint and, afterward, query performance.

Memory Footprint. In Vectorwise the memory consumption of many queries, particularly the TPC-H queries, is dominated by the size of hash tables. Therefore, during the TPC-H power run, we measured hash tables sizes. Figure 4-6 shows the compression ratios we measured.

Domain-Guided Prefix Suppression (*CHT alone*), without Optimistic Splitting and USSR, was able to reduce hash table size by up to 4×. However, due to certain hurdles, the compression ratio is often limited to 2×: (a) Aggregates are not compressible without Optimistic Splitting. (b) Without the USSR, each string has to be a 64-bit pointer into a string heap. On recent hardware, this requires storing at least 48 bits with Domain-Guided Prefix Suppression. (c) As CHT does not make

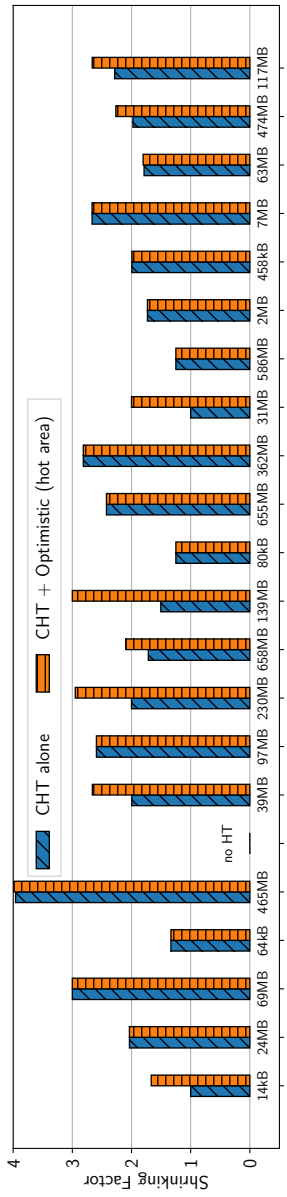


Figure 4-6: Reduction in hash table memory footprint over TPC-H with baseline hash table memory footprint (below the bar)

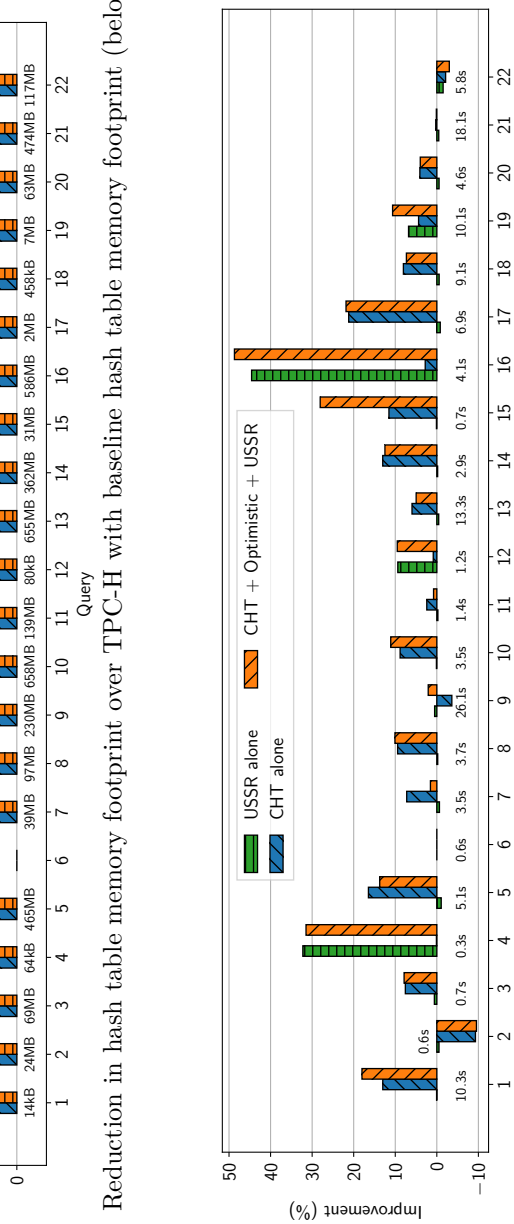


Figure 4-7: Improvement over TPC-H power run with baseline times (under the bar)

sense for CPU cache-resident hash tables, we do not enable it if the hash table is small, based on optimizer estimates. The impact of (a) and (b) on the active working set will be reduced using Optimistic Splitting and the USSR.

Optimistic Splitting aims at improving performance through more efficient cache utilization by separating the hash table into a thin frequently accessed table (hot area) and a rarely accessed table (cold area). In combination with the USSR we measured a 2–4× smaller hot area (*CHT + Optimistic (hot area)*) in many TPC-H queries.

However, Optimistic Splitting in fact *increases* (rather than reduces) the overall memory consumption as it introduces additional data. For example, splitting a 128-bit `SUM` aggregate will introduce an additional aggregate with a smaller size, but the full 128-bit aggregate will still reside in the cold area. Table 4.2 shows the relative memory footprint of vanilla Vectorwise against the combination of Domain-Guided Prefix Suppression, Optimistic Splitting and the USSR. Over TPC-H we measured up to 2.1× lower memory consumption. However, in comparison to Domain-Guided Prefix Suppression alone, Optimistic Splitting achieves an inferior compression ratio. The main idea behind Optimistic Splitting is to reduce memory *pressure* rather than overall memory *consumption*.

Query Performance. To demonstrate the performance benefits of the USSR, Domain-Guided Prefix Suppression and Optimistic Splitting, we visualize the query response times of all 22 TPC-H queries in Figure 4-7. We split our analysis into three stages. First, we evaluate the impact achieved by only using USSR. Then we discuss the effects of only using Domain-Guided Prefix Suppression. Finally, the influence of the combination of all three techniques will be discussed.

The idea of the USSR is to boost operations on frequent strings. However, TPC-H is not an extremely string-intensive benchmark. Nonetheless, by using the Unique Strings Self-aligned Region (*USSR alone*) three queries (Q4, Q12 and Q16) showed significant performance gains. All three benefit from faster string hashing and equality comparisons provided by the USSR and improve by up to 45%.

Apart from the string-specific USSR, Domain-Guided Prefix Suppression aims at shrinking hash tables and providing operations on compressed data. We found that Domain-Guided Prefix Suppression accelerates most queries (*CHT alone*) by up to 30%. In most queries we noticed an improvement of at least 10%. This is caused by the more efficient expression evaluation that smaller data types provide and the more cache-efficient hash table that allows equality comparisons directly on

Table 4.2: Reduction in hash table memory footprint on TPC-H comparing vanilla Vectorwise against optimistically compressed hash tables including hot and cold area

Query	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Factor	0.8	1.6	1.5	1.0	2.0	0.0	1.1	2.0	1.4	1.4	1.0	1.0	1.4	2.1	1.0	1.0	1.3	1.6	1.6	1.3	1.4	1.6

Table 4.3: Speedup and USSR statistics for workbook *CommonGovernment*

Query	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Speedup	**2.1**	**1.4**	**2.2**	**1.4**	**1.3**	**1.0**	**1.2**	**1.0**	**1.5**	**1.8**	**1.1**	**2.2**	**2.2**	**1.8**	**1.5**	**2.1**	**1.4**	**1.1**	**1.4**	**1.1**
USSR Size (kB)	1.8	0.5	2.0	0.3	66.1	512.0	83.2	512.0	12.7	7.2	112.4	1.9	1.8	7.2	1.8	2.0	1.8	110.3	0.3	512.0
Rejection Ratio (%)	0.0	0.0	0.0	0.0	0.0	18.3	0.0	32.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	21.1
#Rejected	0	0	0	0	0	37627	0	30204	0	0	0	0	0	0	0	0	0	17	0	13742
#Candidates	1312	2720	1056	1504	73200	205440	77296	92208	656	6768	110704	1072	1232	6752	3792	1360	3808	99744	1728	65222
#Strings in USSR	46	16	49	11	2251	12227	2835	12218	252	165	3041	48	45	167	51	49	51	2990	11	21343
Average String Length	23	3	20	5	18	26	19	29	21	25	23	22	22	25	18	22	19	23	5	11
Baseline Runtime (s)	0.15	0.27	0.13	0.17	0.37	3.48	0.39	0.54	0.18	0.16	0.29	0.14	0.14	0.16	0.18	0.14	0.17	0.27	0.18	0.51
Baseline HT size (MB)	0.05	0.09	0.05	0.09	0.14	82.11	0.14	9.12	0.07	0.05	0.11	0.05	0.05	0.05	0.05	0.05	0.05	0.11	0.09	8.11

compressed keys. Notably, the regression in Q2 was caused by type casting overhead due to opportunistic shrinking of data types. We highlight that the purpose of Domain-Guided Prefix Suppression is mostly to reduce the memory footprint and not necessarily to speedup query evaluation.

When combining all three techniques (Domain-Guided Prefix Suppression, USSR and Optimistic Splitting) we measured gains up to 40% (*CHT + Optimistic + USSR*). We measured additional improvements from 5%, in Q1, up to 10%, in Q15. Both queries benefited from the Optimistic `SUM` aggregate, which boosted the aggregate computation.

We ran all 22 queries with intra-query parallelism and noticed similar performance improvements. However, as these runs were considerably more noisy and would not contribute significant new information, we excluded them from this chapter.

4.5.2 Public BI Benchmark

It has been noted that synthetic benchmarks like TPC-H do not capture all relevant aspects of real workloads [CG12, BAK17]. Recently, a workload study was published [VHF+18] based on the Tableau Public[1] Business Intelligence (BI) free cloud service. It analyzes its workbooks (data and queries generated by the Tableau BI tool) and specifically notes that users make extensive use of string data types (i.e. strings are by far the most common data type; used for 49% of all values). Not only is text data prevalent in these workbooks, but it is also observed that date columns, numeric and decimal columns are often stored as strings; arguably suboptimally, but often this is related to data cleaning issues. Regrettably, this study did not publish the data and queries as an open benchmark, also upon our request to Tableau. Inspired by this work, we manually downloaded the 48 biggest Tableau Public workbooks (400 GB data) and extracted the SQL statements from its query log. This workload is now available in open-source as the **Public BI Benchmark** [2]. As a representative example, we focus on one of its workbooks: *CommonGovernment*.

CommonGovernment. We extracted all 43 queries and all 13 tables. Each table contains around 8 GiB of data in CSV format. Unlike TPC-H, each table contains many string columns and columns that contain `NULL` values are common. We executed each query sequentially, and Table 4.3 shows the measured effects on the runtime.

[1] `https://public.tableau.com`
[2] `https://github.com/cwida/public_bi_benchmark`

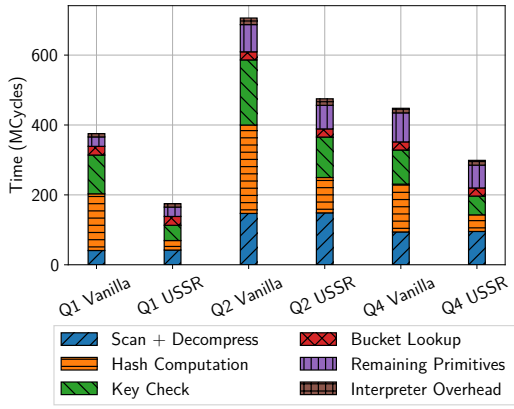


Figure 4-8: Query time breakdown for selected queries of workbook *CommonGovernment*

The workbook *CommonGovernment* is string-intensive: using only the USSR, we measured a speedup of up to $\approx 2\times$ (55% improvement). These speedups are caused by the fact that on the one hand, many strings originate from a small domain of unique strings and thus become resident in the USSR. On the other hand, many strings are long enough to significantly impact string operations to cause a speedup of the whole query.

Q6, Q8 and Q20 show no significant benefits from the USSR mainly because the string columns have a large unified dictionary (that does not even fit fully in the USSR). While dictionary-coded decompression in Vectorwise has a sub-cycle per/-tuple cost, the effort of setting up the dictionary array when the scan moves to a new disk block increases, when the per-block dictionary size increases. With the USSR, this setting-up effort becomes significantly higher, as all dictionary strings must be looked up in the USSR linear hash table. Moreover, with larger dictionaries per block, each dictionary string has a lower repetition count during execution; so the amortization of the setting-up investment by faster hashing and comparison decreases. Still, we see that we make a good trade-off, as queries Q8 and Q20 still get (marginally) faster, and only Q6 is marginally slower.

In general, the Public BI workload is characterized by few joins and many aggregations [VHF+18], where these aggregations produce small results—few or in the thousands, but almost never in the millions of tuples. This means that the hash tables needed for aggregation are often CPU cache-resident. Therefore, CHT is not triggered and that the USSR is what most matters in this workload, so we focus only on that.

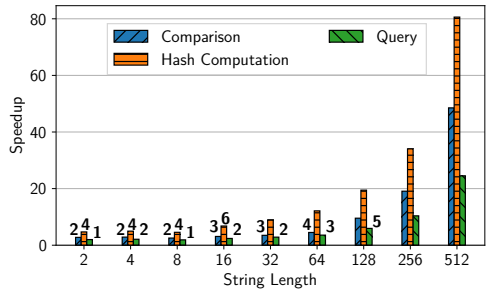


Figure 4-9: Group-By on string keys: Speedup vs. length

Breakdown. On the string-intensive workbook *CommonGovernment* the USSR caused many queries to improve up to ≈ 2×. Figure 4-8 shows a breakdown of the query time for Q1, Q2 and Q4. For Q1, we now discuss how close 2× comes to the optimal speedup: The USSR led to a roughly 7× faster hash computation and a 2× faster key check (check whether keys are equal). This led to a total speedup of 2×. Assuming zero cycles would be spent on hash computation and key checks, then one could achieve a total of 3×. This, however, is a theoretical optimum and is unlikely to be achieved when performing these operations.

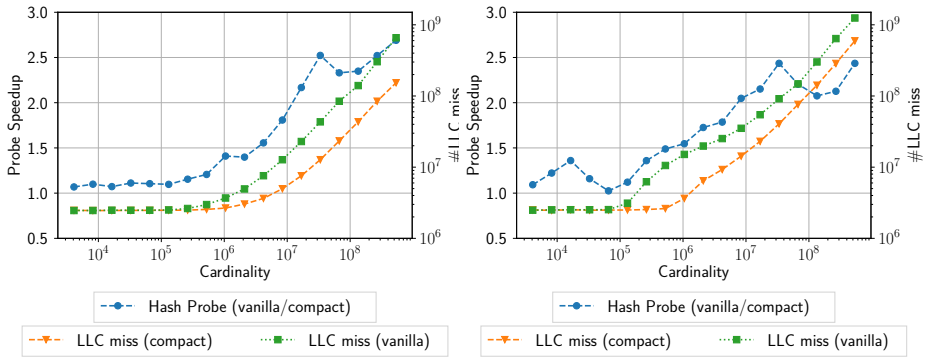
4.5.3 Micro-Bench: USSR and Group-By

We now move to a number of micro-benchmarks to focus on individual performance aspects of string processing with the USSR. We start with the performance on a `SELECT COUNT(*) FROM T GROUP BY s` query. These strings came from a domain of 10 unique strings, all strings had the same length. Figure 4-9 shows the speedups that can be achieved using the USSR. We profiled the time spent on string comparisons when checking the keys inside group by's hash table. These results show significant speedups reaching from a 2× to 50× faster string comparison. Similarly, we profiled the time spent on computing hash of the string keys. This results in speedups reaching from 4× for small strings, to 80× for large strings.

Besides the significant speedup in terms of string comparison and hash computation, we also noticed significant speedup of the whole query, up to ≈ 25×.

4.5.4 Micro-Bench: Join Probe Performance

We now micro-benchmark Domain-Guided Prefix Suppression, regarding hash table lookup performance. Our experiment consists of a simple join query where we vary the size of the inner relation and the domain of the key columns. We experimented with two and four key columns, four payload columns with values $v \in [0, 10]$.



(a) 4 keys $k_1, ..., k_4 \in [0, 1.000]$ whereas schema suggests 64-bit integers

(b) 2 keys $k_1, k_2 \in [0, 10^6]$ whereas schema suggests 128-bit integers

Figure 4-10: Hash probe speedup vs. build-side cardinality using 4 payload columns $p_1, ..., p_4 \in [0, 10]$

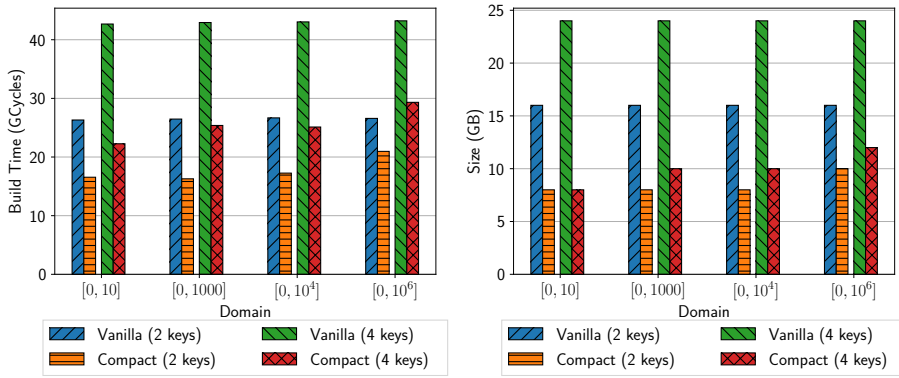
Figure 4-10 visualizes the speedup, as well as, the L3/last-level cache (LLC) misses measured. We observe an up to 2.5× faster hash probe, including the tuple reconstruction cost. The measured speedups tend to increase with hash table size (size of inner relation). For large hash tables with more than 10^6 rows, the speedups were caused by the significantly smaller and, consequently, more cache-resident hash table. For hash tables with less than 10^6 rows, performance was mostly affected by the more efficient comparisons directly on compressed data.

4.5.5 Micro-Bench: Hash Join Key Domain

The compression ratio and access performance of Domain-Guided Prefix Suppression depends on the domain of the input values. We now micro-benchmark the effects of input domain(s) on hash join build time and compression ratio. We executed a simple (hash) join query with a large fixed-size inner (build) relation over multiple domains, with multiple keys (2 and 4) and without any payload columns. We measured build time, as well as, hash table size.

First, we investigated the impact of Domain-Guided Prefix Suppression on the build phase. Figure 4-11a shows the results. We noticed a pronounced increase in performance, ranging from 25% up to 2×.

As mentioned, the performance of Domain-Guided Prefix Suppression depends on the compression ratio. Figure 4-11b shows the hash table size with and without our compression technique. For small domains, here $[0, 10]$ and $[0, 1000]$, we notice a compression ratio from 2× to 2.5×. With two keys, the compression ratio is limited



(a) Build time (b) Hash table size

Figure 4-11: Hash join performance vs. key domain

by the output type size of our transformation, where we have to round up to that type's size. Using more keys, we can fill the output word more densely and, hence, achieve a higher compression ratio up to 2.5×. For larger domains, $[0, 10^6]$, our scheme cannot achieve a high compression ratio, since there are simply not enough redundant 0-bits to suppress.

4.5.6 Micro-Bench: Memory Footprint against other Hash Tables

We benchmarked the memory footprint of our compressed hash tables against linear, Concise [BLP+14] and uncompressed bucket-chained hash tables. Table 4.4 shows the reduction in memory footprint.

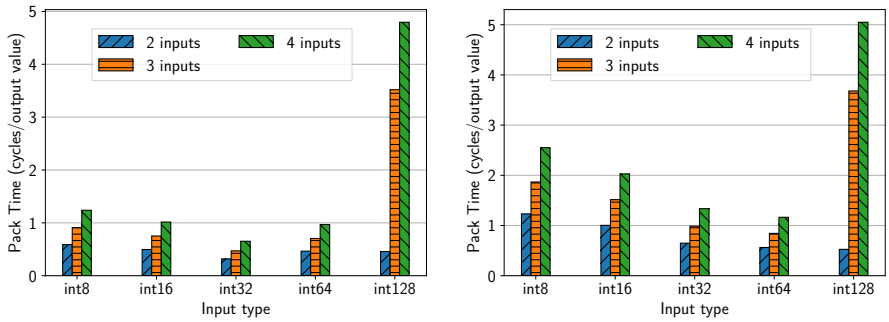
We observed that especially for large and wide hash tables, common in analytical queries, our compressed hash table easily out-performs the three other designs by $\approx 2\times$ to $7\times$. For extremely thin hash tables, our compressed bucket-chained hash table, shows additional overhead as `next` pointer, as well as `bucket` pointer needs to be stored. However, this is merely a limitation of our implementation. By applying Domain-Guided Prefix Suppression to the Concise Hash Table, one could easily construct a much more memory-efficient hash table.

4.5.7 Micro-Bench: Compression Overhead

Our next micro-benchmark showcases that Domain-Guided Prefix Suppression is very lightweight and compression overhead for common types is negligible. The compression performance is visualized in Figure 4-12. For native integer types,

Table 4.4: Reduction in memory footprint of our compressed hash table vs. other hash table designs. With n values $\in [0, 2^{16})$ and 50% fill rate. Higher is better.

|Inner|	#64-bit Values						
	1	2	4	8	16	24	32
Linear Hash Table							
1k	2.0×	3.2×	4.6×	5.8×	6.7×	7.1×	7.3×
1M	1.1×	2.0×	3.2×	4.6×	5.8×	6.4×	6.7×
1G	1.1×	2.0×	3.2×	4.6×	5.8×	6.4×	6.7×
Concise Hash Table [BLP+14]							
1k	1.1×	1.6×	2.3×	2.9×	3.4×	3.6×	3.7×
1M	0.6×	1.0×	1.6×	2.3×	2.9×	3.2×	3.4×
1G	0.6×	1.0×	1.6×	2.3×	2.9×	3.2×	3.4×
Bucket-chained Hash Table							
1k	1.8×	2.2×	2.7×	3.2×	3.5×	3.7×	3.7×
1M	1.4×	1.8×	2.2×	2.7×	3.2×	3.4×	3.5×
1G	1.4×	1.8×	2.2×	2.7×	3.2×	3.4×	3.5×



(a) Packing values into 32-bit integers (b) Packing values into 64-bit integers

Figure 4-12: Compression performance of bit-packing the first 8 bits of each input

namely 8, 16, 32 and 64-bit, our implementation can compress between 1 and 2 output values per cycle. In contrast, packing 128-bit integers is significantly slower. However, 128-bit integers rarely occur in real-world data sets and mainly stem from aggregates, for which we use Optimistic Splitting (such that packing/unpacking 128-bits values is not needed).

4.5.8 Micro-Bench: Optimistic Splitting

Finally, we micro-benchmark the effectiveness of Optimistic Splitting for `SUM` aggregates. We evaluated the performance of 128-bit optimistic `SUM` in the following experiment:

We sum up 64-bit integers into a 128-bit aggregate. We implemented a vectorized "full" aggregation that adds each 64-bit value into the 128-bit aggregate and an optimistic version with a 64-bit partial aggregate. Each vectorized aggregation

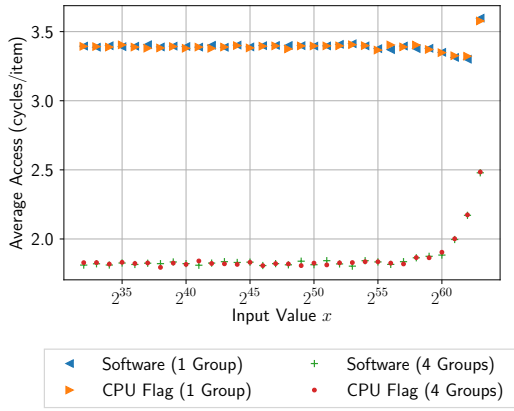
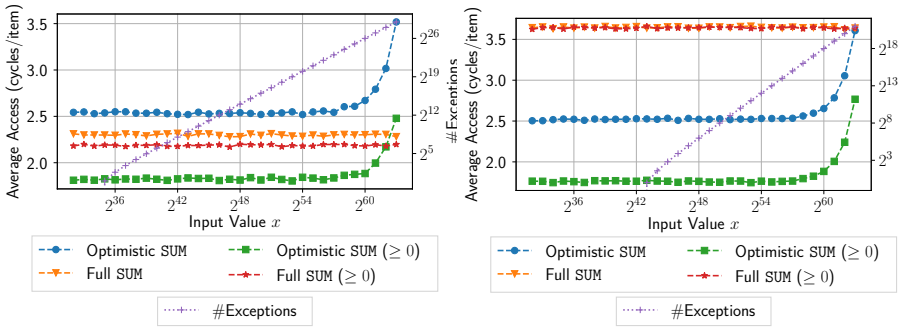


Figure 4-13: No noticeable difference between optimistic `SUM` overflow checking in software vs. CPU overflow flag



(a) 4 groups (b) 1024 groups

Figure 4-14: Aggregation methods for 128-bit `SUM` aggregate with 64-bit integers as input

primitive processes cache-resident vectors of input values, indices, aggregates, and data. We used 2^{32} input values each equal to a constant x. Each value is then summed up per group. To control overflows, we varied the constant x and the number of groups. All aggregates (full and partial) are stored in a table in row-wise layout (NSM).

First, we evaluate overflow checking in software (as in our earlier `spec_sum` pseudocode) vs. overflow checking in hardware (i.e., using the CPU overflow flag). Figure 4-13 shows the average computation time of each aggregate using different techniques. We want to highlight that, for 1 and 4 groups, we measured no significant difference between them. This verifies our design choice to implement the Optimistic `SUM` using software-based overflow logic, which is also portable.

Second, we compare the Optimistic `SUM` against the full 128-bit `SUM`. Figure 4-14 plots the average runtime of each aggregation method against the input values. We have separated two cases: (a) the full-blown optimistic `SUM` and (b) if one can prove, using MinMax information, the absence of negative values, one can optimize the `SUM` further. Regardless of the number of groups in the aggregation (4 or, 1024), it can be observed that (positive-only) Optimistic `SUM` significantly outperforms the full `SUM` for positive inputs and for values $\leq 2^{61}$. The Optimistic `SUM`'s performance heavily depends on the number of exceptions as for each exception it requires two reads, two writes, two additions, as well as two/three branches. For 4 groups, the Optimistic `SUM` is slower than full `SUM`, still the optimized version for only positive inputs is the fastest for values up to 2^{61}. For 1024 groups, we observed that the Optimistic `SUM` in both flavors, outperforms the full `SUM` by $2\times$. This is caused by more efficient memory access.

4.6 Conclusion

In this chapter, we explored efficient approaches for exploiting compressed data inside query processing pipelines:

Compressed Hash Tables. Hash tables are often used data structures for efficient query processing. Typically, hash tables are used to in join and group-by operations. However, accessing hash tables becomes more time-consuming the larger they are due to increasing cache miss penalties. Therefore, we explored the idea of using compression to ease the cache miss penalties. We explored three methods: Domain-Guided Prefix Suppression, Optimistic Splitting and the USSR.

Domain-Guided Prefix Suppression compresses keys and values based on their Min/Max information. Once, the Min/Max of each value is known, we know the number of bits required to store each value (works for integers and fixed-point numbers, both are often used in RDBMS). With this information, we generate a compressed data layout for each hash table row. In essence, this layout tries to pack the bits tightly (i.e. bit-packing multiple columns together but allowing gaps). During runtime, we efficiently compress and decompress using the generated layout. Notably, key checks can be realized, without decompression, by transforming the keys into the same representation as the compressed data. Consequently, this allows checking multiple key columns using one key check (assuming they can be compressed into one word). In certain cases, Domain-Guided Prefix Suppression is not very effective: (a) aggregates require decompressing each value, updating and compressing it again (repeated compression/decompression overhead) and (b)

strings as it is challenging to shrink 64-bit pointers (pointing to the string data). Point (a) is tackled by Optimistic Splitting, while point (b) can be solved through dictionary compression (e.g. using Unique Strings Self-aligned Region) and Optimistic Splitting.

Optimistic Splitting decomposes frequently accessed area (hot) from rarely accessed (cold) area. To achieve this, we assume that most values will be in a given range. These values will be stored in the hot area. For the remaining values outside that range, we store a marker in the hot area (marking it as an exception) and store the data in the cold area. Consequently, during runtime most accesses will access only the hot area, and only in exceptional cases, we have to access the cold area too. By tuning (shrinking) the range, we can further minimize the hot area and, therefore, make the working set more cache resident.

In practice, this works very well for aggregates (especially `SUM`s). While it can be applied to other types (keys or values in hash tables) as well, it requires extensive tuning to guarantee that most values access only the hot region.

Unique Strings Self-aligned Region (USSR) is an opportunistic on-the-fly dictionary for frequent strings. The USSR accelerates hash computation and equality comparisons on frequently occurring strings. Furthermore, in combination with Optimistic Splitting, the USSR allows representing frequent strings using short 16-bit integers (indexes into the USSR), instead of full 64-bit pointers. These small 16-bit integers are then inserted into the hot area of the compressed hash table.

Summary. In summary, we have observed that, even though the query plan stayed constant, there is a multitude of possible implementation options exploiting specific data properties to more efficiently process data: Domain-Guided Prefix Suppression depends on the range of values (like Compact Data Types from Chapter 3), Optimistic Splitting depends on the frequently used range of values, i.e. speculative extension of Compact Data Types, and the USSR depends on the number of distinct strings as well as the string lengths.

Lessons Learned from Manual Exploration. While the "tricks" from the previous and this chapter improved query performance, they:

(a) Required a tedious, repetitive and, in the case of this chapter, substantial engineering effort, and

(b) Relied on properties that are, unfortunately, not always known beforehand, i.e. before evaluating the physical query plan. Most notably, cardinalities (hash

table sizes, number of groups in group-by) are difficult to predict [LGM+15], especially in complex query plans. The same is true for more complicated statistics such as data distributions required to determine frequently used ranges.

(c) Very often the best implementation is influenced by the hardware (see Chapter 7). Additionally, other factors outside the DBMS can affect performance (concurrent processes, CPU temperature, ...). Generally, these factors are extremely challenging to predict using cost models, especially their effect on the performance of the *whole* query.

The following two chapters (Chapters 5 and 6) tackle (a) i.e. reduce the engineering effort to explore new points. Points (b) and (c) are addressed by Chapter 8 which proposes to try specific implementations while the query is running and, thus, finds the best implementation *micro-adaptively*.

CHAPTER 5

Encapsulating the Essence in VOILA

> Any problem in computer science can be solved with another level of indirection.
>
> David Wheeler

5.1 Introduction

From the previous chapters, we have motivating evidence that the design space offers many opportunities (e.g. by exploiting data properties or different operator implementations). However, these exploratory steps were extremely time-consuming, as they required writing and testing code. Moreover, it is not clear how these design space points translate to other query execution paradigms. Therefore, for large-scale exploration, this process is, obviously, not suitable.

In this and the following chapter, we move towards a systematic and more scalable approach of exploring the design space. In particular, here, we first abstract query execution using a domain-specific language (VOILA). In the first section, we discuss why existing domain-specific languages were not suitable and explain why VOILA

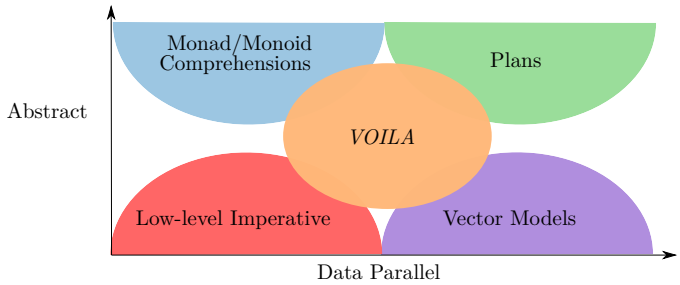


Figure 5-1: VOILA strikes the sweet spot between data parallelism and abstraction.

hits the sweet spot between multiple paradigms, as illustrated in Figure 5-1. Afterward, we formally specify the semantics of VOILA programs. In the third section, we show how often-used plan operators can be expressed in VOILA, followed by a discussion of how VOILA has been extended to integrate multi-thread parallelism.

5.2 VOILA

During query evaluation, database systems often apply the same algorithms and data structures, i.e. the same physical operators (hash join, hash group-by etc.). The major difference lies in their physical execution strategy (compiled/interpreted tuple/vector/column-at-a-time). We argue that operators should be described in a way such that we, later, can synthesize different execution strategies. Our domain-specific language VOILA (Variable Operator Implementation LAnguage) is tailored for this purpose. It describes the algorithmic details of an operator while abstracting the physical execution strategy away.

5.2.1 Core Concepts

High-level Languages are not well-suited. Many high-level languages can express algorithms relevant to database engines, for which one can generate many different implementations (flavors). Notable, and well-known, examples include MIL [Bon02], Voodoo [PMZM16], QMonad [SKP+16] and Weld [PTS+17]. However, these languages lack the ability to describe algorithmic details. Suppose we want to express a hash table lookup. Due to their level of abstraction MAL, QMonad and Voodoo are unable to represent a simple hash table lookup. Instead, they enforce the usage of higher level concepts, like a hash join. Consequently, when generating many flavors from this description, one would have to re-implement many different joins. Being slightly more low-level, Weld can represent dictionary lookups

via a primitive building block. This has two major disadvantages: (a) It is not easily possible to optimize specific fragments of the hash table lookup (e.g. fusing key checks for composite keys) without requiring a new lookup implementation. (b) When synthesizing different execution strategies, new black-box hash table lookups have to be re-implemented for each strategy.

Besides their inability to express algorithmic details, high-level languages often introduce complex nested data structures for intermediate values. Compared to simple intermediates (scalars, arrays), complex intermediates are typically slower to access (e.g. compare accessing a nested linked-list to a flat array) and consume more memory. To mitigate this problem, high-level languages have optimization passes that deal with removing complex intermediates. However, removing these intermediates is a very costly process (in NP-time), called deforestation [Wad88].

Both properties, the inability to express algorithm details and the problems caused by complex intermediates, render high-level languages unsuited for our purpose.

VOILA to the rescue. To mitigate these disadvantages, we propose to decompose complex operations (hash table lookup) into multiple simpler operations. In VOILA, we decompose a hash table lookup into: hash computation, bucket lookup, key check (gather & equality check), navigation to the next bucket and a loop to iterate over hash collisions. This decomposition describes *how* a hash table lookup is to be performed, but still omits specific details such as hash table design (bucket-chained, linear probing, ...), data layout (row-wise, columnar, hybrid) or execution strategies (data-centric, vectorized, interleaved prefetching, ...). From the example it is evident that VOILA is more high-level than low-level languages (e.g. C, LLVM IR [LA04] or ScalLite [SKP+16]), but more low-level than languages such as Voodoo and MIL/MAL. Thus, it bridges the gap between typical high- and low-level languages. After decomposing algorithms into high-level primitives, we can synthesize a specific implementation for each primitive and, thus, the whole algorithm.

Block-wise Execution. To abstract the physical execution from the logical description, operations in VOILA always operate on multiple values (vectors) at once. Scaling the vector size allows covering value-at-a-time (vector size 1) to column-at-a-time (vector size ∞) as well as the design points in between (e.g. SIMD with 8 values-at-a-time). Most operations on vectors can happen completely data-parallel. The rationale of first-class support of block-wise execution is that re-discovering data-parallelism from sequential code is very hard, e.g. neither GCC nor LLVM properly vectorize loops with branches.

Inferred Types. Expressions and variables in VOILA do not specify data types because they can be inferred automatically, from schema and program. This provides additional flexibility to add type-based optimizations (e.g. exploiting thinner data types for faster arithmetic see Chapter 3, or faster hash table access as explained in Chapter 4).

Predication. In our use-case, it is common to only process parts of the data, e.g. when tuples get logically removed (filtered out). In VOILA filters create predicates, which can be attached to operations, but can also be inferred. Note that predicates are required for synthesizing an efficient vectorized implementation á la Vectorwise [BZN05] from VOILA.

Operator Context. Commonly, when lowering higher-level into a lower-level language, e.g. physical plan into LLVM IR, the operator context disappears. After lowering, it will be very hard or impossible to determine which operations belong to which operators. To be able to, later, synthesize iterator-based operators from VOILA, we chose to keep the operator context.

5.2.2 Language

VOILA describes operators as imperative programs with high-level primitives. Each description consists of a list of statements (evaluation, assignment, `LOOP`, `EMIT`). Statements contain expressions. Expressions can either be literals (variables, constants) or functions on expressions (see Table 5.1). Statements, as well as expressions, logically operate on data vectors. After filtering, instead of forcing materialization of vectors, i.e. removing the deselected items, VOILA allows augmenting statements and expressions with predicates (`op | predicate`). Predicates can be thought of a (bit)mask that annotates that if the predicate yields `true` on a tuple, the operation can safely be applied. Though, that representation is conceptual – in the data-centric VOILA backend, for-instance, no bitmaps exist and the predicate will be translated into a branch. In VOILA, operators, as well as statements therein, are stateful, they e.g. maintain a hash table. The `EMIT` statement moves tuples of vectors, resulting from expressions, to the following operator. Logically, all expressions and statements operate on data vectors. Expressions either, in case of functions, apply the function element-wise on the data vector(s) – all vectors are required to have the same length – or, in case of a constant, broadcast a constant to all elements. The result of expressions can be stored in variables. Assignments behave similar to other imperative languages and, logically, copy all values of the data vector into the destination variable. This allows updating the same variable.

Table 5.1: Expressions & Statements in VOILA. `x`, `y` denote values or expressions. `c` denotes a table column. `ht` denotes a hash table. `b` denotes a hash bucket, also an expression.

Operation	Description
Comparison/arithmetic/logic	
`eq(x, y)`	`x == y`
`not(x)`	`!x`
`add(x, y)`	`x + y`
`cast_i32(x)`	Casts `x` to (signed) 32-bit integer
`hash(x)`	Computes hash of `x`
...	
Hash Table	
`bucket_insert(ht, b)`	Create new bucket(s) in hash table
`bucket_lookup(ht, b)`	Given hash values, find initial buckets
`bucket_next(ht, b)`	Given bucket(s), find next bucket(s) in chain
Table/Array	
`scatter(c, b, x)`	Scatter values into bucket(s)
`gather(c, b)`	Gather values from buckets
`read_pos(t)`	Allocate next consecutive read position from table `t`
`write_pos(t)`	Allocate next consecutive write position from table `t`
`write(c, p, x)`	Consecutive write data to column starting from position `p`
`read(c, p)`	Consecutive Read from column starting from position `p`
Table Aggregates	
`aggr_count(c, b)`	Count active values (via predicate) in table's column `c` at index `b`
`aggr_sum(c, b, x)`	Sum values in table's column `c` at index `b`
...	
Data Inflow	
`scan_pos(t)`	Allocate next consecutive scan position
`scan(c, p)`	Returns column chunk from position `p`
Predicate	
`seltrue(x)`	Selected if `x` is true
`selfalse(x)`	`seltrue(not(x))`
`selvalid(x)`	Selected if `x` is valid. `[read, write, scan]_pos` can return an invalid position
`selunion(x, y)`	Selected if `x` is true or if `y` is true (`x` and `y` are both predicates)

Listing 5.1: Hash group-by in VOILA.

```
LOCAL HASHTABLE ht(k1 KEY, sum1 VALUE)

GroupBy(T) {
  h = hash(T[0])
  |miss = seltrue(true)

  LOOP |miss { // repeat until every tuple is processed
    bucket = bucket_lookup(ht, h)
    empty = eq(bucket, 0)
    |hit = selfalse(empty)
    |miss = seltrue(empty)

    LOOP |hit { // hash probing
      htkey = gather(ht.k1, bucket)
      equal = eq(htkey, T[0])
      |found = seltrue(equal)

      // compute aggregates
      aggr_sum(ht.sum1, bucket |found, T[1])

      // continue with non-matching tuples
      |hit = selfalse(equal)
      bucket = bucket_next(ht, bucket |hit)
      empty = eq(bucket, 0)
      |miss = selunion(|miss, seltrue(empty))
      |hit = selfalse(empty)
    }

    // optimistically insert non-matching tuples
    new_pos = bucket_insert(ht, h |miss)
    // copy key T[0] into column 'k1'
    |can_scatter = selvalid(new_pos)
    scatter(ht.k1, new_pos |can_scatter, T[0])
} }
```

Using different predicates, one can overwrite different positions of the same variable. Besides assignments, VOILA also allows fixed-point iteration via loops, similar to C's `while` statement. Different is that in VOILA the loop condition is a predicate and is only true, as long as at ≥ 1 items in the vector qualify.

Example. We explain VOILA using the hash group-by in Listing 5.1 as an example. First we declare the required data structures (line 1), then we describe the operator: Commonly an operator receives an input (`T`) which is a tuple of vectors. In this case, we use the `T` to find the final hash bucket, and to directly compute aggregates (commented out). We first extract the key (`T[0]`) and hash the value. Afterward, we initialize the predicate `miss` to select all tuples. As long as there are `miss`es, we repeat the following process (7):

We look up the first bucket and check whether it is 0, i.e. empty. For buckets that are $\neq 0$, we repeatedly follow the bucket chain (13), check the keys (13 & 14) and compute the aggregate(s) using these positions (19). Afterward, we continue with the buckets that did not match any keys (21) and follow the bucket chain (23). If we notice the end of the bucket chain, we have new `miss`es (values that have to be

inserted) and append them to the existing misses (25). Then, we try to insert the `misses` (30) and copy the keys (33) which might fail. Finally, we repeat the insertion process until we found a bucket for every tuple.

5.3 Formal Semantics of VOILA

We formally define VOILA's semantics bottom-up: We start with basic expressions, afterward step-wise broaden the semantics to statements, operators, and query.

5.3.1 Expressions

In VOILA, expressions can have predicates attached. Predicates indicate which values inside vectors are valid. In case, there is no predicate attached, we attach a predicate that will return `true` for every value. We intend to only define the result of an expression when the predicate is `true` (1). Otherwise, we define it as undefined ($\perp$).

Predicates. To conveniently apply predicates to functions, we define ϕ as the application of a function f to its arguments a_1, a_2, ... in the presence of a predicate $p \in \{0, 1\}$:

$$\phi(p, f, a_1, a_2, \dots) := \begin{cases} f(a_1, a_2, \dots) & \text{if } p = 1 \\ \perp & \text{otherwise} \end{cases}$$

Element-wise Application. To apply regular functions onto vectors, we define element-wise application (π). Let vectors be functions from an index set I to the result set R ($I \to R$). For a vector $\vec{v}$, we can define $I = \{1, 2, \dots, dim(\vec{v})\}$. Let $I_{\vec{a}}$ denote the index set of vector $\vec{a}$. We define the trivial element-wise application (π') as:

$$\pi'(i, f, a_1, a_2, \dots) := \begin{cases} f(a_1(i), a_2(i), \dots) & \text{if } c(i, a_1, a_2, \dots) \\ \perp & \text{otherwise} \end{cases}$$

with
$$c(i, a_1, a_2 \dots) := \big((a_1(i) \neq \perp) \wedge (a_2(i) \neq \perp) \wedge \dots\big) \\ \wedge \big((i \in I_{a_1}) \wedge (i \in I_{a_2}) \wedge \dots\big) .$$

The function c defines which elements of the vectors are valid. An element is valid if is defined in terms of (a) $\perp$ and (b) the index set. The resulting index set of π' is the intersection of all input index sets: $I_{\pi'} = I_{a_1} \cap I_{a_2} \dots$.

We define the element-wise application (π) as the inverse transformation of vectors to functions applied on π'.

Expressions apply a function f to its arguments and a global state W. We define the result of an expression η as the composition of predication (ϕ) and application (π):

$$\eta(W, f, \vec{p}, \vec{a_1}, \vec{a_2} \dots) := \pi(\phi(f, W), \vec{p}, \vec{a_1}, \vec{a_2}, \dots)$$

Using η, we define the expressions in VOILA as the element-wise application of a function f. Expressions that allow sequential access, `read` and `scan`, have both similar semantics to `gather` with the difference that the indices are sequential starting from a scalar offset (o). We transform expressions with sequential access into the same shape as normal expressions in VOILA described by a function f. Therefore, we add the identity vector $\vec{id}$ as an additional argument to π and, consequently, $f(a_1, a_2 \dots)$. We denote such modified functions as $f_{seq}(a_1, a_2, \dots, i)$ where a_1, a_2 are the arguments and i the i-th position of the identity vector.

Table 5.2 defines most expressions as such function f. To keep the notation concise, we implicitly broadcast scalar expressions (`ht`, `c`, `p`) to a constant vector with infinite dimension. The remaining expressions (`read_pos`, `scan_pos` and `write_pos`, `bucket_insert`) can have side effects. Thus, we postpone their definition to the next section by rewriting them into statements: Let e be such an expression and x some unique identifier, then we rewrite $e(\vec{a}_1, \vec{a}_2, \dots)$ into `assign`$(x, e(\vec{a}_1, \vec{a}_2, \dots))$ and reference x in the expression(s) referring to the result of e.

5.3.2 Statements

Side-Effects. Similar to low-level imperative languages, e.g. C, statements can have side effects . To formally encapsulate side effects, we define a "world state" W. Each statement $S :: (W, A) \to W$ has input arguments A and modifies W. A chain of two statements S_1, S_2 would hand though the world state W: $W_{final} = S_2(S_1(W_{init}, A_1), A_2)$. By induction, we can construct chains of arbitrary length.

Global State. Our constructed world, or global state, W contains mappings for variables ($W.V_{var\ name}$), as well as mappings for data structures ($W.D_{struct\ name}$). Further, we define getters and setters: A getter returns the specified value, i.e. $get(w, m) := w.m$. A setter "modifies" the global state W by creating a new state W'. $W' := set(w, m, v)$ creates a copy of w, namely W', with $w.m = v$.

Variables. Statements allow assigning values to variables. Therefore, we extend W with an associative set that maps variable names to values ($W.V$). Assignments can update variables in $W.V$ with values stemming from the evaluation of an expression, which can read variables.

Table 5.2: Expressions in VOILA defined as function f.

Operation	Semantics
Comparison/arithmetic/logic	
`eq(x, y)`	$f(W, x, y) = x = y$
`eq(x, y)`	$f(W, x) = \neg(x)$
`add(x, y)`	$f(W, x, y) = x + y$
`cast_i32(x)`	$f(W, x) = x$
`hash(x)`	$f(W, x) = hash(x)$
...	
Hash Table	
`bucket_lookup(ht, b)`	$f(W, ht, b) = get(W, D_{ht;bucket}[b])$
`bucket_next(ht, b)`	$f(W, ht, b) = get(W, D_{ht;next}[b])$
Table/Array	
`gather(c, b)`	$f(W, c, b) = get(W, D_c[b])$.
`read(c, p)`	$f_{seq}(W, c, p, i) = get(W, D_c[p + i])$.
Data Inflow	
`scan_pos(t)`	`read_pos(t)`
`scan(c, p)`	`read(c, p)`
Predicate	
`seltrue(x)`	$f(W, x) = x$
`selvalid(x)`	$f(W, x) \neq \bot$
`selunion(x, y)`	`or(x, y)`
...	
Other	
Variable v	$f(W, v) = get(W, V_v)$
Constant c	$f(W, c) = bcast(c)$

Element-wise Application. Similar to Expressions, we define the element-wise application for statements. The difference is that statements and (rewritten) expressions have side effects and, thus, we need to (a) carry the global state around and (b) specify an evaluation order. Therefore, we define the element-wise application until an index m as:

$$A'_m(W, s, i, \vec{p}, \vec{a_1}, \vec{a_2}, \dots) := \begin{cases} A'_m(W', s, i+1, & \text{if } i \leq m \\ \quad \vec{p}, \vec{a_1}, \vec{a_2}, \dots) & \\ W' & \text{otherwise} \end{cases}$$

$$\text{with } W' = \begin{cases} s(W, i, \vec{a_{1,1}}, \vec{a_{2,1}}, \dots) & \text{if } (p = 1) \wedge (i \leq m) \\ W & \text{otherwise} \end{cases}.$$

Using A'_m, we define the element-wise application A as:

$$A(W, s, p, a_1, a_2, \dots) := A'_m(W, s, 1, \vec{p}, \vec{a_1}, \vec{a_2}, \dots)$$

Table 5.3: Statements in VOILA. i is the index inside vectors, defined in Section 5.3.2 (Element-wise Application).

Pattern	Semantics
$\texttt{assign}(W, v, \vec{e})$	$s(W, i, v, \vec{e}) = set\big(W, V_v, \theta(W, \vec{e})_i\big)$
Writers	
$\texttt{write}(c, e_{pos}, e_{\vec{val}})$	$\texttt{scatter}(c, bcast(e_{pos}), e_{\vec{val}})$
$\texttt{scatter}(c, e_{\vec{idx}}, e_{\vec{val}})$	$s(W, i, c, e_{\vec{idx}}, e_{\vec{val}}) = set\big(W,$ $D_c\big[\theta(W, e_{\vec{idx}})[i]\big], \theta(W, e_{\vec{val}})[i]\big)$
Aggregates	
Let f be an aggregation function, e.g. $f_{\text{sum}}(e_{old}, e_{val}) := e_{old} + e_{val}$	
$\texttt{aggr\_}*(c, e_{\vec{idx}}, e_{\vec{val}})$	$s(W, i, c, e_{\vec{idx}}, e_{\vec{val}}) = W'$ with $k = \theta(W, e_{\vec{idx}})[i]$ $W' = set(W, D_c[k],$ $f\big(get(W, D_c[k]), \theta(W, e_{\vec{val}})\big)$
(Rewritten) Position Allocators	
For $*\texttt{\_pos} \in \{\texttt{read\_pos}, \texttt{scan\_pos}, \texttt{write\_pos}\}$	
$\texttt{assign}\big(W, v,$ $*\texttt{\_pos}(\texttt{t})\big)$	$W' = set(T, V_v, \theta(W, r))$ with $r = get(W, D_{t;*\_pos})$ $T = set\Big(W, D_{t;*\_pos}, \theta\big(W, \texttt{add}(r, dim(r))\big)\Big)$

with $m = min\big\{dim(\vec{p}), dim(\vec{a_1}), dim(\vec{a_2}), \dots\big\}$, $\vec{p} = \theta(W, p)$, $\vec{a_1} = \theta(W, a_1)$, $\vec{a_2} = \theta(W, a_2)$ ... Using A, we can define statements as the application of a function s onto each element. With the element-wise application and the following helpers, most statements can be defined as in Table 5.3:

- $\theta(W, e)$ evaluates an expression e and returns its value.
- $C[i]$ accesses a value inside C at position i. This makes C a mapping from i to values.
- $\vec{e} = bcast(d)$ broadcasts d to all entries of $\vec{e}$ ($\forall i \vec{e_i} = d$).

`bucket_insert` tries to create $k = dim(b)$ buckets. Collisions might happen (conflicting indices inside a vector). Therefore, the result can be (a) a successful insertion of the bucket or (b) a failure to insert. In the latter case, the insertion procedure will have to be repeated. We combined both and return a bucket index, or $\perp$ for failure. We can define $\texttt{assign}(W, v, \texttt{bucket\_insert}(ht, b))$ as:

$$s(W, i, ht, b) = \begin{cases} set(W_3, V_v, r) & \text{if } conflict(\vec{b}, i) = 0 \\ set(W, V_v, \perp) & \text{otherwise} \end{cases}$$

with:

- $W_1 := set\big(W, D_{ht;next}[b], get(W, D_{ht;bucket}[b])\big)$,

- $(W_2, r) := allocate(W_1)$ allocates a position in the hash table and returns new state W_2 and position r.
- $W_3 := set(W_2, D_{ht;bucket}[b], r)$

$$conflict(\vec{b}, i) := \begin{cases} \left| \{\vec{b}[i]\} \cap \{\vec{b}[1], \ldots, \vec{b}[i-1]\} \right| & \text{if } i > 1 \\ 0 & \text{otherwise} \end{cases}$$

`LOOP` repeats a certain scope until a fixed-point is reached, i.e. condition is not satisfied. Let W be the global state and S be a statement, then we define `LOOP` as:

$$L(W, S, P) = \begin{cases} L(S(W), S, P) & \text{if } count(\theta(W, P), 1) > 0 \\ W & \text{otherwise} \end{cases}$$

5.3.3 Operators

In VOILA an operator is described by a function `op(input)`. Let the current pipeline be P, $|P|$ the number of operators in P and P_i the specific function of operator i.

`EMIT`. The `EMIT` statement transports tuples from the current operator to the next operator, in the pipeline, or outputs tuples, last operator in the last pipeline, to the user. With knowledge of the current pipeline P, the following operator is known and static (will not change). This allows us to rewrite the VOILA program into a program without `EMIT` To handle `EMIT`, we fully transform the VOILA program.

For every operator o (with operator function $P_o(W, x)$, with x being its input, or for scans), we replace every occurrence of `EMIT(x)` (typically only one), with $P_{o+1}(x)$ when $o{+}1 < |P|$ (otherwise it would print tuples to the user). We repeat this process until no `EMIT` can be replaced. As a consequence, there will only be *one* remaining operator function (every operator has been inlined into the scan operator P_1) and the remaining `EMIT`s add tuples to the result R. We define the result R as a list, part of the global state W, and let $append(L, x)$ a function that appends a value x at the end of the list L. Like regular statements, we define (the remaining) `EMIT` in terms of W as:

$$W' = \begin{cases} set(W, R, append(get(W, R), x)) & \text{if } p = 1 \\ W & \text{otherwise} \end{cases}$$

Execution & Termination. With the above toolkit, we can answer a query Q. Let the query consist of multiple pipelines $Q_1, Q_2, \ldots$. We construct an initial

W_0 with empty result R, empty set of variables V and set of data structures D consisting of base tables. Using the initial state W_0, we can evaluate each pipeline consecutively, resulting in a new state (a W') and feed W' into the next pipeline, and so on. We evaluate a pipeline using $eval(W, Q_n) := eval(W, Q_{n;1})$ which evaluates the first operator function ($Q_{n;1}$, the P_1 of a Q_n) of the pipeline until the end of the operator function is reached. We repeat this process until all pipelines have been evaluated. Afterward, the final result of the query is stored as $W.R$, in the final state W.

5.4 Common Relational Operators in VOILA

We translate physical plan operators (such as hash join, hash group-by) into VOILA. In this section, we show how commonly used operators can be implemented. In particular, we focus on scan, hash group-by, hash join and filter.

5.4.1 Scan

The scan operator allocates ranges (morsels) to be processed (`scan_morsel`) dynamically. In each such range it iterates over the tuple in the range (`scan_pos`), scans (`scan`) them and emits the constructed tuple:

```
Scan() {
  morsel = scan_morsel(BASE_TABLE1)
  |valid_morsel = selvalid(morsel)

  LOOP |valid_morsel {
    // iterate over morsel
    pos = scan_pos(morsel)
    |valid_pos = selvalid(pos)

    LOOP |valid_pos {
      k1 = scan(BASE_TABLE.k1, pos)
      k2 = scan(BASE_TABLE.k2, pos)
      EMIT (k1, k2)

      // goto next position in morsel
      pos = scan_pos(morsel)
      |valid_pos = selvalid(pos)
    }

    // go to next morsel
    morsel = scan_morsel(BASE_TABLE1)
    |valid_morsel = selvalid(morsel)
} }
```

5.4.2 Hash Group-By

The group-by consists of two phases: First it builds the hash table with the individual groups, potentially computes aggregates and, afterward, the hash table will be scanned. We logically separate these phases into two LOLEPOPs: *GroupBy* and *GroupByScan*.

GroupBy computes the index of unique each group in the hash table. It can be implemented as illustrated in Listing 5.1.

The unique index, from *GroupByScan* (stored in `T[0]`), is then used to calculate the aggregates for each group.

```
LOCAL HASHTABLE ht(key KEY, sum_a VALUE, cnt_a VALUE)

GroupByScan() {
  morsel = read_morsel(ht)
  |valid_morsel = selvalid(morsel)

  LOOP |valid_morsel {
    // consume morsel
    pos = read_pos(ht)
    |valid_pos = selvalid(morsel)

    LOOP |valid_pos {
      // read data from value space of HT
      c = read(ht.cnt_a, pos)

      // prevent output of empty groups
      exists_pred = gt(c, 0)
      |exists = seltrue(exists_pred)

      s = read(ht.sum_a, pos |exists)
      avg1 = div(s, c)

      // get key(s)
      key = read(ht.key, pos |exists)

      // produce output
      EMIT (key, avg1) |exists

      // get next read position
      pos = read_pos(morsel)
      |valid_pos = selvalid(pos)
    }

    // next morsel
    morsel = read_morsel(ht)
    |valid_morsel = selvalid(morsel)
} }
```

5.4.3 Hash Join

The hash join implementation consists of three phases: (1) Materializing the inner (build) relation (*HashJoinMaterialize*), (2) building the bucket chains (*HashJoinBuild*) and (3) probing the outer relation and reconstructing the output tuples (*HashJoinProbeFk1*). Note that *HashJoinBuild* has a hard-coded implementation.

During the hash join's build phase, we first materialize the inner relation:

```
SHARED HASHTABLE ht(k1 KEY, k2 KEY, hash HASH,
  c1 VALUE, ...)

HashJoinMaterialize(T) {
  // allocate space to materialize |T[0]| tuples
  pos = write_pos(ht, T[0])
  h = hash(T[0])

  // write columns to thread-local space
  write(ht.k1, pos, T[0])
  write(ht.k2, pos, T[1])
  write(ht.c1, pos, T[2])
  write(ht.hash, pos, h)
  // ...
}
```

Once the hash table is built, we have a different pipeline that allows to probe tuples through it. The following code shows an optimized LOLEPOP for probing in case of a foreign-key join (≤ 1 matches):

```
SHARED HASHTABLE ht(key KEY, payl VALUE)

HashJoinProbeFk1(T) {
  // find initial buckets
  h = hash(T.key)
  bucket = bucket_lookup(ht, h)
  empty = eq(bucket, 0)
  |active = selfalse(empty)

  // iterate through bucket chain
  LOOP |active {
    lookup = gather(ht.key, bucket|active)
    match = eq(lookup, T.key)
    |hit = seltrue(match)

    payl = gather(ht.payl, bucket | hit)
    EMIT (T ++ payl) |hit

    |active = selfalse(match) // omit for non-fk1 join
    bucket = bucket_next(ht, bucket|active)
    empty = eq(bucket, 0)
    |active = selfalse(empty)
} }
```

Note that *HashJoinProbeFk1* can also be generalized by omitting the line:

```
|active = selfalse(match)
```

5.4.4 Filter

Filters, or selections, are trivial to implement in VOILA:

```
Filter(T) {
  |p = seltrue(predicate)
  EMIT T |p
}
```

5.5 Multi-core Parallelism in VOILA

To take advantage of modern multi/many-core hardware, VOILA must be able to efficiently support intra-query parallelism. Therefore, we extended VOILA, with only a few changes, to support such parallelism.

5.5.1 Morsel-driven Parallelism

Morsel-driven parallelism [LBKN14] parallelizes each pipeline using a task-based model together with operators aware of parallelism. First, the query is split into pipelines, query fragments consisting of a chain of operators from a data source to a data sink. Afterward, each pipeline is parallelized.

In the data source, the input is dynamically partitioned into chunks, so-called morsels. Afterward, the pipeline is executed on that morsel. Eventually, a morsel will reach the end of the pipeline (typically a materializing operator). The behavior depends on the operator, in the following we explain hash group-by and hash join.

Hash Group-By. The parallel hash group-by works in three stages: First, each thread will pre-aggregate locally using a cache-resident hash table. Afterward, the hash table is (hash) partitioned. In the last step, each thread will scan its corresponding partitions to group them again thread-locally. Eventually, each thread holds local hash tables partitioned by the key. Pre-aggregation typically eliminates heavy hitters and reduces the amount of data that, later, needs to be shuffled/sent across cores, whereas, partitioned group-by allows the handling of many groups efficiently.

Hash Join. Leis et al. [LBKN14] describe the parallel hash join as a three-stage operator: First, each thread materializes the inner/build relation in a thread-local

space. Second, every thread will scan the materialized relation, starting with its local space and build the global shared hash table. In the third and last step, the shared hash table is probed by concurrently running threads.

Pros and Cons. It is resilient to skew, as well as, elastic by partitioning data sources (tables, hash tables) into small chunks and scheduling them in a balanced fashion. Through combining task-driven scheduling and operators optimized for parallelism, it achieves state-of-the-art scalability [LBKN14].

However, integrating morsel-driven parallelism into an existing system requires significant change. We show that integrating it into VOILA is relatively easy and straight-forward.

5.5.2 Integration

To integrate morsel-driven parallelism into VOILA, we (a) extended the data structures, as well as, (b) allowed morsel-based access for `scan`, `read` and `write`.

Data Structures. To support the variety of tables and hash tables required by morsel-driven parallelism we extended our table data structure by (a) a fully thread-local (hash) tables (for final group-by), (b) thread-local hash tables that can be flushed into partitions (for pre-aggregation) and (c) shared (hash) tables (for hash join)

Morsel-based Access. During scans or table reads, each thread will call the operations `scan_pos` and `read_pos`, which will return the position of the next tuple. We extended that interface by introducing two new operations `scan_morsel` and `read_morsel` which retrieve the next morsel. The, afterward, following operations `scan/read_pos` will then only return positions to tuples inside the given morsel.

5.6 Conclusion

VOILA provides an extensible representation where each operation operates on data vectors. Due to its "medium" level (neither fully low-, nor fully high-level either) nature, VOILA avoids issues other languages run into, most notably:

- The removal of complex intermediates (i.e. Deforestation [Wad88]). VOILA avoids such complex intermediates.
- The representation of data-parallelism (e.g. required from generation of SIMD code). By using vectors to represent data, in VOILA, the whole program is

data-parallel by design. While most operations in VOILA are data-parallel (e.g. `bucket_lookup`, `add`), not all operations have to be (e.g. `seltrue`). In VOILA, the presence of sequential (non-data-parallel) operations does not preclude other operations to run in a data-parallel fashion.

Besides its features on the language-level, VOILA is powerful enough to encode common operators used for query evaluation (Section 5.5.4, most notably join and group-by). Additionally, VOILA can easily be extended to support efficient parallelization (Section 5.5.5).

The following chapter will reveal the flexibility of VOILA by synthesizing different high-performance query execution paradigms from a single description (in VOILA).

CHAPTER 6

Synthesizing Engines from VOILA

> Insanity is doing the same thing, over and over again, but expecting different results
>
> Narcotics Anonymous

6.1 Introduction

The previous chapter discussed VOILA, a domain-specific language that describes commonly used query operators. This chapter discusses synthesizing state-of-the-art implementations from VOILA, as illustrated in Figure 6-1. We intend to generate a plethora of such implementations to gather more knowledge about the design space.

We discuss two kinds of translation methods (a) direct back-ends which directly translate VOILA into data-centric and vectorized code and (b) FUJI which decomposes code generation into different components that can be arbitrarily re-composed, and translated into an intermediate language (CLite) first. Furthermore, FUJI can mix different implementations within a pipeline.

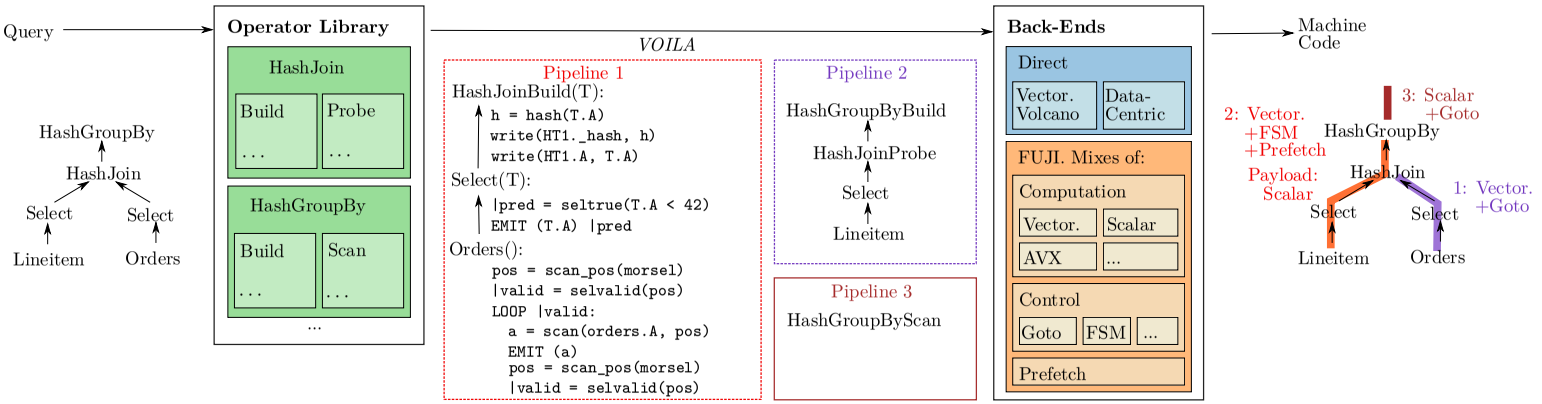


Figure 6-1: Architecture of our VOILA-based synthesis framework

Translate(Operator o, Input T):

1. Loops: `LOOP |p ...` → `while (p) { ... }`
2. Remove predicates:
   - `r = x |p` → `if (p) { r = x }`
   - `p = seltrue(x)` → `p = x`
   - `p = selfalse(x)` → `p = !x`
3. Implement operations:
   - `r = hash(x)` → `r = HASH(x)`
   - `r = read(col, idx)` → `r = col[idx]`
   - `scatter(col, idx, val)` → `col[idx] = val`
   - `EMIT x` → Translate($o + 1$, `x`)
   - ...

Figure 6-2: Translation of statements/expression from VOILA to data-centric program. Order of application matters.

6.2 Direct Synthesizer Back-ends

Using descriptions in VOILA, we can synthesize code that uses different execution styles. We created backends that generate the state-of-the-art paradigms in C++: data-centric as used in HyPer [Neu11], and iterator-based vectorized [BZN05] code, as used in Vectorwise.

6.2.1 Data-Centric Program

We first re-cap data-centric compilation and afterward describe how to synthesize data-centric code.

Data-Centric Compilation, first, splits query plans into pipelines. Pipelines start from scans (base table, group-by etc.) and end in a materializing operator. For each pipeline, data-centric compilation fuses all operators into one loop while, typically, generating scalar code for the operator's body. Thus, we focus on synthesizing scalar code, but our synthesis strategy is not limited to it.

Synthesis. Similar to the original approach by Neumann [Neu11], we first split the query plan into pipelines and inline all operators in one pipeline into one loop. This gives us data-centric pipelines in VOILA, which we then lower to executable code. During the translation, we assume a vector size of 1 (i.e. scalar) and directly expand operations in VOILA using the set of rules listed in Figure 6-2.

6.2.2 Iterator-based Vectorized Program

Alternatively, one can also translate VOILA into an iterator-based and vectorized program. Currently, our framework can only synthesize unary operators and, thus, we split binary operators into pipelines as for data-centric code.

Iterator-based Operators. It is a traditional approach to implement physical operators as iterators by providing an `open-next-close` interface. Using this interface, operators pull the next tuple from the child operator(s) by calling `next`. This both reduces the size of intermediates materialized in memory (one tuple) as cache-efficiency (tuple is produced and immediately consumed by the next operator).

Vectorized Interpretation. Traditionally, the iterator-based model only returns one tuple at a time. It has been shown that this leads to high interpretation overhead [BZN05]. As a mitigation, the iterator-model can be extended to return multiple tuples. To further cut down interpretation costs, basic expressions also need to operate on multiple tuples. This is known as "vectorized interpretation" [BZN05].

Synthesis. To generate an iterator-based vectorized program, we synthesize an operator implementation that implements the `open-next-close` interface. Inside the operator we need to construct expressions (`open`), evaluate them (`next`), deallocate resources (`close`), as well as, maintain the operator's state. Since we keep the operator context for each operator, we macro-expand a basic operator template.

For each expression in VOILA, we need to generate code that constructs an expression (`Expr a_plus_b("add", a, b);`). At a later point, the top-most expression will trigger the recursive evaluation of its input expressions (lazy evaluation). For statements, we generate specialized code:

- `EMIT` is translated into code that moves tuples (expressions) into the operator's output and returns tuples.
- `LOOP` translates into a loop, including evaluating the loop predicate. It can happen that loops refer to in-flight, not yet evaluated, expressions either from out-side the loop, or from a previous iteration. In such cases, we need to evaluate these in-flight expressions.
- All remaining statements enforce evaluation of their input expressions.

For each operator, we build expression trees which are evaluated, either, via statements, or when the result of the operator is requested (`next`).

6.3 FUJI – A flexible back-end

Our two direct backends can generate executable code for queries, using operators described in VOILA according to two entirely different state-of-the-art flavors: data-centric and vectorized. However, it is not limited to these two. Therefore, we designed a third back-end: Flexible Unified JIT Infrastructure (FUJI), which is

Table 6.1: Known points in the FUJI's design space

Flavor	Computation	Control	Prefetch	Buffering
Hyper [Neu11]	Scalar	Goto	✗	✗
X100 [BZN05]	Vec. Primitive	Goto	✗	✗
AMAC [KFG15]	Scalar	Conc. FSMs	✓	✗
ROF [MMP17]	Scalar	Goto	✓	✓
IMV [FZW19]	SIMD	Conc. FSMs	✓	✓

more *generic*. It makes code generation (1) more flexible and (2) easier to extend and debug. It (a) decomposes code generation into components, (b) allows freely mixing them, and (c) is logically splitting code generation into different modules serving different purposes.

6.3.1 Component-based Flavor-Generation

We decompose code generation into basic components: Computation, Prefetching, Control and Buffering.

Computation. The computation component translates expression trees into scalar operations (e.g. Hyper), SIMD operations or calls to vectorized primitives (functions that process column chunks in a tight loop).

Prefetching. Recent work has shown that software prefetching can significantly improve performance [KFG15, MMP17, FZW19] and therefore should be a part of modern query engines.

Control. The Control component decides how `EMIT` and `LOOP` statements are translated. This can be a goto-based program or multiple finite state machines (FSMs). Multiple FSMs have the advantage that each FSM can run concurrently and allow overlapping prefetching with other operations, e.g. one FSM issues the memory loads via prefetch instructions, while waiting the other FSMs can proceed.

Buffering. Selective operators (e.g. filters) or predicates can remove tuples from the flow. However, to achieve full utilization of SIMD lanes (or ALU in general) it is, in some cases, advisable to physically eliminate filtered-out tuples (i.e. typically materializing a chunk of the relation). Typically, buffering becomes pricier with more columns, but leads to gains at subsequent operators/operations.

This decomposition covers the state-of-the-art (Table 6.1) and the space in between.

6.3.2 Flexible Unified JIT Infrastructure (FUJI)

Typically, code generators tend to be giant monoliths. For example, the early code generator of Hyper was about 10K lines of code [Neu11]. To increase flexibility and extensibility, we split the logic of our FUJI back-end into multiple modules: Generic Codegen, Target Codegen and CLite.

Generic Codegen. The generic code generator generates CLite code for operators, statements, buffering logic (buffer refill and flushing), variables, position allocators. The remaining expressions are handled by the specific target code generators which, then, synthesize optimized code.

Target Codegen. The target-specific code generator generates specific implementations for expressions, buffering of intermediates and prefetching. We implemented 3 targets: `scalar`, `vector` and `avx512`. The `scalar` target generates data-centric code [Neu11] similar to Section 6.2.1. The `vector` target generates calls to vectorized primitives, which is an alternative implementation of vectorized execution, compared to iterator-based vectorized program (Section 6.2.2). In addition FUJI, currently, provides an `avx512` target, which processes blocks of 8 values-at-a-time and, if possible, uses AVX intrisincs. Note, not all operators are (a) possible using only the AVX instruction sets (e.g. `bucket_insert`), nor (b) benefit from it (e.g. aggregations or sub-word gathers). Selective processing is implemented using AVX-512 bit-masks. If the usage of AVX is not possible, for each of the 8 values, we check the mask and generate scalar code.

CLite. Both, generic and target code generators, generate CLite-code, our second domain-specific language. CLite is a simplified version of C without infix operators, macros, loops (go-to instead). It constitutes a lightweight abstraction that (a) provides a convenient interface to construct programs, (b) allows specific optimizations and (c) helps to synthesize different control-flow techniques. From the program in CLite, we currently generate a C++ program, but plan to compile to LLVM IR or machine code directly.

Synthesizing Control Flow. When synthesizing code, we differentiate between two control-flow strategies: (a) simple goto-based programs and (b) finite state machines (FSMs). Both can easily be synthesized from CLite which can be seen in Figure 6-3: For a goto-based program, CLite blocks are lowered into labels (`LABEL: BODY`) and branches into go-to statements (`goto NEXT;`). To generate an FSM, CLite blocks are lowered into an FSM state (`case STATE_ID: BODY`) and branches schedule the next states (`state.state = NEXT; break;`). The generated code is, then,

```
Fragment f;
Block l1(f), l2(f);
Builder b(l1);

Var x = f.var("int", "x");
Var y = f.var("int", "y");
Expr c = f.literal(42);
b.assign(y,
  b.func("+", b.ref(x), c));
b.effect(
  b.func("print", b.ref(y)));
b.branch(l2);
```

(a) Simple CLite example

```
int x,y;



{
  l1:
    y = x + 42;
    print(y);
    goto l2;
  l2:
    ...
}
```

(b) Goto Program

```
int state=1,x,y;

while (1) {

  switch (state) {
  case 1:
    y = x + 42;
    print(y);
    state = 2; break;
  case 2:
    ...
} }
```

(c) Finite State Machine

```
struct {int state=1,x,y;} S[N];
unsigned lwt = 0;
while (1) {
  auto& s = S[(lwt++) % N];
  switch (s.state) {
  case 1:
    s.y = s.x + 42;
    print(s.y);
    s.state = 2; break;
  case 2:
    ...
} }
```

(d) N Concurrent FSMs

Figure 6-3: Synthesizing Control-Flow from CLite.

wrapped into a loop and switch statement. To generate concurrent FSMs, we extend the FSM-code by wrapping local variables into a per-FSM state (`S`), and adding scheduling logic (`lwt`).

Minimizing State `S`. Especially, the performance of concurrent FSMs is very sensitive to the number of variables stored in the per-FSM state `S`, as additional overheads occur when variables are accessed: (a) indirection overhead because, instead of in CPU registers, variables are stored inside an array of `struct` and (b) additional cache pressure with an increasing size of `S`. Therefore, we added an optimization pass, that promotes CLite variables to regular variables (can be stored in regular CPU registers). This is possible, whenever variables are only read/written inside the same block, and, obviously, for constants. This optimization pass minimizes indirections as well as eases cache pressure.

6.3.3 Mixing Flavors (`BLEND`)

To finally generate an astronomical number of engines, the FUJI back-end should allow combining different flavors. Therefore, we extended VOILA with operations that allow changing the generated flavor, and extended FUJI with the ability to generate transitions between flavors.

One Flavor per Pipeline. One of the easiest ways of mixing flavors in a query is to choose a different flavor per pipeline. In FUJI, this is trivial because it just means instantiating a different code generator for each pipeline.

Mixing Flavors in a Pipeline. A more flexible method is to combine multiple flavors inside a pipeline, what we call *blending*.

We extended VOILA with `BLEND`, a statement which defines a flavor for a scope (subprogram with its statements and in-flight variables). Then, we can create different blends by setting a default/main flavor and introducing `BLEND`s which define flavors for program fragments. Note that this allows recursive stacking of `BLEND`s.

Translating `BLEND`. A `BLEND` defines a child flavor within a parent flavor. When translating a `BLEND`, we compose a new code generator (as described in Section 6.3.1) and buffer in-flight data during the transition from parent to child flavor, and back. Buffering can be done in many ways. One can imagine buffering columnar, row-wise buffers or mixes. To allow different buffering implementations, we construct an extremely versatile, yet simple, interface: (1) `buffer_read_pos`/`buffer_write_pos` allocate slots for reading/writing, (2) `buffer_read`/`buffer_write` read/write data from/to the buffer, and (3) `buffer_read_commit`/`buffer_write_commit` commit used slots. For example, to write a row to the buffer, we first allocate a destination slot using `p = buffer_write_pos`. Then, we write each attribute/cell `a` to the slot `p` via `buffer_write(p, a)`. Afterward, we complete the write via `buffer_write_commit(p)`. As this interface allows many possible buffer implementations, FUJI leaves specific implementation choices to the code generators.

Using the buffering interface, we implement `BLEND`. A `BLEND` introduces two buffers: an input and an output buffer. Data flows from a source into the input buffer. When the input buffer is full, we read values from the input buffer, and run the code inside `BLEND` (generated in a different flavor). This is producing output values which are then written into the output buffer. When the output buffer is full, we read values from the output buffer, which then flow towards the sink. We use source and sink rather generically: In the trivial case, one `BLEND` inside a pipeline, the source refers to the VOILA code before the `BLEND` whereas the sink is the code after the `BLEND`. However, when nesting or chaining `BLEND`s, source/sink can as well read/write another `BLEND`'s buffer.

Buffering Design Choices. To minimize allocation overheads, we use fixed-sized ring buffers. Typically, when wrapping around, vectorized writes can become non-contiguous. In that case, we leave empty space at the end, wrap around and write

Table 6.2: Highly diverse runtimes. SF 100. 24 threads.

#BLENDs	#Queries	Runtime (s)				
		Min	$Q_{0.25}$	Median	$Q_{0.75}$	Max
5	1	7.22	7.22	7.22	7.22	7.22
6	11	3.87	6.54	8.03	9.86	17.85
7	85	5.18	7.20	8.16	10.14	311.77
8	449	4.81	8.30	10.06	13.32	353.42
9	1511	4.70	9.05	10.98	15.51	318.32
10	9216	3.90	9.75	12.19	17.21	347.92
Total	11273	3.87	9.63	11.92	16.85	353.42

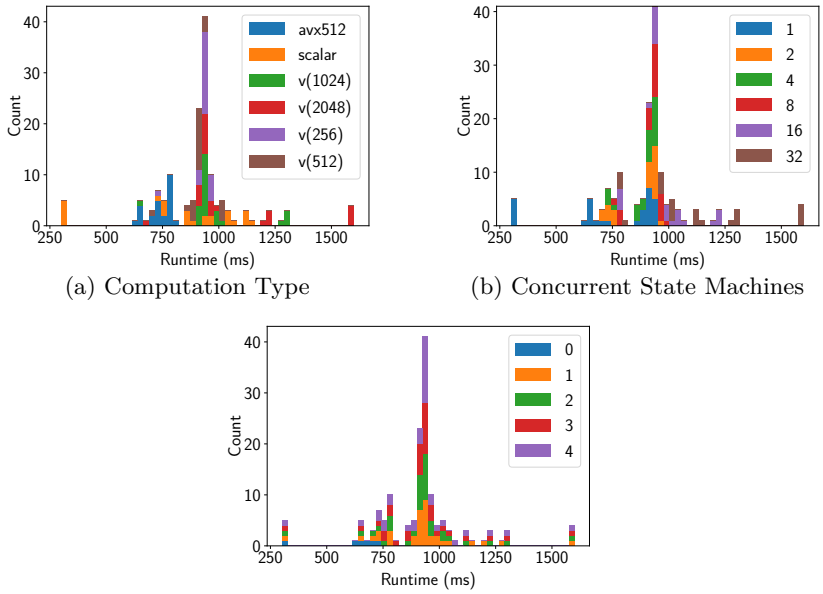
contiguously. The buffer size impacts performance significantly. A buffer that is too small will be flushed too often, incurring branch miss-predictions. If the buffer is too large, additional cache misses can have a negative impact. We differentiate between the physical buffer size and a high watermark, the effective size. The buffer size ensures the writes fit, whereas the high watermark controls buffer performance. We use high watermark of `max(2*n, 2k)` and a size of `max(2*s*n, 64k)` with `n` being the input vector size (e.g. 8 for AVX-512) and `s` the number of concurrent FSMs.

6.4 Evaluation

We implemented the VOILA compiler with the two direct back-ends and FUJI in C++. All queries use the TPC-H dataset with varying scale factors (SF). The experiments were performed on a dual-socket Intel Xeon Gold 6126 with 24 SMT cores (12 physical cores) and 19.25 MB L3 cache each. The system is equipped with 384 GB of main memory.

6.4.1 Design Space Exploration

We explored the design space of TPC-H Q9 span through mixing different flavors per pipeline and `BLEND`ing different flavors inside the same pipeline. Instead of allowing fully flexible `BLEND` operations, we limited them to specific points: (a) hash join key check, (b) hash join payload gather, (c) projections/arithmetic and (d) filters. Further, we limited base flavors to the pipelines that contribute $> 15\%$ to total performance. We further restricted the space by limiting essential parameters: computation type to the basic types (`scalar`, `avx512` and `vector`), prefetching to a boolean (`0`, `1`) and the number of concurrent state machines to reasonable small values (`1`, `2`, `4`, `8`). Since Q9 the design space is too large for full exploration, we sampled the design space. We synthesized roughly 10,000 queries from VOILA using uniformly random combinations of base flavors (data-centric, prefetching,



(a) Computation Type

(b) Concurrent State Machines

(c) Prefetching

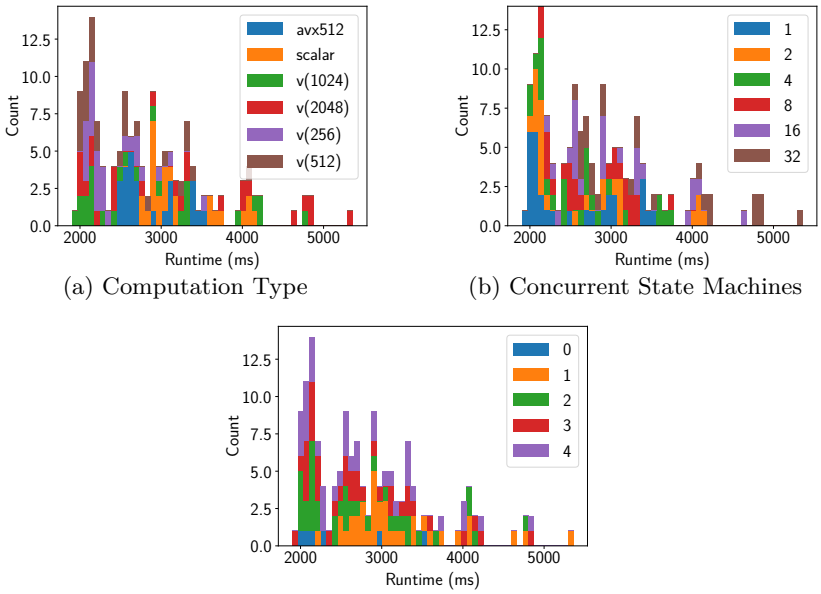
Figure 6-4: Q1: Breakdown of *the same* histogram into computation type, prefetching and concurrent state machines. Many flavors are far from optimal. No benefit from prefetching. 24 threads, SF 100

state machines etc.) as well as mixes of them, inside the same pipeline and between pipelines. This covers roughly $4 * 10^{-4}\%$ of the described space. The runtimes are summarized in Table 6.2.

Our uniform space sample frequently contains many `BLEND`s. Compared to the best runtime found (3.87), many queries perform worse ($\geq 2\times$ higher median). There is a tail of runtimes $> 4\times$ slower than the best time and extreme outliers that are $\approx 100\times$ slower. With an increasing amount of `BLEND`s (mixes) the space increases exponentially. Due to uniform sampling, we obtained more samples. Further, there is a tendency that more `BLEND`s lead to higher runtime (increasing median, increasing 25- and 75-percentile $Q_{0.75}$). This can be explained by the increasing buffering effort per `BLEND`. Indicating that further methods to reduce buffering overhead are required. Besides that tendency, there are still positive outliers, i.e. using 10 `BLEND` operations the minimum runtime is competitive to the best runtime using 5 mixes.

6.4.2 Impact of Components on Runtimes

Given a specific query, we investigate the impact of specific flavors and their components onto the total runtime. Therefore, we generated roughly 150 base flavors



(a) Computation Type

(b) Concurrent State Machines

(c) Prefetching

Figure 6-5: Q9: Breakdown of *the same* histogram into computation type, prefetching and concurrent state machines. Runtimes vary. `scalar` flavors tend to lead to worse performance. 24 threads, SF 100

(combinations of paradigms, prefetching and concurrent state machines) and ran the compiled query. The option we call prefetching encodes different prefetching localities as follows: `0` means no prefetching, `1` locality 0 (prefetch into all cache-levels up to L1), `2` locality 1 (prefetch up to L2), `3` locality 2 (prefetch into L3), `4` non-temporal (short-term/evict soon) [pre20]. The precise meaning of the locality hints depends on the hardware. In particular, we analyze the impact of FUJI's components on two queries: TPC-H Q1 and Q9.

Q1. The results for Q1 are visualized in Figure 6-4. On first sight the plot reveals that most flavors are suboptimal, but outliers, positive as well as negative, exist. The best flavors are roughly 3× better than average. These are based on `scalar` processing and do not use concurrent state machines. There is no clear benefit of prefetching as Q1 fits into cache, but incurs no significant overhead either. Vectorized (`vector` flavors tend to perform worse than `scalar` and `avx512` due to (a) materialization overhead (reading/writing vectors) and (b) in-efficient access to row-wise data when updating the aggregates. Further overhead is introduced by adding concurrent state machines, eventually leading to the worst flavors up to 50% worse than average. The performance of `avx512` is in between `scalar` and `vector`.

Table 6.3: Competitive performance. Runtimes of flavors generated from VOILA are comparable with recent handwritten implementations. Runtimes in s.

Flavor	Q1	Q3	Q6	Q9
SF 10				
Typer [KLK+18]	0.5	1.1	0.2	3.1
Direct Hyper	0.6 (0.9×)	1.2 (0.9×)	0.3 (0.9×)	3.1 (1.0×)
FUJI Scalar	0.5 (1.1×)	1.2 (0.9×)	0.2 (1.3×)	3.1 (1.0×)
Tectorwise [KLK+18]	1.0	0.7	0.2	1.6
Direct Vector	1.0 (1.0×)	0.8 (0.8×)	0.2 (1.1×)	2.0 (0.8×)
FUJI Vector	1.0 (1.0×)	0.7 (0.9×)	0.3 (0.7×)	1.8 (0.9×)
SF 100				
Typer [KLK+18]	5.5	13.3	1.8	40.9
Direct Hyper	5.9 (0.9×)	12.2 (1.1×)	2.8 (0.6×)	31.9 (1.3×)
FUJI Scalar	6.0 (0.9×)	11.5 (1.2×)	1.8 (1.0×)	32.7 (1.2×)
Tectorwise [KLK+18]	9.2	8.0	1.7	21.2
Direct Vector	10.6 (0.9×)	7.6 (1.1×)	2.3 (0.7×)	17.7 (1.2×)
FUJI Vector	10.4 (0.9×)	6.7 (1.2×)	3.6 (0.5×)	16.0 (1.3×)

Q9 paints a different picture, as Figure 6-5 shows. In general, block-based flavors (`avx512`, `vector`) tend to outperform `scalar`. The best flavors are `vector`ized ones, with ≤ 8 concurrent state machines and prefetching. Unlike Q1, we observed benefit from using concurrent state machines and prefetching. Similar to Q1, `avx512` tends to be in the middle between `vector` and `scalar`.

Summary. From both queries, it can be said that `scalar` flavors outperform in simple aggregation queries. `vector`ized flavors shine in complex join queries. Block-based flavors (`avx512`) are the "safe" choice that does not perform best, but does not lose badly either.

6.4.3 VOILA vs. Hand-Optimized Code

We compare the runtime of VOILA-synthesized queries to state-of-the-art paradigms (a) data-centric compilation and (b) vectorized execution. As a baseline, we used the *handwritten* and *optimized* implementations by Kersten et al. [KLK+18]. Kersten et al. have shown that their implementations behave similar to the original systems Hyper and Vectorwise. We synthesized code for the basic data-centric and vectorized flavors (no prefetching, only one state-machine/goto-based) (a) `(scalar, 1, 0)`, (b) `(vector(1024), 1, 0)`, (c) using the direct Hyper back-end and (d) using the Iterator-based vectorized back-end. Table 6.3 shows our results.

Besides the vectorized Q6, we observed similar performance over all queries in a range of ±30%.

Table 6.4: VOILA can compete with hand-optimized prefetching, with further optimizations. Time in ms.

Name	Time	Speedup over `FUJI (avx512, 8, 1)`
`FUJI (avx512, 8, 1)`	1358	
Interleaved Multi-Vector. (IMV) [FZW19]	1297	1.1 ×
-*Indirections*	912	1.5 ×
-*HyperBuild*	825	1.6 ×
-*NoBuffering*	800	1.7 ×
Relaxed Operator Fusion (ROF) [MMP17]	1141	1.2 ×

Vectorized Q6. Q6 is slower because our synthesized code diverges from the Tectorwise implementation. Our implementation, first, computes all predicates and, afterward, builds the selection vector from the conjunction of the predicates. For very selective queries, Q6 in particular, this introduces overhead for eliminated rows. Tectorwise builds a selection vector for every predicate and, therefore, can avoid this additional computational effort. Therefore, we modified the plan to build the selection vector for every predicate, similar to Tectorwise. With the modified plan, our FUJI-generated vectorized Q6, runs in 1.6s and performs roughly on par with Tectorwise (1.7s).

6.4.4 VOILA vs. State-of-the-Art Prefetching

The recent re-emergence of prefetching methods highlighted the importance of intelligent data structure access for overall query performance. In this experiment, we compare VOILA-synthesized queries to hand-optimized implementations of such as Interleaved Multi-Vectorization (IMV) [FZW19] and Relaxed Operator Fusion (ROF) [MMP17].

The source code of IMV [FZW] revealed an already allocated perfectly sized hash table (size taken from a previous run). At runtime, IMV just inserts values into that hash table and builds bucket chains on-the-fly. Therefore, we implemented a flavor of IMV that can build a hash table of unknown size (*HyperBuild*). VOILA currently lacks a notation for buffering (eliminating filtered out tuples) and produces more intermediate states in concurrent state machines than strictly necessary. To enable an "apples-to-apples", in IMV, we disabled buffering (*NoBuffering*) and added 3 additional states to the state machine (*Indirections*). As baseline, we chose a FUJI flavor that resembles IMV: `(avx512, 8, 1)`. Both are using prefetching, multiple concurrent state machines and feature an implementation in AVX-512. We ran all queries single-threaded and used SF 10. Table 6.4 shows our results.

Table 6.5: VOILA-synthesized queries are significantly faster than other open-source systems. DuckDB & LegoBase do not support parallelism. Runtimes in s, on TPC-H SF 10.

System	Q1	Q3	Q6	Q9
Single-threaded				
VOILA	0.59	0.93	0.15	2.07
DuckDB	5.71 (9.5×)	2.25 (2.4×)	0.64 (4.3×)	36.26 (17.5×)
LegoBase	0.78 (1.3×)	5.19 (5.5×)	0.29 (1.3×)	32.69 (15.8×)
24 threads				
VOILA	0.05	0.25	0.03	0.19
MonetDB	1.15 (24.3×)	0.36 (1.5×)	0.72 (28.8×)	0.30 (1.6×)
Weld	0.39 (8.3×)	3.05 (12.3×)	0.11 (4.3×)	6.90 (35.5×)

Compared to IMV, the VOILA-generated query achieves a similar performance. Once we remove certain factors that ensured a fair comparison (*Indirections*, *NoBuffering*, *HyperBuild*), IMV becomes up to 60% faster. We see this as an indication that future versions of VOILA should include (a) buffering and (b) further measures to minimize the number of states (in concurrent state machines). Compared to ROF, VOILA is roughly 20% slower. A crucial difference is that VOILA does not support buffering yet.

6.4.5 VOILA vs. State-of-the-Art Open-Source

We compare VOILA to high-performance open-source systems: Weld [PTS+17], a domain-specific language for data analytics, DBLAB/LegoBase [SKP+16, dbl], an elaborate query compiler, DuckDB [RM19], a vectorized in-memory DBMS, and MonetDB [IGN+12], an in-memory DBMS executing queries column-at-a-time. In the process, we had to make adjustments to the queries in Weld and LegoBase. Weld does not support group-by on strings – required for Q9 – therefore, we gave Weld the unfair advantage of using string dictionaries. We translated `n_name` into an integer and resolve the string at the end of the query. LegoBase allows many different query-specific optimizations, e.g. string compression and partitioning, that are not "TPC-H compliant" [SKP+16]. To enable a fair comparison, we used the TPC-H compliant settings proposed by Shaikhha et al. [SKP+16]. We compare the performance of queries generated by these systems to best flavor synthesized from VOILA. Table 6.5 shows the results.

Queries generated from VOILA are up to 17.5× faster, without parallelism, and up to 35.5× faster, using all cores.

Single-threaded, VOILA-synthesized queries ran, across the board, 30% – 17.5× faster than DuckDB and LegoBase. Due to its early stage, DuckDB does not extensively focus on query performance, which explains its 4.3× – 9.5× slower performance on simple aggregation queries (Q1 & Q6).

However, performance is the main focus of LegoBase compiler. LegoBase performs similarly (±30%) on simple aggregation queries. For complex join queries, LegoBase performs significantly worse (5.5× – 15.8× slower), than VOILA. LegoBase tends to produce suboptimal code, partly caused by complex intermediate structures resulting from expressing a join (`dict[(key1, key2), list[(val1, val2]]`) which are very challenging to remove [Wad88].

Multi-threaded, all queries generated using Weld are 4.3× – 35.5× slower than queries generated from VOILA. Weld tends to perform better on the simple aggregation queries and ran only 4.3× – 8.3× slower than VOILA. Weld performs worse on complex join queries, 12.3× – 35.5× slower than VOILA. Part of this overhead in Weld is caused by complex intermediate structures. Additionally, in Weld, it is impossible to express primary-foreign-key joins (joins that can only produce one or none match) which further exaggerates already existing inefficiencies. We noticed that, for Q1 and Q6, single-threaded Weld is substantially faster and performs roughly on par with VOILA. VOILA outperforms MonetDB, especially on simple queries, by up to 28.8×. On join queries, MonetDB performs better, but VOILA is still 50% faster.

6.4.6 Engineering Aspects

We investigate the complexity of our code generation modules and compare development times to hand-writing queries.

Back-End Complexity. To understand the complexity of back-end modules, we measured the lines of code (LOC) excluding debug information, comments and empty lines.

Our compilation back-end modules are rather compact (600 – 1300 LOC). Direct back-ends tend to be simpler as they just concatenate strings (600 – 700 LOC). FUJI back-ends are more complex (600 – 1300 LOC) as (a) they include buffering logic for `BLEND` and (b) interact with other modules (FUJI front-end, FUJI generic back-end). Compared to the code generator of Weld [PTS+17] which contains 20k LOC, our back-ends are 16 – 33× more compact. Compared to Hyper's code generator, which was 10k LOC in size [Neu11], our data-centric back-end modules are 16.7×

more compact. The smaller size of our back-end modules makes them rather easy to engineer.

Personal Experiences. In our personal experience, writing a direct back-end took roughly 5 days. Compared to 1–3 months expected to handwritten specific flavors for the queries, this is a 6 – 18× speedup! FUJI back-ends, without buffering, were roughly equivalent. Buffering in FUJI back-ends provided a small extra hurdle of roughly 1-2 extra days. Compared to the code generators, most time was consumed by engineering a generic runtime framework.

6.5 Conclusion

This chapter shed some light on the practical aspects of VOILA. In particular, how VOILA can be transformed into executable code:

Generating efficient Code from VOILA. For analytical queries, a low processing overhead (i.e. efficiency) is paramount. Currently, there are two execution paradigms that can be considered state-of-the-art: Data-centric execution [Neu11] and vectorized execution [BZN05]. However, it has not been possible to synthesize both implementations from *one* representation before.

VOILA can be directly lowered into C++ (direct back-ends in Section 6.6.2). Notable examples are the data-centric direct back-end (Section 6.2.1) and the Iterator-based Vectorized (Section 6.2.2). VOILA allows synthesizing both state-of-the-art paradigms from one representation. When synthesized from VOILA, both paradigms, show comparable performance to handwritten and optimized implementations (which were shown to perform similarly to the systems [KLK+18] that originally implemented these paradigms).

Beyond the State-of-Art. Direct back-ends have the issue that for each design space point, one would need to craft one back-end. FUJI – a new code generator – mitigates this. It does so by decomposing code generation into different components. Each component then synthesizes specific parts. At code generation time, these components can be recomposed to create various hybrids (e.g. with/without prefetching, SIMD).

Design Space Exploration and Prototyping. Especially, for design space exploration, one would need to write a new back-end to prototype "crazy new ideas". In Section 6.4.6, we estimated the time required to engineer a new back-end. Compared to compiling SQL or physical plans to low-level code (e.g. LLVM IR or C++),

new back-end are relatively compact and relatively easily created. The reason is that VOILA operations often overlap (re-appear) in multiple query plan operators as well as LOLEPOPs. Most notably, group-by and join use very similar code patterns (hash chain probing). Consequently, this reduces the re-implementation effort (e.g. hash chain probing has to be only implemented once), in terms of time and space (code generator size).

FUJI went a step further, by decomposing code synthesizers into components, new ideas can be prototyped just by implementing a new component. Consequently, this can save significant time and code generator size, while allowing to easily explore neighboring points in the design space, the other components are unaffected and can be enabled at code generation time.

Needless to say, new back-ends or components would automatically benefit from the surrounding framework (data loading, queries, parallelization etc.).

Hardware-Dependency. So far, we assumed that the performance of points (in the design space) translates to different machines (if flavor α works well on machine A, it also performs well on machine B). In the following chapter, we put this simplification to the test.

CHAPTER 7

Performance is Relative

> Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth.
>
> Marcus Aurelius

7.1 Introduction

Lately, we have seen multiple new hardware architectures moving from niche to mainstream, Most notably ARM-based architectures have moved from niche, embedded devices, to ubiquitous smartphones, until reaching server-grade hardware (e.g. Amazon Graviton, Apple M1). While it is commonly understood that the best execution tactic (flavor) is known and (somewhat) predictable [KLK+18], it can be that changes in the underlying hardware affected the performance characteristics. Therefore, the statement "the best execution tactic is known and predictable" may not be universally true, but would require some extra nuance.

In this chapter, we verify this statement by testing multiple query execution paradigms on machines with different architectures (X86, ARM, PowerPC and RISC-V).

Table 7.1: The hardware bouquet used for the experiments. It consists of machines with different architectures (X86, ARM, PowerPC and RISC-V).

	Skylake-X	8275CL	Epyc	Graviton 1	Graviton 2	M1	Power8	Power9	910
Platform	X86	X86	X86	ARMv8.0	ARMv8.2	ARMv8.4	PPC	PPC	RISC-V
Architecture	Skylake-X	Cascade Lake-SP	Zen 2	Cortex-A72	Neoverse N1	Fire-/Icestorm	POWER8	POWER9	C-910
Processor Model	Xeon Gold 6126	Xeon Platinum 8275CL	Epyc 7552	Graviton 1	Graviton 2		POWER8	POWER9	C-910
Threads per Core	1	2	2	1	1	1	8	4	1
Cores per Socket	12	24	48	4	64	4+4	8	16	2
Sockets	2	2	1	4	1	1	2	2	1
NUMA Nodes	2	2	1	1	1 [Ama21b]	1	2	2	1
RAM (GiB)	192	192	192	32	128 [aws21a]	16	256	128	4
L3 (MiB)	19.25	35.75	192	-	32 [aws21a]	-	64	120	-
L2 (kiB)	1024	1024	512	2048 [aws21b]	1024 [aws21a]	4096/2048	512	512	2048
L1d (kiB)	32	32	32	32 [aws21b]	64 [aws21a]	128/64	64	32	64
L1i (kiB)	32	32	32	48 [aws21b]	64 [aws21c]	192/128	32	32	64
Freq. max (Ghz)	3.7	3.6	3.5	2.3 [aws21b]	2.5 [aws21c]	3.2/2	3.0	3.8	1.2
Freq. min (Ghz)	1.0	1.2	1.8	2.3 [aws21b]	2.5 [aws21c]	0.6/0.6	2.0	2.1	1.2
AWS Instance	-	c5.metal	c5a.24xlarge	a1.metal	r6gd.metal	-	-	-	-

7.2 Methodology

We conduct micro- to macro-level experiments with varying and, rather, diverse hardware.

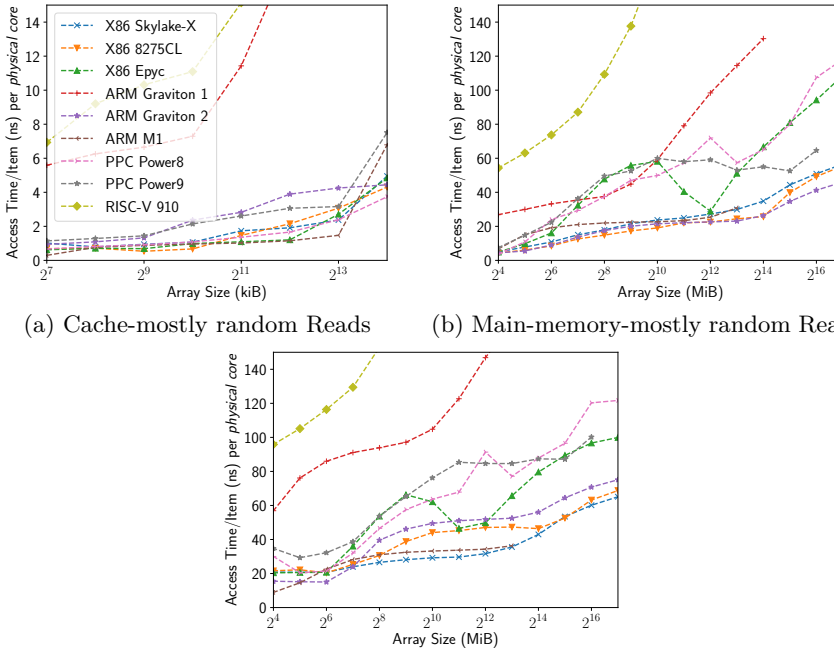
Hardware. For our experiments, we used 3 X86 machines (Skylake-X, 8275CL and Epyc), 3 ARM machines (Graviton 1, 2 and M1), 2 PowerPC machines (Power8 and Power9) and one RISC-V machine (910). Details can be found in Table 7.1. Some machines have noteworthy special features: The Graviton 2 has accelerated access to always-encrypted memory, as well as acceleration for fast compression and decompression [Ama21b]. The M1 features a heterogeneous design of 4 fast CPU cores (Firestorm), 4 slow CPU cores (Icestorm), an integrated GPU and acceleration for Neural Networks. The 910 appears to be an early prototype of a RISC-V-based machine – rather a development board – and appears to target functionality testing, rather than performance (e.g. seems to lack proper cooling). Therefore, we only included the 910 into the basic micro-benchmarks, as its performance seems to be the worst of our hardware bouquet. Furthermore, neither Graviton 1, nor M1, nor 910 seem to have an L3 cache.

Synthesizing Efficient Implementations. Rather than implementing the required queries by hand, we synthesize them. VOILA (Chapters 5 and 6) allows synthesizing many implementations from one query description. It generates data-centric and vectorized flavors that perform on-par with handwritten implementations and, implicitly, the systems Hyper [Neu11] and Vectorwise [BZN05]. VOILA also allows generating mixes that facilitate prefetching and efficient overlapping of prefetching with useful computation, similar to AMAC [KFG15] and IMV [FZW19]. Here, we use the VOILA-based synthesis framework to generate the implementations required for our experiments on holistic query performance. To investigate such impacts, we ported the VOILA synthesis framework to non-X86 architectures.

7.3 Micro-Benchmarks

We start with micro-benchmarks that stress specific aspects of the underlying hardware: (a) memory access, (b) data-parallel computation and (c) control flow & data dependencies.

For each type, we implemented vectorized primitives, functions that operate on columnar vectors in a tight loop. We ran the micro-benchmarks multi-threaded using all available threads to the operating system (i.e. including SMT, if available).



(a) Cache-mostly random Reads

(b) Main-memory-mostly random Reads

(c) Main-memory-mostly random `SUM`

Figure 7-1: Memory-heavy workloads.

We report per-tuple timings in nanoseconds, normalized by the number of SMT threads (e.g. for 1 real core and 8 SMT threads, we divide the time by 8).

7.3.1 Memory Access

In modern database engines, memory access is a well-known bottleneck for certain queries [BKM08]. Therefore, we investigate memory access performance. We differentiate between (1) cache-mostly random reads, (2) bigger-than-cache random reads and (3) the `SUM` aggregate as a read-update-write workload. Each experiment accesses an array of 64-bit integers at pseudo-random indices.

Cache-mostly random Reads. The runtimes for each random access read from a small array are plotted in Figure 7-1a. In general, we see very little difference, but two extreme outliers: The Graviton 1 and the 910 feature very slow cache access, of which the 910 shows the worst performance. Faster than Graviton 1, but slower than the others (Skylake-X, 8275CL, Eypc, Power8) are Power9 and Graviton 2. Both provide relatively slow access to small-to-medium-small arrays. The Graviton 2, however, can "catch up with the crowd", for slightly bigger arrays.

Main-Memory-mostly random Reads. We now move to large arrays which likely do not fully reside in caches. That means that on NUMA machines, we might see NUMA overheads (data needs to be shipped from one socket to another). As massive arrays do not fit into one NUMA node, we decided to interleave the whole array over all NUMA nodes (round-robin assignment of pages to NUMA nodes, page `i` assigned to node `i % num_nodes`). Essentially, this resembles the initial bucket lookup of the scalable non-partitioned hash join described by Leis et al. [LBKN14] which also effectively interleaves the array over all NUMA nodes.

For large arrays, the random reads are visualized in Figure 7-1b. We see a rather big divergence, especially with growing array size.

Skylake-X, 8275CL and Graviton 2 perform best, of which the Graviton 2 outperforms the other two for arrays $> 2^{14}$ integers. It has to be said that Graviton 2 offers direct non-NUMA memory access whereas, both, the Skylake-X and the 8275CL, are NUMA machines and might require transferring data from other sockets over a shared bus. While this is a rather uncommon architectural decision, this gives the Graviton 2 an advantage, as it can avoid costly data transfers over a shared bus and use fast local memory instead. The other competitive non-NUMA machine is the Epyc, but it tends to feature rather slow memory access for L3-mostly (until 2^7 MiB) and main-memory-mostly random reads.

`SUM` Aggregate. The `SUM` is a read-update-write workload and will, therefore, for many core systems involve a cache invalidation cost (invalidate, write back and possibly read from another core or socket). However, due to the relatively large size of the array, the chance of such false sharing (writing a cache line that is, later, read and modified by another core or socket) is relatively low. Figure 7-1c shows our measurements. The measurements look similar to random-read workload, but tend to be slower. We can observe that the Skylake-X and 8275CL outperform for large arrays, followed by the Graviton 2, which outperforms for arrays $\leq 2^7$ MiB.

Conclusions. Both, the Graviton 1 and the 910, show slow memory access across the board. Power8 and Epyc are slower for larger arrays. The Skylake-X and 8275CL are fast across the board. Graviton 2 is relatively slow (compared to Skylake-X, 8275CL) on smaller arrays ($\leq 2^{13}$ kiB), but outperforms for large arrays ($\geq 2^{15}$ MiB).

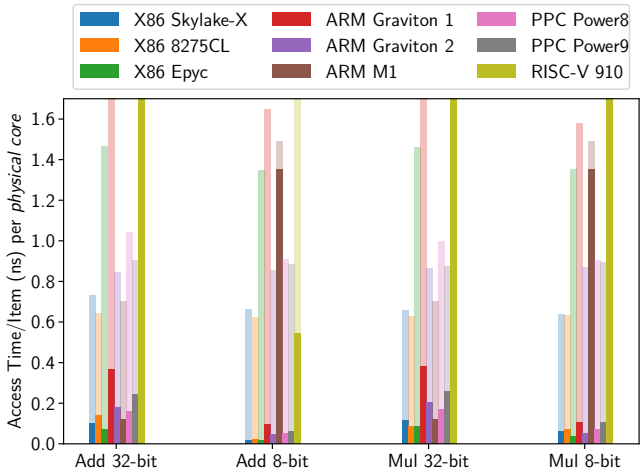


Figure 7-2: Computation without selection vector, amenable to SIMD acceleration (opaque). Computation with selection vector introduces indirection (transparent).

7.3.2 Data-Parallel Computation

Besides memory access, the other important cornerstone for query performance is computational power. As examples of computation-heavy workloads, we performed a series of additions and multiplications in a tight loop. We investigate the performance of relatively cheap (addition) vs. relatively pricey (multiplication) operations, and the impact of thin data types (Chapter 3). For vectorized kernels, there are two alternative paths that influence performance: (a) the non-selective path that only fetches data from input vectors, computes the result and writes the output, and (b) the selective path which introduces an additional indirection.

The non-selective path accesses plain arrays (vectors) in sequential order and is, thus, amenable to acceleration with SIMD. Thanks to the trivial access pattern, the compiler will typically automatically SIMD-ize this code path.

Selective execution intends to only process selected tuples (i.e. efficiently ignore values that are filtered out). We implemented this path, like described by the classic vectorized execution model [BZN05], using selection vectors that describe which indices in the vectors are alive. Due to this indirection, the compiler will typically not SIMD-ize this path. To reduce branch prediction overhead (in the `for` loop), we unrolled this path 8 times.

Performance. Our results can be found in Figure 7-2. At the first glance, we see that selective computations (transparent) are significantly pricier than non-selective computations (opaque). This is due to the extra indirection, which is (a) expensive

Table 7.2: Graviton 1 & 2 provide consistent performance over multiple degrees of parallelism (DOP, N), while many others show significant slowdowns. (32-bit integer multiplication without selection vector, without adjustment for SMT, DOP beyond real cores are marked in italics on gray)

Hardware		Time/Item in ns (slowdown) on given DOP N				
Machine	T	N=1	$T/8$	$T/4$	$T/2$	T
X86 Skylake-X	24	0.08	0.08 (1.0×)	0.09 (1.1×)	0.09 (1.2×)	0.12 (1.5×)
X86 8275CL	96	0.08	0.08 (1.0×)	0.10 (1.4×)	0.17 (2.3×)	*0.18 (2.3×)*
X86 Epyc	96	0.08	0.08 (1.0×)	0.08 (1.0×)	0.12 (1.5×)	*0.17 (2.3×)*
ARM Graviton 1	16	0.34	0.34 (1.0×)	0.36 (1.1×)	0.35 (1.0×)	0.38 (1.1×)
ARM Graviton 2	64	0.20	0.20 (1.0×)	0.20 (1.0×)	0.20 (1.0×)	0.20 (1.0×)
ARM M1	8	0.07	*is* $N = 1$	0.08 (1.1×)	0.08 (1.1×)	0.12 (1.7×)
PPC Power8	128	0.26	0.38 (1.4×)	*0.68 (2.6×)*	*1.27 (4.8×)*	*1.38 (5.2×)*
PPC Power9	128	0.19	0.27 (1.4×)	0.52 (2.7×)	*0.95 (5.0×)*	*1.04 (5.4×)*

and (b) prevents efficient SIMD-ization. To alleviate this overhead, in a vectorized system one would, typically, ignore the selection vector, when the vector more than, say, 30% full ("full computation" [RBZ13]).

Between machines, we also see significant differences. The 910 shows "off the charts" performance for 32-bit additions and multiplications, and is roughly 24× slower than the Skylake-X or the Epyc. The Graviton 1 significantly faster than the 910, but is roughly 4× slower than the Skylake-X or the Epyc. The fastest machines are Skylake-X, 8275CL and Epyc which are roughly 2× faster than the other machines. For ARM platforms, the slowdown is likely caused by the lack of compiler auto-vectorization. For non-selective workloads, thin data types provide significant benefit on most platforms. However, this benefit disappears when the selection vector is used. A notable exception is the M1, which not only lacks vectorization benefits but also incurs additional overhead for loading and storing 8-bit integers. For the 8-bit addition, the M1 even performs worse than the 910.

Scalability. Typically, when using all available cores, processors tend to reduce computational throughput significantly. This is typically due to heat emission. Each chip has a given thermal budget (thermal design power, TDP): If the budget is reached, heat emission needs to be curtailed. Therefore, cores clock down and, thus, scale computational throughput down.

In this experiment, we run many SIMD-izable (non-selective) 32-bit integer multiplications with varying degrees of parallelism (DOP, or number of threads N). We scale the DOP from 1 (no parallelism) to the level of parallelism the hardware provides (T). Note that this experiment evaluates the *best-case*, as the scalability of OLAP queries is typically limited by other factors, such as memory-access. Table 7.2 shows our results.

Listing 7.1: Vectorized kernel: Creating a selection vector using branches

```
int select_true(int* res, i8* a, int* sel, int n) {
  int r = 0;
  if (sel) {
    for (int i=0; i<n; i++) {
      if (a[sel[i]]) res[r++] = sel[i];
    }
  } else {
    for (int i=0; i<n; i++) {
      if (a[i]) res[r++] = i;
    }
  }
  return r;
}
```

With increasing DOP, we see a tendency to significant slowdowns of 50% to 2.7× without using SMT cores and up to roughly 5× with SMT cores. However, except for the Graviton-based platforms which, evidently, do not clock down significantly. M1 showed less throughput beyond 4 cores. This can be explained by the design of the M1 that combines 4 fast and 4 slow cores, i.e. workloads with >4 threads ($> T/2$) will also use the slower cores.

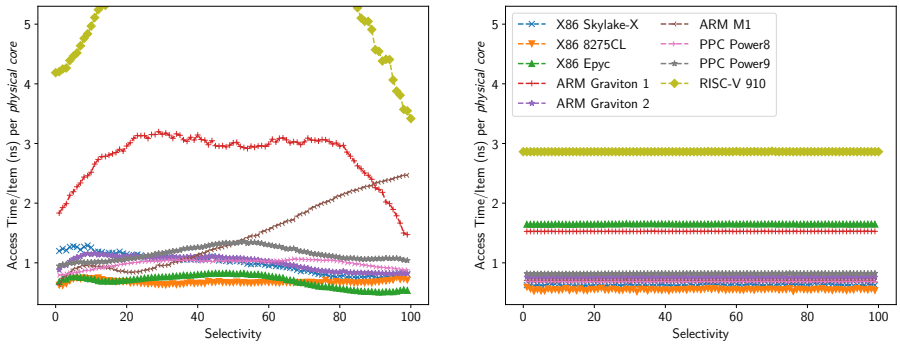
7.3.3 Control Flow & Data Dependencies

Besides data-parallel computation, modern engines typically also rely on fast control flow and data dependencies. Depending on the hardware (e.g. pipeline length), branch misses can become quite costly, and data dependencies reduce the CPU pipeline parallelism. In vectorized engines, such operations appear when creating a selection vector (e.g. in a filter or hash-based operators). Therefore, we benchmark the performance of selection vector creation.

Selection vectors can be built in multiple ways: Most commonly, they are created using (a) branches or (b) data dependencies. Alternatively, one can create selection vector using X86-specific AVX-512 `vpcompressstore` [KLK+18]. While this method is often faster [KLK+18], creating selection vectors using AVX-512 is not portable to other hardware architectures.

A branch-based implementation, as in Listing 7.1, stresses the branch predictor. For very high/low selectivities, the branch becomes predictable. The closer the selectivity comes to 50%, the more unpredictable the branch becomes.

As an alternative to creating selection vectors using branches, one can introduce a data dependency. In pseudocode, in Listing 7.1, this means replacing `if (a[k]) res[r++] = k` by `res[r] = k; r+= a[k]`. Obviously, this avoids the overhead of mispredicting branches, but might introduce additional costs for predictable branches.



(a) Branch-based creation (b) Dependency-based creation

Figure 7-3: Control-Flow & Data-Dependency-intensive workload: Creating a selection vector from an 8-bit boolean predicate.

Results. For the branch-based creation of selection vectors, our results can be found in Figure 7-3a. We refer to selectivity as the fraction of tuples that pass the filter (100% = all pass). Typically, one would expect lower timings (faster) for very low and high selectivities because the branch becomes predicable. Around 50% selectivity, one would expect the worst performance as the branch is unpredictable. We can observe this behavior on the Epyc and Power9. The plot of the Graviton 1 stands out due to very high cost, and edge behavior at middle-low and -high selectivities, which are costlier than branches of 50%. Also interesting, is the plot of the M1, which exhibits an asymmetrical shape where taking the branch (`if(a[i])`) is pricier than skipping it.

We now juxtapose the performance of branch-based (Figure 7-3a) with the dependency-based selection vector creation (Figure 7-3b). In general, dependency-based creation tends to outperform, with one notable exception: The Epyc for which the branch-based created is faster. For the 8275CL, both variants are roughly equal. Thus, it can be said that the best way to create a selection vector depends on the hardware at hand. In a system, it is advisable to determine the choice between branch- and dependency-based selection vector building adaptively at runtime.

7.3.4 Case Study: Hash Join

As we gradually move towards macro-benchmarks, we now investigate the performance of a hash join. In particular, we are interested whether the heuristic *vectorized execution excels in data-access-heavy workloads* is true on different hardware/platforms.

Table 7.3: Best execution paradigm unclear for simple join query. Best flavor as tuple (Computation Type, Prefetch, #FSMs), data-centric (blue) and prefetching (italics).

Machine	Best Flavor		Well-known Flavors	
	Name	best (ms)	x100 (ms)	hyper (ms)
X86 Skylake-X	*vec(1024),3,1*	173	209	239
X86 8275CL	*vec(512),3,2*	176	192	236
X86 Epyc	*scalar,4,16*	134	139	157
ARM Graviton 1	scalar,0,1	412	440	412
ARM Graviton 2	vec(1024),0,1	101	101	115
ARM M1	*vec(1024),4,1*	228	273	297
PPC Power8	*vec(512),2,1*	488	498	507
PPC Power9	*scalar,3,1*	317	339	317

Therefore, we synthesized the following SQL query using the VOILA-based synthesis framework (Chapter 5):

```
SELECT count(*) FROM lineitem, orders
WHERE o_orderdate < date '1996-01-01'
  AND l_quantity < 50 AND l_orderkey = o_orderkey
```

We ran this query in multiple flavors on the TPC-H data set with scale factor 10. Table 7.3 shows the best flavor as well as the runtimes of the data-centric (*hyper*) and vectorized (*x100*) execution.[1]

We see that the majority of the best flavors are indeed vectorized. To our surprise, data-centric flavors can beat vectorized flavors. Typically, the winning data-centric flavors need elaborate prefetching to outperform, with one notable exception: On the Graviton 1, the plain data-centric flavor (without Finite State Machines and without prefetching) outperforms the vectorized flavors.

Favorable Features for Data-Centric Execution. We believe that on the Graviton 1 the $3-4\times$ slower computation (*slower cores*) (Figure 7-2) favors data-centric execution because the more efficient computation (data-centric, less issued instructions) outweighs the less efficient memory-access.

The other machines feature faster cores. The *massive L3 caches*, on the Epyc and Power9, tend to benefit data-centric flavors, as huge L3 caches leads to effectively faster memory access (more data in faster memory) and, thus, making more efficient memory access (using vectorized execution) less important. This is further exaggerated via *SMT*, which can effectively hide memory access latency by executing another thread. On the Epyc, both features (SMT and large L3) are barely enough

[1] We use x100 as short identifier for vectorized execution, in reference to MonetDB/X100 [BZN05] – which later became Vectorwise and currently is called Vector.

to allow a data-centric flavor to win (134ms vs. 139ms, roughly on noise level). We notice similar behavior on the Power8/Power9, which feature a large L3 cache and, compared to the Epyc, a higher degree of SMT (8 threads on Power8, or 4 threads on Power9, vs. 2 threads per core).

In summary, we can say that, certain hardware properties (slow cores, large L3 cache and SMT) have a tendency to favor data-centric execution, for joins.

7.4 Macro-Benchmarks

While micro-benchmarks, provide useful insights into extreme cases, it is difficult to draw conclusions on holistic query performance. Queries are more complex than simple operations and are, thus, rarely completely limited by either memory bandwidth, or computational throughput. Using VOILA, we generated implementations for TPC-H Q1, Q3, Q6 and Q9. For our benchmarks, we used the TPC-H data set with scale factor 10. For each query, we sampled 50 different execution flavors from the universe that VOILA can generate, always including the two most well-known ones: pure data-centric compilation [Neu11] and pure vectorized execution [BZN05].

7.4.1 Query Performance

Here, we compare the runtime of data-centric (*hyper*) and vectorized (*x100*) on varying hardware in terms of overall system performance and per-core performance. The results are visualized in Table 7.4.

Overall Performance. We notice a significant diversity in overall runtimes of up to, roughly, 10× between the fastest and the slowest machine. Common wisdom would suggest that X86 would perform best, but surprisingly, the ARM Graviton 2 significantly outperforms all others. Compared to the runner-up (Skylake-X), it performs up to 3× faster (Q9 hyper) and, often executes queries roughly 2× faster (Q1, Q3, Q9).

On Q9, we see data-centric flavors outperforming vectorized flavors, for the Epyc, Graviton 1 and Power9. This is due to the hardware properties we identified in Section 7.3.4: massive L3 caches (Epyc, Power9), SMT (Epyc, Power9) as well as slow cores (Graviton 1). These factors favor data-centric execution on join-intensive workloads such as Q9 in particular.

Another important query is Q3, which is less join-heavy and is, therefore, less suited to vectorized execution. We noticed that data-centric outperforms on 8275CL, Epyc,

Table 7.4: Graviton 2 beats all other machines in overall performance. M1 leads in performance per core. Runtimes of well-known query execution paradigms.

Machine	Q1		Q3		Q6		Q9	
	x100	hyper	x100	hyper	x100	hyper	x100	hyper
Runtime (milliseconds)								
X86 Skylake-X	79	54	261	282	28	35	228	291
X86 8275CL	93	84	480	397	70	112	232	261
X86 Epyc	81	65	241	238	51	52	193	180
ARM Graviton 1	188	107	447	447	55	45	720	715
ARM Graviton 2	**42**	**29**	**162**	**158**	**20**	**22**	**95**	**109**
ARM M1	216	86	313	440	138	404	432	590
PPC Power8	404	384	1094	1132	336	337	627	636
PPC Power9	239	225	645	631	190	192	406	393
*Runtime * Number of real cores (seconds)*								
X86 Skylake-X	1.9	1.3	6.3	6.8	**0.7**	0.8	5.5	7.0
X86 8275CL	4.5	4.1	23.1	19.0	3.4	5.4	11.1	12.5
X86 Epyc	3.9	3.1	11.6	11.4	2.4	2.5	9.3	8.6
ARM Graviton 1	3.0	1.7	7.1	7.2	0.9	**0.7**	11.5	11.4
ARM Graviton 2	2.7	1.9	10.4	10.1	1.3	1.4	6.1	7.0
ARM M1	**1.7**	**0.7**	**2.5**	**3.5**	1.1	3.2	**3.5**	**4.7**
PPC Power8	6.5	6.1	17.5	18.1	5.4	5.4	10.0	10.2
PPC Power9	7.6	7.2	20.6	20.2	6.1	6.1	13.0	12.6

Graviton 2, Power8 and Power9. This is partially caused by the hardware factors we identified (L3 size, SMT, slow cores) and, partly, by the structure of the query.

Per-Core Performance. We scale the multi-threaded performance up to the number of real cores. This provides a per-core performance metric that includes potential multi-core scalability bottlenecks (e.g. down clocking, memory bandwidth). Implicitly, this metric favors systems with a lower number of cores (typically close to desktop systems).

The machines with a lower number of cores (Graviton 1, M1 and Skylake-X) score best. This is partly an artifact of queries not scaling perfectly sometimes, but it indicates real per-core performance (i.e., in a parallel workload). This per-core performance is dominated by ARM platforms, except for *Q6 x100*. But to our surprise, the Graviton 2, *with 64 cores*, comes close to the best 3. This indicates that Graviton 2 scales quite well, up to all 64 cores, on whole queries and not just in micro-benchmarks (Section 7.3.1 and Section 7.3.2).

7.4.2 Optimal Flavor

Here, we investigate which execution paradigm (flavor) is the best for each query. Using the VOILA-based synthesis framework, we generated basic flavors, i.e. one flavor per query (no mixes). Table 7.5 shows the flavors with the lowest average runtime.

Table 7.5: Best query execution paradigm unclear, even for specific queries (e.g. Q9). Best flavor as tuple (Computation Type, Prefetch, #FSMs). Data-centric (scalar) flavors are marked in blue color, prefetching in italics.

Machine	Q1	Q3	Q6	Q9
X86 Skylake-X	scalar,0,1	*vec(2048),3,1*	*vec(1024),3,1*	*vec(1024),2,1*
X86 8275CL	*scalar,2,1*	*scalar,3,8*	*vec(512),4,1*	*scalar,2,32*
X86 Epyc	*scalar,2,1*	*vec(256),1,1*	*vec(1024),2,1*	vec(512),0,1
ARM Graviton 1	scalar,0,1	vec(512),0,1	*scalar,4,1*	vec(256),0,1
ARM Graviton 2	*scalar,2,1*	scalar,0,1	*vec(2048),2,1*	vec(512),0,1
ARM M1	scalar,0,1	*vec(2048),2,1*	*scalar,3,1*	*vec(1024),2,1*
PPC Power8	scalar,0,1	*vec(1024),2,1*	vec(256),0,1	*scalar,2,2*
PPC Power9	*scalar,3,1*	vec(512),0,1	*vec(256),2,1*	*scalar,2,8*

Best Flavor. We can see that some configurations, most notably Skylake-X and Epyc, exhibit the behavior described by Kersten et al. [KLK+18] that data-centric wins in Q1 and vectorized wins in Q3 and Q9. However, surprisingly, we found data-centric flavors winning on the join-heavy Q3 and Q9. Notably, these are augmented data-centric flavors with prefetching and multiple finite state-machines (FSMs) that allow overlapping prefetching with useful computation. In particular, these augmented data-centric flavors perform well on large machines with multiple threads per core (SMT), i.e. 8275CL, Power8 and Power9.

Although, the winning flavor on Q1 is data-centric (scalar), we can see the use of prefetching for a query that runs mostly in cache. This is caused by the low overhead introduced by prefetching rather than the actual benefit of prefetching (improvements between data-centric and best flavor in Q1 are on noise level, i.e. $<$ 15%).

Are the "best Flavors" really better? Table 7.6 shows the improvement of the best flavors over the well-known data-centric (*hyper*) and vectorized (*x100*). Often, the best flavor outperformed the well-known ones by 10-31%. In some cases, the difference was on noise level, but in other cases the best flavor significantly outperforms well-known ones by up to 220%. Therefore, we can conclude that there is significant performance diversity, which – in some cases – can be exploited. However, taking advantage of this diversity, in practice, would require significantly more flexible engines.

7.4.3 Costs & "Bang for the Buck"

While the ARM Graviton 2 might outperform the other machines on performance, it may not necessarily provide the best performance and a price-adjusted basis.

Table 7.6: Best flavors outperform by up to 220%. Runtime improvement over plain vectorized (*x100*)/data-centric (*hyper*).

Machine	Best vs. *x100* (%)				Best vs. *hyper* (%)			
	Q1	Q3	Q6	Q9	Q1	Q3	Q6	Q9
X86 Skylake-X	47	3	5	13	*is*	12	29	44
X86 8275CL	14	54	12	16	4	27	79	31
X86 Epyc	29	8	3	19	3	6	7	11
ARM Graviton 1	77	1	22	12	*is*	1	0	11
ARM Graviton 2	62	3	15	2	12	*is*	23	17
ARM M1	150	7	10	9	*is*	51	220	49
PPC Power8	5	9	3	9	*is*	12	3	11
PPC Power9	7	3	3	10	1	1	5	7

Table 7.7: ARM Gravitons are significantly cheaper and provide most "bang for the buck".

Machine	$\frac{\$}{\text{hour}}$	$\frac{\text{Cents per real core}}{\text{hour}}$	Q9 (ms)	1M × Q9 ($)
X86 Skylake-X (price est.)	1.3392	5.6	228	84
X86 8275CL	0.9122	1.9	232	59
X86 Epyc	0.9122	1.9	193	49
ARM Graviton 1	**0.0788**	**0.5**	720	**16**
ARM Graviton 2	0.7024	1.1	**95**	19

Therefore, we investigated the costs for renting the hardware, used for our experiments, and discuss cost performance trade-offs.

For pricing, we used the spot prices reported on AWS [Ama21c]. Unfortunately, PowerPC architectures and the M1 were not available. Even though we did not use AWS for our Skylake-X machine, we found a similar instance type (*z1d.12xlarge*) that we used for pricing.

Cost. We visualized the costs in Table 7.7. From that table, it is evident that ARM-based instances are up to 12× cheaper per hour and 11× cheaper per core. The most expensive instance is the Skylake-X. It is also the best performing X86 machine (Q1, Q3, Q6 and Q9) and is only beaten on Q9 by the Epyc.

Cost per Q9 run. Typically, faster machines are pricier. Therefore, we calculated the cost for 1 million runs of vectorized flavor of TPC-H Q9 with scale factor 10.

On this metric, the ARM instances outperform by $> 2\times$. Compared to the cheapest X86-instance (Epyc), the Graviton 1 is 3× cheaper per run, whereas the Graviton 2 is 2.5× cheaper.

7.5 Conclusion

VOILA allows generating many semantically equivalent implementations of query execution paradigms. In this chapter, VOILA was used to benchmark these implementations on a bouquet of hardware consisting not only of X86, but also PowerPC, ARM and RISC-V machines.

The experiments suggest that the performance of query execution paradigms depends on the hardware at hand (e.g. cache sizes) and structure of the query (e.g. join heavy, many cache misses). Neither data-centric nor vectorized execution is universally good or bad. Previously, it has been assumed that Vectorized Execution wins in join-heavy workloads [KLK+18], which is evidently *not universally true*.

The trade-off between data-centric and vectorized execution can be pushed to benefit data-centric through certain hardware features: (a) large caches, (b) high levels of SMT (simultaneous multi-threading), or (c) slower cores).

Besides, the experiments have shown that ARM architectures can exhibit comparable, if not better, performance (compared to state-of-the-art X86 architectures).

In summary, modern, more so future, query engines have to provide robust performance and an increasingly diverse set of hardware (most notably X86 and ARM). Frequently, the best solution (query execution paradigm) is – if predictable at all – very difficult to predict on one hardware platform. In practice, an engine has to run on a broad set of hardware and needs to cope with dynamic effects (e.g. interference with other queries, processes etc.), which makes predicting the best paradigm even more challenging.

Therefore, it is not inconceivable, to abandon the idea that we can predict everything perfectly, but rather find good solutions through trial and error. The next chapter will discuss a prototype of such an adaptive (and learning) engine.

CHAPTER 8

Adaptive Query Execution using Virtual Machines

> Write a paper promising salvation, make it a 'structured' something or a 'virtual' something, or 'abstract', 'distributed' or 'higher-order' or 'applicative' and you can almost be certain of having started a new cult.
>
> Edsger Dijkstra

8.1 Introduction

In the previous chapter, we noted that there is no such thing as the best query execution paradigm, for all queries and all possible hardware combinations. If a future query engine were to exploit performance diversity rather than falling victim to it, it would need to synthesize different flavors at runtime and somehow find the best flavor. With VOILA, described in Chapters 5 and 6, we have the means to synthesize many different execution styles (flavors) from a single description.

In this chapter, we go one step further. We dynamically synthesize and try many flavors, *while the query is running*. Our prototype, Excalibur, is based on a virtual machine (VM) that abstracts many nasty details away (e.g. code synthesis and interpretation) and, therefore, provides an efficient and convenient framework to switch between different execution flavors (e.g. minimizing excessive compilation overhead).

As the previous chapter described, it is hard to impossible to predict the best flavor via cost models. Therefore, we try to find the best flavor via trial and error, i.e. make a choice according to some policy and measure the *actual runtime* on a sample. This approach is often also known as adaptivity or micro-adaptivity [RBZ13]. The major challenge is that significant exploration of the design space requires significant time. This excessive time consumption frequently makes exploration at runtime impractical for large spaces. Note that in practice, runtime behavior might change and points might need to be revisited.

In Excalibur, we carefully control the time spent on risky exploration by using a budget β (typically 30% query runtime). Consequently, this leaves even less time for exploration and makes the order in which points are explored more relevant. Therefore, we explore different strategies to search the design space more efficiently.

8.2 Background

This section introduces preliminary concepts: The domain-specific language VOILA, the Multi-Armed Bandit problem and the Upper Confidence Bound algorithm.

VOILA. Excalibur builds on top of the domain-specific language VOILA (Chapter 5). VOILA allows describing operators in a way that exposes data-parallelism and, thus, implicitly allows synthesizing SIMD-ized code or vectorized execution. Using VOILA, different back-ends can generate *very* different code styles and query execution paradigms (Chapter 6). In Chapter 6, We have shown that the VOILA synthesis framework can cover a large design space.

Multi-Armed Bandits (MAB). In practice, we often have choices, but we do not know which one is best. Instead, we want to find the best choice at runtime (online learning). We can either explore (try new or re-try old choices) or exploit (use the best choice found, so far). This is commonly abstracted using the MAB problem. The MAB problem can be imagined as a row of slot machines, and we want to maximize our possible reward by using the machine most favorable to us. To

achieve this, we need to observe the distributions of all slot machines (exploration). Once we are confident about the distributions, we can pull the lever on the most favorable slot machine (exploitation) and pocket the rewards. The goal is to solve this problem with a low, hopefully sublinear, regret (loss compared to the best possible choice).

Upper Confidence Bound (UCB). One algorithm that solves MAB optimally is the Upper Confidence Bound algorithm (UCB) [ACBF02]. UCB tends to be an elegant and effective solution to the MAB problem with an attractive sublinear regret. For each arm i, we define a score $ucb_i(T)$ and we always choose the arm with the highest score, i.e. $argmax_i\ ucb_i(T)$, at a time-step T (number of calls to the algorithm), For an arm i, let N_i be the number of samples collected so far, X_i the empirical mean of rewards, and an independent constant c. The score is defined as:

$$ucb_i(T) = \begin{cases} \infty & \text{if } N_i = 0 \\ X_i + c * \sqrt{\frac{log(T)}{N_i}} & \text{otherwise} \end{cases} \tag{8.1}$$

8.3 Excalibur

Excalibur is a system prototype[1] intending to make query execution flexible & dynamic. It allows trying and exploiting many different execution styles (flavors), while executing the query. We now walk through its architecture in Figure 8-1.

To execute a query, its query plan is handed over to Excalibur, along with readers that allow scanning the base tables involved in the query. From there on, Excalibur translates the plan into its own plan representation (Low-Level Plan). In the Low-Level Plan representation, the query is split into pipelines (Pipeline 1 and Pipeline 2) with simple operator chains inside each pipeline. These operator chains can be and are pipelined to minimize the size of intermediates. Afterward, we expand plan operators into code in the domain-specific language VOILA (shown for Pipeline 1). The VOILA program is used to generate byte code which can efficiently be interpreted. This step also involves generating the required code fragments that are invoked by the byte code. These fragments could already be cached and then do not require compilation. After the code generation has finished, the pipeline is evaluated by interpreting each operator in a (vectorized) iterator-based fashion, i.e. calling a `next()` method that returns batches of tuples (typically 1024) produced by the operator. Inside the `next()` method, the byte code interpreter calls compiled code fragments for each byte code.

[1] Source code and scripts can be found under `https://github.com/t1mm3/db_excalibur`

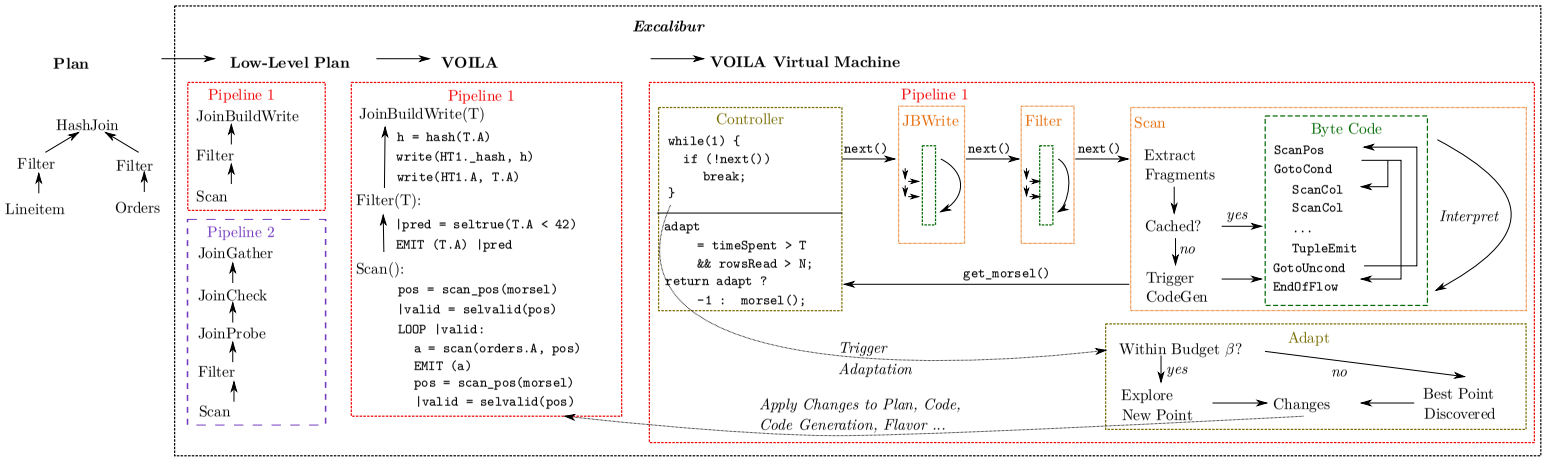


Figure 8-1: Excalibur Architecture. Excalibur uses multiple layers to generate code and allows adaptive re-optimization based on runtime feedback. Instead of full compilation and re-compilation, Excalibur allows reusing already compiled fragments.

Instead of fully evaluating the pipeline, we can interrupt execution after a certain number of tuples or CPU cycles. This is handled by the Controller, which triggers the evaluation of the top-most operator (in the chain) and suspends evaluation by choking the scan (`get_morsel()` returning 0 tuples). This interrupt allows modifying the current execution flavor ("Trigger Adaptation" in Figure 8-1). Whether Excalibur can explore new flavors or rather exploit the already explored points is decided through a Budget β. If there is enough budget, new flavors are explored, otherwise the best discovered flavor will be run. After Trigger Adaptation, changes are applied to the VOILA byte code and execution of the pipeline is resumed. For example, to switch to executing a query in data-centric flavor, we inline all operators into the top-most operator, compile this into one code fragment and reconfigure the byte code of the top operator, deactivating the other operators.

8.3.1 Execution Model

Excalibur uses two levels of relational operators: (a) high-level operators, like e.g. `HashJoin`, and (b) low-level operators that encode which operations a high-level operator, e.g. `HashJoin`, must perform.

High-Level operators are rather a logical construct than a part of physical query execution. They own the state shared by the low-level operators (most notably data structures) and provide high-level features such as progress estimation (needed later).

Low-level operators specify the physical implementation of a corresponding high-level operator. Each low-level operator uses the vectorized Volcano model, i.e. it is an iterator with a `next()` method that returns multiple tuples stored as an array of columnar vectors. While our low-level operators share some similarity with LOLEPOPs [Loh88], they are different from DB2 BLU [RAB+13] or Starburst [HCL+90]. Specifically, we further decompose the join into sub-operators.

Instead of a monolithic `HashJoin` operator, we use a sequence of `JoinProbe`, `JoinCheck` and `JoinGather`, of course after building the hash table using `JoinBuild`. Consequently, our joins can be easily extended in the future, e.g. `JoinProbe` can be replaced by using a perfect hash [BLP+14, BNE13] or see Chapter 3. Low-level operators are the physical unit of query execution. An operator can be white-box (expressed in our domain-specific language VOILA) or black-box, which allows integrating operators for which no representation in VOILA exists (e.g. the `Output` operator that materializes the query result). White-box operators expose VOILA code and, therefore, qualify for compilation and interpretation-compilation hybrids.

Table 8.1: Byte Code instructions. Certain instructions are not strictly necessary but exist for performance-purpose, these instructions are marked with an asterisk (*).

Byte Code Instruction	Description
`GotoCond`	If condition == constant: Goto "line"
`GotoUncond`	Goto "line"
`EndOfFlow`	Signal end of stream
`End`	End of program
`Copy`	Copy value/vector
`Emit`	Returns tuples from operator
`ScanPos`	Allocates a position for reading a table
`ScanCol`	Reads a column chunk from `ScanPos`
`SelNum`	Turns position inside table into predicate
`WritePos`	Allocate a position for writing a table
`CompiledFragment`	Call compiled VOILA fragment
`BucketInsert`*	*(Complex) VOILA operation*
`SelUnion`*	*(Complex) VOILA operation*

Almost all relational operators in Excalibur are implemented using VOILA (white-box). Notable black-box exceptions are `Output`, which produces the query result, and `JoinBuild`, that builds the hash table of a join, after the inner relation has been materialized (resembling the Morsel-driven parallel hash join with a shared hash table [LBKN14]). Excalibur is an interpreter (VM) that evaluates VOILA plans, while being able to leverage VOILA's flexibility.

8.3.2 Interpretation

Excalibur executes query plans as block-based pull iterators (i.e. vectorized execution [BZN05]), exploiting the fact that VOILA programs can always be executed as vectorized primitives, which provides low-latency efficient interpreted execution as a starting point.

Vectorized Byte Code. The VOILA code is translated into an easily and efficiently interpretable representation (byte code). Our byte code encodes auxiliary operations required to execute (vectorized) VOILA code, while keeping VOILA code mostly encapsulated in fragments. VOILA fragments are invoked via the instruction `CompiledFragment`. The all supported instructions are shown in Table 8.1.

Generating Byte Code. While generating the byte code, we check for fragments that need to be compiled. This could be atomic operations (e.g. adding two columns) or complex fragments (e.g. gathering a multi-column join probe result). For each, we generate a corresponding `CompiledFragment` instruction and trigger compilation.

Listing 8.1: Example JIT-ed vectorized primitive computing $-x$ and $x * y$. Only the selective scalar path is required, the other paths can be omitted to decrease compilation time.

```
void jit_1(PrimArg* arg) {
  int i=0;

  // Deserialize inputs and outputs
  int* sel = arg->sources[0]->first;
  int num = arg->sources[0]->num;
  long* in_val1 = arg->sources[1]->first;
  long* in_val2 = arg->sources[2]->first;
  long* out_val1 = arg->sinks[0]->first;
  long* out_val2 = arg->sinks[1]->first;

  if (ignore_selvector(sel, num, true, 2*64, 2)) { // optional
    // Optional unrolling
    for (; i+16<num; i+=16) { /* ... */ }

    for (; i<num; i++) {
      out_val1[i] = -in_val1[i];
      out_val2[i] = in_val1[i]*in_val2[i];
    }
  } else { // Use selection vector, mandatory
    // Optional unrolling
    for (; i+16<num; i+=16) { /* ... */ }

    for (; i<num; i++) {
      out_val1[sel[i]] = -in_val1[sel[i]];
      out_val2[sel[i]] = in_val1[sel[i]]*in_val2[sel[i]];
} } }
```

8.3.3 Compilation into Vectorized Primitives

VOILA fragments are compiled into machine code with LLVM [LA04], a widely used framework for building compilers. Especially for short-running queries, compilation is quite costly ($10 - 100$ ms).

Caching. Fortunately, compilation can often be omitted be caching frequently used fragments. Especially for simple code fragments (e.g. consisting of 1-2 VOILA operations) this method is quite effective, as small fragments can often be re-used. Re-use can happen inside the same pipeline, query, or across queries. Essentially, this caching mechanism approximates vectorized execution (very simple cached fragment with only one operation = vectorized primitive) while still allowing complex custom-tailored fragments.

Parallel Compilation. Besides reducing compilation time (thanks to caching), code fragments also provide a means to parallelize compilation, even inside a single pipeline. Code fragments are independent pieces of VOILA code that are glued together by the surrounding byte code. Therefore, code fragments can be compiled independently of each other, which allows parallelizing compilation.

Listing 8.2: Ignore selection vector for dense predicates without filtered out tuples, average bits per VOILA node are above a certain limit.

```
bool ignore_selvector(int* sel, int& num, bool can_full_eval,
    double sum_bits, double num_nodes) {
  if (!num && !can_full_eval) return false;

  double score = sum_bits / num_nodes / SCORE_DIVISOR;
  double min_size = (scope * VECTOR_SIZE) / (score + 1.0);
  return num > min_size;
}
```

Compiling Vectorized Primitives. We generate vectorized primitives, functions that operate on columnar chunks of data. Note that data-centric compilation fits into the vectorized model (e.g. Hyper uses morsels [LBKN14], table chunks like vectors, delivered by its vectorized scan operator that decompresses DataBlocks [LMF+16]).

The basic function template is illustrated in Listing 8.1. It iterates over the input predicate (selection vector, which in our system always exists, and then evaluate the VOILA code value-at-a-time. Furthermore, this generic template allows interesting variations:

- We can choose to ignore the predicate. This, however, is not always possible (e.g. for example operations that can raise an error), but can lead to better SIMD performance (Chapter 3). Choosing only requires a quick density check on the selection vector, like illustrated in Listing 8.2.
- Important code paths can be unrolled. This means splitting the loop into the unrolled loop that processes N values (e.g. 16 using SIMD) at once and a residual loop that processes the remainder.
- Code can be annotated to enable/disable SIMD-ization of the code, or define different SIMD widths (e.g. triggering AVX2 instead of AVX-512 to prevent down-clocking on some processors [Kra17, dow22]).

Later, we expand this template for specific flavors such as vectorized and data-centric execution. Note that the performance of these variations is hard to predict. Therefore, Excalibur will choose the best one dynamically at query runtime.

8.3.4 Code Cache

The idea is to *fingerprint* code fragments and look the fingerprint up in the cache. For this cache, however, lookup performance under updates is crucial. Therefore, we use an asynchronous eviction process that does not require write latches during lookups.

Asynchronous Eviction. Instead of replacing during lookups, we have an asynchronous process that cleans up excess fragments in the cache. Therefore, during lookups we just need to update a reference counter and a last-updated timestamp, using atomics. This only requires a shared latch to prevent concurrent updates.

Eviction. Periodically, cleanup is triggered. We mark the N least recently used fragments evictable. When eviction is triggered again and if they have not been touched in between, we safely evict them.

Adapting N. Typically, we aim for a constant cache size ($\leq T$ fragments) with a margin for new fragments (say 10%). Let F be the current number of fragments in the cache. To stay within the bounds, we have to evict $T - F$ fragments. However, we cannot guarantee that our eviction process will clean up $N = T - F$ fragments because they might have been updated/used in between (there is a time lag). Therefore, we measure the number of fragments, we were able to evict, calculate the eviction rate (out of X, we evicted Y) and over-allocate the number of eviction candidates by the corresponding factor ($\frac{X}{Y}$) during the next iteration.

Footprint per Fragment. Ideally, all code fragments are cached and do not incur JIT-overhead. Practically, however, this is not feasible. The important question is, how many fragments can realistically be cached, i.e. what is their memory footprint.

Each code fragment can be compiled in parallel, thus LLVM requires each fragment to use its own instances of `LLVMContext` and `TargetMachine`, an abstraction for hardware-specific details. This led to a memory footprint of roughly 400 kB per fragment, while, for simple fragments, the machine code fits in roughly 1 kB. The extra footprint stems from LLVM which is only needed during compilation. Therefore, after compilation is done, we can safely deallocate LLVM-related objects. This, however, is a non-intended LLVM use-case and requires providing a custom memory manager, which it uses to store compiled machine code. After compilation is done, we dispose the allocated LLVM compilation utilities and just keep the machine code, which is now owned by our memory manager. This pushes the footprint of a code fragment to around 10 kB (40× smaller than the naive implementation). This currently allows roughly 100,000 fragments in 1 GB code cache.

The footprint of cached code fragments could be improved further by sharing pages between multiple fragments. Sharing pages, however, is non-trivial to do in a portable manner because one needs to allocate physical pages, modify page flags (make writable, remove writable flag and make executable), handle concurrency and, of course, find a memory layout that satisfies the requirements of the CPU.

8.4 Code Generation Flavors

Excalibur's rather generic means of query execution allows very different execution flavors. Keep in mind that there are no well-defined guidelines, rather vague rules of thumb, to decide which execution flavor is best to execute a query (see Chapter 7). This makes it impossible to decide the best flavor beforehand because its performance depends on the current environment (hardware at hand, #cores used ...). Therefore, we provide a bouquet of paradigms and choose the best flavor adaptively at runtime. In the following, we describe two different flavors (a) atomic fragments, resembling vectorized execution [BZN05] and (b) fused statements which is similar to data-centric compilation [Neu11].

8.4.1 Atomic Fragments (Vectorized Execution)

Our base (and fallback) flavor is to only compile the smallest possible (indivisible = atomic) fragments. For VOILA operators, this means that such fragments are basic operations in VOILA (e.g. `add`, `bucket_lookup`, `seltrue`). Interestingly, when compiling atomic fragments, the resulting strategy is basically vectorized execution very similar to MonetDB/X100 [BZN05] that became Vectorwise and later Vector. Consequently, this (1) generates many small fragments that can be compiled in parallel and can likely be re-used (2) allows efficient memory access, inherited from vectorized execution and (3) has good chances for micro-adaptive [RBZ13] optimizations like full-execution by ignoring the selection vector. Since this is the default base flavor, it is used whenever we decide to not use any other flavors, which happens for short-running queries, or when the other flavors yield worse performance.

Specialized Implementations for Complex Operations. VOILA has two complex operations: `bucket_insert`, which allocates new buckets in a hash table but can fail, and `selunion`, which `OR`s two predicates together (in the vectorized model concatenates two selection vectors). For these operations, we provide specialized hard-coded implementations.

8.4.2 Fused Statements (Data-Centric)

Data-centric compilation [Neu11] is the extreme of compound primitives (or fused expressions), as it inlines the whole pipeline into a single function. For, a static engine with black-box operators, this inlining process is impossible as operator borders, typically, cannot be crossed. In Excalibur, operators can be *black-box*, i.e. hard-coded with one static implementation like `Output` (delivers query results), or white-box, yielding VOILA code that can be analyzed, modified, inlined etc.

Note that the (performance-wise) most impactful operators (join, group-by, filter, projection) are white-box operators, which allows us to inline them. Hence, the presence of black-box operators breaks the inlining into multiple fragments. From the inlined VOILA code, we can, then, generate data-centric code (Chapter 6).

8.5 (Micro-)Adaptive Execution

We use vectorized execution as our base execution flavor and, during query execution, try to further improve performance by generating different execution flavors and observing whether they improve performance. Note that there is both a choice of execution flavor, and granularity (which parts of the query plan to use it in). The combination of these two choices we call execution *tactic*.

Exploration vs. Exploitation. During execution, we attempt two different things: (a) find the best possible execution tactic (*exploration*) and (b) use the best found tactic to improve runtime (*exploitation*). Consequently, to improve the runtime, we need to spend cycles exploring potentially not very useful tactics with no clear guarantees for success, i.e. a risky bet. In addition, we want to learn good tactics and exploit them as much as possible. Such problems are typically formalized as multi-armed bandits (MAB).

A naive MAB approach would be to explore *all* possible tactics *at least once*, and then exploit the best one. Note that the set of possible tactics is gigantic, especially since combinations of choices (query fragmentation and flavor) get flattened into separate points in the search space (actions in the MAB formalism)[2]. To limit the amount of alternative code fragments that need to be compiled and tested, we focus on *sparsely searching* the design space, followed by exploiting the best point found.

8.5.1 Constraints on Adaptive Execution

Suppose, we want to improve the runtime of a query fragment, and we were given some method to decrease its runtime by 4× (4× speedup, $s = 4$). If this fragment only constitutes 50% of query runtime ($f = 0.5$), the overall expected speedup will

[2]Alternatively, our problem could be modelled as a combinatorical MAB, by skipping the flattening and assuming that combinations of actions behave like the sum of its parts. This is a powerful concept that allows learning, e.g. the shortest paths or rankings. Solution approaches typically require an oracle to predict best actions [LS20], something we do not know a priori. A notable approach is the Follow-the-Perturbed-Leader algorithm [LS20], which introduces additional and complex tuning knobs, like well-chosen distributions for the perturbation (add random noise) to balance exploitation and exploration.

drop to a disappointing 1.6×. Further, suppose that we find this faster fragment not at the beginning of the query, but rather in the middle (at 50% progress, $\phi = 0.5$), then the final speedup will decrease further. In the following, we aim at finding a sweet spot for micro-adaptive optimizations which will guide the choices made by Excalibur.

Amdahl's Law. We model the impact of (adaptive) choices using Amdahl's law [Amd67]. Normally, Amdahl's law is used to compute the speedup of parallelizable computations with a sequential fraction. Here, instead of parallelizing, we just accelerate the previously parallel fraction by a given factor.

We apply Amdahl's law for the progress ϕ: $S = (\phi + \frac{1-\phi}{y})^{-1}$ with y being the speedup at the specific progress (ϕ) in the query. Then, we apply Amdahl's law to determine y based on improving a fraction of the query f: $y = (1 - f + \frac{f}{s})^{-1}$, combining both yields:

$$S = \frac{1}{\phi + (1-\phi)\left(1 - f + \frac{f}{s}\right)} \tag{8.2}$$

From Equation (8.2) we can derive that ideally we have to make good decisions (a) early and (b) on a large portion of the pipeline.

Limits on Exploration. The problem with exploring in constant intervals, as e.g. proposed by Raducanu et al. [RBZ13], is that towards the end of the query, it is still seeking better alternatives (exploring). Even though, their potential benefit cannot yield a significant improvement anymore (because it is found late).

To mitigate this exploration problem, we define a specific exploration budget (30% query runtime). Specifying a budget has two major advantages: (1) it forces most of the exploration to be done at the beginning of the pipeline (greedily) and (2) it limits the negative impact of over-exploring. Using a budget makes adaptivity a favorable asymmetric bet (limited loss, unbounded gain).

When running a query, we estimate the progress of the current pipeline (by tracking the data source). By estimating the progress and measuring the time spent for achieving the progress, we estimate the absolute budget used for exploration (in cycles)[3]: Absolute budget $B = (t + \frac{\phi}{t} * (1-\phi)) * \beta$ with relative budget β (typically $0.3 = 30\%$ of query time), time t and progress $p \in [0, 1]$.

[3]These estimation techniques have previously been used by Kohn et al. [KLN18] and Gubner [Gub14].

Table 8.2: Mutation nodes. A sequence of such nodes allows describing a specific point in the design space.

Mutation	Description
`JitFragm(begin, end, flavorMod)`	Compile fragment between `begin` and `end`, apply given `flavorMod`
`SetScope(begin, end, flavorMod)`	Set `flavorMod` for statements and expressions between `begin` and `end`
`Inline()`	Inline all VOILA operators
`SetDefault(flavorMod)`	Set default `flavorMod` for the whole pipeline
`SetConf(vectorSize, fullEval)`	Set vector size and different full evaluation threshold points
`BloomFilter(op)`	Enable Bloom filter [Blo70, GTLB19] at operator `op`
`SwapOps(a, b)`	Swap operators `a` and `b`

If exploration (and compilation) exceeds this budget, exploration is canceled, and the residual budget is returned. Note, in case the query decelerates (starts running suboptimally), the budget will increase, hence giving opportunity for more exploration. Further, we stop generating new tactics after 40% progress as we do not expect significant overall/net performance gains afterward.

Other Applications. Equation (8.2) has many applications. For example, it can be applied to offloading work to accelerators. When an accelerator improves performance of an operation covering 40% of query performance by 10× and is triggered at the start, the best overall improvement we can possibly expect is a meager 1.5×. If our accelerator improves performance by 100×, all else equal, we can maximally expect a rather disappointing 1.7×.

8.5.2 Exploitation

After the exploration budget is consumed, or the space is fully explored, our adaptive framework switches to exploiting the best points found so far. We choose the point with the lowest cost (CPU cycles per input tuple). However, during exploitation, we still maintain our performance metrics, i.e. if performance of the current best choice degrades, we can still retry the already generated tactics.

8.5.3 Encoding the Design Space

Excalibur allows switching between tactics (i.e. different flavors applied to different fragments). Each tactic is a point in the design space. Here we discuss how Excalibur encodes points in that space.

Mutation Sequences. We define a point in the design space as a sequence of mutations that are created through rules. Currently, we have mutation nodes for

Table 8.3: Rules create and extend mutation sequences.

Rule	Description
`JitBiggestFragment(flavorMod, reqInline)`	JIT compiles the biggest fragment with `flavorMod` & introduces `Inline` before when `reqInline` is *true*
`ReorderFilterBySel` `BloomFilterMostSelJoin`	Modifies plan to order filters by selectivity. Introduces `BloomFilter` into most selective hash join.
`SetScopeFlavor(flavorMod)`	Find most expensive scope, introduces `SetScope`
`SetScopeFlavorSel(flavorMod)`	Like `SetScopeFlavor`, but scope must include VOILA's `SelTrue`, `SelFalse`
`SetScopeFlavorMem(flavorMod)`	Like `SetScopeFlavor`, but scope must include VOILA's `BucketLookup`, `BucketNext`, `BucketScatter`, `BucketGather`
`SetDefaultFlavor(flavorMod)`	Introduces `SetDefault`, if `flavorMod` is not already set.
`SetConfig(vectorSize, fullEval, scoreDiv, simdOpts)`	Introduces `SetConf`, if not already set.

(a) plan changes, (b) local configuration changes, (c) fragment JIT-ting and (d) flavor specifications. The specific nodes are listed in Table 8.2. Additionally, mutations can also have parameters. Most notably, `flavorMod` defines: specific unroll factors and SIMD widths for selective and, similarly, for the non-selective path of the vectorized primitive. Additionally, it allows using predicated execution using techniques described by Crotty et al. [CGK20] or using conditional moves (`cmov`).

For instance, we can choose to combine `SwapOps` and `JitFragm`. `SwapOps` first modifies the plan and, afterward, `JitFragm` would JIT-compile a specific fragment. Full data-centric execution can be expressed using `JitFragm` by selecting the whole pipeline.

Rule-based Generation. During exploration, we extend existing or create new mutation sequences (i.e. extending empty sequence). Table 8.3 shows the rule templates currently used in Excalibur. In practice, we expand the rule templates with common values for flavor and configurations. Rule-based generation provides two advantages: (a) it is easily extensible and (b) we can iteratively expand the design space by applying rules onto a previous mutation sequence. Considering that over time the number of rules will likely grow, the design space will grow exponentially (assuming rules are not mutually exclusive). The rules in Table 8.3 were chosen to provide a minimal set of useful optimizations without inflating the design space too much. Still, materializing this large space is not practical due to compilation time and code-cache space overhead.

Therefore, how we explore the space matters.

8.6 Exploration Strategies

In this section, we present different exploration strategies, reaching from a simple randomized search, to hill-climbing with hard-coded heuristics and Monte Carlo Tree Search (MCTS).

8.6.1 Randomized Exploration (rand)

One could explore the space using random choices, and there are good reasons for this: randomized search is relatively easy to implement, can fully explore the space and can provide a "good" coverage of the space [GPK94]. But for gigantic spaces, randomized exploration might easily get "lost in the space" (i.e. not focus on interesting sub-spaces) and can take *extremely* long until the space is fully explored. This, while the budget of a short running query can only afford running a limited number of tactics.

8.6.2 Hard-Coded Heuristic (heur)

An intuitive approach is to try what database architects believe are good choices. This further makes the assumption that simpler choices, with shorter mutation sequences, are better (similar to Occam's razor). Therefore, we define a list of rules to apply and try:

1. Reorder filters by increasing selectivity
2. Introduce Bloom filter for selective joins
3. Heuristically JIT fragments:
   - If SelTrue/SelFalse and $\sigma < 95\%$ and $\sigma > 5\%$: Do not cross
   - If MemAccess and Cyc/Tup $> N_1$: Do not cross
4. Try fully data-centric
5. Try different vector sizes
6. Give up: Exploit

Clearly, iteratively improving an execution tactic by applying these rules in order reflects its creator's biases and potentially ignores large parts of the search space. Moreover, when new generator rules are added, this approach needs to be maintained (extended and re-evaluated) which is a recurring time-consuming process. Note that the other strategies (randomized and, the following, Monte Carlo Tree Search) do not require hard-coded decisions and, hence, are less influenced by creator's biases and are maintenance-free.

8.6.3 Monte Carlo Tree Search (MCTS)

When rethinking our approach of exploring large spaces, we can draw analogies to Artificial Intelligence (AI) used in complex games (e.g. chess or go). For an AI to make the next choice, it commonly first builds a tree (state/search tree) that represents all possible choices made by players, multiple steps ahead. For complex games, like chess, these trees quickly become very (exponentially) large. One of the relatively new approaches, e.g. used in AlphaGo alongside Neural Networks, is Monte Carlo Tree Search (MCTS) [SHM+16].

Generic MCTS is a randomized approach to search a tree. It starts with exploring parts of the tree and, will, given enough time, eventually have fully explored the tree. MCTS has 4 phases which are repeated, until typically a time limit is reached:

1. *Selection*: A node will be selected using some policy.
2. *Simulation*: Multiple paths from the selected node to the leaves are simulated.
3. *Node Expansion*: The selected node is expanded.
4. *Back Propagation*: The information from the simulation is propagated back towards the root.

The Selection phase has significant impact on which areas of the large tree are explored. Ideally, these should be the most important areas. Suppose, we had some measure of reward, then we could visit areas with the highest reward first (exploitation). This, however, should be balanced with more risky exploration. To balance exploration and exploitation with some measure of reward is a classical MAB problem, with arms corresponding to child nodes. A problem that can be solved optimally using the Upper Confidence Bound algorithm (UCB, Equation (8.1)).

UCB applied to trees (UCT). For trees, or MCTS in specific, there is a variant of UCB, called UCT [KS06]. Until a given point in time, the following variables specific historical metrics collected so far: Let X_i be the empirical mean (of rewards) for child node i, c be some constant, t be the number of samples in the parent node and s be the number of samples of the current node:

$$uct_i = X_i + c * \sqrt{\frac{t}{s}}$$

During the Selection phase, this score is computed for each potential child node, then the child node with the highest score is chosen. This is repeated until a leaf is found. Later, during Back Propagation, we update the metrics (reward and number of samples) for our selected node and all nodes on the path towards the root.

Application to our Exploration Problem. The application of MCTS to our problem, efficiently exploring the design space, is relatively straightforward. Each mutation node becomes a node in the MCTS. We adapt the Simulation and Node Expansion steps: during Simulation, we execute the mutation sequences that result from the path in the tree and collect runtime statistics. Then, we expand an existing node simply by applying our generator rules. In our case, MCTS brings two major advantages: (1) MCTS almost never evaluates the full search space, unless given a huge exploration budget. Instead, it focuses on promising sub-spaces, which is a crucial advantage for exploring large spaces. (2) Assuming the same pipeline is run multiple times, we can extend the existing tree in the new run (i.e. learning). Especially in a cold run, our tree does not yet contain useful information (many UCT scores are ∞ i.e. the highest possible score). In the following, we aim at further improving the order in which we traverse the search space by heuristics for breaking ties between UCT scores.

Propagating Information across Branches. Suppose we already have partial knowledge, e.g. we know that whole-pipeline data-centric works well on this platform. But that knowledge is part of a different branch of the tree, one we have not traversed yet during Selection. Consequently, we would have to re-discover that data-centric execution is beneficial. To ease the burden of re-discovering good choices, we remember rewards and #samples for all mutations (many tree nodes can encode a mutation in different branches, also tree nodes can encode mutation sequences). This allows us to formulate this guessing problem as a MAB, but this time for mutations (and mutation sequences), rather than MCTS tree nodes. This steers the exploration into the direction with the highest confidence (using UCB). In practice, this turns out to work quite well, because on the first level of the tree (i.e. close to the root), we likely discover most possible decisions.

Maximum Distance. If, we are, however, at the very beginning of building the tree, we do not have such knowledge yet. In this case, we try to steer away from already explored nodes (or clusters). Therefore, given a set of already explored siblings (children of the same parent node), we should preferably explore the most dissimilar point next. We define the similarity of two nodes x and y as $1 - d(x, y)$ according to a distance function d. Using d, we select the nodes with the maximal distance to already explored nodes, to break ties. If there are multiple such nodes, we chose randomly.

Gower Distance. Our choice chains (mutation sequences) are rather complex objects, composed of categorical integers, quantitative integers and lists thereof.

Therefore, we use the Gower distance [Gow71], a distance function able to handle complex objects. It is defined as the arithmetic mean of its components K:

$$d(x,y) = \frac{1}{|K|} \sum_{k \in K} d'(x,y,k) \tag{8.3}$$

Each component has slightly different formula (d') depending on its type. We only describe quantitative (d'_q) and categorical components d'_c, other types are defined as well but are not relevant here.

For a quantitative component k and its range r_k, we define

$$d'_q(x,y,k) := \frac{|x_k - y_k|}{r_k}$$

For a categorical component k, we define

$$d'_c(x,y,k) := \begin{cases} 1 & \text{if } x_k \neq y_k \\ 0 & \text{otherwise} \end{cases}$$

Since we only need to measure the distance between sibling nodes in the tree, we can directly apply Equation (8.3) for two nodes x and y. Note that to find the node(s) with maximal distance, we need to compute the pair-wise distances between all siblings. Thus, for trees with many siblings, computing the distance can become costly. The trees we create are typically not very wide (nodes have roughly up to 40 siblings). When extending the mutation nodes and mutation rules over time, trees widen. In this case, we use a random sample of sibling nodes to compute the distances.

8.6.4 Remembering the Past

The proposed exploration strategies have to re-explore the search space before any positive changes can be done, *for each query*. Figure 8-2a illustrates this for TPC-H Q1. One can increase the Risk Budget to explore a bigger part of the space, but consequently, overall query performance suffers as precious CPU cycles are wasted:

In longer-running workloads, we can exploit past knowledge.

While we cannot fully rely on the accuracy of the past (underlying data may have changed, causing different performance), we should not fully disregard past knowledge.

Quick Start - Remembering Good Points. After exploring a pipeline, or query, we can remember the best choices. On the next iteration of the query, we can start checking whether these choices are still the best. While this delays the regular exploration process by a few steps, it directly feeds good points back into the exploration process. We call this Quick Start.

We implemented Quick Start by generating a fingerprint of the pipeline and mapping the fingerprint to the historic data. The historic data contains a mapping from the (design space) point to a histogram of runtimes. Both mappings can grow considerable, if they grew over a certain threshold, we use sampling to determine the surviving data points. Our fingerprints contain operator types as well operator properties (e.g. global aggregation, key join). An improved mapping could also include performance information (e.g. pipeline throughput tuples/cycle, selectivities) or system state (e.g. #threads used). Currently, we use an exact mapping between fingerprint and historical data. But, especially, when integrating performance information into the fingerprint, a best-effort match would be more desirable.

Incremental Monte Carlo Tree Search (MCTS). MCTS has the convenient property that we can continue building the tree with following runs of the same query. Consequentially, MCTS can incrementally learn more about the design space, iteration by iteration. The challenge is to identify the same pipeline, for which we use the same fingerprinting scheme as for Quick Start.

8.7 Experimental Evaluation

In this section, we provide a compact experimental evaluation of the Excalibur VM, which we implemented in C++. For each operator, it first generates VOILA code (Chapter 5, which it then translates into LLVM IR using a particular flavor; and then into machine code on-the-fly (using LLVM's C++ API). The VM supports multiple adaptive decisions (presented in Table 8.2) and a bouquet of exploration strategies (discussed in Section 8.8.6).

Hardware. In small-scale experiments, scale factor 50 and below, we used a dual-socket Intel Xeon Gold 6126 with 24 SMT cores (12 physical cores) and 19.25 MB L3 cache each. The system is equipped with 187 GB of main memory. For large-scale experiments with scale factors ≥ 100, the previous system did not have enough main memory. Therefore, for large-scale experiments, we used an (older) quad-socket Xeon E5-4657L v2 with 96 SMT cores (48 "real" cores) in total, 30 MB L3 cache per chip and a total of 1 TB main memory.

Table 8.4: Excalibur often significantly outperforms other systems optimized for analytics (TPC-H SF50, multi-threaded).

Name	Runtime (ms)			
	Q1	Q3	Q6	Q9
Umbra [NF20]	287 (1.5×)	**326** (0.9×)	91 (1.8×)	854 (1.2×)
DuckDB [dud]	1325 (6.9×)	2338 (6.7×)	341 (6.6×)	15306 (21.0×)
MonetDB [IGN+12]	5488 (28.6×)	1089 (3.1×)	190 (3.7×)	1178 (1.6×)
Excalibur (heur)	**192**	349	**52**	**730**

Table 8.5: Compared to handwritten & optimized implementations, Excalibur's implementation of vectorized & data-centric execution still "leaves room for improvement".

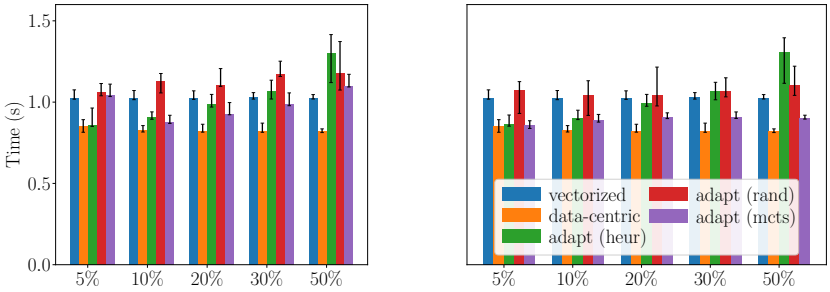
Name	Runtime (ms)			
	Q1	Q3	Q6	Q9
Vectorized Execution				
Tectorwise [KLK+18]	248 (1.0×)	**294** (0.7×)	66 (1.3×)	793 (0.9×)
Excalibur (vec)	225	394	**49**	917
Data-Centric Execution				
Typer [KLK+18]	**137** (0.8×)	437 (0.8×)	73 (1.2×)	1193 (0.9×)
Excalibur (dc)	163	541	61	1337
Overall				
Tectorwise [KLK+18]	248 (1.3×)	**294** (0.8×)	66 (1.3×)	793 (1.1×)
Typer [KLK+18]	**137** (0.7×)	437 (1.3×)	73 (1.4×)	1193 (1.6×)
Excalibur (heur)	192	349	**52**	**730**

Structure. First, we compare Excalibur to other state-of-the-art systems as well as handwritten implementations. Then, we analyze the impact of the Risk Budget onto finding possible improvements and, consequently, performance. Afterward, we compare the different exploration strategies on the TPC-H data set and investigate the adaptation regarding various parameter values in TPC-H Q6. Then we investigate the impact of the code cache and, lastly, show the adaptation over the runtime of a query.

8.7.1 State-of-the-Art Competitors vs. Excalibur

To judge the performance of Excalibur on a relative as well as absolute level, we compare it to state-of-art systems and handwritten implementations. We selected a diverse set of queries: TPC-H Q1, Q3, Q6 and Q9. We ran these queries against the TPC-H data set with scale factor 50 and used all available hardware threads.

Systems. First, we compare to systems optimized for analytical performance. In particular, we chose the well-known state-of-the-art systems. This includes the



(a) *Naive* Exploration: β must be high enough to discover good choices, but low enough to not waste too much time.

(b) With *Quick Start*, we already have a good guess on the best flavor. Thus, we can lower β.

Figure 8-2: Impact of Risk Budget β

open-source systems MonetDB [IGN+12], featuring classical columnar execution, and, the vectorized system, DuckDB [dud]. In addition, we compare to the data-centric system Umbra [NF20] which uses a simple VM-based approach to dynamically switch between different JIT-compiled flavors [KLN21]. The results are summarized in Table 8.4. We can see that Excalibur outperforms the three other systems on most queries, as none of the fixed execution strategies (column-at-a-time, data-centric, vectorized) dominates across all queries and Excalibur adaptively finds a good strategy and the code generated using VOILA has competitive raw performance.

Handwritten Implementations. To further delve in raw performance, we compare with the hand-optimized implementations of state-of-the-art query execution paradigms by Kersten et al. [KLK+18]: Typer, an instance of data-centric compilation [Neu11] and Tectorwise, an implementation of vectorized execution [BZN05]. Both, Typer and Tectorwise, perform roughly on par with the system that pioneered its respective paradigms Hyper and Vectorwise [KLK+18]. Table 8.5 shows the results. Most queries perform roughly on par. However, we noticed that the implementations of Excalibur are slightly slower. Most notably, the data-centric implementation of Q1, where LLVM "optimizes" our data-centric code by, instead of merging branches, replaces them with conditional move instructions (`cmov`).

8.7.2 Impact of Risk Budget

In this experiment, we measure the effect the *Risk Budget*, the budget for adaptive exploration, has on overall performance. We chose a relatively simple query, TPC-H Q1 (on SF10, single-threaded), where the best execution paradigms currently known

are data-centric, or variations thereof. Consequently, our system has to switch to an entirely different execution paradigm, which first has to be discovered. Here, we differentiate between non-learning exploration strategies (*naive* exploration), without knowledge of the past, and learning strategies, able to leverage past knowledge.

Naive Exploration. Figure 8-2a visualizes impact of varying Risk Budgets on overall query performance for non-learning exploration strategies. We can see that there is no clear optimal budget, and it depends on the exploration strategy: We need a minimum budget to be able to discover a reasonably good solution, but using too much is counter-productive. For large search spaces, it is challenging to adapt to the better flavor in time, especially with a low budget.

Learning Exploration. Using Quick Start, we remember good points and, in the next run, explore them early. Figure 8-2b shows that using Quick Start allows lowering the Risk Budget needed to find good points. For example, using the MCTS strategy, it could be lowered to 5% whereas without learning, even with a Risk Budget of 50% we were not likely to discover good points.

8.7.3 Various Scale Factors & Multi-Threading

Analytical queries tend to behave differently with (a) varying data sizes as well as (b) with/without parallelism (i.e. multi-threading). Larger tables significantly impact query performance, e.g. hash tables grow bigger leading to increased memory access cost. Similarly, parallelism also causes different performance characteristics. When running using one core, that core can consume most of the system's memory bandwidth. On the other hand, when using all available cores, memory bandwidth has to be shared between them, which leads to higher memory access cost on each core (more cycles spent on memory accesses/waiting for memory locally). Since, these factors impact performance, it is reasonable to assume they can also impact the best flavor of a query. Therefore, we experiment on different scale factors of the TPC-H data set.

Medium-Scale Multi-Threaded. We start with a multi-threaded experiment on scale factor 50. The resulting runtimes are visualized in Figure 8-3. We see that there is significant performance diversity between the data-centric and vectorized flavors, most notably in Q9 and Q18, but less extreme also in Q1 and Q3. It is visible that Excalibur can adapt to the best flavor, depending on the exploration strategy used. Notably, adaptive strategies can beat static flavors (e.g. on Q9 heuristic beats vectorized and data-centric leading to roughly 2× improvement). Usually, the heuristic strategy (*heur*) behaves best thanks to the relatively small space explored

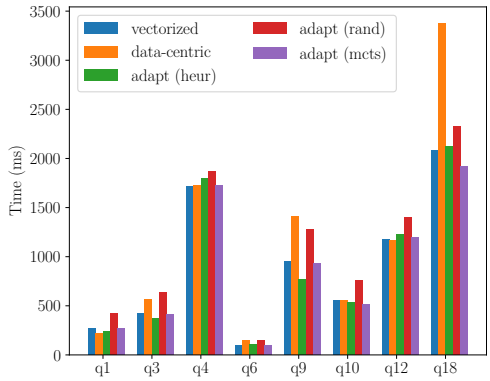


Figure 8-3: On medium-sized data sets, Excalibur can adapt to the best flavors (TPC-H SF50, multi-threaded).

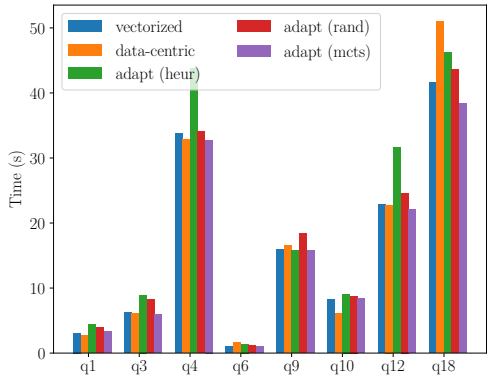
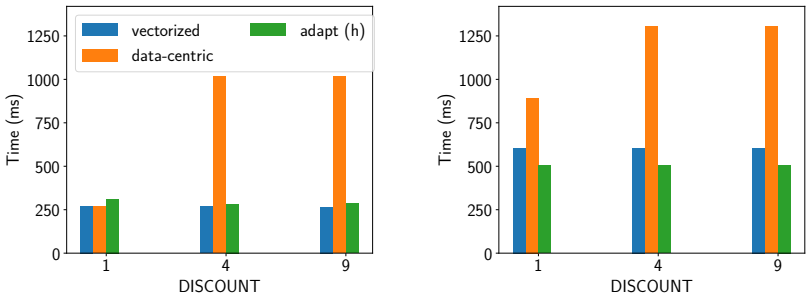


Figure 8-4: On the older hardware platform used for the larger scale, the difference between data-centric and vectorized execution blurs. The extra execution time does allow *mcts* to consistently beat *heur* (TPC-H SF300, multi-threaded).

(certain hard-coded points), but is closely followed by the Monte Carlo Tree Search-based strategy (*mcts*). Less elaborate strategies (i.e. the randomized approach *rand*) tend to perform worse than MCTS, this is due to (a) the learning nature, trees can be extended over multiple runs, and (b) better exploration behavior, as good candidate subtrees more likely to be re-visited. In these multi-threaded experiments, the absolute budget is relatively low (high throughput, queries run quickly), and thus exploration strategies do not have much time to discover good points.

Large-Scale Multi-Threaded. Large data sets, however, give Excalibur more time for exploration, thanks to the higher query runtime. Thus, we ran the same queries on a roughly 6× bigger data set. Figure 8-4 shows the resulting runtimes. The underlying hardware has more main memory and cores, but the older CPU



(a) `YEAR=1999`, `QUANTITY=1` (b) `YEAR=1992`, `QUANTITY=1`

Figure 8-5: Adaptive execution beats static on Q6 with varying parameters. Values for `DISCOUNT` have been multiplied by 100 (i.e. 0.01 becomes 1). (TPC-H SF300, multi-threaded)

means queries have relatively higher runtime and performance of flavors behaves differently. But also here Excalibur adapts to the best flavor. The heuristic exploration is now consistently outperformed by the MCTS strategy: thanks to significantly higher query runtime, the absolute risk budget is higher (proportional to query runtime) and allows exploring more points in the design space.

8.7.4 Adaptation to Varying Query Parameters

Especially in real-world queries, cardinalities are extremely challenging to predict and frequently *wrong by orders of magnitude* [LGM+15]. One possible solution is to use real-life observed metrics to optimize the query at runtime (i.e. adaptivity) by e.g. re-ordering filters. This is (1) challenging for JIT-compiling systems, as it would require expensive re-compilation and (2) affects the best flavor. Therefore, we experiment how our JIT-compiling system Excalibur adapts to changing selectivities. In particular, we evaluate TPC-H Q6 with different parameters:

```
SELECT SUM(l_extendedprice*l_discount) AS revenue FROM lineitem
WHERE  l_shipdate >= DATE '[DATE]' AND l_quantity < [QUANTITY]
  AND  l_shipdate < DATE '[DATE]' + INTERVAL '1' YEAR
  AND  l_discount BETWEEN [DISCOUNT] - 0.01 AND  [DISCOUNT] + 0.01
```

Different parameter choices consequently lead to different selectivities in each of the `WHERE` clauses. For simplicity, we chose the `DATE` to start `01-01` (January 1st) in a specific year and from hereon only specify the `YEAR`. Figure 8-5 shows our results. We observe that full vectorized execution often is the best tactic, beating full data-centric execution. However, adaptive execution using the heuristic search strategy can identify this: it generally is on par with the best static tactic, and in Figure 8-5b even beats it.

Table 8.6: Runtime of smaller scale factors is significantly affected by compilation latency, which can be eased using a code cache or using parallelism. Impact of code cache on query runtime for TPC-H SF0.1 without adaptive execution.

Cache Size (#fragments)	Runtime (s)					
	1 Thread			8 Threads		
	Q1	Q9	Q18	Q1	Q9	Q18
0	29.1	54.6	59.0	5.1 (6×)	10.6 (5×)	11.2 (5×)
8	13.9 (2×)	29.6 (2×)	28.8 (2×)	2.9 (10×)	6.4 (9×)	7.7 (8×)
16	11.1 (3×)	25.9 (2×)	25.5 (2×)	2.6 (11×)	6.7 (8×)	6.0 (10×)
32	4.5 (6×)	19.3 (3×)	19.1 (3×)	1.8 (16×)	5.3 (10×)	4.8 (12×)
64	1.1 (27×)	6.0 (9×)	6.0 (10×)	0.4 (69×)	2.1 (26×)	2.3 (25×)
128	1.1 (26×)	1.9 (28×)	2.0 (30×)	0.4 (68×)	0.8 (68×)	0.9 (68×)
1024	1.1 (26×)	2.0 (28×)	2.0 (30×)	0.4 (72×)	0.8 (68×)	0.8 (74×)
16384	1.1 (26×)	2.0 (28×)	2.0 (30×)	0.4 (68×)	0.8 (68×)	0.8 (73×)

8.7.5 Code Cache

For short-running queries, compilation latency tends to be a major bottleneck. In our model, many fragments can be cached, thus reducing compilation latency. In the following, we investigate the impact of the code cache's size on the query runtime. This size refers to the number of fragments stored. 0 refers to the code cache being disabled. Table 8.6 shows the results.

General Observations. We observed that with increasing the size of the code cache, query runtime improves. For simple queries, like Q1, with 32 fragments cached, we can improve the runtime by 6×. The plateau is reached at about 64 cached fragments, where the runtime is roughly 26× faster than without the code cache. More complex queries, like Q18, contain more code fragments and, therefore, require larger code caches. In case of Q18, a code cache size of around 128 fragments captures all code fragments at that point, runtime is roughly 30× faster than without the code cache.

Multi-Threaded Compilation. If we compare single-threaded execution (mostly compilation time on SF0.1) against multi-threaded, we see that compilation time improves significantly $(5 - 6\times)$. Unfortunately, compilation does not seem to scale linearly. When compiling code fragments concurrently, we do not introduce any additional locking (besides checking the code cache, reserving an entry in the cache and the execution pipeline waiting until compilation is complete). Thus, we suspect the additional overhead must come from LLVM.

High Compilation Time without Code Cache. It can be seen that without code cache (size 0), the initial compilation time is extremely high as all vector-

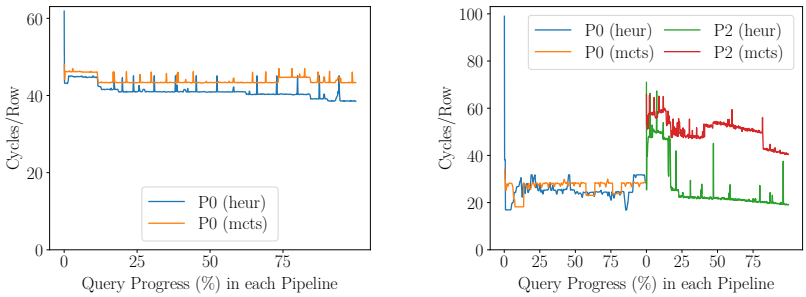
ized primitives (code fragments) need to be generated. This has multiple reasons: Machine Code generation in Excalibur is not optimized: We generate relatively straightforward LLVM IR from VOILA and rely on compiler optimizations (like auto-vectorization/SIMD-zation, and others commonly included in `-O3`) afterward. This can, of course, be improved. For example, we could directly emit SIMD-ized code, i.e. skipping compiler auto-vectorization. We consider optimizing query compilation latency a very relevant, yet rather orthogonal challenge for our research into the possibility and benefits of an adaptive fine-grained runtime exploration of the execution strategy search space. Thus, we expect low-latency JIT query compilation techniques pioneered in other code-generating systems, most notably Hyper [Neu11] and Umbra [NF20] (both regrettably not in open source), to be beneficial for Excalibur; e.g. better register allocation [KLN18], directly emitting assembly [KLN21], avoiding certain combinations of optimizations [Neu11]. Improvements in compilation latency will be even more impactful in Excalibur than Hyper and Umbra, as we generate multiple alternative code fragments during execution. This currently also leads to additional setup and tear-down costs (`LLVMContext` and `TargetMachine`).

8.7.6 Adaptation over Query Runtime

Over the runtime of a query, Excalibur tries to find better execution tactics. To highlight this adaptive behavior, we visualize the execution traces as measured by Excalibur without Quick Start. If a pipeline has less than 10 samples, we omit it in the plot. Note that the x-axis shows query progress rather than the absolute time, consequently all pipelines have the same length (from 0% to 100%).

Q1. The results for Q1 are visualized in Figure 8-6a. Both exploration strategies, heuristic and MCTS, start at around 60 cycles/row, and from there on quickly choose better flavors. Unsurprisingly, we can see that with the hard-coded heuristic (*heur*) strategy more quickly find a faster flavor. The reasons are that (a) the heuristic is heavily biased and (b) only explores a rather small space. The other strategy (*mcts*) explores a significantly larger space and is, in this time frame, unable to find a flavor faster than 45 cycles/row.

Q12. The results for Q12 are visualized in Figure 8-6b. Again, the heuristic tends to outperform. In the first pipeline $P0$, it finds a faster flavor (30 vs. 100 cycles/row initially). But it is closely tracked by *mcts*. In the second pipeline in the plot ($P2$), the heuristic outperforms by a wider margin (20 vs. 70 cycles/row initially). Also here, *mcts* tends to find better flavors (40 cycles/row) but is, due to the large search space, unlikely to find the winning flavor in the time frame.



(a) Q1 (b) Q12

Figure 8-6: Over time, different flavors are tried and adapted. Pi denotes pipeline i. (TPC-H SF50, single-threaded).

8.8 Conclusion

This chapter discussed how a virtual machine (VM) can *automatically* tailor the implementation of a query to the underlying hardware. Most notably, this VM did (micro-)adaptively find "good" implementations while running the query and, thus, *did not rely on complex and fragile cost models.*

System Architecture. Essentially, we built a vectorized engine prototype (Excalibur) augmented with white-box operators, where each operator is implemented in VOILA (see Chapter 5). Queries are executed pipeline per pipeline. For each pipeline, Excalibur generates all code fragments ("primitives") required for vectorized execution. Afterward, it executes the pipeline, first in a vectorized fashion and later with different execution styles (e.g. data-centric [Neu11]).

To generate the different styles, Excalibur features a flexible code generator emitting LLVM IR, which LLVM turns into optimized machine code. Since code generation is rather time-consuming, Excalibur caches frequently used code fragments.

Exploration Strategy. Typically, micro-adaptive approaches rely on randomized exploration of the design space to discover "good" points to be exploited. Since the possible design space is large for non-trivial queries (and gigantic for complex queries) such a VM requires a more efficient exploration strategy. The most naive way is to implement a heuristic by hand, which arguably will be biased towards specific points. Thus, this hard-coded heuristic will only late, or not at all, visit unfavorable points, no matter whether their performance is good or bad.

Monte Carlo Tree Search (MCTS). Therefore, we proposed to adapt Monte Carlo Tree Search to this problem because MCTS (a) guarantees full exploration of

the space (given unlimited time), (b) allows to abort exploration at any time and (c) features adaptive exploration (branches with better points are preferred).

With limited runtime (i.e. not extremely long-running queries), there are usually not enough iterations through the MCTS to provide enough information for adaptivity to trigger (i.e. lack of samples). Thus, MCTS explores, in an arguably undefined order. In that case, we propose heuristics to gather more information:

- We propagate information across branches. Often similar choices reflect similar performance (arguably, biased). Therefore, we try to observe how basic choices perform and prefer applying better-performing choices.
- We use the distance between different points to determine the next points to be explored, i.e. we look for points with the highest distance.

Still, for the more complex queries we evaluated, the search space was not fully explored. This was mainly due to a lack of time spent on exploration. Thus, we propose to utilize knowledge gained from past runs of the same query (i.e. same plan of the pipeline). In particular, we propose to (a) remember the best point found in the previous runs (Quick Start) and (b) extend the MCTS over multiple runs of the query (i.e. Learning).

CHAPTER 9

Conclusion & Future Work

> Hindsight is not only clearer than perception-in-the-moment but also unfair to those who actually lived through the moment.
>
> Edwin S. Shneidman

9.1 Contributions

Let us briefly recap the research questions defined in Section 1.1.1:

1. How far can query plans be optimized to the specific instance?
2. How can query engines exploit increasingly heterogeneous modern hardware?

The contributions to answer these questions are summarized below.

9.1.1 Exploring the Design Space of Q1

In Chapter 3, we explored the design space of TPC-H Q1. We asked: How would a human implement Q1, knowing data distributions, cardinalities, ...? As an answer, we explored the design space and developed two techniques: Compact Data Types and In-Register Aggregation.

Compact Data Types. Since the value ranges of columns are limited, we can shrink data types (e.g. shrink 64-bit to 16-bit integer) to tightly represent data, rather than using the user-supplied schema. This improves the performance of arithmetic, due to better exploitation of SIMD (4× smaller data type fits 4× more data into the same SIMD register).

In-Register Aggregation. The number of groups is rather low (4). For low numbers of groups, we developed a special group-by technique that re-orders the data on-the-fly such that aggregates can be computed in an ordered fashion (ordered by group, within a vector/table chunk). The re-ordering (partial shuffle) is not extremely cheap. However, the improvements gained from the more efficient aggregation, which now happens in-register (no repeated loads and stores to memory), more than compensates for the overhead introduced by re-ordering (in case of Q1).

Summary. Both techniques are one step further towards instance-specific optimizations, while still being general enough to apply to a wider class of queries. Unfortunately, time has shown that they were (and are) not perfectly tailored to the query and, thus, did not deliver optimal performance: Later Nowakiewicz et al. [NBH+18] found that the performance of group-by/aggregation can be further improved via JIT compilation and exploiting the very low number of groups). Alternatively, the runtime can be improved further, when execution can be offloaded to the GPU (i.e. CPU and GPU work *together* on the query) [TGR+18].

9.1.2 Compressing Hash Tables & Strings

In Chapter 4, we explored fast and efficient methods for compressing hash tables to speed up query processing. We proposed Prefix-Guided Null Suppression, Optimistic Splitting and the Unique Strings Self-aligned Region (USSR).

Prefix-Guided Null Suppression. Compressing hash tables requires a compression method that allows fast random access to specific rows. Prefix-Guided Null Suppression is one possible method. To allow fast access, it (a) compresses rows separately, but with one common layout and (b) each value (attribute) is bit-packed

(after normalization[1]). Notably, key checks do not require decompression (keys can be compressed into the same format) and can be implemented using a few integer comparisons (e.g. multiple 64-bit integers could be compared with a single 64-bit comparison). The compression rate depends on the data distribution (ranges) of the inputs, on TPC-H, our method has shown significant gains in memory footprint.

Optimistic Splitting. Often, code paths are optimized for the worst case. For example, if we compute the `SUM` of 64-bit integer, we can just naively add each 64-bit integer to a 128-bit accumulator[2]. Assuming the worst-case can be costly, then it can make sense to optimize for the *average case* instead, and provide potentially pricey handling for the worst case. For example, if we sum many 64-bit integers, we can either assume a final data type of 128-bit (naive method), or build partial sums in 64-bit integers (average case) and flush them when needed (on under/overflow, exception). Building partial sums tries to capitalize on the hope that values *mostly* fit into some range and, thus, under/overflows will only occur infrequently. This is the basic idea behind Optimistic Splitting, which proposes splitting the code paths for (a) average case and (b) exception. We explored Optimistic Splitting for aggregates (`MIN`, `MAX`, `COUNT`, `SUM` and `AVG`) and USSR-encoded strings. Optimistic Splitting can significantly improve the size of the working set, by storing the average case inside the hash table and the exception outside.

However, we believe this technique is more general and can be applied more widely. The challenge is finding the average case and, then, optimizing for that case (Optimize for 90% of values? What distribution do values have?).

Unique Strings Self-aligned Region (USSR). Strings are ubiquitous in real-world data sets [VHF+18]. However, string operations tend to be slow as they require multiple instructions and typically also require pointer chasing. Often, strings repeat and are, thus, stored dictionary-compressed, as a code that identifies the string's position in the dictionary. Comparing dictionary codes instead of the full string is significantly cheaper (no pointer chasing, fewer instructions). However, this has so far been only possible with global dictionaries, which comes with significant disadvantages (update-ability, synchronization for multi node systems/clusters). Unique Strings Self-aligned Region (USSR) is a query-wide string dictionary that is created while the query is running (*on-the-fly*) and encodes frequent strings. Consequently, this reduces memory footprint and certain string operations, such as equality check or computing the hash. Equality checks within the

[1] Normalization subtracts the domain minimum to (a) avoid negative integers (highest bits are 1s) and domains with offsets (e.g. years starting from 1900 AD instead of 200 BC.).

[2] Using a 128-bit integer avoids overflow detection per row, but comes with expensive additions.

USSR "dictionary codes" can be compared directly and hash computations will just return a pre-computed hash, stored in the USSR (hash has to be computed upon insertion).

Summary. All three techniques can be combined such that (a) hash table's memory footprint can be lowered and (b) query runtime improved. All three exploit data distributions: Null Suppression relies on the range of values, Optimistic Splitting on a frequency of certain sub-ranges and the USSR requires a relatively low amount of unique strings. The applicability of these techniques depends on the query and data distributions.

9.1.3 VOILA & Synthesis from VOILA

In Chapters 5 and 6, we describe how a domain-specific language (DSL) can be used to abstract hardware- and implementation-specific details and, later, synthesize, such details via code generation.

Variable Operator Implementation LAnguage (VOILA). In Chapter 6, we explore the idea of abstracting implementation details using domain-specific languages. The challenge is that most existing DSLs are not well-suited for that purpose (abstract implementation details to synthesize them later). We presented Variable Operator Implementation LAnguage (VOILA), as one possible solution. Furthermore, we formally defined the semantics and showed how widely used relational operators can be implemented in VOILA.

Synthesis from VOILA. Chapter 6 shows different examples of how efficient implementations can be generated from VOILA. We focus on two different classes of code generators: Flavor-specific "direct" back-ends and the more flexible back-end, FUJI.

Direct Back-ends for Data-Centric and Vectorized Execution. The well-known query execution paradigms Data-Centric Execution [Neu11] and Vectorized Execution [BZN05] can relatively easily be generated from VOILA code. However, they each require a slightly different back-end implementation. Therefore, for exploring the design space of specific queries, direct back-ends are unlikely to be the optimal choice. Arguably, still more efficient than starting from scratch because the surrounding VOILA infrastructure, including queries and testing, can be reused.

Flexible Unified JIT Infrastructure (FUJI). For design space exploration, direct back-ends require quite excessive manual labor (i.e. crafting one back-end per

flavor). Therefore, the next logical step is a more flexible code generation: Flexible Unified JIT Infrastructure (FUJI). FUJI allows generating many flavors and mixes thereof from *one* program in VOILA. We aim for a more automated design-space exploration. FUJI takes the first steps by providing very flexible code generation, while the VOILA infrastructure allows benchmarking multiple generated points.

Summary. With VOILA we presented a DSL able to abstract implementation and hardware-specific details. The aspect of abstraction is useful for forward compatibility with future hardware features and architectures (requires back-ends to synthesize these features later on). For example, Data-Centric [Neu11] and Vectorized Execution [BZN05] exhibit different performance characteristics [KLK+18]. With VOILA, we have one DSL able to generate both very different flavors. Each back-end explores specific points of the low-level design space (implementation details).

9.1.4 Performance Diversity

In Chapter 7, we investigate how certain flavors behave on various hardware architectures.

Performance Diversity. We found that there is significant performance diversity. The best flavor is not necessarily the one that the study by Kersten et al. [KLK+18] would predict. Most notably, Data-Centric Execution was able to (sometimes) outperform Vectorized Execution, depending on the hardware environment. Certain hardware features appeared to favor Data-Centric Execution for join queries, i.e. (a) large caches, (b) high levels of SMT (simultaneous multi-threading) and (c) slow(er) cores. In summary, the best flavor tends to depend on the whole system consisting of query structure, data distribution and hardware environment. Importantly, the hardware environment is often a factor that is out of our control.

ARM became Competitive. In addition, we tested on ARM-based systems and noticed that they can outperform the high-end X86-based systems (Graviton 2 outperformed all other systems in the queries we tested, on query runtime).

9.1.5 Excalibur

With Chapter 8, we go to the next logical step from VOILA: Create a prototype that automatically explores the design space, more efficiently and at runtime.

Framework. Excalibur is a prototype that exploits instance-specific optimizations, automatically and while the query is running (*on-the-fly*). It mixes many optimiza-

tions from reordering filters, Bloom filters to different implementation flavors from VOILA (Vectorized [BZN05] interpreted execution, Data-Centric [Neu11] to different vectorized primitives). The transformations are encoded as a chain of choices, which are built via Exploration Strategies, that add certain choices (optimizations).

Excalibur is open-source and provides an interface for scans and can, thus, be used as a highly efficient *embedded database engine*.

Exploration Strategies. The challenge with the exploration in the VOILA framework was its in-efficiency. VOILA basically tried to sample the space, making random choices. While random sampling is an unbiased strategy, finding "good" points will take rather long (assuming non-trivial queries). Especially, when trying to further improve query runtime on-the-fly random sampling will find beneficial points, typically, too late[3], if at all. It is a reasonable assumption that other "good" points can probably be found in the neighborhood of already discovered "good" points.

Monte Carlo Tree Search (MCTS). We adapted Monte Carlo Tree Search (MCTS) to our exploration problem (exploration choices form a tree). MCTS has two major advantages: (a) given enough time, it will explore the whole space and (b) beneficial branches will be explored more heavily. Further, we augmented the choice of unexplored nodes (in the tree) based on distance (of choice chains) and similarly with other known choices (maximum dissimilarity).

Learning. Still, often exploration still requires excessive time. Simply, more efficient exploration is not enough. Fortunately, it is possible to incrementally extend the MCTS over multiple runs of the same (similar) query (based on the plan of the current pipeline). Additionally, we proposed to remember the last "good" points per query, such that the following runs can start with the currently best flavor.

Summary. By design, Excalibur can generate different code paths, via VOILA, and choose the currently best one (if found). Beyond that, Excalibur can apply higher level optimizations (reorder filters, enable Bloom filters) in a code-generating system. Given the right set of optimizations, Excalibur can automatically optimize to the specific instance.

To exploit heterogeneous hardware, optimization steps could offload to accelerators (GPU, FPGA), optimize VOILA code for new instructions, prefetching etc. Excalibur would then try, whether these optimizations are beneficial.

[3]Possible runtime improvements are easily ruined by over-exploring (compilation and runtime overhead). Furthermore, to achieve good improvements, a good flavor has to be found *as early as possible*. The further the query progresses, the less the improvement becomes.

9.2 Reflections & Future Work

In this final section, we reflect upon the important challenges tackled in this thesis and highlight ideas for future work. We start with the automated discovery (of good implementations), followed by exploration strategies and offloading to heterogeneous hardware. Finally, we give a final reflection on the results of this thesis.

9.2.1 Seeking the Holy Grail — Automated Discovery

With the advantage of hindsight, it appears that trying to optimize for the specific instance is like a never-ending search, often yielding disappointing results.

While VOILA simplifies that process, it can only synthesize specific patterns (i.e. makes it hard to impossible to explore *all* possible points). So far, only Supercompilation [Tur86] can do that, but is practically unfeasible as it requires too many mutation steps to even get close to known paradigms.

The best gains (improvement in runtime), we have found using human ingenuity (Chapters 3 and 4). Still, we applied already known patterns (use less space to represent data, re-order data on-the-fly, use data-parallelism whenever possible, and simplify the representation of complex data types).

We believe that this is rather a matter of finding the right abstractions. For future work, the question becomes whether VOILA uses the right patterns or whether there are "atomic" patterns able to generate VOILA's patterns.

9.2.2 Exploration Strategies

With the VOILA framework and Excalibur, we highlighted that it is more efficient to explore the space using combinations of fixed patterns, compared to "brute-force" methods like Supercompilation [Tur86] or manual development.

(More) Practical Exploration. While the ability to synthesize different implementations does allow exploring the design space (e.g. by random sampling or exhaustive search), it is not very practical as "good" points are possibly explored too late. For example, in an adaptive VM (like Excalibur) finding a better implementation too late means query execution, in the current run, will not improve. In Chapter 8, we explored using heuristics and an adapted Monte Carlo Tree Search (MCTS). Heuristics require human ingenuity to bias exploration towards good points. Therefore, using heuristics is not a generic approach, as it cannot find certain points too late (if exhaustive search is used) or at all (if we

prune the space). MCTS, however, will guarantee to explore the whole space (if given enough time) and tends towards extending beneficial branches of the search tree. Still, there is no guarantee that good implementations are combinations of good choices.

Perhaps, exploration strategies should be rethought: What we want is to find a path through the design space starting with the highest chances of success. However, quantifying the likeliness will be hard without knowledge (from random samples, or from similar points).

One could also interpret the adaptive system as a sequence of question-answer pairs and, somehow, try to minimize the number of questions (Active Learning). However, the optimal answer might still require a number of questions proportional to the size of the space, as we do not know (yet?) any rules to efficiently skip points (if there was an order, we could use bisection or interpolation search).

Hybrid - Pruning using Cost Models. We assumed that an adaptive (online learning) approach would be the "best" because it avoids creating and maintaining very specific cost models (not only hardware- but also environment-specific). The problem is that the adaptive approach requires relatively smart exploration (which is biased towards good points) due to the lack of exploration time. There is a point to be made for cost models. Since we still do not know much about the design space, it seems possible that cost models can perform reasonably well. This can be either for predicting good points in the design space, or to prune points unlikely to be good. It seems reasonable that a basic machine model (e.g. for CPU and memory) can be created, which can then simulate many flavors and disqualify flavors with excessive cost. If the simulation on the cost model is faster than the real run (likely the case, given a simple model), pruning would be very efficient, as it could quickly eliminate large sub-spaces.

The Non-Stationary Case. In Chapter 8, we assumed that there exists *one* optimal point that never changes during the execution of the pipeline (stationary). However, this assumption is not always true in practice. For example, selectivities can change during execution of the pipeline (e.g. join has 10% during the first half and 90% selectivity during the second half). Therefore, the optimal point can change as well.

Handling non-stationary cases, typically, requires periodically re-exploring points to check whether the situation changed, and then trying to find the new optimal point. Obviously, re-exploration will incur some overhead. In an adaptive VM like

Excalibur, it can become rather costly, if the required code fragments are not cached (triggering compilation).

9.2.3 Offloading to Heterogeneous Hardware

One aspect of the upcoming modern hardware that has not received much attention in this thesis is: *Can VOILA generate code for heterogeneous hardware, and can Excalibur efficiently offload (and dynamically) offload to it?*

In theory, the building blocks should be there. Morsel-driven parallelism allows offloading pipelines to accelerators [GTLB19]. Since VOILA supports Morsel-driven parallelism (Section 5.5.5), it should be possible to implement VOILA back-ends for CUDA or OpenCL and integrate them into Excalibur. This would allow to dynamically offload parts of a query to accelerator devices (and decide automatically whether it makes sense to do so). Finding the optimal flavor will introduce additional code generation effort to do the fine-tuning [TGR+18] as well as finding the optimal number of streams (from CPU to GPU, and back).

But, with these parts in place, Excalibur could become the go-to system for future heterogeneous hardware. New back-ends can be added, and the adaptive VM would automatically handle the offloading.

9.2.4 Final Reflections

We set out to find a better implementation of a query, a point in the design space, given a specific setup and data (instance-specific optimization)

We found some better points in the design space. Especially, for TPC-H Q1 and similar queries, these points yielded quite significant improvements (Chapter 3). But also for queries with strings, hash joins and aggregations we were able to reduce the memory footprint and significantly improve query runtimes (Chapter 4). The methods from both chapters are relatively widely applicable and can relatively easily be integrated into existing (vectorized) systems.

Still, the question, of how to find better points efficiently and exploit them, remains only partially solved. VOILA (Chapters 5 and 6) attempted to explore the space, by abstracting and re-synthesizing execution-specific details. Unfortunately, the design space seemed more disappointing than originally hoped, and new points, with significant improvements, tend to be rather rare (Chapter 6). But there is the possibility that we simply looked in the wrong corner of design space, or the problem itself might rely more on human ingenuity than we thought.

Excalibur (Chapter 8) improved upon the VOILA framework by (a) its ability to exploit the design space efficiently via adaptive JIT compilation and (b) improved exploration methods (heuristic and an augmented Monte Carlo Tree Search). Excalibur is a rather generic prototype that can be used for further research on new exploration methods or ways to generate code. The personal lesson learned from Excalibur is that neither cost-based nor online learning (adaptive) exploration seemed to be the optimum (good cost models are difficult to create and maintain; adaptivity requires numerous trials). Thus, with a bit of luck, there might be a better method that we just have not discovered yet.

Bibliography

[ABH+13] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[ACBF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[Alc20] Paul Alcorn. Zhaoxin KaiXian x86 CPU Tested: The Rise of China's Chips. `https://www.tomshardware.com/features/zhaoxin-kx-u6780a-x86-cpu-tested`, 2020. Accessed: 2022-02-11.

[Ama19] Amazon. Announcing New Amazon EC2 M6g, C6g, and R6g Instances Powered by Next-Generation Arm-based AWS Graviton2 Processors. `https://aws.amazon.com/about-aws/whats-new/2019/12/announcing-new-amazon-ec2-m6g-c6g-and-r6g-instances-powered-by-next-generation-arm-based-aws-graviton2-processors/?nc1=h_ls`, 2019. Accessed: 2024-08-20.

[Ama20] Amazon. Announcing new Amazon EC2 M6gd, C6gd, and R6gd instances powered by AWS Graviton2 processors. `https://aws.amazon.com/de/about-aws/whats-new/2020/07/announcing-new-amazon-ec2-instances-powered-aws-graviton2-processors/`, 2020. Accessed: 2024-08-20.

[Ama21a] Amazon. `https://aws.amazon.com/en/ec2/graviton/`, 2021. Accessed: 2021-03-02.

[Ama21b] Amazon. `https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deep_dive_on_Arm-based_EC2_instances_powered_by_AWS_Graviton_CMP322-R1.pdf`, 2021. Accessed: 2021-03-02.

[Ama21c] Amazon. `https://aws.amazon.com/de/ec2/spot/pricing/`, 2021. Accessed: 2021-03-19.

[Ama23] Amazon. https://aws.amazon.com/de/blogs/aws/new-graviton3-based-general-purpose-m7g-and-memory-optimized-r7g-amazon-ec2-instances/, 2023. Accessed: 2023-10-29.

[Amd67] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS '67 (Spring), page 483–485, 1967.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *SIGMOD*, pages 671–682, 2006.

[and20] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode, 2020. Accessed: 2022-02-11.

[App21] Apple. https://www.apple.com/mac/m1/, 2021. Accessed: 2021-03-02.

[aws21a] https://www.forbes.com/sites/moorinsights/2019/12/03/aws-goes-all-in-on-arm-based-graviton2-processors-with-ec2-6th-gen-instances/, 2021. Accessed: 2021-03-02.

[aws21b] https://en.wikichip.org/wiki/annapurna_labs/alpine/al73400, 2021. Accessed: 2021-03-02.

[aws21c] https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd, 2021. Accessed: 2021-03-02.

[BAK17] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *TPCTC*, pages 103–119, 2017.

[BBF+10] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 708–725, 2010.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

[BK99] Peter A. Boncz and Martin L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB Journal*, 8(2):101–119, 1999.

[BKF+18] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating custom code for efficient query execution on heterogeneous processors. *VLDB Journal*, 27(6):797–822, 2018.

[BKM08] Peter Boncz, Martin Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008.

[Ble90] Guy E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, 1990.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[BLP+14] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. Memory-efficient hash joins. *PVLDB*, 8(4):353–364, 2014.

[BNE13] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76, 2013.

[Bon02] Peter Boncz. *Monet: A next-generation DBMS kernel for query-intensive applications*. Universiteit van Amsterdam, 2002.

[BTAÖ13] 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*, pages 362–373, 2013.

[BZN05] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[CAB+81] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. A History and Evaluation of System R. *Commun. ACM*, pages 632–646, 1981.

[CG12] Alain Crolotte and Ahmad Ghazal. Introducing Skew into the TPC-H Benchmark. In *TPCTC*, pages 137–145, 2012.

[CGK20] Andrew Crotty, Alex Galakatos, and Tim Kraska. Getting swole: Generating access-aware code with predicate pullups. In *ICDE*, pages 1273–1284, 2020.

[Cod83] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69, 1983.

[dbl] `https://github.com/epfldata/dblab`.

[DG16] Johan De Gelas. `https://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores`, 2016. Accessed August 20, 2024.

[dow22] `https://stackoverflow.com/a/56861355`, 2022. Accessed: 2022-02-16.

[dud] `https://duckdb.org/`.

[EHZH+22] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. Benchmarking the Second Generation of Intel SGX Hardware. In *DaMoN*, pages 1–8, 2022.

[EPI24] EPI. Accelerator Processor Stream. `https://www.european-processor-initiative.eu/accelerator/`, 2024. Accessed: 2024-08-26.

[euR24] `https://www.european-processor-initiative.eu/`, 2024. Accessed: 2024-08-20.

[FCP+12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: Data management for modern business applications. *SIGMOD Record*, pages 45–51, 2012.

[Feg16] Leonidas Fegaras. An Algebra for Distributed Big Data Analytics. 2016.

[FM95] Leonidas Fegaras and David Maier. Towards an Effective Calculus for Object Query Languages. SIGMOD, pages 47–58, 1995.

[Fru14] Andrei Frumusanu. A Closer Look at Android RunTime (ART) in Android L. `https://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/`, 2014. Accessed: 2022-02-11.

[FW88] Alex Ferguson and Philip Wadler. When will deforestation stop. In *Proc. of 1988 Glasgow Workshop on Functional Programming*, pages 39–56, 1988.

[FZW] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. `https://github.com/fzhedu/db-imv/blob/master/src/imv/engine.cpp`, line 226. Accessed June 10, 2020.

[FZW19] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. Interleaved Multi-vectorizing. *PVLDB*, 13(3):226–238, 2019.

[GBE+23] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. Bringing compiling databases to risc architectures. *PVLDB*, 16(6):1222–1234, 2023.

[GES+09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6):465–478, 2009.

[gfn20] What are the AVX-512 Galois-field-related instructions for? `https://stackoverflow.com/questions/59124720/what-are-the-avx-512-galois-field-related-instructions-for`, 2020. Accessed: 2023-08-05.

[Gör13] Steve Göring. Effiziente In-Memory Verarbeitung von SPARQL-Anfragen auf großen Datenmengen. Master's thesis, Technical University of Ilmenau, 2013.

[Gow71] John C Gower. A general coefficient of similarity and some of its properties. *Biometrics*, pages 857–871, 1971.

[GPF06] Andreas Gal, Christian W Probst, and Michael Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, 2006.

[GPK94] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Fast, Randomized Join-Order Selection - Why Use Transformations? In *VLDB'94*, pages 85–95, 1994.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[Gra94] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.

[Gru04] Torsten Grust. Monad comprehensions: a versatile representation for queries. In *The Functional Approach to Data Management*, pages 288–311. Springer, 2004.

[GTLB19] Tim Gubner, Diego Tomé, Harald Lang, and Peter Boncz. Fluid Co-processing: GPU Bloom-filters for CPU Joins. In *DaMoN*, pages 9:1–9:10, 2019.

[Gub14] Tim Gubner. Achieving many-core scalability in Vectorwise. Master's thesis, Technical University of Ilmenau, 2014.

[Gub18] Tim Gubner. Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware. In *Proc. ICDE*, 2018.

[HCL+90] Laura M. Haas, Wendy Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Trans. on Knowl. and Data Eng.*, pages 143–160, 1990.

[Hua19] Huawei. Huawei Unveils Industry's Highest-Performance ARM-based CPU. `https://www.huawei.com/en/news/2019/1/huawei-unveils-highest-performance-arm-based-cpu`, 2019. Accessed: 2022-09-24.

[HZN+10] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter Boncz. Positional Update Handling in Column Stores. In *SIGMOD*, pages 543–554, 2010.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. MonetDB: Two Decades of Research in Column-oriented Database. *IEEE Data Engineering Bulletin*, 2012.

[Inta] Intel. Beyond Direct Memory Access: Reducing the Data Center Tax with Intel Data Streaming Accelerator. `https://www.intel.com/content/www/us/en/developer/articles/technical/beyond-direct-memory-access-datacenter-tax-dsa.html`. Accessed: 2023-11-19.

[Intb] Intel. Intel® Software Guard Extensions (Intel® SGX) Developer Guide. `https://cdrdv2-public.intel.com/671581/intel-sgx-developer-guide.pdf`. Accessed: 2024-08-20.

[Intc] Intel. Power AI Anywhere with Built-In AI Acceleration. `https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/ai-engines.html`. Accessed: 2023-11-19.

[Intd] Intel. What Is Intel Advanced Matrix Extensions (Intel AMX)? `https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-amx.html`. Accessed: 2023-11-19.

[Int16] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, September 2016. [Accessed on June 28, 2017].

[Int23] Intel. Enhance Business with Faster Insights with Intel IAA. `https://www.intel.com/content/www/us/en/content-details/787805/enhance-business-with-faster-insights-with-intel-iaa.html`, 2023. Accessed: 2023-11-19.

[JMH+16] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *SIGMOD*, pages 281–293, 2016.

[KFG15] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *PVLDB*, 9(4):252–263, 2015.

[KLK+18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, pages 2209–2222, 2018.

[KLN18] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive Execution of Compiled Queries. ICDE, 2018.

[KLN21] Timo Kersten, Viktor Leis, and Thomas Neumann. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal*, pages 1–23, 2021.

[Kra17] Vlad Krasnov. On the dangers of Intel's frequency scaling. `https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/`, 2017. Accessed: 2022-02-16.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293, 2006.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, Palo Alto, California, 2004.

[LAB+20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[LBKN14] 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*, pages 743–754, 2014.

[LCH+11] 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. SIGMOD, pages 1177–1184, 2011.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[LLC23] Yinan Li, Jianan Lu, and Badrish Chandramouli. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *SIGMOD*, 1(2):1–26, 2023.

[LMF+16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIDMOG*, pages 311–326, 2016.

[Loh88] Guy M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. *SIGMOD Record*, pages 18–27, 1988.

[LPK+20] Harald Lang, Linnea Passing, Andreas Kipf, Peter Boncz, Thomas Neumann, and Alfons Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. *The VLDB Journal*, 29(2):757–774, 2020.

[LR20] Lottie Lynn and Matthew Reynolds. PS5 specs and features, including SSD, ray tracing, GPU and CPU for the PlayStation 5 explained. `https://www.eurogamer.net/articles/ps5-specs-features-ssd-ray-tracing-cpu-gpu-6300`, 2020. Accessed: 2022-02-11.

[LS20] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

[Łus11] Alicja Łuszczak. Simple Solutions for Compressed Execution in Vectorized Database System. Master's thesis, Vrije Universiteit Amsterdam, 2011.

[Mic] Microsoft. Xbox Series X. `https://www.xbox.com/en-US/consoles/xbox-series-x#specs`. Accessed: 2022-02-11.

[MMP17] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed Operator Fusion for In-memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *PVLDB*, 11(1):1–13, 2017.

[MRF14] Ingo Müller, Cornelius Ratsch, and Franz Färber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*, pages 283–294, 2014.

[NBH+18] Michal Nowakiewicz, Eric Boutin, Eric Hanson, Robert Walzer, and Akash Katipally. BIPie: Fast Selection and Aggregation on Encoded Data Using Operator Specialization. In *SIGMOD*, pages 1447–1459, 2018.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[NF20] Thomas Neumann and Michael J Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*, 2020.

[NVI] NVIDIA. NVIDIA Grace CPU and Arm Architecture. `https://www.nvidia.com/en-us/data-center/grace-cpu/`. Accessed: 2024-08-17.

[Pal] Mike Pall. https://luajit.org/luajit.html. Accessed June 10, 2023.

[Pal09] Mike Pall. Luajit 2.0 intellectual property disclosure and research opportunities. http://lua-users.org/lists/lua-l/2009-11/msg00089.html, 2009. Accessed June 10, 2023.

[PMZM16] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, pages 1707–1718, 2016.

[Pos11] Cyclic Tag System. https://wiki.postgresql.org/index.php?title=Cyclic_Tag_System&oldid=15106, 2011. Accessed: 2023-07-02.

[pre20] What are _mm_prefetch() locality hints? https://stackoverflow.com/questions/46521694/what-are-mm-prefetch-locality-hints, 2020. Accessed: 2020-07-01.

[PTS+17] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR '17*, 2017.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.

[RAB+13] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[RBŁ21] Greg Rahn, Alexander Behm, and Alicja Łuszczak. https://databricks.com/blog/2021/06/17/announcing-photon-public-preview-the-next-generation-query-engine-on-the-databricks-lakehouse-platform.html, 2021. Accessed: 2021-09-14.

[RBZ13] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro Adaptivity in Vectorwise. SIGMOD, pages 1231–1242, 2013.

[RHGM18] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *DBTEST*, 2018.

[ris21] https://www.nextplatform.com/2020/08/21/alibaba-on-the-bleeding-edge-of-risc-v-with-xt910/, 2021. Accessed: 2021-03-02.

[RM19] Mark Raasveldt and Hannes Mühleisen. DuckDB: An Embeddable Analytical Database. In *SIGMOD*, page 1981–1984, 2019.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-oriented DBMS. In *PVLDB*, pages 553–564, 2005.

[Sam12] Samsung. Samsung Exynos 5 Dual Powers the New Google Chromebook. https://news.samsung.com/global/samsung-exynos-5-dual-powers-the-new-google-chromebook, 2012. Accessed: 2024-08-26.

[SCD16] Stefan Schuh, Xiao Chen, and Jens Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*, pages 1961–1976, 2016.

[Shi13] Anand Lal Shimpi. AMD's Jaguar Architecture: The CPU Powering Xbox One, PlayStation 4, Kabini & Temash. https://www.anandtech.com/show/6976/amds-jaguar-architecture-the-cpu-powering-xbox-one-playstation-4-kabini-temash/4, 2013. Accessed: 2022-02-11.

[Shi21] Anton Shilov. Alibaba Develops Its Own 5nm 128-Core Arm-Based Server Chip. https://www.tomshardware.com/news/alibaba-unveils-128-core-server-cpu, 2021. Accessed: 2022-07-16.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529:484–489, 2016.

[SKP+16] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to Architect a Query Compiler. In *SIGMOD*, pages 1907–1922, 2016.

[Son] Sony. Tech Specs. https://www.playstation.com/en-us/ps4/tech-specs/. Accessed: 2022-02-11.

[SQL20] The SQLite Bytecode Engine. https://www.sqlite.org/opcode.html, 2020. Accessed: 2020-06-24.

[SR86] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. SIGMOD, pages 340–355, 1986.

[SZB11] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. Compilation in Query Execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, 2011.

[TGR+18] Diego Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter Boncz. Optimizing Group-By and Aggregation using GPU-CPU Co-Processing. In *ADMS@ VLDB*, pages 1–10, 2018.

[Tur86] Valentin F Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

[UPD+18] Annett Ungethum, Johannes Pietrzyk, Patrick Damme, Dirk Habich, and Wolfgang Lehner. Conflict detection-based run-length encoding-AVX-512 CD instruction set in action. In *ICDE*, pages 96–101, 2018.

[Var21] Rahoul Varma. The rise of Chromebooks. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/rise-of-chromebooks, 2021. Accessed: 2024-08-20.

[VHF+18] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTEST*, 2018.

[Wad88] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *ESOP*, pages 231–248, 1988.

[XCZ+21] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro*, 41:67–75, 2021.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN@SIGMOD*, pages 47–54, 2008.

CHAPTER 10

Summary

For decades, computers have been used for analyzing data, more recently (still decades ago) via database management systems (DBMS). DBMS are systems that facilitate data storage, modification and analysis, and on top of that, provide certain guarantees. With huge, and often increasing, data volumes, the efficiency of data analysis becomes increasingly important.

Commonly, queries are formulated in some high-level language (typically SQL) for which the DBMS has to find an "optimal" implementation in a lower-level language (which is afterward either interpreted or directly executed). This, typically, means restricting possible choices, which DBMS traditionally do by either fixing certain choices (like e.g. query execution technique/paradigm) or finding a cost-based optimum (for algorithms, ordering, etc.). The generated implementation (flavor) is rather oblivious to the data set (except for possibly relying on optimizer statistics) and environment (hardware, operating system ...), and is, therefore, likely not optimal on the data set and environment "at hand". With more knowledge of the current instance (data set, query, and environment), more efficient implementations, so-called *instance-specific optimizations*, should be possible.

This thesis investigates instance-specific optimizations and focuses on answering two questions: (1) How far can query plans be optimized to the specific instance and (2) how can query engines exploit increasingly heterogeneous modern hardware?

First, we explore the design space (of possible instance-specific optimizations) manually. We found certain techniques that can improve query performance: Thinner data types (Compact Data Types) can fit more data into SIMD registers which improves throughput for arithmetic operations and In-Register Aggregation is a special aggregation optimized for a group-bys with a low number of groups. Furthermore, it is possible to compress hash tables (assuming data distributions allow it) and optimistically compress strings using dictionaries. Both not only improve memory footprint but also runtime.

Second, we simplify the exploration of the design space by using a domain-specific language (VOILA) and synthesizing specific flavors from VOILA. We show that VOILA can not only generate state-of-the-art implementations (e.g. Vectorized and Data-centric Execution) with competitive performance, but also can generate variants as well as mixes thereof (across and within pipelines).

Third, we highlight that the optimal flavor depends on the environment and certain hardware choices can favor certain flavors. Therefore, more or less esoteric hardware choices can weaken/break rules of thumb (e.g. Vectorized Execution outperforms on join-heavy workloads). As the environment and hardware "at hand" are often not fully known ahead of time (i.e. when the DBMS is developed), neither is the optimal flavor. Therefore, more flexible and dynamic query execution is needed. One possible solution is adding another layer of abstraction, by using a virtual machine to execute queries.

Fourth, we propose such a virtual machine (Excalibur) that explores and exploits different instance-specific optimizations, while the query is running (*on-the-fly*). Possible flavors are explored automatically and good flavors are remembered, within and across similar queries. Excalibur demonstrates how a highly flexible and adaptive query engine could look like and how it can be integrated into a framework that uses Vectorized Execution.

In summary, this thesis explored the design space of instance-specific optimizations. We, first, explored the space manually and found points with improved performance. Afterward, we used a domain-specific language to simplify and automate the exploration process. Finally, we described a prototype that explores the space of possible optimizations, while the query is running.

CHAPTER 11

Publications

This thesis is based on the following publications:

- **Exploring Query Execution Strategies for JIT, Vectorization and SIMD**
  Tim Gubner and Peter Boncz
  VLDB - Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS) 2017
- **Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware**
  Tim Gubner
  ICDE - PhD Symposium 2018
- **Efficient Query Processing with Optimistically Compressed Hash Tables & Strings in the USSR**
  Tim Gubner, Viktor Leis and Peter Boncz
  IEEE International Conference on Data Engineering (ICDE) 2020
- **Charting the Design Space of Query Execution using VOILA**
  Tim Gubner and Peter Boncz
  International Conference on Very Large Data Bases (VLDB) 2021

- **Highlighting the Performance Diversity of Analytical Queries using VOILA**
  Tim Gubner and Peter Boncz
  VLDB - Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS) 2021
- **Optimistically Compressed Hash Tables & Strings in the USSR**[1]
  Tim Gubner, Viktor Leis and Peter Boncz
  SIGMOD Record 2021
- **Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA**
  Tim Gubner and Peter Boncz
  International Conference on Very Large Data Bases (VLDB) 2023

Further set of publications not included in this thesis:

- **Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing**
  Mark Raasveldt, Pedro Holanda, Tim Gubner and Hannes Mühleisen
  SIGMOD - International Workshop on Testing Database Systems (DBTest) 2018
- **Optimizing Group-By And Aggregation using GPU-CPU Co-Processing**
  Diego Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg and Peter Boncz
  VLDB - Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS) 2018
- **Fluid Co-processing: GPU Bloom-filters for CPU Joins**
  Tim Gubner, Diego Tomé, Harald Lang and Peter Boncz
  SIGMOD - Workshop on Data Management on New Hardware (DaMoN) 2019

[1] Republished version of 2020 ICDE paper that won the award for the best research paper of the conference.

SIKS Dissertation Series

2016

01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VUA), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UvA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VUA), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VUA), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UvA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VUA), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UvA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VUA), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VUA), Refining Statistical Data on the Web
19 Julia Efremova (TU/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UvA), Context & Semantics in News & Web Search

21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VUA), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UvA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VUA), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (TiU), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UvA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TU/e), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UvA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UvA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UvA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (TiU), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

---

2017

01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation

03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UvA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VUA), Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TU/e), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (TiU), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UvA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UvA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VUA), Logics for causal inference under uncertainty
23 David Graus (UvA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VUA), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VUA), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (TiU), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (TiU), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VUA), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TU/e), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR

40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
46 Jan Schneider (OU), Sensor-based Learning Support
47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
48 Angel Suarez (OU), Collaborative inquiry-based learning

2018

01 Han van der Aa (VUA), Comparing and Aligning Process Representations
02 Felix Mannhardt (TU/e), Multi-perspective Process Mining
03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
05 Hugo Huurdeman (UvA), Supporting the Complex Dynamics of the Information Seeking Process
06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems
09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
11 Mahdi Sargolzaei (UvA), Enabling Framework for Service-oriented Collaborative Networks
12 Xixi Lu (TU/e), Using behavioral context in process mining
13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
14 Bart Joosten (TiU), Detecting Social Signals with Spatiotemporal Gabor Filters
15 Naser Davarzani (UM), Biomarker discovery in heart failure
16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TU/e), On Graph Sample Clustering
18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OU), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araújo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TU/e), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (TiU), Emotion Recognition from Mandarin Speech

	30	Wouter Beek (VUA), The "K" in "semantic web" stands for "knowledge": scaling semantics to the web
2019		
	01	Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
	02	Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
	03	Eduardo Gonzalez Lopez de Murillas (TU/e), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
	04	Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
	05	Sebastiaan van Zelst (TU/e), Process Mining with Streaming Data
	06	Chris Dijkshoorn (VUA), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
	07	Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
	08	Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
	09	Fahimeh Alizadeh Moghaddam (UvA), Self-adaptation for energy efficiency in software systems
	10	Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
	11	Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
	12	Jacqueline Heinerman (VUA), Better Together
	13	Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
	14	Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
	15	Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
	16	Guangming Li (TU/e), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
	17	Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
	18	Gerard Wagenaar (UU), Artefacts in Agile Team Communication
	19	Vincent Koeman (TUD), Tools for Developing Cognitive Agents
	20	Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
	21	Cong Liu (TU/e), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
	22	Martin van den Berg (VUA),Improving IT Decisions with Enterprise Architecture
	23	Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
	24	Anca Dumitrache (VUA), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
	25	Emiel van Miltenburg (VUA), Pragmatic factors in (automatic) image description
	26	Prince Singh (UT), An Integration Platform for Synchromodal Transport
	27	Alessandra Antonaci (OU), The Gamification Design Process applied to (Massive) Open Online Courses
	28	Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
	29	Daniel Formolo (VUA), Using virtual agents for simulation and training of social skills in safety-critical circumstances
	30	Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
	31	Milan Jelisavcic (VUA), Alive and Kicking: Baby Steps in Robotics
	32	Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
	33	Anil Yaman (TU/e), Evolution of Biologically Inspired Learning in Artificial Neural Networks

34 Negar Ahmadi (TU/e), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
35 Lisa Facey-Shaw (OU), Gamification with digital badges in learning programming
36 Kevin Ackermans (OU), Designing Video-Enhanced Rubrics to Master Complex Skills
37 Jian Fang (TUD), Database Acceleration on FPGAs
38 Akos Kadar (OU), Learning visually grounded and multilingual representations

---

2020

01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
04 Maarten van Gompel (RUN), Context as Linguistic Bridges
05 Yulong Pei (TU/e), On local and global structure mining
06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
07 Wim van der Vegt (OU), Towards a software architecture for reusable game components
08 Ali Mirsoleimani (UL),Structured Parallel Programming for Monte Carlo Tree Search
09 Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
10 Alifah Syamsiyah (TU/e), In-database Preprocessing for Process Mining
11 Sepideh Mesbah (TUD), Semantic-Enhanced Training Data AugmentationMethods for Long-Tail Entity Recognition Models
12 Ward van Breda (VUA), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
13 Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
14 Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
15 Konstantinos Georgiadis (OU), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
16 Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling
17 Daniele Di Mitri (OU), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
18 Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
19 Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
20 Albert Hankel (VUA), Embedding Green ICT Maturity in Organisations
21 Karine da Silva Miras de Araujo (VUA), Where is the robot?: Life as it could be
22 Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar
23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
24 Lenin da Nóbrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
25 Xin Du (TU/e), The Uncertainty in Exceptional Model Mining
26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
29 Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
30 Bob Zadok Blok (UL), Creatief, Creatiever, Creatiefst

31 Gongjin Lan (VUA), Learning better – From Baby to Better
32 Jason Rhuggenaath (TU/e), Revenue management in online markets: pricing and online advertising
33 Rick Gilsing (TU/e), Supporting service-dominant business model evaluation in the context of business model innovation
34 Anna Bon (UM), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
35 Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production

2021

01 Francisco Xavier Dos Santos Fonseca (TUD),Location-based Games for Social Interaction in Public Space
02 Rijk Mercuur (TUD), Simulating Human Routines: Integrating Social Practice Theory in Agent-Based Models
03 Seyyed Hadi Hashemi (UvA), Modeling Users Interacting with Smart Devices
04 Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
05 Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems
06 Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
07 Armel Lefebvre (UU), Research data management for open science
08 Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
09 Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Children's Collaboration Through Play
10 Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
11 Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision
12 Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
13 Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
14 Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
15 Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
16 Esam A. H. Ghaleb (UM), Bimodal emotion recognition from audio-visual cues
17 Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks
18 Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
19 Roberto Verdecchia (VUA), Architectural Technical Debt: Identification and Management
20 Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems
21 Pedro Thiago Timbó Holanda (CWI), Progressive Indexes
22 Sihang Qiu (TUD), Conversational Crowdsourcing
23 Hugo Manuel Proença (UL), Robust rules for prediction and description
24 Kaijie Zhu (TU/e), On Efficient Temporal Subgraph Query Processing
25 Eoin Martino Grua (VUA), The Future of E-Health is Mobile: Combining AI and Self-Adaptation to Create Adaptive E-Health Mobile Applications
26 Benno Kruit (CWI/VUA), Reading the Grid: Extending Knowledge Bases from Human-readable Tables
27 Jelte van Waterschoot (UT), Personalized and Personal Conversations: Designing Agents Who Want to Connect With You
28 Christoph Selig (UL), Understanding the Heterogeneity of Corporate Entrepreneurship Programs

2022

01 Judith van Stegeren (UT), Flavor text generation for role-playing video games

02 Paulo da Costa (TU/e), Data-driven Prognostics and Logistics Optimisation: A Deep Learning Journey
03 Ali el Hassouni (VUA), A Model A Day Keeps The Doctor Away: Reinforcement Learning For Personalized Healthcare
04 Ünal Aksu (UU), A Cross-Organizational Process Mining Framework
05 Shiwei Liu (TU/e), Sparse Neural Network Training with In-Time Over-Parameterization
06 Reza Refaei Afshar (TU/e), Machine Learning for Ad Publishers in Real Time Bidding
07 Sambit Praharaj (OU), Measuring the Unmeasurable? Towards Automatic Co-located Collaboration Analytics
08 Maikel L. van Eck (TU/e), Process Mining for Smart Product Design
09 Oana Andreea Inel (VUA), Understanding Events: A Diversity-driven Human-Machine Approach
10 Felipe Moraes Gomes (TUD), Examining the Effectiveness of Collaborative Search Engines
11 Mirjam de Haas (UT), Staying engaged in child-robot interaction, a quantitative approach to studying preschoolers' engagement with robots and tasks during second-language tutoring
12 Guanyi Chen (UU), Computational Generation of Chinese Noun Phrases
13 Xander Wilcke (VUA), Machine Learning on Multimodal Knowledge Graphs: Opportunities, Challenges, and Methods for Learning on Real-World Heterogeneous and Spatially-Oriented Knowledge
14 Michiel Overeem (UU), Evolution of Low-Code Platforms
15 Jelmer Jan Koorn (UU), Work in Process: Unearthing Meaning using Process Mining
16 Pieter Gijsbers (TU/e), Systems for AutoML Research
17 Laura van der Lubbe (VUA), Empowering vulnerable people with serious games and gamification
18 Paris Mavromoustakos Blom (TiU), Player Affect Modelling and Video Game Personalisation
19 Bilge Yigit Ozkan (UU), Cybersecurity Maturity Assessment and Standardisation
20 Fakhra Jabeen (VUA), Dark Side of the Digital Media - Computational Analysis of Negative Human Behaviors on Social Media
21 Seethu Mariyam Christopher (UM), Intelligent Toys for Physical and Cognitive Assessments
22 Alexandra Sierra Rativa (TiU), Virtual Character Design and its potential to foster Empathy, Immersion, and Collaboration Skills in Video Games and Virtual Reality Simulations
23 Ilir Kola (TUD), Enabling Social Situation Awareness in Support Agents
24 Samaneh Heidari (UU), Agents with Social Norms and Values - A framework for agent based social simulations with social norms and personal values
25 Anna L.D. Latour (UL), Optimal decision-making under constraints and uncertainty
26 Anne Dirkson (UL), Knowledge Discovery from Patient Forums: Gaining novel medical insights from patient experiences
27 Christos Athanasiadis (UM), Emotion-aware cross-modal domain adaptation in video sequences
28 Onuralp Ulusoy (UU), Privacy in Collaborative Systems
29 Jan Kolkmeier (UT), From Head Transform to Mind Transplant: Social Interactions in Mixed Reality
30 Dean De Leo (CWI), Analysis of Dynamic Graphs on Sparse Arrays
31 Konstantinos Traganos (TU/e), Tackling Complexity in Smart Manufacturing with Advanced Manufacturing Process Management
32 Cezara Pastrav (UU), Social simulation for socio-ecological systems
33 Brinn Hekkelman (CWI/TUD), Fair Mechanisms for Smart Grid Congestion Management

34 Nimat Ullah (VUA), Mind Your Behaviour: Computational Modelling of Emotion & Desire Regulation for Behaviour Change
35 Mike E.U. Ligthart (VUA), Shaping the Child-Robot Relationship: Interaction Design Patterns for a Sustainable Interaction

2023

01 Bojan Simoski (VUA), Untangling the Puzzle of Digital Health Interventions
02 Mariana Rachel Dias da Silva (TiU), Grounded or in flight? What our bodies can tell us about the whereabouts of our thoughts
03 Shabnam Najafian (TUD), User Modeling for Privacy-preserving Explanations in Group Recommendations
04 Gineke Wiggers (UL), The Relevance of Impact: bibliometric-enhanced legal information retrieval
05 Anton Bouter (CWI), Optimal Mixing Evolutionary Algorithms for Large-Scale Real-Valued Optimization, Including Real-World Medical Applications
06 António Pereira Barata (UL), Reliable and Fair Machine Learning for Risk Assessment
07 Tianjin Huang (TU/e), The Roles of Adversarial Examples on Trustworthiness of Deep Learning
08 Lu Yin (TU/e), Knowledge Elicitation using Psychometric Learning
09 Xu Wang (VUA), Scientific Dataset Recommendation with Semantic Techniques
10 Dennis J.N.J. Soemers (UM), Learning State-Action Features for General Game Playing
11 Fawad Taj (VUA), Towards Motivating Machines: Computational Modeling of the Mechanism of Actions for Effective Digital Health Behavior Change Applications
12 Tessel Bogaard (VUA), Using Metadata to Understand Search Behavior in Digital Libraries
13 Injy Sarhan (UU), Open Information Extraction for Knowledge Representation
14 Selma Čaušević (TUD), Energy resilience through self-organization
15 Alvaro Henrique Chaim Correia (TU/e), Insights on Learning Tractable Probabilistic Graphical Models
16 Peter Blomsma (TiU), Building Embodied Conversational Agents: Observations on human nonverbal behaviour as a resource for the development of artificial characters
17 Meike Nauta (UT), Explainable AI and Interpretable Computer Vision – From Oversight to Insight
18 Gustavo Penha (TUD), Designing and Diagnosing Models for Conversational Search and Recommendation
19 George Aalbers (TiU), Digital Traces of the Mind: Using Smartphones to Capture Signals of Well-Being in Individuals
20 Arkadiy Dushatskiy (TUD), Expensive Optimization with Model-Based Evolutionary Algorithms applied to Medical Image Segmentation using Deep Learning
21 Gerrit Jan de Bruin (UL), Network Analysis Methods for Smart Inspection in the Transport Domain
22 Alireza Shojaifar (UU), Volitional Cybersecurity
23 Theo Theunissen (UU), Documentation in Continuous Software Development
24 Agathe Balayn (TUD), Practices Towards Hazardous Failure Diagnosis in Machine Learning
25 Jurian Baas (UU), Entity Resolution on Historical Knowledge Graphs
26 Loek Tonnaer (TU/e), Linearly Symmetry-Based Disentangled Representations and their Out-of-Distribution Behaviour
27 Ghada Sokar (TU/e), Learning Continually Under Changing Data Distributions
28 Floris den Hengst (VUA), Learning to Behave: Reinforcement Learning in Human Contexts
29 Tim Draws (TUD), Understanding Viewpoint Biases in Web Search Results

2024

01 Daphne Miedema (TU/e), On Learning SQL: Disentangling concepts in data systems education
02 Emile van Krieken (VUA), Optimisation in Neurosymbolic Learning Systems
03 Feri Wijayanto (RUN), Automated Model Selection for Rasch and Mediation Analysis
04 Mike Huisman (UL), Understanding Deep Meta-Learning
05 Yiyong Gou (UM), Aerial Robotic Operations: Multi-environment Cooperative Inspection & Construction Crack Autonomous Repair
06 Azqa Nadeem (TUD), Understanding Adversary Behavior via XAI: Leveraging Sequence Clustering to Extract Threat Intelligence
07 Parisa Shayan (TiU), Modeling User Behavior in Learning Management Systems
08 Xin Zhou (UvA), From Empowering to Motivating: Enhancing Policy Enforcement through Process Design and Incentive Implementation
09 Giso Dal (UT), Probabilistic Inference Using Partitioned Bayesian Networks
10 Cristina-Iulia Bucur (VUA), Linkflows: Towards Genuine Semantic Publishing in Science
11 withdrawn
12 Peide Zhu (TUD), Towards Robust Automatic Question Generation For Learning
13 Enrico Liscio (TUD), Context-Specific Value Inference via Hybrid Intelligence
14 Larissa Capobianco Shimomura (TU/e), On Graph Generating Dependencies and their Applications in Data Profiling
15 Ting Liu (VUA), A Gut Feeling: Biomedical Knowledge Graphs for Interrelating the Gut Microbiome and Mental Health
16 Arthur Barbosa Câmara (TUD), Designing Search-as-Learning Systems
17 Razieh Alidoosti (VUA), Ethics-aware Software Architecture Design
18 Laurens Stoop (UU), Data Driven Understanding of Energy-Meteorological Variability and its Impact on Energy System Operations
19 Azadeh Mozafari Mehr (TU/e), Multi-perspective Conformance Checking: Identifying and Understanding Patterns of Anomalous Behavior
20 Ritsart Anne Plantenga (UL), Omgang met Regels
21 Federica Vinella (UU), Crowdsourcing User-Centered Teams
22 Zeynep Ozturk Yurt (TU/e), Beyond Routine: Extending BPM for Knowledge-Intensive Processes with Controllable Dynamic Contexts
23 Jie Luo (VUA), Lamarck's Revenge: Inheritance of Learned Traits Improves Robot Evolution
24 Nirmal Roy (TUD), Exploring the effects of interactive interfaces on user search behaviour
25 Alisa Rieger (TUD), Striving for Responsible Opinion Formation in Web Search on Debated Topics