



Efficient Data Processing on Modern Hardware

Maximilian Bandle



Efficient Data Processing on Modern Hardware

Maximilian Bandle

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Georg Carle

Prüfer*innen der Dissertation:

1. Prof. Dr. Jana Giceva Makreshanska
2. Prof. Dr. Thomas Neumann
3. Prof. Dr. Zsolt Istvan

Die Dissertation wurde am 21.09.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 30.12.2023 angenommen.

Abstract

Database systems and SQL have been the cornerstone of data processing for over 50 years. They are relevant to this day since they continue to evolve along with changing hardware setups and workload requirements. This work focuses on the challenges of efficient data processing in high-performance database systems. We follow the path data takes from filtered ingestion to combining data and finally take a holistic view of the query processing system.

First, we look at the implementation of database operators by evaluating approximate filters and joins in-depth to identify the best variant depending on workload characteristics. We pick up the ongoing debate on the best main-memory join implementation and answer it with an in-system comparison. Combining the analysis of both operators, we compile a list of factors determining whether partitioning pays off and is worth integrating.

Second, this thesis addresses the challenge of devising a concept for a more flexible data flow architecture. This work splits up the existing operators and shows how the resulting sub-operators can form more generic data flows. Sub-operators furthermore enable the database system to target more different architectures and raise the question of how to approach resource heterogeneity, which has become omnipresent in the cloud.

We conclude with a discussion of future research directions that arise from the contributions made in this dissertation. These contributions include multi-dimensional analyses of filters and joins and a conceptual architecture for lowering diverse workloads into sub-operators. We highlight the potential of compiling database systems to hide both the underlying complexity of the ongoing disaggregation and the continuous advent of new hardware platforms.

Zusammenfassung

Datenbanksysteme und SQL sind seit über 50 Jahren die Eckpfeiler der Datenverarbeitung. Sie sind auch heute noch relevant, da sie sich mit den sich stetig ändernden Hardwarekonfigurationen und Workloads weiterentwickeln. Diese Arbeit konzentriert sich auf die Herausforderungen von effizienter Datenverarbeitung in hochperformanten Datenbanksystemen. Wir verfolgen den Weg, den die Daten von der gefilterten Aufnahme bis zur Kombination der Daten nehmen, und betrachten schließlich holistisch das Verhalten des Gesamtsystems.









Zunächst befassen wir uns mit der Implementierung von Datenbankoperatoren, indem wir Bitfilter und Joins eingehend evaluieren, um die beste Variante in Abhängigkeit von den gegebenen Anforderungen zu ermitteln. Wir greifen die laufende Debatte über die beste Join-Implementierung in Hauptspeicher-Datenbanken auf und beantworten sie mit einem Vergleich im System. Indem wir die Analyse beider Operatoren kombinieren, stellen wir eine Liste von Faktoren zusammen, die bestimmen, ob sich Partitionierung auszahlt und ob es die Integration rechtfertigt.

Zweitens befasst sich diese Arbeit mit der Herausforderung, ein Konzept für eine flexiblere Datenflussarchitektur zu entwickeln. In dieser Arbeit werden die vorhandenen Operatoren zerlegt und wir zeigen, wie sich die entstehenden Sub-Operatoren zu vielfältigeren Datenflüssen kombinieren lassen. Sub-Operatoren ermöglichen es dem Datenbanksystem außerdem, verschiedene Architekturen zu unterstützen, und werfen die Frage auf, wie mit der in der Cloud allgegenwärtigen Heterogenität der Systeme umgegangen werden kann.

Wir schließen mit einer Diskussion über zukünftige Forschungsrichtungen, die sich aus den Beiträgen dieser Dissertation ergeben. Zu diesen Beiträgen gehören mehrdimensionale Analysen von Filtern und Joins, sowie eine konzeptionelle Architektur zur Aufteilung der Datenbank-Operatoren in Sub-Operatoren. Wir zeigen das Potenzial der Kompilierung von Datenbanksystemen auf, um sowohl die zugrunde liegende Komplexität der fortschreitenden Disaggregation als auch das kontinuierliche Aufkommen neuer Hardwareplattformen vor Anwendern zu verbergen.

Acknowledgments

During my time at TUM, I was fortunate to receive help, advice, and guidance from many people. I would especially like to thank:

-  Jana and Thomas for giving me both guidance and freedom to work on the topics contained in this thesis and many more.
-  Prof. Kemper, Viktor, and Andy for showing and convincing me that database systems research is far more than just writing SQL queries.
-  Prof. Georg Carle and Prof. Zsolt Istvan for serving on my doctoral examination.
-  Tobias and Ferdinand for our collaborations, which led to a VLDB paper each.
-  Alice, Maximilian, Max, and Miguel for splitting the teaching duty and sharing our enthusiasm for database systems and prototyping.
-  Philipp for sharing an office and having the same thrive for furnishing and decoration.
-  Flo, Alex, and all our employees and working students for giving me the freedom, flexibility, time, and support to work on this thesis.
-  Thuy Ngan for the constant support no matter what kind.

I would furthermore like to thank all my current and former colleagues at the chair, Adrian, Aliaksei, Alex, Alexis, Altan, André, Bernhard, Chris, Christoph, Daniel, Dominik, Harald, Jan, Linnea, Lukas, Max, Moritz, Michael F., Michael J., Michalis, Per, Simon, Stefan, Timo, Theresa, Tobias G., and Tobias S. for fruitful research discussions and for tolerating the lots of new door signs and decorations.

Finally, I thank my friends and family for their support and also sometimes the diversions throughout this journey. This thesis would not have been possible without all of you.

Funding.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement number 725286).



Preface

Excerpts of this thesis have been published in advance.

Chapter 2 has previously been published in:

Tobias Schmidt, Maximilian Bandle, and Jana Giceva. “A four-dimensional Analysis of Partitioned Approximate Filters”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2355–2368

Chapter 3 has previously been published in:

Maximilian Bandle, Jana Giceva, and Thomas Neumann. “To Partition, or Not to Partition, That is the Join Question in a Real System”. In: *SIGMOD Conference.* ACM, 2021, pp. 168–180

Chapter 4 has previously been published in:

Maximilian Bandle and Jana Giceva. “Database Technology for the Masses: Sub-Operators as First-Class Entities”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2483–2490

In addition to these publications, the author of this thesis also co-authored the following related work, which is not part of this thesis

Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. “Adopting Worst-Case Optimal Joins in Relational Database Systems”. In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1891–1904

Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. “Profiling dataflow systems on multiple abstraction levels”. In: *EuroSys.* ACM, 2021, pp. 474–489

Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. “Bringing Compiling Databases to RISC Architectures”. In: *Proc. VLDB Endow.* 16.6 (2023), pp. 1222–1234

Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. “Programming Fully Disaggregated Systems”. In: *HotOS.* USENIX Association, 2023

Adrian Riedl, Philipp Fent, Maximilian Bandle, and Thomas Neumann.
“Exploiting Code Generation for Efficient LIKE Pattern Matching”. In:
ADMS@VLDB. 2023

Contents

Acknowledgments	i
Preface	iii
1 Introduction	1
1.1 Background and Motivation	2
1.2 Research Questions	4
1.3 Challenges and Methodology	4
1.4 Contributions and Outline	5
2 Partitioned Approximate Filters	7
2.1 Motivation	8
2.2 Related Work	10
2.3 Approximate Filters	11
2.3.1 Filters in DBMS	11
2.3.2 Configuration Parameters	15
2.3.3 Bloom Filters	15
2.3.4 Fingerprint Filters	17
2.4 Implementation and Optimizations	18
2.4.1 Partitioning	19
2.4.2 Vectorization	21
2.4.3 Multi-Threading	22
2.5 Evaluation	24
2.5.1 Experimental Setup	24
2.5.2 Lookup Performance	25
2.5.3 Build Performance	31
2.6 Lessons Learned	33
2.7 Discussion	34
3 Partitioned Joins in a DBMS	35
3.1 Motivation	36

3.2	Related Work	38
3.3	Partitioned Radix Joins	41
3.3.1	Basic Partitioned Join	41
3.3.2	Parallel Radix Join by Balkesen et al.	42
3.3.3	Optimized Radix Join:	43
3.4	Joins in main-memory DBMS	44
3.4.1	Data-Centric Code-Generation	44
3.4.2	Materialization Strategy	45
3.4.3	Non-Partitioned Hash Join	45
3.4.4	Partitioned Hash Join	45
3.4.5	Morsel-Driven Partitioning	46
3.4.6	Final Join Phase	49
3.4.7	Bloom Filters	49
3.5	Evaluation	50
3.5.1	Experimental Setup	51
3.5.2	Performance characterization and comparison to related work	53
3.5.3	TPC-H Evaluation	58
3.5.4	Isolating the effects of different factors	62
3.6	Discussion	67
4	Sub-Operators as First-Class Entities	71
4.1	Motivation	72
4.2	Sub-operators as first class entities	74
4.2.1	Sub-operators and interfaces	74
4.2.2	Dataflows beyond standard SQL	79
4.2.3	Cross-platform compilation and execution	80
4.2.4	Hardware integration	81
4.2.5	Impact on system design	82
4.3	Multi-Level Query Optimizer	83
4.4	Query Compiler	85
4.4.1	CPU Heterogeneity: RISC on the rise	86
4.4.2	Compilation Strategies	88
4.4.3	Performance Tradeoffs: Latency vs. Throughput	92
4.4.4	Architecture Conscious Sub-Operator Design	94
4.5	Execution engine	95
4.6	Related work	96
4.7	Discussion	97
5	Conclusion	99
	Bibliography	101

List of Figures

1.1	Statistics about Data Processed and Processing Power.	1
2.1	Speedup gained by optimizing insert and lookup operations for 100 M keys.	8
2.2	Performance with 100 M elements (10 threads).	9
2.3	Hash Joins in Umbra	12
2.4	False-Positive Rate of for Bloom Filters with $m = 16$	13
2.5	Overview of Bloom Filter variants	16
2.6	Build and lookup performance for partitioned and non-partitioned filters.	20
2.7	Throughput-optimal number of partitions.	21
2.8	Speedup using vectorized filters (AVX512); the baseline are scalar filters (partitioning is enabled).	22
2.9	Scalability of the blocked Bloom filter on different machines	23
2.10	Bloom vs. fingerprint filters.	25
2.11	Lookup performance for 100M elements	26
2.12	Lookup performance and best-performing filter. The white line separates Bloom from fingerprint variants.	27
2.13	Slice from 1 M elements for constant size or FPR.	27
2.14	Best-performing Bloom filter variant for lookup.	29
2.15	k of best-performing filter	30
2.16	Vectorized vs. non-vectorized filters.	31
2.17	Partitioned vs. Non-Partitioned variant.	32
2.18	Best-performing filter for construction.	33
3.1	Relative performance of partitioned and non-partitioned hash join	36
3.2	Tuple Size and Join Partners in TPC-H and prior work	37
3.3	Radix-Partitioning in prior work	42
3.4	Pipelining in radix and hash joins. Hash joins can pass the probe tuples through multiple joins while radix joins have to materialize both inputs every time.	44
3.5	Schematic Overview of our Partitioned Join.	46
3.6	Schematic overview of our two-pass partitioning performing an eight-way split using 3 bits.	47

3.7	Bloom filters with selective joins. Tuples without a join partner are filtered early and are not materialized.	50
3.8	Scalability and comparison to Balkesen et al.	54
3.9	Scalability on different machines	55
3.10	Memory Bandwidth for 24B wide tuples	56
3.11	Throughput of all TPC-H queries containing joins with every join replaced by the one under test	57
3.12	Relative impact per join for selected TPC-H queries (without Late Materialization)	61
3.13	Q21 Join Tree annotated with build and probe size	61
3.14	Impact of pre-filtering the probe side using a Bloom-filter based early probe	63
3.15	Impact of payload size on join performance	63
3.16	Impact of pipeline depth	65
3.17	Impact of different Zipf factors	66
3.18	Speedup of different join implementations over the optimized radix join .	68
4.1	Sub-operators are building blocks for more complex data operations . . .	72
4.2	Overview of a sub-operator-based database engine	76
4.3	Composing a partitioner using sub-operators. The new <i>partition</i> sub-operator can be used in hash-joins.	77
4.4	Schematic overview of one k-means iteration.	80
4.5	Overview of the sub-operator system stack.	83
4.6	The query optimizer generates the optimal plan for the given target system using specialized sub-operators based on dataflow and cost functions. . .	84
4.7	different levels	85
4.8	Prognosis of ARM usage in the server market.	87
4.9	Qualitative overview of different query-compilation strategies' properties	89
4.10	Comparison of different execution strategies in a database system. . . .	93
4.11	Design process of domain-specific and architecture-aware sub-operators for dataflow systems.	95

List of Tables

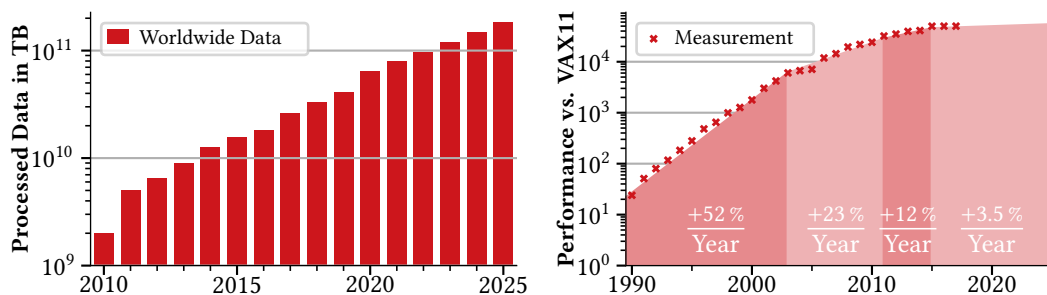
2.1	Common configuration parameters for approximate filters	14
2.2	Throughput on Skylake-X [Keys/s (scale-up)].	23
2.3	Partition Parameters for Experiments	24
3.1	Workloads from Prior Work	52
3.2	Hardware Platforms	52
3.3	Throughput [T/s] with and without Late Materialization	64
3.4	Workload Characteristics for Partitioned Joins	68
3.5	Workloads for Join Processing	69
4.1	Example of possible sub-operators	75

CHAPTER 1

Introduction

Data processing involves systematically converting raw data from multiple sources and transforming them into meaningful insights. It is a foundational application area for computing because it automates the data retrieval and transformation process. This automation allows data processing to work in real-time and scale with the demands. Scalability is crucial here since along with data volume growing exponentially as shown in Figure 1.1a, the required processing power has to keep up. The stagnating raw processing power, visualized in Figure 1.1b, cannot keep up with the growing data processing demand. This gap between the actual processing power and the demand necessitates streamlined and more performant data processing.

Database systems offer the demanded high performance to the user while also offering consistency guarantees when storing and retrieving data. This makes them a central piece of data processing. To achieve high performance, database systems



(a) Yearly Processed Data [86]

(b) Normalized Processor Performance [83]

Figure 1.1: Statistics about Data Processed and Processing Power show an increased gap between the rapid growth of processed data and the stagnating processor performance.

are closely interlinked to modern hardware, which introduces both opportunities and complexities. Current hardware architecture has shifted away from pursuing higher clock speeds to multicore CPUs and now also tailored on-chip accelerators or distributed cloud infrastructures. This evolution increased the raw processing power by orders of magnitudes. However, due to higher complexity, it also raised the need for understanding the new hardware to utilize their performance in data processing.

This dissertation delves into the combination of database systems and the evolving hardware landscape. By following the fundamental steps in a data processing pipeline and examining how they interact with modern hardware, we contribute to more efficient database design and operation. Through systematic analysis and thoroughly benchmarked implementations, this dissertation's goal is to advance data processing and enhance our understanding of how modern hardware can be effectively utilized to optimize the performance of database systems.

1.1 Background and Motivation

Databases have been one of the building blocks for electronic data processing since the 1950s and provide efficient access to individual data chunks instead of batch-processing whole tapes. They furthermore offer strong guarantees, commonly referred to as ACID, for transactional workloads and kept the same relational storage model for over 50 years [45]. The ubiquity of SQL as lingua franca for data access [38] across various computing platforms, including mainframes [44], cloud databases [11, 50, 170], and mobile phones [188, 193], further demonstrates the enduring significance of databases.

While SQL and the relational model offer a user-oriented perspective on data access and structure, database systems retain the flexibility to determine the actual implementation of features and data storage on physical media [45]. This flexibility, initially perceived as a potential obstacle to standardization, has in fact contributed to the continued relevance of SQL and the relational model. Modern Database Systems have embraced novel techniques such as massively-parallel join algorithms [4, 17, 108], vectorized or compiled execution methods [95, 181], and distributed query processing [170], while still relying on the relational model as their foundation.

Independent of the described advancements, a substantial portion of database tasks involves filtering [179] and connecting stored data through joins [143]. Users commonly engage with aggregated data through dashboards and automated processes [173], building upon the foundational capabilities of databases [204]. In our research, we focus on database engineering using our system Umbra [140]. Utilizing a SQL frontend, Umbra is designed around a state-of-the-art optimizer [23, 58, 70, 144], code-generation [65, 75, 96], and implements a wide range of specialized relational and physical operators [17, 57, 66, 69, 104, 143, 168, 171, 180, 184, 190, 209]. Umbra is under constant development and allows us to directly integrate and test our research

in a state-of-the-art database system [56] with industry-standard benchmarks [26, 116, 198, 199].

Nearly every database query involves accessing data from storage media, either by directly loading frequently accessed data from memory or retrieving it from disk for larger datasets [5, 79, 114]. To optimize query performance, database systems commonly employ index structures like B-Trees [19, 47] or Log-Structured-Merge (LSM) Trees [120, 148], reducing data traversal during queries. Even before accessing the actual data, efficient pre-filtering mechanisms are crucial for query optimization [65]. Approximate bit filters, including bloom [25] or fingerprint filters [62, 74, 203], are key to accelerating data retrieval. These filters offer the advantage of significant space savings and high throughput at the expense of potential false positives, which are subsequently filtered out [179]. This approach is seamlessly integrated into various stages of the database system, including join processing [17, 113], data distribution between nodes [34], and data loading from disk [51].

With the data now appropriately filtered, the next step involves connecting datasets or relational tables. The relational model builds upon connecting information from multiple tables, offering a robust foundation for structured data storage [45]. As SQL does not specify the join algorithm [38], there is a multitude of alternatives available ranging from nested loop joins that iterate through both tables to sort-merge joins [4] or hash joins [108, 113, 143]. Sort-Merge joins, which leverage sorted data, were prevalent until the 70s and were replaced by hash joins for efficiency reasons in the 1980s [153]. Around 2000, Boncz et al. [28, 125, 126] predicted that memory access would be the future bottleneck. Back then processor throughput limited the processing speed. They worked on multi-pass radix-partitioning to overcome the Translation Lookaside Buffer (TLB) thrashing problem of the original hardware-conscious join by Shatdal et al. [187] and investigated optimized materialization strategies [127]. Several authors have since then compared join implementations [4, 13, 14, 15, 24, 99, 108, 178, 207] and came to different conclusions about which hash join variant to use. The findings range from using non-partitioned or partitioned as the main implementation to combining both approaches and using the radix-partitioned join as a booster with filters attached like semi-join reducers [17, 56].

Considering both filter and join operator, we notice synergies, as certain components, such as the bloom filter in the semi-join reducer, occur multiple times and thus can be re-used [16, 17]. With a growing number of specialized operators [66, 168, 171, 209] and accelerators [146, 149, 162, 163, 202], this aspect becomes increasingly pertinent. Most database operators use similar re-configurable sub-operators to scan, materialize, or process data. These sub-operators have a common notion of data input or output, whether in the form of data streams for in-place processing like the join's probe side or buffers for processing the data as a whole, e.g., the join's build side [16, 103, 106, 210]. Splitting operators into sub-parts makes it reusable and simplifies the construction or embedding of non-SQL data flows by recombining sub-operators.

Having such a sub-operator representation offers future possibilities with the trend going towards distributed computing [11, 50, 170]. Depending on how we can define the operator boundaries, certain sub-tasks can be offloaded to dedicated hardware targets, which are tailored for decompression or filtering [162, 163, 202]. The increasing complexity of modern hardware heavily influences the architecture and implementation of today's highly efficient data flow engines, including database systems. Cloud providers already adopt heterogeneous hardware to accelerate data processing. Machine learning workloads utilize these accelerators, primarily GPUs, which diversify how and where data is processed. Thus, database systems must tackle the workload diversification on heterogeneous hardware to keep the data in the database system and assist the data analytics pipelines.

1.2 Research Questions

This dissertation first looks into two key aspects of efficient data processing, filtering and joining, and then continues with an outlook on how data processing will change with the general availability of heterogeneous resources. Some of the research questions this dissertation addresses are:

R 1: Approximate Filters Given the plethora of approximate filter variants, what are the best-fitting filters for different data processing scenarios?

R 2: Radix-Partitioned Joins Given the ongoing debate on architecting main-memory joins, how can we fairly compare partitioned and non-partitioned joins?

R 3: Partitioning When if at all does partitioning pay off for synthetic and real-world workloads, and is partitioning worth integrating?

R 4: Data Flow Architecture Given a notion of data flow and certain target architectures, how can we best interface between them?

R 5: Resource Heterogeneity How can the database engine handle machine and workload diversity?

1.3 Challenges and Methodology

In light of a constantly changing hardware landscape and processing demands, the presented research questions are imperative for developing database engines. More precisely, database engines must continuously overcome challenges to decide on the

best-fitting implementation on a case-by-case basis to cater to the individual requirements of each data flow pipeline. One challenge is the increasing heterogeneity in hardware, which enlarges the options when implementing algorithms. This includes considerations on efficient multi-threading, the usage of vector instruction, or specialized accelerators which are on the rise. With CPU performance slowly stagnating, the time is right to research how database components can use resources most efficiently.

Furthermore, the resources cannot simply scale with the processing demands since, e.g., available hardware and power are naturally limited. So, the software has to get more sustainable together with the hardware it is running on. Sustainability for the software goes in two directions. On the one hand, of course, more efficient software uses the available resources better and thus leads to more energy-efficient computation. On the other hand, the architecture can be designed to be more reusable thereby improving sustainability. By having an overview of which parts and algorithms to focus your engineering on, database development in itself becomes more efficient.

We tackle the aforementioned challenges through an in-depth analysis of several major parts of the database system. We use microbenchmarks to highlight the problem from all sides and understand how the hardware behaves under certain tasks and loads. Since it is a challenge in itself to find representative workloads, we also perform benchmarks inside of our database system. This allows us to see how a component performs non-isolated and, for example, identify the factors that influence when partitioning pays off. Based on these learnings from building operators and porting the DBMS to another architecture, we break down typical relational operators into a concept of sub-operators. We provide a nuanced analysis including related work on how the proposed operator set can perform arbitrary data flow operations. This leads to a discussion of how to engineer the future DMBS with modern hardware in mind.

1.4 Contributions and Outline

The thesis structure follows the structure of a data processing pipeline. First, we focus on improving and understanding performance in real database systems. We implement and analyze joins and filters, focusing on when partitioning pays off.

Second, we look at the implications of adapting database systems to hardware disaggregation and specialized accelerators. We propose the sub-operators framework that allows full exploitation of the heterogeneous hardware capabilities and describe how it changes the whole stack based on our experience.

C 1: Optimized partitioned multi-threaded filters (addresses R1, R3) The presented filter variations adopt the idea of partitioning to approximate filters. This leads to a parallelized build phase for all filter variants at the cost of a slightly increased

false-positive rate. We present the optimizations and how they affect the four most relevant approximate filters (Bloom, Cuckoo, Morton, and Xor) in Section 2.4.

C 2: Four-dimensional analysis of approximate filters (addresses R1) We compare the approximate filters in four key dimensions, false-positive rate, space consumption, build/lookup throughput, to recommend the best filter for different scenarios. The comparison includes bloom and fingerprint filters and looks into optimizations like partitioning and vectorization. Section 2.5 helps to find the most suitable filter and its configuration parameters by analyzing the key dimensions in-depth.

C 3: Fully integrated Radix-Partitioned Join (addresses R2) We fully integrate a radix-partitioned hash join into our main-memory centric DBMS Umbra [140] as explained in Section 3.4. To the best of our knowledge, this is the first implementation of a radix join in a DBMS, using data-centric code generation [96]. Additionally, we embed a Bloom-filtered semi-join reducer that significantly reduces materialization overhead for certain queries. The join integrated into Umbra is on par or even outperforms stand-alone related work implementations and can thus be used to compare both variants inside of a system.

C 4: In-depth hash join comparison (addresses R2, R3) We compare the bloom-filtered radix join implementation within Umbra [140] against a state-of-the-art hash join [108, 113, 132] using extensive microbenchmarks and the full TPC-H benchmark. Section 3.5 presents experimentally validated values for workload characteristics needed to observe any performance benefits when using radix-partitioned joins and answers the question of whether implementing partitioning pays off.

C 5: Sub-Operators as Building Blocks (addresses R4, R5) Sub-operators provide a holistic system architecture to split up and optimize data processing tasks on different hardware targets. They are reconfigurable and can be composed into relational and iterative dataflows while being future-proof for resource disaggregation and heterogeneous hardware. Chapter 4 introduces a notion of data flows using a sample set of sub-operators, which can serve as a basis for building a reconfigurable general-purpose query engine. The findings of this chapter also help the ongoing transition of database engines to heterogeneous cloud systems or non-x86 compute platforms.

Conclusion and Outlook Finally, we conclude the dissertation in Chapter 5 with a summary of the findings and an outlook on how the presented results will change the upcoming research and how they already influenced the community.

CHAPTER 2

Partitioned Approximate Filters

*Excerpts of this chapter have been published in [179].
With contributions from Tobias Schmidt.*

With today's data deluge, approximate filters are particularly attractive to avoid expensive operations like remote data/disk accesses. Among the many filter variants available, it is non-trivial to find the most suitable one and its optimal configuration for a specific use-case. We evaluate our open-source implementations for the most relevant filters (Bloom, Cuckoo, Morton, and Xor filters) and compare them in four key dimensions: the false-positive rate, space consumption, and build/lookup throughput. Each filter can switch on and off the same tuning knobs to allow an apples-to-apples comparison, e.g., choose different hash functions or vectorize the implementation.

We improve upon existing state-of-the-art implementations with a new optimization, radix partitioning, which boosts the build and lookup throughput for large filters by up to 9x and 5x. Our in-depth evaluation first studies the impact of all vectorization and partitioning separately before combining them with optimizations like addressing to determine the optimal filter for specific use-cases. While register-blocked Bloom filters offer the highest throughput, fingerprint filters, especially the Xor filters, are best suited when optimizing for small filter sizes or low false-positive rates. For more dynamic workloads that delete keys, Cuckoo filters offer the best performance.

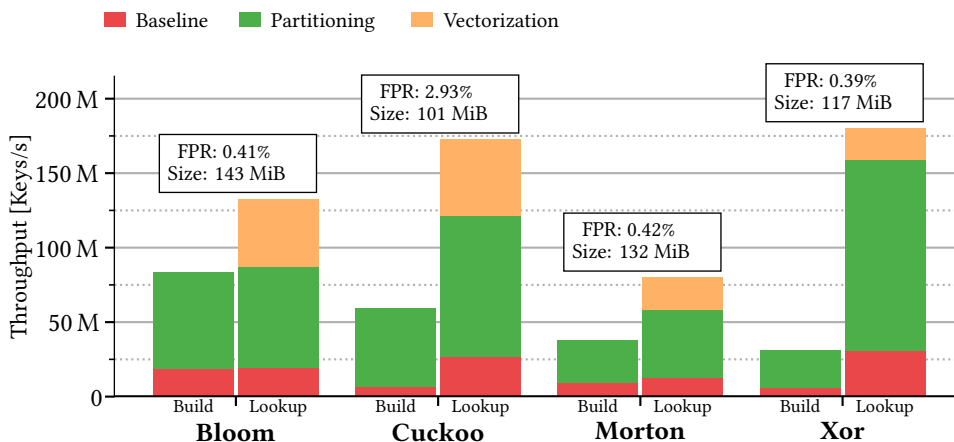


Figure 2.1: Speedup gained by optimizing insert and lookup operations for 100 M keys.

2.1 Motivation

As the volume of generated and processed data increases [169], *efficient access to only the relevant items* is necessary. The goal is both to achieve good performance for the executing workload and to reduce overall pressure on data movement channels by only loading necessary data from storage or over the network.

In this context, approximate filters are particularly useful as they compactly represent the membership of elements in a set, however, at the cost of having false positives. More specifically, the filter always reports contained items as members, i.e., there are no false negatives. For items that are not in the set, the filter returns incorrect results with a certain probability, the false-positive rate ϵ . Small filters can fit in a higher level of the storage (memory) hierarchy, leading to faster access times and lower bandwidth consumption, putting less pressure on the rest of the system’s resources.

It is therefore not a surprise that filters are often used to speed up applications. Log-structured merge (LSM) trees, for instance, check the filter before fetching a page from disk [120]. In databases, filters improve query execution through selective join pushdown, which drops tuples not needed for probing early in the pipeline [109]. Other applications include distributed joins or network applications, where filters reduce the amount of transferred data [34, 105].

We distinguish between two filter families: Bloom filter variants and fingerprint filters. The Bloom filter accesses several bits in a bitmap on lookup or insert [25] and is the most popular filter today [121]. However, fingerprint filters have recently emerged, which store small *signatures of the key* in a hash table-like structure. They are smaller in size and have lower false-positive rates than Bloom filters, at the cost of higher access latencies. Some of the more notable fingerprint filters are the Quotient [22], the Cuckoo [62], the Morton filter [33], and more recently, the Xor filter [74].

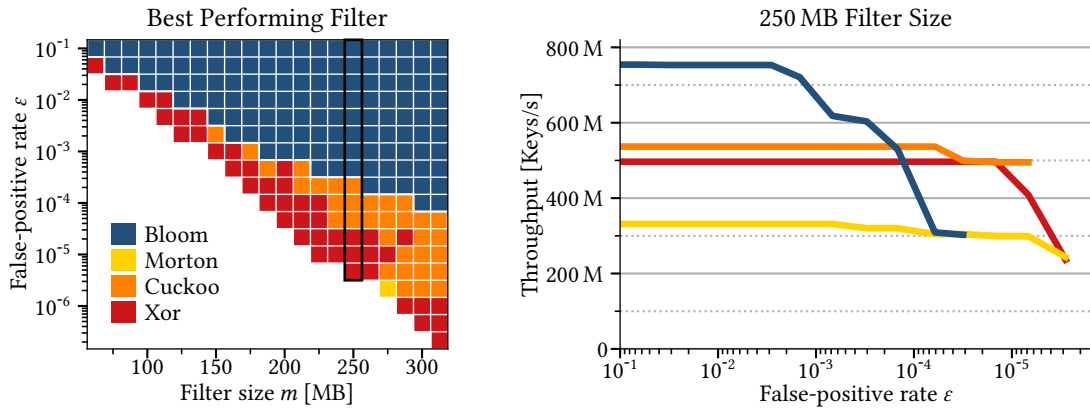


Figure 2.2: Performance with 100 M elements (10 threads).

With this plethora of available alternatives, it is unclear *which filter* to use *when*. Very often, one has to consider multiple dimensions that are relevant to the use-case in mind. Thus, in this chapter, we evaluate the four most promising filters — Bloom, Cuckoo, Morton, and Xor filters — on the following four key dimensions:

False-positive rate (FPR): it affects the application’s performance and hints at the extra bandwidth overhead on shared I/O resources.

Space consumption: we want to minimize the precious space in caches/DRAM to store auxiliary data structures.

Lookup performance: it directly affects the performance of the application.

Build performance: the time it takes to construct the filter.

All four aspects are closely interlinked, and improving one dimension may result in a decline in another (e.g., reducing the FPR may necessitate an increase in size). Which dimension to prioritize when choosing the most suitable filter is application-specific. On the one hand, LSM-Trees primarily aim to reduce the FPR to avoid unnecessary expensive I/O operations while limiting the memory assigned to the filters [51]. On the other hand, an in-memory join cares more about lookup performance than filter size.

Unfortunately, a fair comparison between the filters is not possible today, as most authors introduced different optimizations in their implementations, like vectorization or specialized addressing schemes [74, 109, 205]. We therefore evaluate the *open-source implementations* which integrate all relevant optimizations we provide in [179] for all filters. Furthermore, we apply a new optimization, *radix partitioning* [28, 181], that considerably increases build/lookup performance for all filters when their size exceeds the last-level cache. Figure 2.1 shows the performance of state-of-the-art baselines

compared to the acceleration we get with vectorization and partitioning. In addition to the significant boost of build/lookup throughput, partitioning also allows us to easily parallelize the construction of all filter types, bringing further performance improvements. However, as not all optimizations are always applicable and usually entail tradeoffs, we also investigate *when* to apply *which optimization*.

In the second part of the chapter, we compare the performance of several Bloom filter variants and the three fingerprint filters. Figure 2.2 (left) shows which filter performs best for a given FPR and filter size. Figure 2.2 (right) shows how each filter performs for a fixed size. Here, an LSM-Tree would prefer one of the fingerprint filters (e.g., Cuckoo or Xor) as they offer low FPRs with good performance under a strict memory budget. In contrast, in-memory joins would favor the Bloom filter, as it achieves twice the performance and the cost of a false positive is rather inexpensive [109]. Cuckoo and Xor filters use the given memory best and achieve significantly lower FPRs than Bloom filters.

The rest of this chapter is organized as follows. We first give a brief overview of prior work before giving a practical database example while introducing the filters in Section 2.3. In Section 2.4, we describe vectorization and radix-partitioning we applied and evaluate the impact of each optimization on the baseline. Finally, in Section 2.5, we evaluate the filters for the four key dimensions and propose guidelines for choosing the right filter and which optimizations to enable. We summarize the findings in Section 2.6 and conclude in Section 2.7.

2.2 Related Work

The Bloom filter dates back to 1970 [25], and since then, more than 60 variants have emerged [121]. A substantial portion of these variations extends functionality to add features like key deletion [46, 63, 177], filter resizing [6, 78], or they tailor the Bloom filter to specific applications, such as network scenarios [67, 133, 176]. Nevertheless, augmenting functionality often leads to trade-offs, like reduced throughput, increased filter size, or higher false positive rates. This chapter instead focuses on variants that improve performance through optimizations, including blocking [165] or sectorization [119].

Fingerprint filters improved the space efficiency of the counting Bloom filter [63], which supports deletions but required twice the size of the vanilla Bloom filter [30, 203]. Recently, fingerprint filters resurfaced with Quotient and Cuckoo filters, which improve the existing idea using cuckoo hashing which leads to increased throughput compared to the original Bloom filters. Based on this work, the Morton filter [33] improves the space efficiency by compressing the values in the buckets. Our analysis of fingerprint filters omits the Quotient filter since several publications came to the conclusion that it is suboptimal compared to Cuckoo and Morton filters [62, 152, 205].

The Xor filter is a different variation of fingerprint filters [74]. It promises a very small filter size with high lookup performance and low false-positive rate but is immutable after construction.

Given the plethora of filter alternatives, the discourse has fueled extensive research over the years. The key parameters of interest are versatility (e.g., deletions), false-positive rate (FPR), space consumption, build, and lookup performance.

The filter variants mentioned before each improve against their alternatives in at least one parameter. However, when introducing a new filter, the comparisons naturally tend to spotlight the advantages of newly introduced data structures and their optimizations. For instance, Breslow et al. [33] focussed on how batching enhances the Morton filter's performance but did not extend this approach to the Cuckoo filter. Graf et al. [74] highlighted the FPR and lookup performance of their Xor filter without looking in detail at the build performance. Lang et al. [109] perform an extensive Bloom and Cuckoo filter comparison, though their performance-oriented analysis predominantly focused on FPR and lookup throughput, neglecting aspects like memory footprint or excessive I/O bandwidth utilization. In contrast, the following comprehensive analysis includes all relevant optimizations and filter families for an apples-to-apples comparison.

2.3 Approximate Filters

This chapter gives a brief background of approximative filters by putting them in the context of a database system. We start with a concrete use-case of approximative filters in Umbra [140], which is similar to the semi-join reduces from our work on bloom-filtered radix-partitioned joins [17].

Next, we introduce all tuning knobs we considered [179] and give an intuition of how the filters work to lay the foundation for the extensive evaluation in the following chapter.

2.3.1 Filters in DBMS

Umbra uses approximate filters in the critical path during join processing, as outlined in Section 3.4.7. Figure 2.3 shows a simplified query plan for joining three tables and zooms into a hash join. It visualizes all pipelines where the data is processed chunk by chunk using morsel-driven parallelism as described in Section 3.4.5. Since the pipelines, which make up the data-centric code-generating system, do not have intermediate materialization, we can process each tuple only once. Thus, the build side first collects all tuples and materializes them pre-sorted in result-collectors. After the whole build side is processed, the number of tuples is known, and an appropriately sized hash table is generated and filled with the collected tuples.

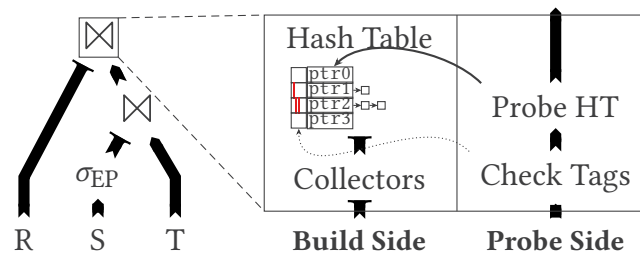


Figure 2.3: Hash Joins in Umbra

The probe side performs lookups without intermediate materialization and directly accesses the hash table in a tight loop. Thus, we can perform multiple hash joins in a row without intermediate buffers, which improves code locality, keeping the tuple values in processor registers. However, the random access introduced by the hash join (c.f. Section 3.4.7) is costly.

Storing Approximate Filters in Pointers

Pointer tagging is a technique to store additional information in the unused bits of the pointer, thereby increasing locality and keeping alignment constraints. In itself, tagging dates back to early IBM computers and is often used in buffer managers, loosely typed languages, or databases [113, 117]. A pointer has two locations to store the tagged bits: In the back, e.g., the last three bits for 8-byte aligned pointers or the first 16 bit when using four-level page tables, which are sufficient to allocate 64 TiB of main memory.¹

We use pointer tags in the main directory of Umbra’s chaining hash table to store a 16-bit bloom filter in the tag, as shown in red in Section 3.4.5. This saves space and allows us to update the pointer, including the tag, with a single atomic compare and swap operation. Thus, before we follow the pointer in the chain, we first check the pointer tag to determine whether the bit for the corresponding element is set, which avoids costly random access [113].

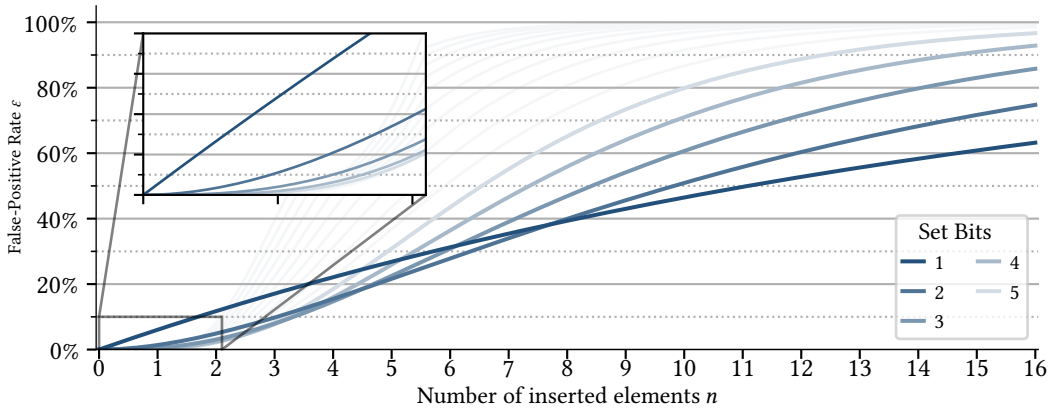
```

1 void* tagPointer(void* ptr, uint16_t tag) {
2     auto taggedPtr = reinterpret_cast<uint64_t>(ptr);
3     taggedPtr = (taggedPtr << 16) | tag;
4     return reinterpret_cast<void*>(taggedPtr);
5 }

```

Listing 2.1: Tagging the lower bits of a pointer

¹While the Linux kernel already has support for 5 levels, it is mostly disabled <https://lwn.net/Articles/717293/>

Figure 2.4: False-Positive Rate of for Bloom Filters with $m = 16$

Our implementation tags the least significant bits of the register by shifting the actual pointer 16 bits to the right to accommodate the tagged bits. Thus, to use the pointer, we must shift it to the right, which moves and cancels out all tagged bits. We prefer tagging the rightmost bits after shifting since this allows us to check the tag directly without further shifting operations.

During the probe phase, Umbra checks the tag first for every tuple, as shown in Figure 2.3. This inexpensive check significantly reduces the number of unnecessary random access operations because it avoids further random access in most cases. The original implementation features a bloom filter that only sets one bit, which leads to a false-positive rate of $\frac{1}{16}$ when having two elements since the bits are set individually from each other.

Beyond using the bloom filter for speeding up only this instance of the hash join, we can also introduce an early probe operator [17, 113] lower in the query plan. This approximative filter may reduce the build side of another hash join and thus speed up the overall query processing. In that case, the bloom filter check for the hash table may be omitted depending on the selectivity. Thus, we can further push down approximative selective predicates in the query tree.

Number of set bits

The false-positive rate ϵ of naïve bloom filters [25] can be estimated based on filter size m , the number of set bits k , also called hash functions, and the number of inserted elements n [31, 179].

$$\epsilon(k, n, m) = \left(1 - \left(1 - \frac{1}{m}\right)^{k \cdot n}\right)^k \approx \left(1 - e^{-\frac{k \cdot n}{m}}\right)^k. \quad (2.1)$$

Figure 2.4 shows the estimated false-positive-rate of different set bits k . When setting fewer bits, on the one hand, the bloom factor performs worse for a low number of inserted elements, but as fewer bits are set, it still is selective for a higher number of tuples. On the other hand, when setting more bits, the bloom filter is very selective for small n and saturates fast, as plenty of bits are set. In our context, with adaptively sized hash tables and an expected load factor strictly lower than 1, we can set multiple bits since short chains are prevalent. Using the known filter size and the number of inserted elements, we can minimize the false-positive rate ε by $k = \ln 2 \cdot (m/n)$.

```

1 // 16 available bits for tagging
2 static constexpr unsigned tagBits = 1u << 4; // 16
3 // set 3 bits for each entry
4 uint64_t computeTag(uint64_t hash) {
5     uint64_t tag = 1ull << ( hash      & (HT::tagBits - 1));
6     tag         |= 1ull << ((hash >> 4) & (HT::tagBits - 1));
7     tag         |= 1ull << ((hash >> 8) & (HT::tagBits - 1));
8     return tag;
9 }
```

Listing 2.2: Naïvely generating 3 Bit tag

Setting more bits, however, does not only have advantages since it improves the filter quality at the cost of more assembly instructions and more bits from our hash value needed, as shown in Listing 2.2. For each bit, we need to take 4 bits from the hash value and use them to set a bit in the tag. Since this is a non-trivial tradeoff, we investigate in the following how this filter variant and other approximate filter variants perform. The following section describes the variants first to give an intuition for how approximate filters work.

Table 2.1: Common configuration parameters for approximate filters (cited from [179])

Symbol	Description
n	Number of elements to insert into filter.
k	Number of bits to check / set (Bloom filter); Fingerprint size in bits (Cuckoo, Morton, Xor filter)
s	Memory scale factor: $m = k \cdot n \cdot s$.
B	Block size in bits (Bloom filter).
W	Sector/word size in bits (Bloom filter).
z	Number of groups per block (Bloom filter).
b	Fingerprints per bucket (Cuckoo, Morton filter).

2.3.2 Configuration Parameters

We use a set of similar configuration parameters for all filter families to make the comparison as simple as possible. Both families use k for the number of bits stored in the filter, called the number of hash functions for bloom filters and fingerprint size for fingerprint filters. Furthermore, we use n to denote the number of elements to insert and calculate the minimum number of bits $M = k \cdot n$. To accommodate that inserts into a fingerprint filter can fail, we over-allocate memory by a factor of s and use $k \cdot s$ bits per key.

Table 2.1 summarizes all common parameters and lists the most important parameters for bloom and fingerprint filters. One of the biggest differences between the two filter families is the influence of the parameters on the false-positive rate. In Bloom filters, the false-positive rate decreases with a size factor s , which has a negligible impact on the fingerprint filters FPR. The size factor s instead mainly determines whether the fingerprint filter can be built and the fingerprint size k influences the false-positive rate.

The following sections provide a brief overview of the two filter families. A more extensive description and evaluation of their implementation and integration into our library can be found in the corresponding paper [179].

2.3.3 Bloom Filters

Bloom Filters revolve around setting bits in a large bit vector depending on the hash value of the tuple. The number of set bits called k in this chapter, is also known as *number of hash functions*. We achieve different false-positive rates depending on the number of set bits, as shown in Figure 2.4.

Thus, the configuration is always a trade-off between space consumption, the number of inserted elements, and the false-positive rate. In general, the bloom filter's false-positive rate degrades the smaller the filter is, or the more elements are inserted. Setting more bits makes the filter more selective for a few inserted elements but saturates the filter faster, leading to worse false-positive rates with more elements contained in the filter. The optimal s (set bits per key) is 1.44 ($1/\ln(2)$) for a naïve bloom filter [133].

In the following, we describe three bloom filter variants we analyzed. While they still set k bits per element, they differ in how they distribute the bits in the filter. Some of the variants put the bits closer together in memory to optimize for faster build and lookup time at the cost of worse space efficiency.

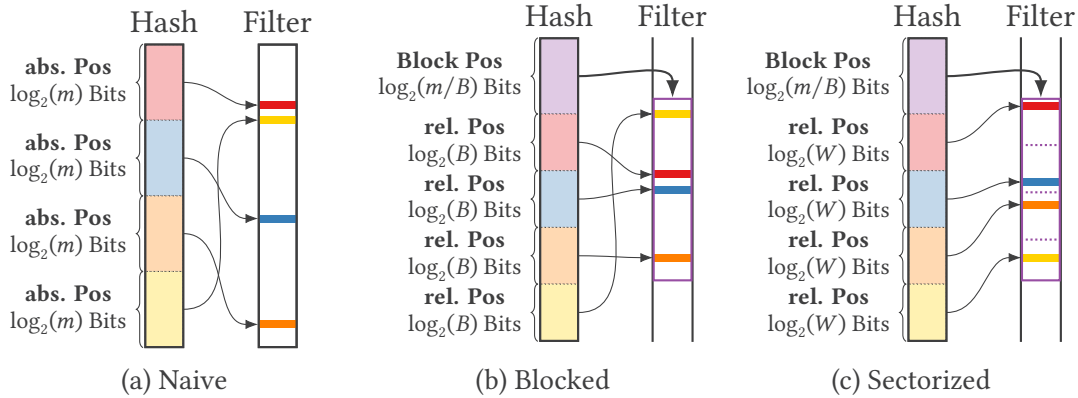


Figure 2.5: Overview of Bloom Filter variants

Naive

In the naïve bloom filter, all bits are statistically decoupled from all other set bits. Thus, it has the best space efficiency of the variants. This complete decoupling, however, impacts performance, as each additional set bit leads to more computational effort and consumed hash bits, and for large filters, to cache or even TLB misses.

Figure 2.5a shows the inserts of four color-coded bits into the filter. Each set bit consumes $\log_2(m)$ bits from the hash function to determine its absolute position in the filter. Thus, the naïve variant needs $k \cdot \log_2(m)$ bits per inserted element, which is the highest amount for all variants but also maximizes the false-positive rate at a given filter size.

Blocked

Blocked Bloom filters operate on blocks, sized B bits each [165], as shown in purple in Figure 2.5b. The block size has to be a power of two for performance reasons. The blocked filter first selects the block to operate in and then only relatively addresses the remaining bits in the selected block.

Two different flavors of this filter are prevalent, register-sized (mostly 64 Bit) and cache-line-sized (512 Bit) blocked filters. Both concentrate all lookup and insert operations on a single cache line introducing only one random access. The set bits for the register-blocked and cache-line-blocked filters can be pre-generated in a normal or AVX512 register. This reduces the memory operations when accessing or altering the filter to one since we can change all bits at the same time.

The false-positive rate is coupled to the block size and worsens with smaller blocks since they reduce the number of possible positions for the individual bits [179]. The smaller the block, the more likely bits overlap, and thus the false-positive rate degrades.

Sectorized

Sectorized Filters operate similarly to blocked bloom filters and introduce another constraint [100, 105]. They separate each block of size B in sectors of size W and ensure that one bit is set in each sector, as illustrated in Figure 2.5c.

This restriction ensures that all bits are distributed evenly across the sectors and simplifies probing and setting the bits in parallel using SIMD instructions. Sectorized filters perform slightly worse in terms of false-positive rate than blocked filters of the same size since the sectors further restrict the bit placement [179]. However, they need fewer bits from the hash function, and all bits can be set individually from each other.

Cache-Sectorized filters loosen the restriction by adding groups between the block and the sector [109]. After finding the block, we select a block in each of the z groups to set the bits in. This additional grouping ensures that the number of set bits k can now be a multiple of z instead of W for the normal sectorized variant.

2.3.4 Fingerprint Filters

Fingerprint filters are hash sets where only fingerprints of size k are stored instead of the complete hash value. Thus, we accept more collisions due to these non-exact hash matches to reduce space consumption. In contrast to bloom filters, the filter size has a minor influence on the false-positive rate. It's influenced by the fingerprint size k instead. If the filter size is too small, the fingerprint filter build may be unsuccessful. Cuckoo and Morton filters can be updated like hash sets, while Xor filters are constructed beforehand and thus static.

Cuckoo Filter

Cuckoo Filters share great similarities with cuckoo hash tables [150]. For each tuple, we compute two candidate buckets, each holding up to b elements. Each lookup then needs to check both buckets to find the key. During insert, when there is still space in one of the buckets, we insert the key. Otherwise, we will relocate the occupying items to their alternative buckets until we make space for our new tuple. One can generate the alternative bucket address from the current bucket address and the contained fingerprint. In case relocating is unsuccessful, the insert fails.

Instead of the complete hash and a pointer to the payload, the cuckoo filter stores only fingerprints of size k , which are generated from the hash values [62]. Like in the cuckoo hash table, the insert to the cuckoo filter can also fail. Increasing the number of fingerprints per bucket b or the filter size increases the likelihood that the filter can be built successfully. Thus, the false-positive rate is mainly influenced by the fingerprint size k and the number of fingerprints per bucket b . Naturally, the FPR

increases with longer fingerprints and decreases with more fingerprints stored per bucket [179].

Morton Filter

Morton filters [33] apply rank-based compression on the buckets of the Cuckoo filter, combining multiple buckets into a block. All bucket's tuples are stored densely after each other along with a counter array, storing how many tuples each block contained. Using the prefix sum of the counter array, we can compute the offset for the lookup. An additional overflow tracking array logs whether one of the tuples was remapped to the alternative bucket. In case there is no bit set for the current bucket, the lookup only needs to test one bucket instead of two in the cuckoo filter.

Xor Filter

Xor Filters [74] also fingerprint filters but are not based on hash sets. The core idea behind Xor filters is to combine l , typically three, values using bitwise XOR operations to store information about element membership. In case, the fingerprint is contained in the filter, the result equals the fingerprint. Otherwise, the result is undefined, and the likelihood of a random match (false-positive) is 2^{-k} . In this chapter, we look at two different construction algorithms integrated in the library [179]: The original Xor Filter [32] and a new fuse graph algorithm [54].

2.4 Implementation and Optimizations

We evaluate the filters presented in the previous section using a C++ implementation, which exploits compile-time optimization where possible [179]. Most parameters, like fingerprint size k , the number of fingerprints per bucket b , or the parameters for sectorization are realized as compile-time constants to avoid runtime-overhead by specialization. The filter's size m , determined by the scale factor s and the number of elements n , is passed at runtime.

Prior work analyzes different hash functions for addressing [62, 109] or use SIMD-operations to vectorize their filter implementation [33, 109]. Our C++ implementation integrates all these optimizations [179] and combines them with partitioning. This novel variant improves build and lookup performance for large filters, maintaining nearly the same false-positive rate and space consumption.

We choose representatives for each filter to evaluate the different optimizations. We use the following configurations [179]:

- **Bloom:** 512-bit blocked Bloom filter ($s = 1.5$).

- **Cuckoo:** the Cuckoo filter with 4 fingerprints per bucket ($s = 1.06$).
- **Morton:** the Morton filter with 3 fingerprints per bucket and an 8-bit OTA ($s = 1.38$).
- **Xor:** the original Xor filter ($s = 1.23$).

All filters use the optimal values for scale factor s [179] and $k = 8$ for set bits, respectively, fingerprint size. We perform all benchmarks using an Intel i9-9900X CPU (Skylake-X, 3.5-4.4 GHz) with 10 cores and 64 GB of main memory, running Ubuntu 20.10 (Kernel 5.8, gcc 10.2), and repeat all measurements five times. For scalability measurements, we use an Xeon Gold 6212U (Kaby Lake, 2.4-3.9 GHz) with 24 cores and an AMD Ryzen 3950X (Zen2, 3.5-4.7 GHz) with 16 cores.

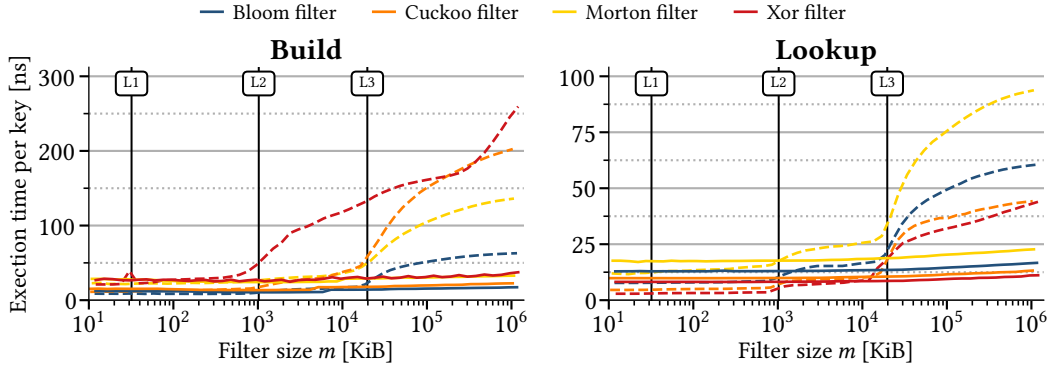
2.4.1 Partitioning

The performance of all filters deteriorates as the number of elements increases. For filters that do not fit into the last-level cache (LLC), lookups are up to one order of magnitude slower than in filters that fit into the L1 cache due to random memory accesses that miss the cache. Prior implementations reduced the number of accessed cache lines by blocking, but even that requires at least one random memory access per operation. We propose, instead, to partition the filters.

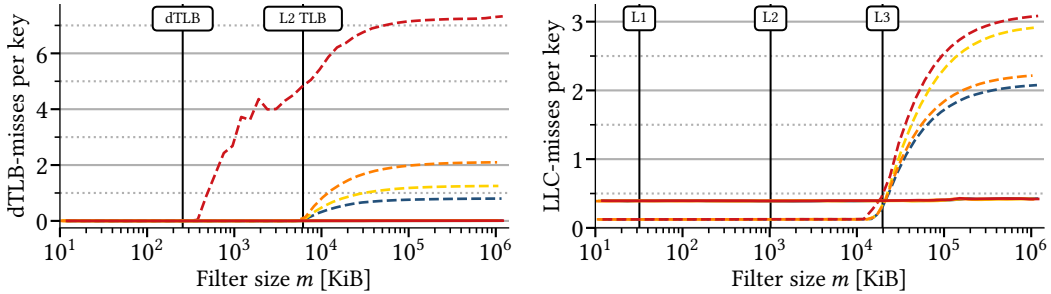
Inspired by radix joins [181], we use radix partitioning to divide the set of keys before building the filter or performing lookups. For construction, one filter is built for each partition. Lookups first determine which filter to use before testing the keys in the respective partition. If the filter fits in the caches, fewer TLB and cache misses occur. We optimize our single-pass radix partitioning implementation using software write-combine buffers and non-temporal streaming stores [207].

Figure 2.6a shows the time per key needed to build the filter and perform a lookup. The partitioned filters (solid lines) include the partitioning time. The partitioned variants outperform the baseline (dashed lines) by around 1 MiB. The Xor filter benefits earlier from partitioning due to its memory-consuming construction algorithm. Without partitioning, the runtime deteriorates due to TLB and cache misses; partitioning keeps the time needed to insert and lookup a key almost constant. The Cuckoo and Xor filters, in particular, benefit from this optimization. Their build times are nearly 10x faster for filters that exceed the LLC. They perform more random accesses than the Bloom and Morton filters and benefit more from the increased spatial locality.

For all four filters, partitioning reduces the overall number of TLB misses by three orders of magnitude, and the number of last-level cache misses by almost one order. Figure 2.6b shows that almost no data TLB misses occur for partitioned filters. At first, partitioning increases the number of LLC misses (cf. Figure 2.6c), but in exchange, the



(a) Construction and lookup times.



(b) Data TLB misses (Build)

(c) LLC-misses (Lookup)

Figure 2.6: Build and lookup performance for partitioned (solid lines) and non-partitioned (dashed lines) filters.

number remains constant even for filters exceeding the L3 cache. Unoptimized filters incur several misses per key as soon as the filter exceeds the LLC.²

While applying radix partitioning in Bloom filters is straightforward, we experienced difficulties with fingerprint filters. More specifically, constructing the Cuckoo or the Morton filters tends to fail for more than 512 partitions. We, therefore, also use the Xor filter’s seed-based retry technique for hash table-based filters with partitioning. If building the filter fails, we xor the keys with a seed and try again.

In Figure 2.7, we show the optimal number of partition bits to use. The Bloom filter shows a clear picture. As soon as the filter size exceeds the L2 cache, partitioning pays off for both building and probing the filter. As anticipated, the optimal number of partitions grows with increasing filter size. Partitioning even improves the performance of fingerprint filters with smaller filter sizes. The Xor filter benefits much

²The hardware prefetcher causes additional cache misses for the blocked Bloom filter and the Morton filter.

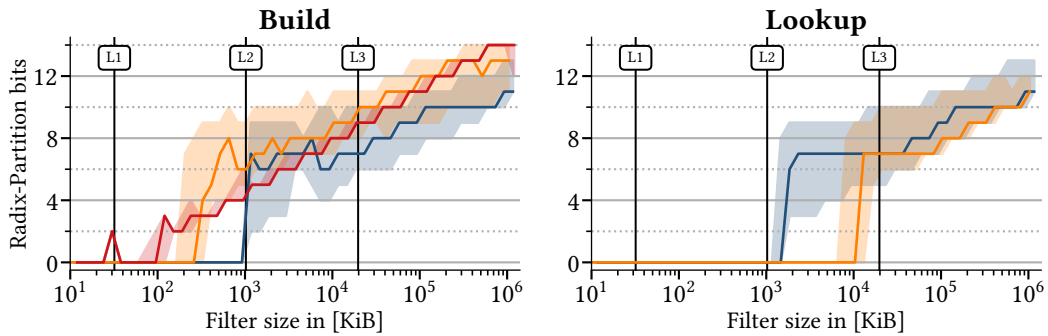


Figure 2.7: Throughput-optimal number of partitions. The colored area shows throughput deviations of less than 5%.

sooner from a partitioned build process. For lookups, all fingerprint filters perform similarly, thus we only show the Cuckoo filter. Figure 2.7 also shows that the precise number of partitioning bits does not have a significant impact on the performance. All filters get within 5% of the optimal performance even when choosing a partition size that differs by an order of magnitude.

However, it is important to note that partitioning the filters introduces an additional requirement, namely batching the keys before performing insert or lookup operations. Without partitioning, all filters except the Xor filter support inserting single keys. Although lookups for single elements are still possible, the performance drops by 10%. The reason for this is the additional work needed to determine the correct filter for the key. Nevertheless, partitioning is the most effective technique to guarantee stable performance for filters that exceed the caches, if we can batch the operations. One particular advantage of partitioning is that it does not affect the FPR, unlike other optimizations that minimize the number of cache misses. RocksDB uses partitioning to split its Bloom filters and store them on disk rather than in memory [60]. An additional top-level index loads the correct partition from disk when needed.

2.4.2 Vectorization

As the number of cache lines accessed per lookup cannot be reduced further for blocked Bloom filters, several authors optimize the computations using SIMD instructions [105, 109, 161]. We found two techniques for vectorizing approximate filter structures: parallelizing the computations for one key (vertical vectorization) or performing multiple lookups in parallel by assigning one key to each SIMD lane (horizontal vectorization). While horizontal vectorization can be used with all filters, vertical vectorization only works with the Impala library’s sectorized Bloom filter [105]. We implement horizontal vectorized lookups for all four filters and their variants. In contrast to existing vectorized implementations for fingerprint filters, we support arbitrary fingerprint sizes.

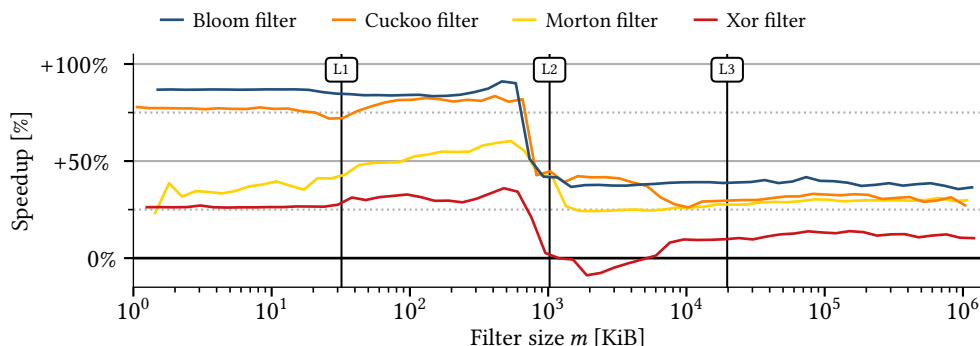


Figure 2.8: Speedup using vectorized filters (AVX512); the baseline are scalar filters (partitioning is enabled).

Our implementations target processors that support the AVX512F and AVX512VL instruction sets.³ We use gather and scatter instructions to implement horizontal vectorization and use masking to avoid branches. Our vectorized filters share most of the code with the scalar implementation. The SIMD instructions are inserted through compiler intrinsics during compilation. In a few cases, such as unaligned memory accesses, the implementations differ: the gather instruction only supports aligned accesses, thus, we have to perform two aligned loads instead of one unaligned load.

Although the number of executed instructions decreases almost eightfold, the vectorized filters are at most twice as fast (cf. Figure 2.8). The vectorized filters spend most of the time fetching data from memory as gather scales only modestly compared to scalar loads. As soon as the filters exceed the L2 cache, the performance of vectorized filters deteriorates due to TLB and cache misses. Partitioning mitigates this effect, but the speedup decreases, as both versions use the same radix partitioning implementation. We also vectorized the construction of the Bloom, Cuckoo, and Xor filter using scatter instructions. However, only the sectorized Bloom filter using vertical vectorization benefits from this optimization.

2.4.3 Multi-Threading

An additional benefit of partitioning is that it simplifies the implementation of task-level parallelism: each thread builds the filters for different partitions and avoids synchronization during construction. We parallelize the radix partitioning as proposed by Balkesen et al. [15] and use the single-threaded algorithms to build each filter. This approach is particularly suitable for fingerprint filters, since synchronizing their construction algorithms is non-trivial. Although it is easier to parallelize the Bloom filters using atomic instructions, partitioning results in less overhead.

³We emulate missing instructions on older platforms and thus expect no performance gain there.

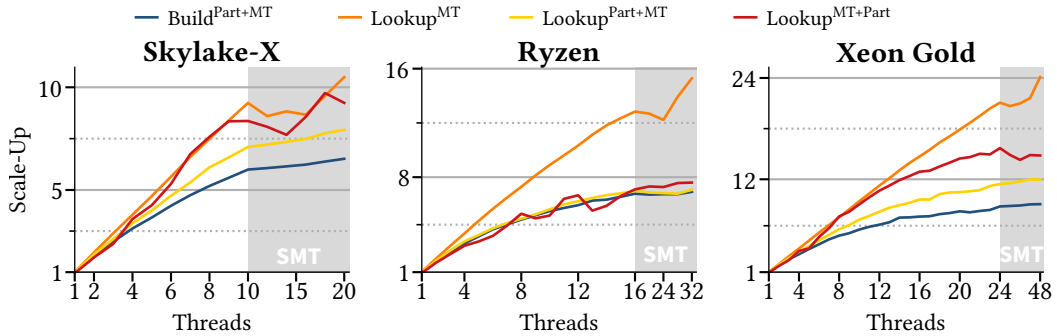


Figure 2.9: Scalability of the blocked Bloom filter on different machines relative to a partitioned version; $\text{Lookup}^{\text{MT}}$ uses the non-partitioned version as baseline. ($n = 100 \text{ M}$)

Table 2.2: Throughput on Skylake-X [Keys/s (scale-up)].

	Bloom	Cuckoo	Morton	Xor
$\text{Build}^{\text{Part+MT}}$	381 M (6.0x)	341 M (6.9x)	235 M (7.2x)	197 M (6.9x)
$\text{Lookup}^{\text{MT}}$	165 M (9.2x)	255 M (9.5x)	113 M (9.6x)	287 M (9.4x)
$\text{Lookup}^{\text{Part+MT}}$	480 M (7.1x)	574 M (6.6x)	368 M (7.6x)	657 M (6.3x)
$\text{Lookup}^{\text{MT+Part}}$	565 M (8.4x)	662 M (7.7x)	342 M (7.1x)	725 M (7.0x)

Lookups, in contrast to insertions, require no synchronization. Once the filter is built, multiple threads can read it simultaneously. In combination with partitioning, two different parallelization schemes are possible: partition the data before assigning one partition to each thread ($\text{Lookup}^{\text{Part+MT}}$) or first split the keys into jobs and then partition them separately ($\text{Lookup}^{\text{MT+Part}}$). The second option has the advantage that no synchronization between the threads is required. However, the spatial locality decreases, since each job accesses the entire filter. The first scheme, in contrast, reads only the part of the filter relevant for the current partition.

Figure 2.9 shows the speedup when building the filters and performing lookups with multiple threads. For construction and lookups with partitioning, we report the numbers relative to the partitioned filter versions. For non-partitioned lookups ($\text{Lookup}^{\text{MT}}$), we use the filter without partitioning as the baseline. The non-partitioned filters scale almost linearly on all three machines. The partitioned filters, on the other hand, scale sub-linearly due to the radix partitioning. The second partitioned lookup variant ($\text{Lookup}^{\text{MT+Part}}$) scales on all three machines better than the first variant ($\text{Lookup}^{\text{Part+MT}}$). Table 2.2 lists the throughput for all four filters using the available hardware threads. Although partitioned filters achieve only sublinear speedup, they are still at least twice as fast as the non-partitioned versions and offer a considerable scale-up for construction.

2.5 Evaluation

We now present an experimental evaluation of our filter implementations, in which we vary all the parameters relevant to the filters. The goal is to identify which filter to use when optimizing for space consumption, throughput, or false-positive rate.

2.5.1 Experimental Setup

We ran all experiments on the Skylake-X machine and used the following filter configurations.

- **Bloom:** All experiments evaluate the naïve, register-blocked, cache-blocked, sectorized, and cache-sectorized variants of our Bloom filter implementation.
- √ **Cuckoo:** We use a performance-optimized configuration that chooses the fingerprints per bucket b depending on k .
- ∧ **Morton:** The number of buckets per block bpb and the OTA size o are powers of two whenever possible and correspond to the configurations from [179].
- × **Xor:** We include both the original and the fuse graph-based method to build the Xor filter.

Parameters & Methodology We benchmark the filter implementations for three different dataset sizes, as shown in Table 2.3, on random data generated by the Mersenne Twister engine from the C++ Standard Template Library. We measure the build and lookup performance on all hardware threads without SMT and use all valid combinations of partitioning and vectorization enabled or disabled. We also varied the memory scale factor s from 4–26 and the fingerprint size k from 1–25. The sectorized and cache-sectorized Bloom filter variants restrict the parameter k and thus have fewer data points respectively. For building the filters, we always enable partitioning to parallelize the construction. To ensure stable results, we report the average performance from five repetitions.

Our in-depth analysis of the filters’ lookup and build performance uses 10 K, 1 M, and 100 M keys by clustering the data by filter size m and false-positive rate. This benchmark finds the filter with the highest throughput for a given FPR and filter size and examines the effects of tailoring one parameter. Besides scaling the parameters s and k , we used all valid combinations of vectorization and partitioning enabled or

Table 2.3: Experiments

Elements	Partitions
10 K	8
1 M	128
100 M	1024

disabled and varied the number of partitions (2^i partitions, $5 \leq i \leq 12$). For every cluster, we report the filter variant with the maximum throughput.

Our experiments resulted in approximately 1 300 data points for each measured filter and size class. In total, we performed about 50 000 measurements for lookups and an additional 35 000 measurements for construct, which cannot use vectorization for unpartitioned builds in the general case.

2.5.2 Lookup Performance

The comparison plot in Figure 2.10 shows how much the best performing of the two filter families differ in lookup time – blue indicates that the bloom filter is faster, red that a fingerprint filter is faster. The gray line divides the measurements into two parts, separating which filter dominates. This line is included in all future plots in this section for reference. Since no filter can reach arbitrarily low FPRs, there are no data points in the graphs’ lower-left halves. In contrast to the other plots, we show fewer data points because the bloom filter cannot reach very low false positive rates, which makes a comparison impossible.

Our results show strong similarities to those of Lang et al. [109]. If higher false positives are acceptable, Bloom filters dominate. Otherwise, if the time penalty for a false-positive is larger, low false-positive rates are more important than the filter lookup time. Thus, fingerprint filters perform better workloads that benefit from low false-positive rates.

In the remainder of this section, we analyze the filters’ false-positive rate and throughput in more detail for 10 K, 1 M, and 100 M elements and evaluate the impact of partitioning. We choose the filter sizes to see how big the difference in performance is when the filter resides in different levels of the cache hierarchy.

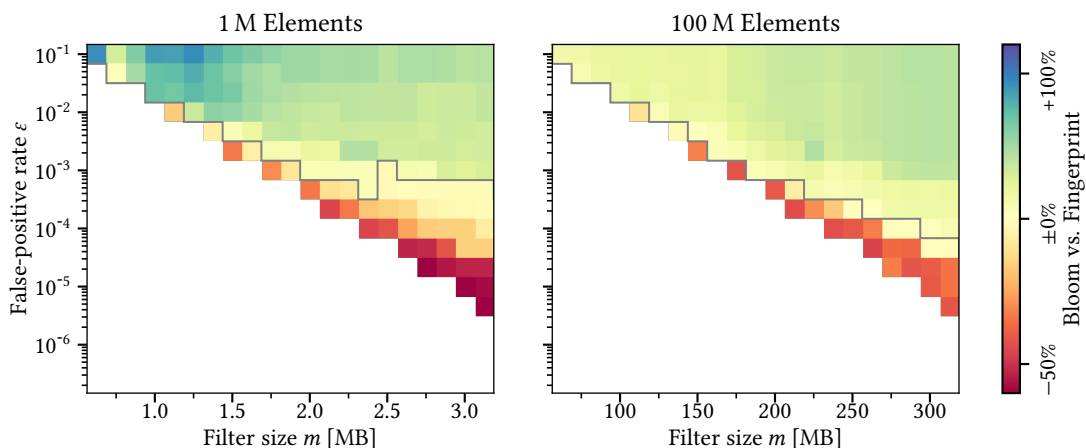


Figure 2.10: Bloom vs. fingerprint filters.

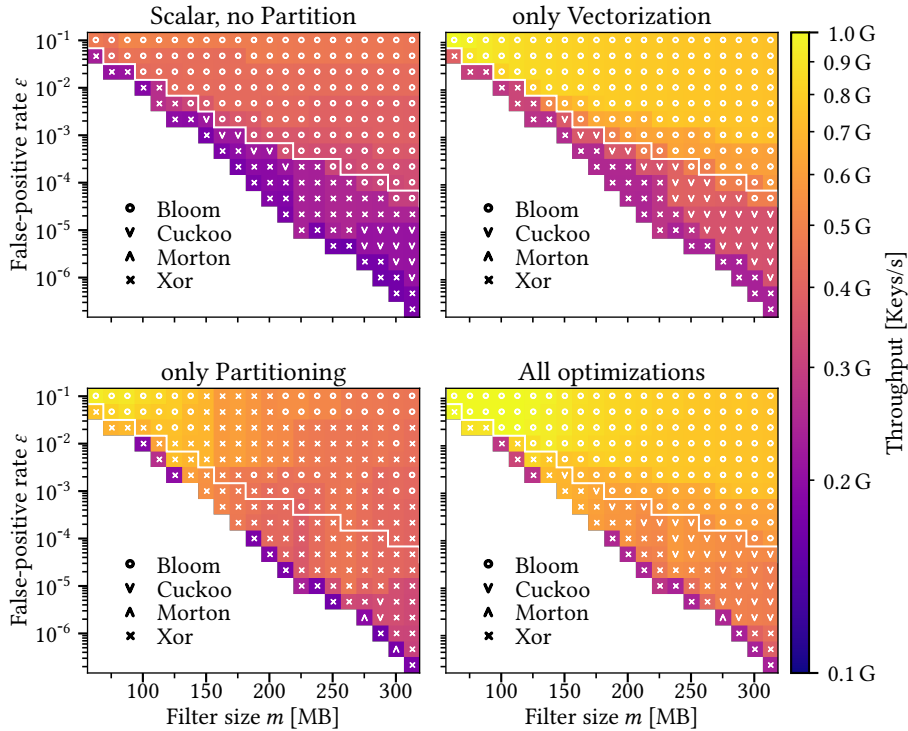


Figure 2.11: Lookup performance for 100 M elements.

Optimal filter variant

Figure 2.11 and Figure 2.12 show the filter with the highest throughput for the given memory budget and the measured false-positive rate. The white line separates Bloom and fingerprint filters with both partitioning and vectorization enabled. Mostly, there is no Bloom filter counterpart reaching a similar FPR as the fingerprint filter, making the fingerprint filter the only option for filters with very low FPRs, as visible in Figure 2.10.

As expected, the overall performance degrades with more indexed elements resulting in increased filter size for a constant false-positive rate. The filters with $10K$ elements are smaller than the L1 cache, so we obtain the highest throughput. For $1M$ elements, the filters only fit into the last-level cache, except for the smallest ones in the top left corner, which still operate in the L2 cache.

We investigate the filters in more detail in a horizontal and vertical slice from the lookup measurements in Figure 2.12 (1 M elements). In the FPR slice (Figure 2.13a), the Bloom filter dominates the performance except for small sizes, where the (partitioned) Xor filter performs best. Only the naïve Bloom filter can attain the false-positive rate using the given space but suffers substantial L1 cache misses even for small sizes due to high k . For a higher memory budget, first the sectorized and then the register-blocked variants take over. In conclusion, larger filters can improve the throughput by using

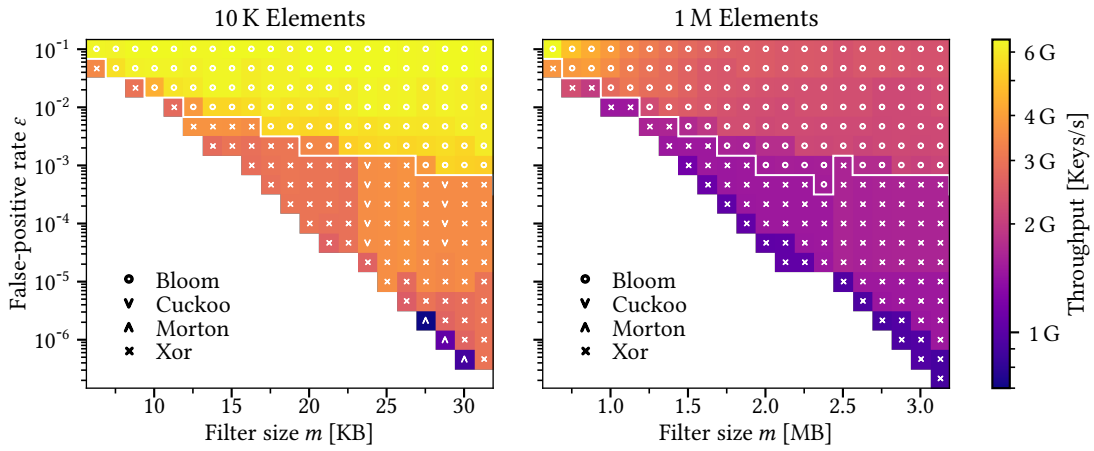


Figure 2.12: Lookup performance and best-performing filter. The white line separates Bloom from fingerprint variants.

optimized variants that increase locality. However, the performance deteriorates for larger sizes as cache misses increase again. Thus, the Xor filter performs best because its random memory accesses are limited to three. The throughput decreases slightly for more than 1.5 MB as the cache misses increase.

When looking at a constant filter size (e.g., 3.0 MB in Figure 2.13b), we see that the fingerprint and Bloom filters perform differently. While Bloom filters can trade a worse FPR for performance, the fingerprint filters’ throughput does not improve for high FPRs. Bloom filters offer more possibilities for performance optimization by clustering the memory loads or reducing the number of hash functions k and, thereby, the

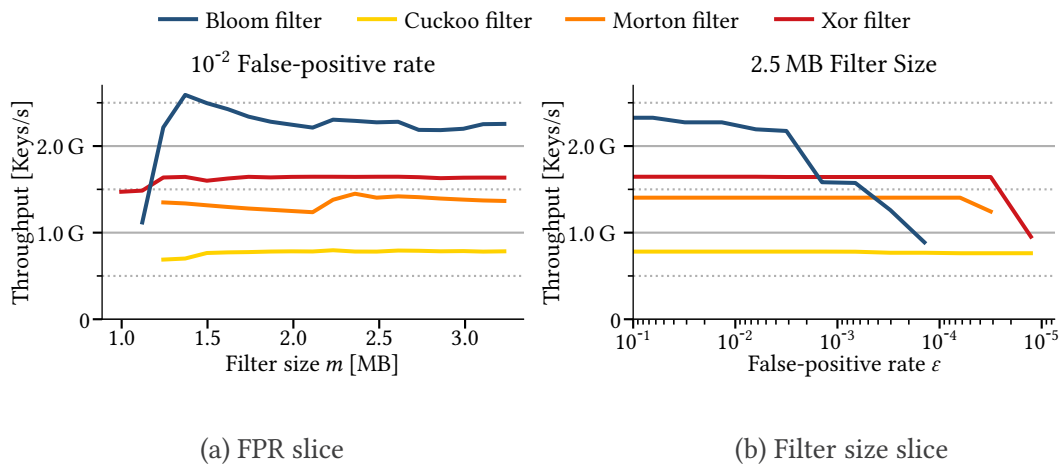


Figure 2.13: Slice from 1 M elements for constant size or FPR.

computational effort at the cost of significantly increasing the FPR. Fingerprint filters, in contrast, can only scale the fingerprint size, which slightly improves performance if a higher FPR is acceptable. Nonetheless, fingerprint filters achieve lower false-positive rates for the given memory budget than the Bloom filters at a modest throughput reduction. This is an attractive trade-off if false positives are expensive, like accessing data over the network or unnecessary disk access in LSM trees, and matches the findings of Lang et al. [109].

For the Bloom filter, partitioning mostly does not pay off. The Xor filter's throughput significantly decreases if partitioning is unavailable and falls behind the Cuckoo filter. Thus, non-partitioned Xor filters are only optimal once very low FPRs are required that the Cuckoo filter cannot attain.

We do not evaluate filters with 10 K elements further as partitioning does not pay off, and the outcomes are similar to 1 M elements.

Recap – Bloom filters offer the highest throughputs but cannot reach low FPRs on a size budget. Fingerprint filters, most notably Cuckoo and Xor, can reach very low FPRs while being a bit slower.

Optimal Bloom filter variant

If opting for high throughput, we next break down which Bloom filter variant performs best in Figure 2.14. Register-blocked filters dominate most of the areas where the Bloom filter performs better than fingerprint filters, as shown in Figure 2.12. Register-blocking trades FPR and size for maximum throughput by reducing memory accesses to a minimum. Consequently, they are the filter of choice for high throughput scenarios like semi-join reducers where false-positives are inexpensive, as shown in Chapter 3. However, blocked bloom filters cannot reach low enough false positive rates, and thus, the other variants dominate the lower part of the figure.

Sectorized and cache-sectorized filters perform second best as they use the entire cache line to reduce the FPR and specialized access patterns to improve performance. The cache-blocked filter offers a slightly better FPR since it does not restrict the set bits' placement. Naïve bloom filters are not competitive in performance but offer by far the lowest possible FPRs at the cost of having a complete random access pattern. For this reason, partitioning improves the naïve filter most as it limits its LLC misses. *Recap* – Bloom filters gradually trade high throughput for better FPRs by decreasing the spatial locality. The range goes from the register-blocked filters that offer the highest throughput to the naïve Bloom filter with the lowest FPR.

Optimal k

Although all filters support arbitrary k for the fingerprint size or the number of hash functions, this flexibility mostly benefits the Bloom filter variants. It directly improves

the performance as fewer memory loads occur and the hash function is evaluated less often. The register-blocked filter typically uses very low k (≤ 4) for maximum performance throughout most of our measurements, as shown in Figure 2.15. Bloom filters mainly regulate their achieved FPR by increasing the number of hash functions k . Thus one can see k increasing from top to bottom, resulting in larger filter sizes as a direct consequence of storing more values to reach a better false-positive rate. For the naïve Bloom filter, this number can increase to 20 when aiming for very low FPRs, resulting in approximately 20 cache misses (without partitioning). However, this is not visible in the figure, as fingerprint filters dominate the lower area.

Fingerprint filters, on the contrary, have an almost constant FPR, and so the value of k does not change that much inside of a column, and we can clearly see blocks of the same k . When scaling up the filter size, we also need to increase the size of the stored fingerprints, which correlates with the x -axis. When using the Xor or Cuckoo filters, powers of two for k perform best and 16-bit fingerprints, in particular, offer a good trade-off. These sizes simplify fingerprint comparison in the vectorized implementations and allow for aligned loads, reducing LLC misses.

Recap – The fewer the number of set bits in a bloom filter, the higher the performance for Bloom filters. In fingerprint filters, choose a power of two as fingerprint size for memory alignment.

Vectorization

The vectorized lookup implementations improve the performance of all filters, as shown in Figure 2.16, which compares the best-performing vectorized and non-vectorized filters. We included the same dividing line as in Figure 2.11 for reference. As expected, we observe the biggest performance boost for small filter sizes that fit

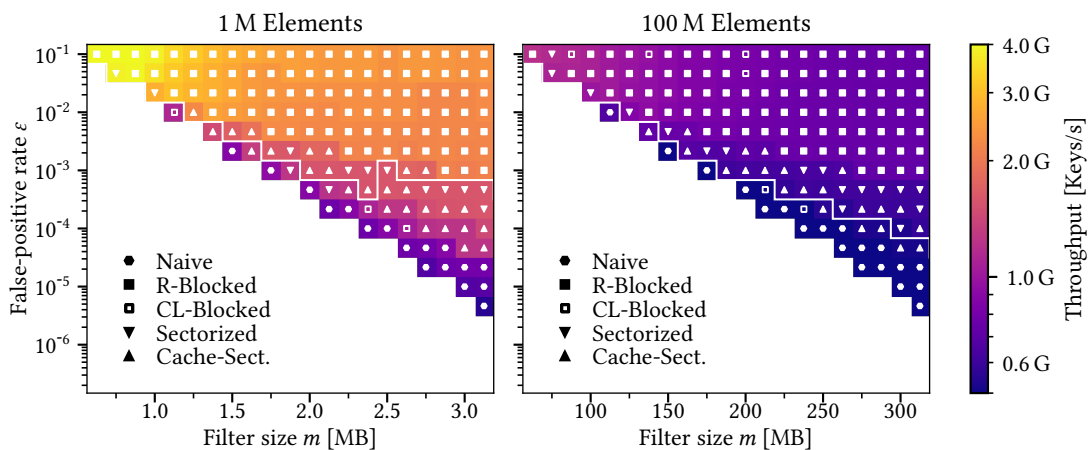


Figure 2.14: Best-performing Bloom filter variant for lookup.

into the L2 cache (cf. Section 2.4.2).

Generally, the register-blocked Bloom filter (cf. Figure 2.14) benefits the most from vectorized lookups. It performs all computations in SIMD registers and loads the required data with a single gather instruction from memory. Although the other blocked and sectorized Bloom variants access only words on the same cache line, they still perform multiple memory loads and therefore benefit less from this optimization.

The Xor filter performs the most random memory accesses and, thus, is at most 50 % faster. The performance of the Cuckoo filter, which accesses at most two random memory words, almost doubles. Without vectorization, the Xor filters dominate large areas for 100 M (cf. Figure 2.11). However, the difference in throughput is mostly less than 20 %, and vectorized Bloom and Cuckoo filters take over.

Recap — Vectorization pays off without impacting the FPR. It boosts all filter implementations and should be applied in any setting that allows concurrent probing of multiple elements.

Partitioning

Since we observed substantial performance improvements with partitioning in Section 2.4.1, we now compare the lookup performance of the fastest partitioned and non-partitioned filters, as shown in Figure 2.17. However, this time the baseline has vectorization and multi-threading enabled, which partly amortizes the achieved speed-up. The two right-hand plots in Figure 2.11 show the fastest filter implementation for 100 M keys. We included the same dividing line for reference.

The tipping point at which partitioning begins to provide benefits is at 1 M elements. The filter that dominates performance in most cases — the register-blocked Bloom filter (cf. Figure 2.14) — benefits the least from partitioning since it already minimizes

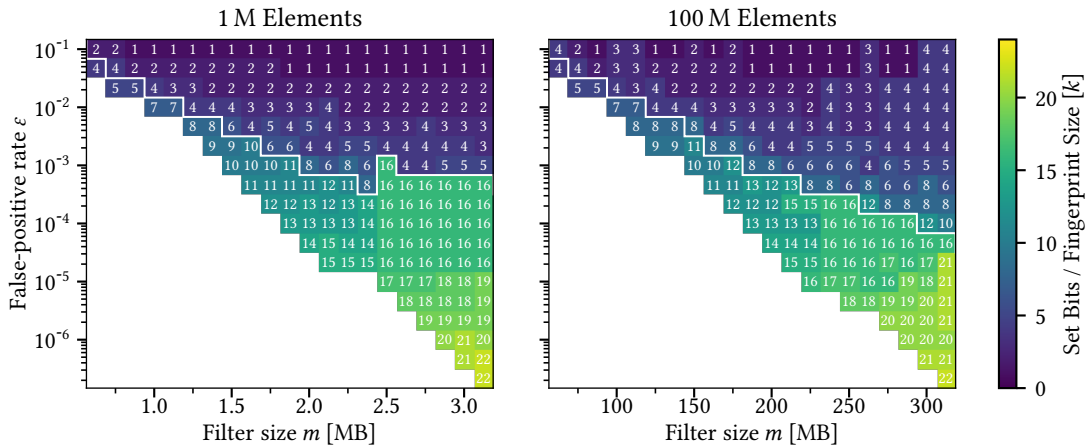


Figure 2.15: k of best-performing filter

memory accesses. Furthermore, we hide the cache miss latencies by building a mask for set bits in the (SIMD) registers while loading the block from memory. When fingerprint filters dominate the performance, they can get up to a third faster, and in a sense, partitioning narrows the gap between the two filter families.⁴

For 100 M elements, partitioning can almost triple the maximum performance of some fingerprint filters, while the peak Bloom performance remains about the same. Even though not visible in the figure, the naïve Bloom filter’s throughput improves, as partitioning reduces the number of cache misses, closing the performance gap to register-blocked and sectorized variants. Overall, partitioning closes the performance gap between the fingerprint filters and the (register-blocked) Bloom filter. In particular, the Xor filter benefits from the optimization and even closes in on the Cuckoo filter (cf. Figure 2.11).

Recap – The larger the number of elements gets, the more partitioning pays off. Most notably, the optimization boosts the Xor filter, narrowing the gap to the Bloom and Cuckoo filters.

2.5.3 Build Performance

The only effective way to parallelize the filter’s construction process is first to partition and then to build one filter per partition. Hence, we always use partitioning, which scales well, as shown in Table 2.2. We omit the results for 10 K elements since partitioning and parallelizing the build does not pay off for small data sets.

⁴In some cases, small partitions hinder the Xor filter from being built, which is the reason for the missing data points in the lower right corner.

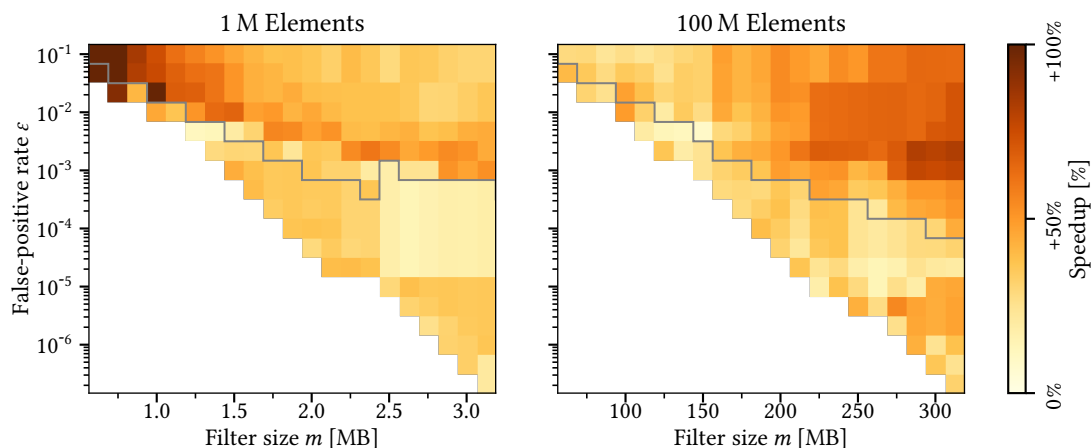


Figure 2.16: Vectorized vs. non-vectorized filters.

Figure 2.18 shows the filter with the best construction time. We include the white line from Figure 2.12 to compare how the divider changes from lookup to build performance. Since the lookup and insert operation on a key is almost the same in Bloom filters, they perform similarly well for construct. While, once again, register-blocking dominates most of the upper area, the sectorized Bloom filter benefits from vectorized inserts and is now the optimal choice for FPRs smaller than 1%. For small FPRs, even the naïve Bloom filter overtakes some of the fingerprint filters.

The Cuckoo filter builds slightly more slowly than the Bloom filters. For small load factors, relocations are rare and insert operations ideally access only the primary and alternate buckets. The Xor filter achieves higher load factors and lower FPRs, but its construction is more involved and thus takes longer. It only outperforms the Cuckoo filter for very low FPRs that hash table-based filters cannot attain or for high load factors. The fuse graph-based algorithm constructs the smallest filters and is the only option for extremely low FPRs. Since it is the slowest of all construction algorithms, we recommend the Xor filter for workloads that require small filters and FPRs and do not often change, like LSM trees.

Recap — If the focus lies on performance, the register-blocked Bloom filter builds the fastest. Other Bloom variants or the Cuckoo filter also offer lower FPRs for the same space consumption at the cost of slightly lower build and lookup throughputs. If the filter is not rebuilt regularly, the Xor filter is an option. It builds the slowest and does not support updates but offers even higher lookup performance and lower FPRs.

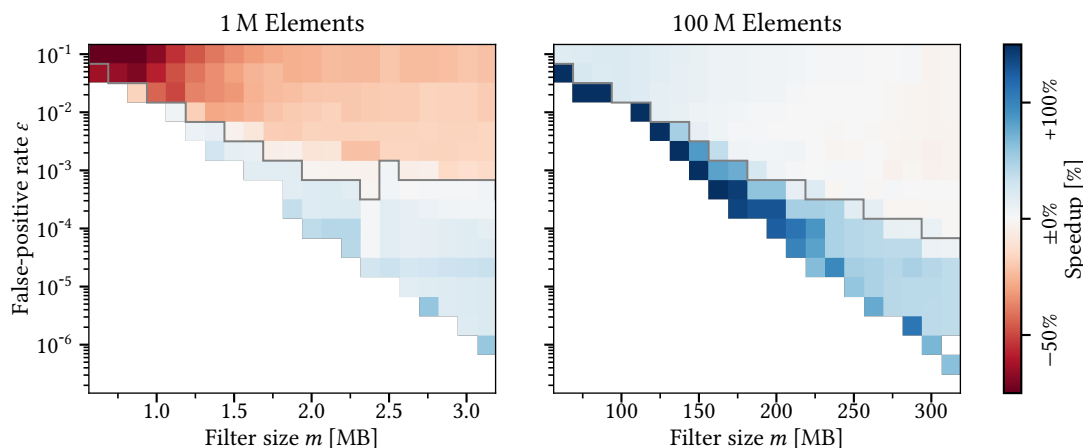


Figure 2.17: Partitioned vs. Non-Partitioned variant.

2.6 Lessons Learned

Each filter variant and optimization offers different performance characteristics that lead to different use cases.

Optimizations Vectorization improves the lookup performance of all four filters irrespective of their size when batching the operations, most notably the (blocked) Bloom filter and the Cuckoo filter. Although it is also possible to vectorize the inserts, we only noticed a stable performance boost for the sectorized Bloom filter. Partitioned filters need to compensate for the radix partitioning and are thus of benefit when the filters exceed the LLC. Even though unpartitioned filters scale better in multi-threaded lookups, partitioning still provides a significant performance boost and decent scalability. Furthermore, it allows for trivial multi-threaded construction of all variants by building a filter per partition. The Xor filter profits the most from partitioning, which narrows the gap between Bloom and fingerprinting filters. Where batching is possible, both optimizations improve the throughput manifold (cf. Figure 2.1).

Filter Variants As expected, Bloom filters dominate both build and lookup performance when high FPRs are acceptable. Most variants, like register-blocking, optimize for higher throughput at the cost of increased FPRs. Other variants are not as fast but offer better FPRs for the same memory budget.

When focusing on space consumption and low false-positive rates, the Xor filter using our new construction algorithm achieves the lowest FPR for the given space and still provides decent lookup throughput. However, the filter has the longest construction time and is immutable. The Cuckoo filter cannot compete with the Xor filter's FPR but instead offers more functionality with similar lookup performance. This filter should be considered particularly with workloads that insert or delete keys after construction. Morton filters effectively improve the Cuckoo filters' space usage

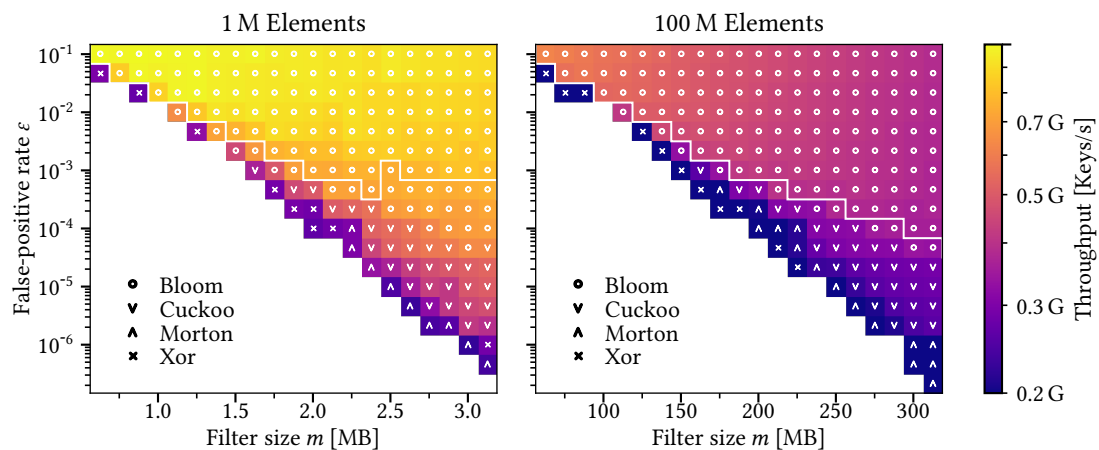


Figure 2.18: Best-performing filter for construction.

and false-positive rate at the cost of decreased lookup and build throughput.

Ultimately, tuning the performance of each filter is still a tradeoff. By spending more time on the filter’s construction, we can use the space more efficiently and reach lower FPRs for the same memory budget. Similarly, when granting slightly more time for lookups, fingerprint filters like the Xor filter pay off because their FPR is lower. When increasing the filter size, preference can be given to either lower false-positive rates or higher performance, using locality-optimized Bloom filter variants or register-friendly fingerprint sizes.

2.7 Discussion

Our work focuses on optimizing the four most promising approximate filters (Bloom, Cuckoo, Morton, and Xor) and identifying the optimal filter for the four key dimensions: false-positive rate, memory footprint, and build/lookup throughput.

To allow for a fair comparison, we use the same implementation for all filters with a wide variety of existing optimizations applied [179]. On top of that, we presented a new optimization — *radix partitioning* — which significantly boosts the performance for large filters without affecting the false-positive rate.

Our analysis shows that of all optimizations, vectorization and partitioning have the most significant impact on all filters. Vectorization is, in particular, beneficial for the compute-bound register-block Bloom filters. Partitioning not only improves the throughput but also makes constructing (even fingerprint) filters trivial to parallelize.

Each of the filter variants has performance characteristics that are attractive to different use-cases as described in Section 2.6. Bloom filters dominate when fast building and probing of the filter is needed and high FPRs are acceptable, making them the most reasonable choice for in-memory query processing. Fingerprint filters offer a better trade-off when focusing on space consumption and achieving low FPRs. Most notably, the Xor filter achieves the best lookup performance for small filters with minimal FPR, making it particularly attractive for applications like LSM-trees, which mostly rely on sub-optimal Bloom filters [51]. However, if applications need more functionality with comparable FPR, the Cuckoo filter is the next best choice, as the Xor filter is immutable.

In summary, the fingerprint filters (Xor and Cuckoo) provide better performance for space-critical or false-positive sensitive workloads like LSM trees or distributed joins. Bloom filters reach the highest throughput and are best suited for applications where the false positives are inexpensive, like in-memory query processing.

CHAPTER 3

Partitioned Joins in a DBMS

Excerpts of this chapter have been published in [17].

An efficient implementation of a hash join has been a highly researched problem for decades. Recently, the radix join has been shown to have superior performance over the alternatives (e.g., the non-partitioned hash join), albeit on synthetic microbenchmarks. Therefore, it is unclear whether one can simply replace the hash join in an RDBMS or use the radix join as a performance booster for selected queries. If the latter, it is still unknown *when* one should rely on the radix join to improve performance.

In this chapter, we address these questions, show how to integrate the radix join in Umbra, a code-generating DBMS, and make it competitive for selective queries by introducing a Bloom-filter based semi-join reducer. We have evaluated how well it runs when used in queries from more representative workloads like TPC-H. Surprisingly, the radix join brings a noticeable improvement in only one out of all 59 joins in TPC-H. Thus, with an extensive range of microbenchmarks, we have isolated the effects of the most important workload factors and synthesized the range of values where partitioning the data for the radix join pays off. Our analysis shows that the benefit of data partitioning quickly diminishes as soon as we deviate from the optimal parameters, and even late materialization rarely helps in real workloads. We thus, conclude that integrating the radix join within a code-generating database rarely justifies the increase in code and optimizer complexity and advise against it for processing real-world workloads.

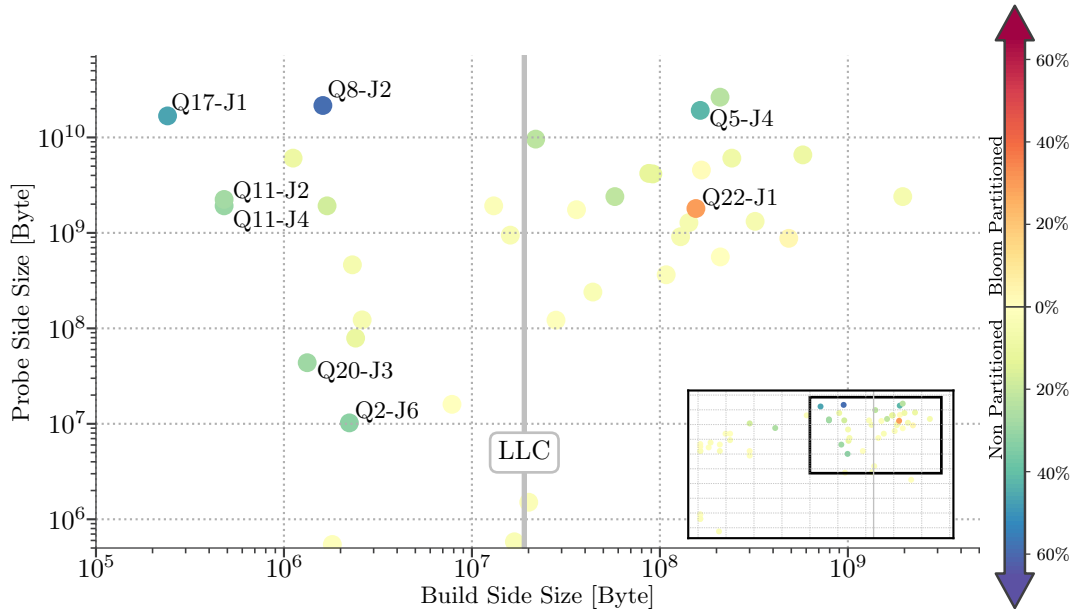


Figure 3.1: Relative performance of Bloom-filtered partitioned and non-partitioned hash join for every join of TPC-H SF 100 labeled as $Q\langle id \rangle\text{-}J\langle order \rangle$

3.1 Motivation

Architectural changes in modern processors have inspired a significant amount of research on finding the optimal join implementation. Over the years, the community has reached the conclusion that hash joins are better than sort-merge joins [13, 99], and that in general algorithm implementations should be tuned to the underlying hardware (i.e., be hardware conscious rather than oblivious) [15, 132, 142, 181].

Recent comprehensive studies have advised that the partitioned radix join performs better than the non-partitioned hash join [15, 181]. What is unclear, however, is if the radix join should completely replace the hash join as a major workhorse in the database engine, or if it should be used as a performance booster. The former is unlikely, as the radix-partitioning phase is only needed when the build side does not naturally fit into the caches; otherwise, the extra pass over the data and the necessary data materialization comes with a non-negligible overhead. The latter is a more difficult question. Using the radix-join as a booster implies that we should know *when* to use it. Unfortunately, existing research has only evaluated the performance of the two on synthetic microbenchmarks, which are not representative of what we typically get in real workloads.

In this work, we investigate *how* to best integrate the state-of-the-art radix join algorithm in a compiling main-memory DBMS and *when* to use it instead of the non-partitioned hash join. Our radix join performance is comparable to prior work’s

stand-alone implementations while also supporting all variants of equi-joins, including outer-, mark-, semi-, and anti-joins [143]. All query plans can use it as a drop-in replacement for the non-partitioned hash join used otherwise. Our system does data-centric query compilation [142] and applies relaxed operator fusion, which enables software-based prefetching [132]. This allows us to make a comprehensive comparison between the two join implementations in a much broader scope of workloads and factors than the analysis done by prior work.

More specifically, we do the following:

Compare the hash joins in a system-wide setup: All joins under testing are integrated and tested within a compiling in-memory DBMS. Both the partitioned and the non-partitioned join are state of the art and hardware-conscious, using optimizations such as software-prefetching, software-write combining, and non-temporal stores [14].

Evaluate the holistic impact of the join on query execution: Existing work compares the joins in isolation and simplifies the settings by relying on materialized input data or omitting result materialization by merely counting the matching tuples [15, 98, 181]. Unfortunately, these simplifications cannot always be applied as joins appear in many stages of query execution. By integrating the joins within a full DBMS, we also investigate their *implicit* effects on the entire execution of a query.

Use representative datasets: In contrast to prior work that only considered narrow tuples (8–16 bytes) [218] and dense data-distributions [98, 181], we use the TPC-H benchmark for our performance evaluation (c.f., Figure 3.1). As shown in Figure 3.2, the TPC-H queries operate on a more extensive range of selectivities and tuple sizes.

To our surprise, despite the encouraging microbenchmark results from prior work, the optimized radix join [15, 181] was not competitive in the TPC-H benchmark. When analyzing the joins, we noticed that for most queries the majority of the shuffled tuples in the partitioning phase might not even be present in the final result (cf. Figure 3.2).

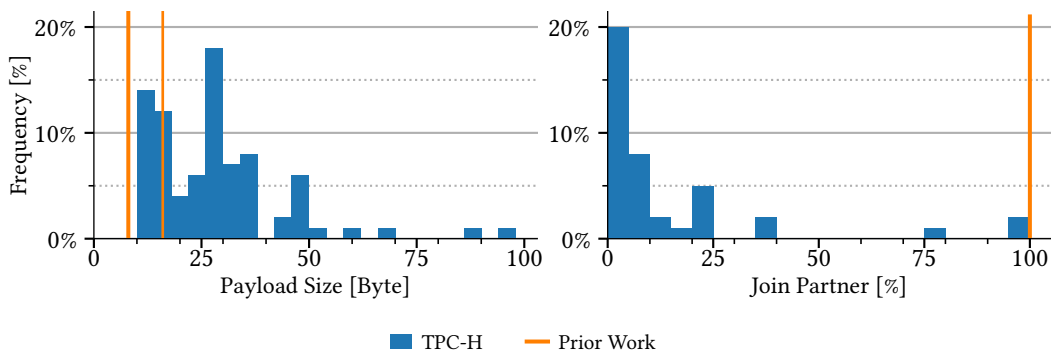


Figure 3.2: Tuple Size and Join Partners in TPC-H and prior work

Therefore, we introduced a filter for the probe phase that drops tuples as early as possible to save computation time and reduce unnecessary materialization overhead.

While this optimization makes the radix join more competitive, it provides measurable benefits in merely one of 59 equi-joins contained in the TPC-H workload (cf. Figure 3.1). By further investigation through a series of microbenchmarks we discovered that the benefits of the radix join diminish quickly when one of the workload’s characteristics (e.g., payload size, data distribution, materialization strategy, join selectivity, etc.) deviates from the optimum. In fact, we can barely achieve any benefit for non-optimal cases.

This chapter contains the following key contributions:

- We fully integrate the radix join into a main-memory DBMS. To the best of our knowledge, this is the first implementation of a radix join in a DBMS, using data-centric code generation [96].
- We reproduce the measurements from prior work [15, 181] and show that our implementation is on par with the state-of-the-art.
- We embed a Bloom-filtered semi-join reducer that significantly reduces materialization overhead for queries with medium and high selectivity.
- We compare our radix join implementation and the Bloom-filtered version *within* Umbra [140] against a state-of-the-art hash join [108, 113, 132] using the TPC-H benchmark.
- With extensive microbenchmarks, we synthesize the range of values for the workload characteristics needed to observe *any* performance benefits when using the radix join.

Following on the insights from our extensive evaluation, we *express serious reservations to implementing the radix join*. Its usage as a booster is limited to a small set of workloads and thus rarely justifies the increase in code- and optimizer-complexity.

The remainder of this chapter is organized as follows. In the next section, we give a brief overview of previous work. Section 3.3 explains the two implementations we base ours on in detail. In Section 3.4, we describe our integration of a partitioned join into a full-fledged RDMBS. Section 3.5 presents the experimental results. We discuss the insights and conclude in Section 3.6.

3.2 Related Work

The majority of papers agree that in-memory hash joins are faster than sort-merge joins [13, 108]. There is further agreement that hardware-conscious joins are superior [14,

132]. However, it is unclear whether partitioning or prefetching for non-partitioning joins, makes the best use of the hardware resources. Balkesen et al., and Schuh et al. claim that radix partitioned joins are superior [13, 14, 181], while Lang et al. state the opposite [108].

Much attention has been given to parallel implementations of in-memory radix joins by our community in the last two decades. Here, we give an overview.

In 1999, Boncz, Kersten, and Manegold [28, 125] predicted that memory access would be the future bottleneck. Back then, the processor throughput was still a limiting factor, but it was already increasing faster than memory performance for the last ten years. Thus optimizing memory usage does pay off quickly. Based on these findings, they proposed the first memory-efficient parallel hash join approach. (cf. Section 3.3.1) This approach summarized most of the core ideas and proposed a hash-based split phase to allow for high-performant memory accesses. In their following work [127] they refined the parallel hash join approach into the radix join so that it is also cache-aware to avoid the original TLB thrashing problem by Shatdal et al. [187]. The resulting partitioned join provides superior cache friendliness. The theory is that these small chunks can be partitioned by using blockwise nested loop processing. Later, the same group introduced Best Effort Partitioning (BEP) which interleaves probe side materialization and hash table probing in order to reduce the memory footprint and use the CPU resources more efficiently [219].

Kim et al. [99] evaluated partitioned radix joins on multicore systems by comparing them against sort-merge joins. They showed that parallel radix hash joins could outperform sort-merge joins in a multicore setup by a factor of two. Furthermore, they tried to predict how future hardware trends influence the speed of sort-merge joins. Their assumption was that SIMD Instructions lead to a near scalar speed-up of sort-merge joins while radix joins negligibly profit from that development. By the time of writing, the size of maximum SIMD instructions on AMD chipsets is still 256 Byte. Blanas et al. [24] compared their multicore partitioned join algorithms against non-partitioning hash joins. They implemented the non-partitioning hash-joins using a lock-based concurrent chaining hash table and compared against three different partitioned-based algorithms, namely shared, independent, and radix partitioning from Kim et al. [99]. Their experiments show that non-partitioning performs best for nearly all workloads, which leaves the cue that partitioning is more effort than the achieved time savings by better cache locality. Only for uniformly distributed datasets parallel radix distribution pays off.

Albutiu et al. [4] provide a new facet in database join research by focussing on multicore NUMA systems. Their Massively Parallel Sort Merge Join uses a specifically tuned memory access pattern and avoids inter-thread synchronization. In their experiments, this join proves to be faster than the comparable join techniques. Balkesen et al. [13, 15] revisited partitioned and non-partitioned joins and optimized the implementation of Blanas [24]. They did use a much more optimal layout in memory, saving

two-thirds of all memory accesses and making it more cache efficient. Furthermore, the skew handling is more robust. We implemented a comparable algorithm into our main-memory DBMS and will describe the approach in Section 3.3.2. Their optimized radix hash joins outperform the non-partition code and thus contradict the measurements of Blanas et al. [24]. Later, they revisit the comparison against sort-merge joins. Their implementation speeds up 2-3 times over the previous sort-merge joins. But still, the hash join outperforms the sort-merge approach. Hence, they argue to continue using hash joins [13].

Lang et al. [108] focused again on massive parallel hash joins. They present a NUMA-aware non-partitioning hash join, which is capable of outperforming the parallel radix hash join. Some of these ideas were then adopted in the non-materializing morsel-driven parallelism framework by Leis et al. [113], which focuses on a full DBMS rather than microbenchmarks. We implemented Umbra's hash join as described in this approach.

Dittrich et al. [172, 181, 182] provided several papers comparing different state-of-the-art joins and hashing methods and also proposing their algorithms. Based on the work by Balkesen [15], and Lang [108], they came up with altered algorithms and compared them with different NUMA-optimized approaches using software-managed buffers and non-temporal streaming. Their findings again contradict the findings of Lang because it is stated that using hardware-conscious algorithms generally pays off. Among their other contributions, the authors also evaluated the radix joins in a stand-alone TPC-H Query 19 variation, where the size of the relations was significantly reduced by partitioning the reference to the original tuple and cutting all strings to one byte. Due to the lack of complete system integration, their analysis is based on the isolated join time without considering the cost of tuple reconstruction, which biases the conclusions [136].

Fang et al. and Makreshanski et al. [64, 123] focused on the still open problem of estimating the performance of hash joins and identified the access granularity, respectively, and the tuple size as the most critical factor. The larger the tuple size is, the better non-partitioning approaches perform. The reason is that larger tuples tend to destroy the necessary cache locality for partitioned joins. This again makes the tuple size the most critical performance factor, saturating the memory bandwidth.

Khattab et al. [98] are proposing another different approach to solve the problem of deciding whether to use radix-partitioning or not. They developed a hybrid solution called *PolyHJ* based on the contributions of Dittrich [181] and Lang [108] and compared it to the respective papers. However, they based their implementation on array-based joins, which limits its use to densely filled array columns. Recently, Zhang et al. provided an extensive evaluation of different data partitioning schemes for in-memory systems [218]. They compared a simple implementation with the multiway approach of Boncz [28] and streaming buffer approaches for different tuple widths and row or column store formats. They conclude that each technique performs best for a different

workload and the size of the tuples has a big impact on the performance.

Further, we note that there is a lot of related research linked to join processing and radix partitioning: Polychroniou et al. [159] have investigated SIMD partitioning and provide an overview of partition variants [160], which is revisited for radix-partitioning by Schuhknecht et al. [182] and Zhang et al. [218]. Richter et al. [172] compare different hash table implementations, while Barber et al. [18] focus on memory-efficient hash joins. Pirk et al. [157] analyze hash joins in-depth and Shrinivas et al. [189] and Abadi et al. [1] compare materialization strategies in column-store database systems. Dreseler et al. [56] analyze the performance of their system by turning on and off different optimizations including semi-join reducers.

Partitioning also applies to non-CPU-centered data processing. For example, GPU- [158] or FPGA-accelerated [92] approaches have similar goals and use comparable algorithms to distribute the workload better.

3.3 Partitioned Radix Joins

Existing in-memory hash join algorithms can be divided into two camps [181]. On the one hand, we have the non-partitioning variants using a global hash table, which is accessed in parallel. They rely on software-based prefetching to avoid expensive cache misses and random memory accesses when the hash table does not fit in the caches [14, 132]. On the other hand, the radix joins directly reduce cache misses by joining the data partition-wise, where each partition is sized so that the hash table fits in the cache [187]. In this chapter, we assume that both probe and build side reside in already materialized form to be comparable with prior work [15, 181].

3.3.1 Basic Partitioned Join

On a high level, a partitioned join splits both input relations into partitions that are then joined individually.

A basic partitioned join implementation consists of two phases: First, in the partitioning phase, both the build and the probe side are partitioned by using a hashed value of the join condition as key. As a result, both sides are now split into partitions containing their respective join partners. In the second phase, the join is executed per partition. A union of all partitions' results yields the final outcome.

The partitioning algorithm operates in three steps [218]: The first step scans the input and builds a histogram, counting how many elements the partition will consist of. The second step uses the histogram to calculate the total number of tuples and the exact partition boundaries. We allocate an output buffer large enough to fit all tuples and assign each partition a region based on the partition boundaries. Finally, in the third step we scan the data again and materialize each tuple to the correct position

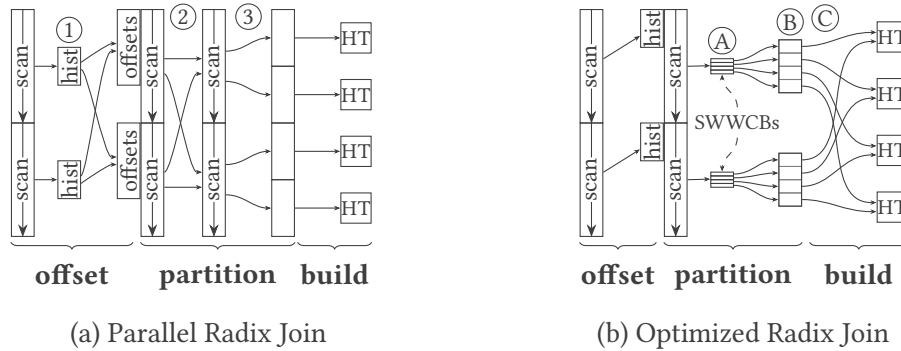


Figure 3.3: Radix-Partitioning in prior work

in the output buffer. Each partition keeps track of the number of written tuples to determine the correct output position.

3.3.2 Parallel Radix Join by Balkesen et al.

Balkesen et al. [15] proposed an efficient, publically available¹ implementation of a radix join. Their join is a refined version of the one by Blanas et al. [24]. Figure 3.3a depicts an overview of their approach.

Two-pass Partitioning: Boncz et al. [28] observed that a single-split partitioned join has a performance problem. It occurs when writing to more partitions in-parallel than the translation lookaside buffer (TLB) has entries, which trashes the TLB. Boncz et al. mitigate the problem by applying multi-pass partitioning, called radix-partitioning, which performs multiple splits subsequently. This limits the number of partitions created in each pass so that it does not exceed the number of TLB entries. Each partitioning pass uses a different subset of bits from the hashed key. Balkesen et al. use two partitioning passes, as shown in Figure 3.3a.

Parallel Partitioning: Running the basic implementation in parallel is challenging because each worker writes to all partitions, leading to high congestion. Kim et al. [99] propose to split the input relation so that each slice can be processed in parallel. All equally sized slices are stored in a task queue. From there, each worker picks a task and performs the steps listed under Section 3.3.1. Following the histogram creation of all tasks, the prefix sums are computed by combining all histograms ①. Based on the prefix sums, each task calculates a dedicated output location and scatters the tuples into partitions without any synchronization ②. The second pass takes the partitions from pass one and splits them again ③.

The final join is done in parallel, using task-based parallelism that also helps with skew.

¹<https://www.systems.ethz.ch/node/334>

3.3.3 Optimized Radix Join:

Balkesen et al. further refined their radix join with software write-combine buffers (SWWCBs) and streaming instructions [13]. We compare our implementations against their optimized radix join in Section 3.5.2. Schuh et al. [181] further optimized the radix join by adding NUMA-awareness (cf. Figure 3.3b).

Software Write-Combine Buffers: Wassenberg et al. [207] propose SWWCBs to speed up radix sorting, which is also beneficial for radix-partitioning [13]. SWWCBs are software-managed data buffers residing in the cache, combining multiple writes. Each buffer is at least one cache line in size and stores the partitioned tuples instead of writing them to their destination directly (A). The buffer is flushed to its destination when it is full, which effectively reduces the pressure on the TLB, and the number of memory writes.

Non-temporal Streaming: Non-temporal streaming instructions mitigate the potential doubled number of writes introduced due to SWWCBs by writing full SWWCBs directly to DRAM. The write bypasses all caches and avoids their pollution (B). However, the data now needs to be aligned at cache line boundaries. This makes combined use of both optimizations sensible. The maximum width of the SIMD registers limits the size a single instruction can write at maximum. In 2016, this was half a cache line, or respectively 256B using AVX2. With AVX512 instructions, modern Intel processors can store a full cache line at once.

NUMA-awareness: Currently, the radix join is not NUMA-aware because each task writes to multiple partitions, which are located all over the output buffer. Thus, each worker potentially has to access different memory regions to store its tuples. Schuh et al. [181] keep the writes local by adding an output chunk per task, which stores the tuples in local partitions. However, now the final partitioning result is not located in one contiguous memory region but in one chunk per task. Hence, the join may have to read from different NUMA nodes (C). Their experimental evaluation shows that the advantages prevail, since only reads may be on different NUMA-nodes. Furthermore, NUMA access is much more balanced, and overall performance increases.

Array-Join Storing the tuples in arrays is another option for densely packed data. Rather than using a hash function to map all keys to specific positions, Dittrich et al. directly use the key to specify the position in an array [172]. Then, a simple array lookup is sufficient to fetch the tuple as long as no duplicates occur. While the data domain is small and densely packed, the size overhead is not significant. Thus, this approach only works for a small data domain, and if we know the value range beforehand.

3.4 Joins in main-memory DBMS

Prior to this dissertation, all work on partitioned joins was evaluated with the join in isolation using microbenchmarks. To take the next step from a stand-alone radix join to a real database system, we integrated radix-partitioned joins into Umbra [140], whose performance is comparable to HyPer or MonetDB [56].

Umbra uses data-centric code-generation [142], relaxed operator fusion [132], arbitrary query unnesting [141], morsel-driven parallelism [113], and accepts the queries using a SQL frontend. We first describe how data-centric code generation works in general, and then how it works for both of our hash join implementations. Following these explanations, we focus on our novel radix-join implementation, which partitions two input dataflows.

3.4.1 Data-Centric Code-Generation

The main difference between a stand-alone join implementation and one integrated into a full-featured RDBMS system is the environment. In the former, the whole system focuses on the join. In the latter, the join is a part of operator pipelines that organize the dataflow, as shown in Figure 3.4. First, a pipeline’s source operator loads the tuple from a materialized state into the CPU. Then, the tuple traverses the pipeline operators and it is finally materialized in the next pipeline breaker [138].

Umbra compiles each pipeline, in particular the dataflow from one source operator to the materialization point, in a bottom-up manner using the produce/consume model [142]. Each operator has to call *produce* on its inputs to delegate the responsibility

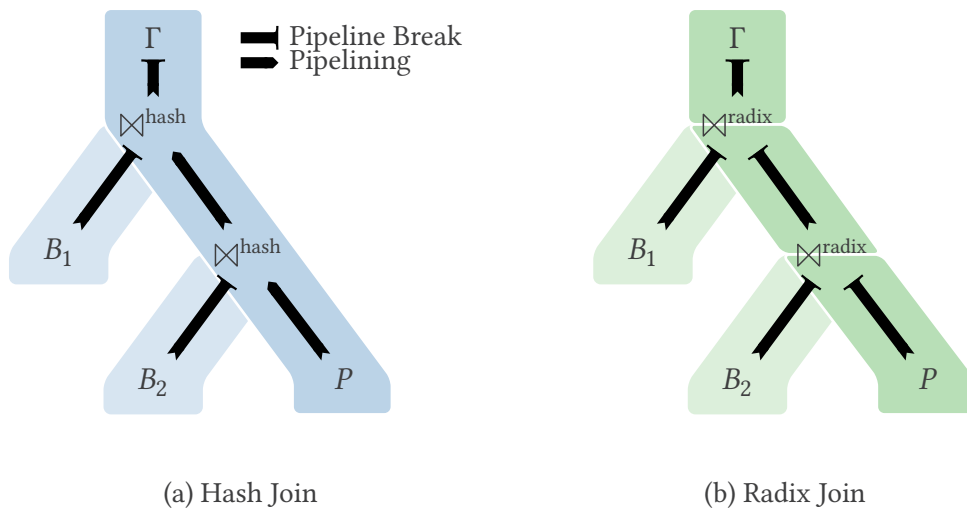


Figure 3.4: Pipelining in radix and hash joins. Hash joins can pass the probe tuples through multiple joins while radix joins have to materialize both inputs every time.

for starting the pipeline. Eventually, the pipeline starter is reached, which cannot delegate further. It begins pushing tuples to its consumer up the pipeline. Once a pipeline breaker is reached, it generates code to materialize all incoming tuples. We use this abstraction to compile data-centric code for arbitrary SQL queries.

3.4.2 Materialization Strategy

Umbra stores relations column-wise in main memory [140]. We use early materialization to reduce random access during pipeline evaluation. Thus, the table scan only reads necessary columns, filters them using SIMD instructions, and stitches them together in tuples passed to the consumer. To avoid materialization overhead, we use sideways information passing [189]. The build side of our hash join, e.g., tells the probe pipeline the required tuples to filter them out early.

To compare effects of the chosen materialization strategy, we integrated Late Materialization. We traverse the query tree from top to find the earliest access to each column. If that does not happen immediately after a table scan, we introduce a late-load operator that retrieves columns based on their tuple id when needed.

3.4.3 Non-Partitioned Hash Join

The non-partitioned hash join does not have to write out the probe side, as shown in Figure 3.4. Each hash join passes the tuples on and performs the join within the pipeline [113]. This so-called operator fusion keeps the tuples in registers for as long as possible. Sadly, it might also hinder inter-tuple parallelism since the code structure is more involved. Relaxed Operator Fusion (ROF) counteracts this problem by loosening the original idea of data-centric code-generation in favor of intermediate materialization. It allows the DBMS to introduce staging points in the query plan, buffering the probe side in cache and trading pipelined tuples with cache-locality [132]. Reading from these buffers enables vectorization optimizations, e.g., branch-free primitives, and software-based prefetching to avoid cache misses. ROF effectively combines the advantages of data-centric code generation with vectorization.

3.4.4 Partitioned Hash Join

In contrast, writing to memory is not optional for the radix join because it builds upon the radix-partitioning phase. These frequent writes loosen the original idea of data-centric code generation, and they also counteract it. So when multiple radix joins are executed after one another, each join has to break the pipeline.²

²When two subsequent joins use the same partition key, we could combine them in a pipeline to avoid the pipeline break and the resulting partitioning overhead.

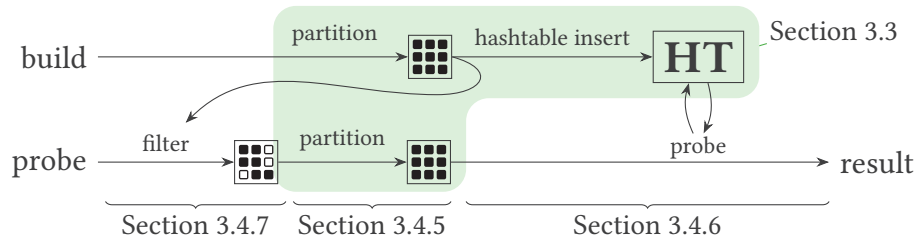


Figure 3.5: Schematic Overview of our Partitioned Join.

Algorithm 1: Full Pipeline Breaker

```

1 Function Radixjoin::produce(requiredColumns):
2   condition ← analyzeJoin(requiredColumns);
3   build ← prepareBuild();
4   left.produce(condition.left.requiredColumns);
5   build.partition();
6   probe ← prepareProbe();
7   right.produce(condition.right.requiredColumns);
8   probe.partition();
9   joinTuples(build, probe);

```

Thus, the radix join is both a full pipeline breaker and a pipeline starter, as shown in Figure 3.4. Algorithm 1 follows along the three phases described in Section 3.3.1. The code first partitions the build side and then the probe side, which breaks both pipelines because all data is now materialized. After both sides are partitioned, the new pipeline is started, which joins the tuples.

The join code of Algorithm 2 mainly consists of tight loops, which is characteristic for the produce/consume model [138]. These tight loops are advantageous for modern CPUs because they maximize data locality by keeping the data in CPU registers as long as possible. The algorithm has to loop over the partitions, build the hashtable, and check whether each tuple is contained. All matching tuples are passed to the consumer in the pipeline.

Because the majority of the work is done during or after materialization, tuple collection is simple. Depending on the current input pipeline, the tuple has to be partitioned either on the build or on the probe side.

3.4.5 Morsel-Driven Partitioning

The pipeline execution is based on morsels, which divide the total workload into smaller blocks, enabling work-stealing [113]. Every source operator has to emit the

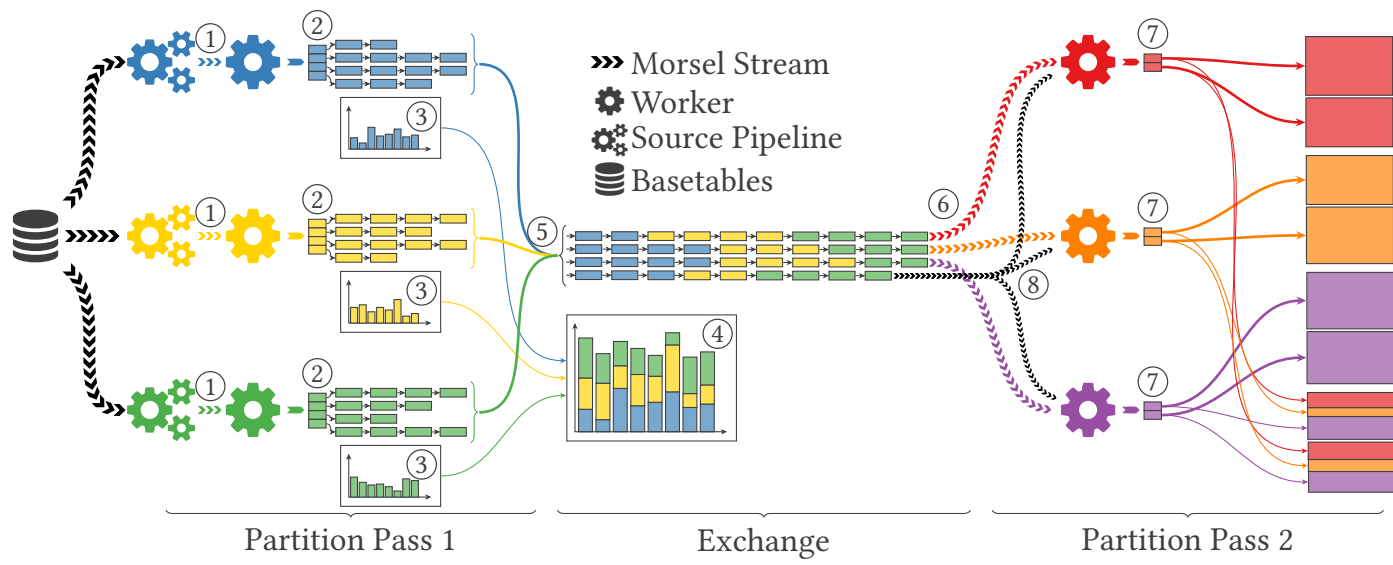


Figure 3.6: Schematic overview of our two-pass partitioning performing an eight-way split using 3 bits.

Algorithm 2: Starting a New Pipeline

```

1 Function Radixjoin::joinTuples(build, probe):
2   for  $p_{\text{build}}, p_{\text{probe}} \leftarrow \{\text{build}, \text{probe}\}$  do
3     hashtable = buildHashtable( $p_{\text{build}}$ );
4     for  $t_{\text{probe}} \in p_{\text{probe}}$  do
5       for  $t_{\text{build}} \in \text{hashtable.probe}(t_{\text{probe}})$  do
6         consumer.consume( $t_{\text{build}} \circ t_{\text{probe}}$ )

```

Algorithm 3: Consume from both input streams

```

1 Function Radixjoin::consume(tuple):
2   if isBuildPipeline() then build.addEntry(tuple);
3   else probe.addEntry(tuple);

```

data into the pipeline morsel-wise. Figure 3.6 shows a detailed overview of the tuple flow inside our partition step, which is used for both build and probe side.

The first pass consumes all morsels of the current source pipeline by picking them from the morsel stream once they finish their previous work ①. This technique allows fine-grained load balancing, even with skewed data. The worker determines the output partition based on the least significant bits of the hash value, which is then paired with the tuple. This is first materialized in the worker’s own worker-local set of SWWCBs ②. As soon as a buffer is full, we use non-temporal streaming instructions to move the tuples to their temporary partition without polluting the caches.

One challenge lies in working with dataflow inputs. This means that we need to materialize the input first without relying on histograms, which is also the reason for using two passes. Hence, each temporary partition is implemented as a linked list of pages. Whenever a page is full, a larger page is prepended and used instead.

Afterward, each worker traverses the linked list and builds a local histogram for the next partition pass ③. Currently, there is no need for communication between the workers.

In the exchange phase, we do two things: First, ④ we compute the exact size of the output partitions based on the prefix sums of the worker-local histograms. Second, ⑤ all workers’ linked lists are combined by concatenating the lists in so-called pre-partitions.

Hence, the database system does not need synchronization between the work packages in the second partitioning pass as each has its dedicated range.

We perform the second partitioning pass morsel-wise as well. The radix join generates its morsels based on the pre-partitions ⑥. We use the same worker to process the entire linked list of one pre-partition. Once again, we use SWWCBs to

combine the writes and then scatter the tuple buffer to its final position ⑦. Further, we implement work-stealing to achieve proper load balancing among the workers, even under the presence of skew ⑧.

During the whole partition process, all workers are writing to either local or dedicated memory areas. Hence, there is no need for synchronization or writing to non-worker-local memory regions, which ensures scalability with different numbers of worker threads and on systems with multiple sockets. The resulting partitions of build and probe side are handed over to the final join phase.

3.4.6 Final Join Phase

Each morsel builds the hash table on the fly using robin-hood hashing, which provides the most robust performance for thread-local workloads [172]. Since moving tuples is expensive, we only store pointers. We avoid costly resizing of the hash table because we know its size in advance. In addition to that, we reuse the hash table's memory segment to avoid costly memory allocation. Thus, we only have to reallocate memory in case the partition size has significant skew.

Robin-Hood hashing [36] reduces the variance of the displacement between the desired and the actual location.

Schuhknecht et al. analyzed different hash table implementations with respect to their performance on specific workloads [172]. They found that robin-hood hashing is ideal for thread-local hash tables providing the most robust performance for various datasets, especially when they are never updated. Conveniently, radix joins construct hash tables thread-local without synchronization. We avoid the costly resize of the hash table because we know precisely how many tuples are inserted and choose the fill-factor accordingly.

Since Robin-Hood hashing minimizes the displacement, we store the maximum displacement to limit the number of probed entries. For non-skewed workloads, we find most tuples instantly and immediately stop retrieving more entries.

Because moving tuples is expensive, we use the hash table only as an index to our tuples instead of copying them. In addition to that, each worker has to join multiple partitions. Thus, we reuse the hash table's memory segment for the next partition as well to avoid costly memory allocation. Ultimately, we only have to allocate memory once in the beginning, and in case the partition size has significant skew.

3.4.7 Bloom Filters

We are now at the point where the join operates on cache-resident partitions, with the cost of partitioning dominating the execution time of the radix join [15, 181]. Materializing the probe side partitions can often become unnecessarily expensive in selective queries. One optimization is to reduce the number of stored tuples for the

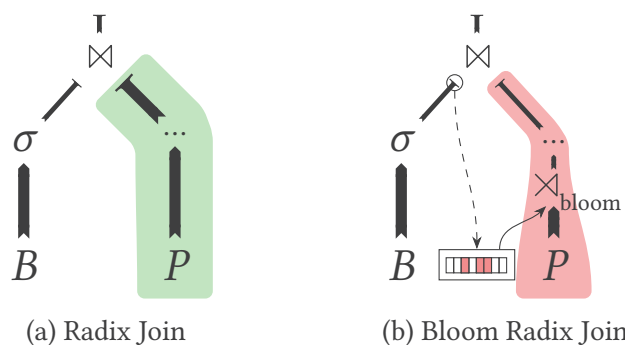


Figure 3.7: Bloom filters with selective joins. Tuples without a join partner are filtered early and are not materialized.

probe side. This is possible because most queries apply selections on the build side before joining the data [29].

Fuzzy semi-join reducers are an established technique for non-partitioned hash joins [110]. They improve the performance of selective joins, as already implemented in our non-partitioned join using tagged pointers [113]. The optimizer pushes the reducers down in the pipeline to prune tuples early (cf. Figure 3.7).

We introduce a Bloom-filter based reducer in our radix join to minimize the cost of materialization. The second pass over the build side generates the filter while partitioning. The filter is probed in the pipeline before partitioning the probe side and is also pushed down when possible.

Following the guidelines from Chapter 2 and by Lang et al. [109], we implemented register-blocked Bloom filters. These filters partition the Bloom filter into register-sized blocks. We have to access exactly one block for each probe, which reduces the number of cache misses to at most one per check. Consequently, the writes to the Bloom filter can be done in parallel without synchronizing as two partitions cannot share blocks. The Bloom-filtered radix join performs around 40% faster for 5% foreign key join partners (cf. Section 3.5.4).

3.5 Evaluation

In the following, we present an experimental evaluation of our radix join against our non-partitioned hash join within Umbra, a full-fledged RDBMS. We answer when and whether partitioning pays off.

3.5.1 Experimental Setup

We briefly give an overview of our setup, including our workloads and the main questions we plan to investigate.

Joins under test

We have compared the following three joins inside Umbra [140]:

Radix-Partitioned Join (RJ): Our radix join implementation with SWWCBs, non-temporal streaming, two-pass partitioning, and thread-local output buffers. It implements all optimizations presented in Section 3.3.

Bloom Radix-Partitioned Join (BRJ): Our Bloom-filtered radix join implementation. It reduces materialization overhead by filtering the probe side (cf. Section 3.4.7).

Buffered Non-Partitioned Hash Join (BHJ): Our non-partitioned join implementation, using a global chaining hashtable with relaxed operator fusion [132]. It features a semi-join reducer based on tagged pointers [113]. Its performance is comparable to HyPer’s hash join [56, 140].

We have validated our joins against state-of-the-art prior work:

Joins from Balkesen et al. (PRJ & NPJ): We have evaluated the aforementioned joins against the partitioned (PRJ) and non-partitioned join (NPJ) of Balkesen et al. [15], which they provide as stand-alone implementations. To allow for a fair comparison, we enabled all optimizations like SWWCBs and non-temporal storing for the PRJ and software-based prefetching for the NPJ.

Workloads

The major part of the evaluation was performed on the TPC-H benchmark [199], which we analyzed on a query and individual join level. It features 22 queries with different workload characteristics (c.f. Figure 3.2).

To compare against related work and refine the TPC-H analysis by isolating certain workload factors, we used microbenchmarks. As a base for these, we reused the workloads of Balkesen et al. [15], whose properties are listed in Table 3.1. We alter the workload for each microbenchmark to isolate particular workload factors that are of interest, e.g., different selectivities or payload sizes.

In our system, we reproduced the setup by generating the build and probe tables using the following SQL statement. We did not preprocess the data and particularly did not generate indexes.

```
CREATE TABLE build(key BIGINT NOT NULL, pay BIGINT NOT NULL);
CREATE TABLE probe(key BIGINT NOT NULL, pay BIGINT NOT NULL);
```

For workload B, we used INT instead of BIGINT to generate 4 B sized columns.

	workload	size [B]	tuple count		size [MiB]	
	used in	key/payload	build	probe	build	probe
A	[15, 24]	8/8	$16 \cdot 2^{20}$	$256 \cdot 2^{20}$	256	4096
B	[13, 15, 99]	4/4	$128 \cdot 10^6$	$128 \cdot 10^6$	977	977

Table 3.1: Workloads from Prior Work

	Skylake-X	Ryzen 9	Sandy Bridge
vendor	Intel	AMD	Intel
model	i9-9900x	3950X	E5-2660v2
sockets	1	1	2
cores (SMT)	10 (x2)	16 (x2)	20 (x2)
clock rate [GHz]	3.5-4.4	3.5-4.7	2.2-3.0
L1 data cache [KiB]	32	32	16
L2 cache [KiB]	1024	512	256
LLC cache [MiB]	19	16 (x4)	25
DRAM size [GB]	64	64	256
DRAM type	DDR4	DDR4	DDR3
DRAM speed [GiB/s]	79.4	47.8	59.9
launch	Q4'18	Q4'19	Q3'13

Table 3.2: Hardware Platforms

Hardware

Unless otherwise noted, we used an Intel i9-9900X (Skylake-X) CPU with 10 cores and 64 GB RAM. By default, we used all available threads, including hyper-threads. Other experiments were conducted on a dual-socket Intel E5-2660v2 (Sandy Bridge) with 10 cores and 256 GB of RAM, and on an AMD 3950X (Ryzen 9) with 16 cores and 64 GB of RAM. Both systems show NUMA effects. The Ryzen chipset is split into four chiplets à four cores with separated L3 cache. Detailed specifications can be found in 3.2. We compiled the code with GCC 9 using the `march=native` flag to enable the AVX512 instruction set, if possible. The RDBMS uses LLVM and clang 9 to compile the queries themselves. CPU counters were obtained using Linux `perf` and the Processor Counter Monitor, formerly known as Intel PCM.

To have a sound comparison, we omitted query compilation time³ because the other implementations were hand-coded and pre-compiled. Before taking any measurements, we warmed up the system and ensured that all data is in memory. We ran

³Query compilation takes negligible time, even for optimized settings as shown in Section 4.4.

all benchmarks at least five times and reported median performance.

Key Questions

We separate the evaluation into three parts:

First, we ran experiments to ensure that our join implementations are competitive to related work (Section 3.5.2), to check how well they scale with the number of threads (Section 3.5.2) and in a NUMA system (Section 3.5.2), and to see how efficiently they use the memory subsystem (Section 3.5.2).

Second, we ran the TPC-H workload to check whether the radix join can completely replace the non-partitioned hash join in our database engine (Section 3.5.3). Since our hypothesis assumes no, we evaluate whether the radix join could be used as a performance booster by analyzing the TPC-H workload on a join-level (Section 3.5.3).

Finally, with an extensive series of microbenchmarks (Section 3.5.4) we searched for ideal range values of workload properties (e.g., selectivity, payload size, pipeline depth, etc.) that emphasize performance advantages of the radix join over the non-partitioned hash join.

3.5.2 Performance characterization and comparison to related work

We have aimed to evaluate the benefits and drawbacks of partitioning within a DBMS objectively. At the same time, this evaluation is only insightful if our implementation offers reasonable performance.

We used PRJ and NPJ by Balkesen et al. [14, 15] with all optimizations enabled to compare its performance against our join implementations. To match the workloads used in the original paper, we have used the following query to join build and probe table and count the resulting tuples.

```
SELECT count(*) FROM probe r, build s WHERE r.k = s.k;
```

One key difference is that Balkesen et al. directly use the key for partitioning, while we create an equally sized hash value and store it with each tuple. This is compensated as we do not store the payload, which is not required for the tuple count.

Scalability

In this experiment, we first compared the performance of our implementations to that of the state-of-the-art. The results are shown in Figure 3.8, which indicates that both the RJ and the BHJ are competitive with PRJ and NPJ. On the one hand, our RJ outperforms the PRJ for workload A, while on the other hand, our BHJ is not as fast as the optimized NPJ on both workloads.

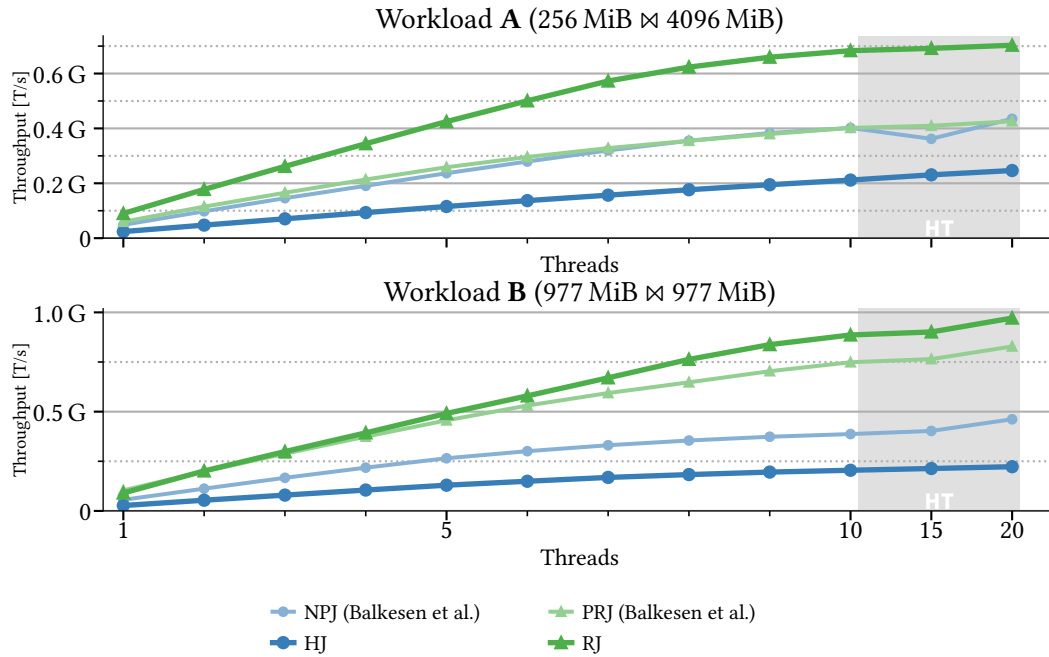


Figure 3.8: Scalability and comparison to Balkesen et al.

Another observation is that all implementations scale well with the number of hardware contexts, although radix joins experience a bigger speed-up than non-partitioned joins. For 10 threads, our RJ implementation speeds up by a factor of 7.5 to 9.5 for workloads A and B, respectively. For workload A, the RJ does not fully scale to 10 threads because the system already reaches the memory bandwidth limit (as we will show in Section 3.5.2). For workload B, the hyper-threads give us about 10% additional performance, since the smaller tuples do not entirely saturate the memory bandwidth. As expected, both non-partitioned hash join implementations benefit more from hyperthreading because it hides their memory access latencies. The NPJ implementation, unlike the BHJ, is optimized for the given workload and performs better. For instance, the NPJ knows the exact hash table size and distribution beforehand.

NUMA effects

In this experiment, we evaluated how well algorithm implementations utilized available hardware resources by scaling the number of cores from one to the maximum number of logical threads available.

We used the other two machines, the dual-socket Intel Sandy Bridge and the AMD Ryzen 9, whose chip has four chiplets (cf. Table 3.2) to show the performance with NUMA.

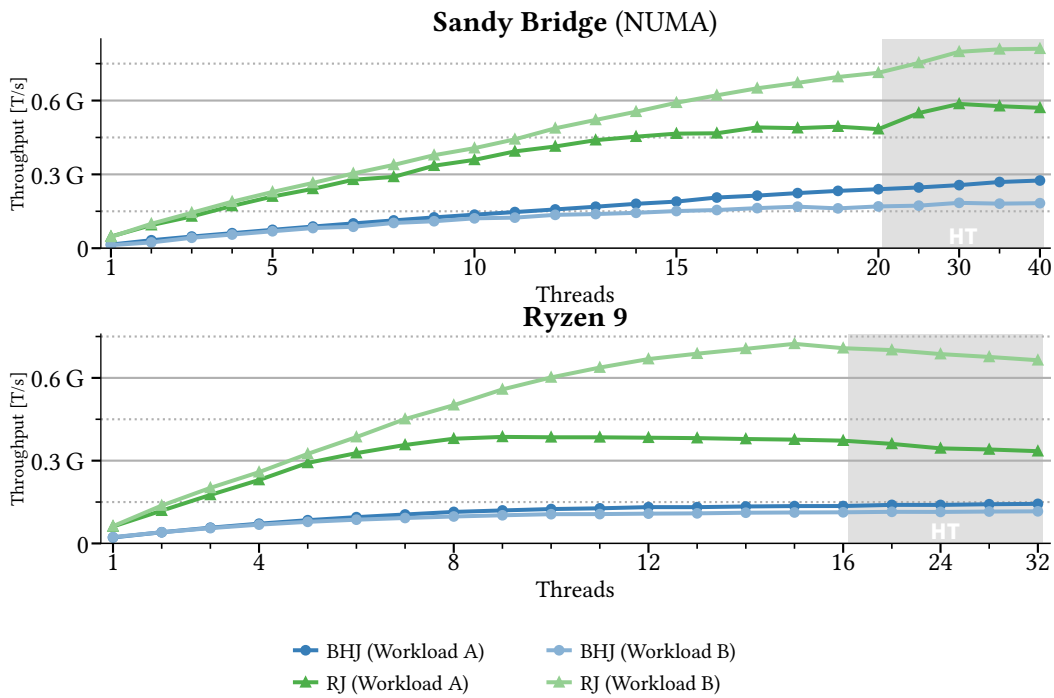


Figure 3.9: Scalability on different machines

The results in Figure 3.9 show that RJ scales well on the Sandy Bridge machine. Its performance increases by a factor of 10 to 16, depending on the workload. The smaller tuples put less pressure on the memory bandwidth, resulting in better scalability. As before, hyper-threads marginally sped up the performance.

On the Ryzen 9, however, we observed a different pattern and the RJs no longer exhibited the linear scalability beyond a certain point. The comparably small memory bandwidth is the key factor as the bandwidth per core is 60 % of the Skylake-X's. Thus, the RJ scaled well initially, but reached the memory bandwidth limit much faster for workload A. As we increased the number of threads further, the RJ slowed down because of memory bandwidth contention. As before, the BHJ performed similarly on all machines and workloads and scaled more independently of the workload.

Memory bandwidth usage

As identified by the two previous experiments, the performance of RJ is significantly affected by its pressure on the memory subsystem. Both when increasing the payload size and when scaling the number of hardware contexts, the performance benefits diminish as we approach the bandwidth limits. Thus, in this experiment, we analyzed the memory bandwidth usage (read and write) for the individual stages of the RJ as

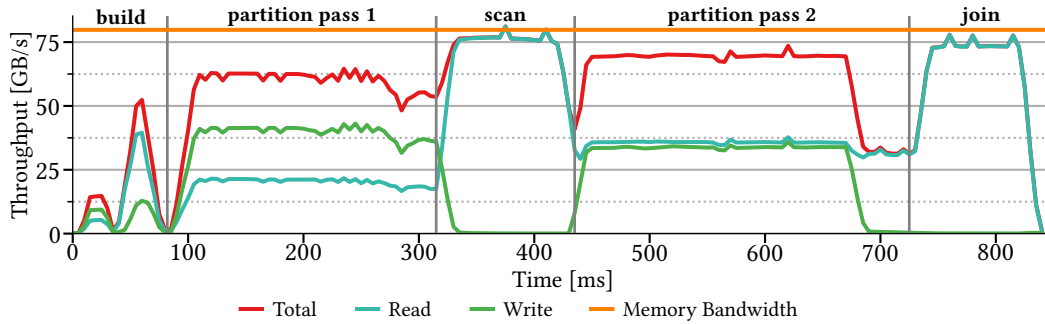


Figure 3.10: Memory Bandwidth for 24B wide tuples

measured using the PCM Tools.⁴

Figure 3.10 shows the read, write, and total memory bandwidth while performing the RJ for the SQL query stated in Section 3.5.4. The x-axis shows the time spent to highlight how expensive each phase of the join is. The build pipeline takes a fraction of the execution time, given that it is 30 times smaller in size than the probe side. The probe pipeline dominates the execution time, mainly due to the materialization phase during the two partitioning passes. We deliberately chose this query since it demonstrates the effects introduced by padding. It is required to use SWWCBS and non-temporal streaming instructions, which outweigh the negative effect of padding. We notice that both the partitioning steps and the join are bandwidth-bound, which confirms the futility of adding more hardware contexts and why increasing the payload size hurts the performance.

The prior three experiments verified the competitiveness of our implementation. It can fully utilize the memory bandwidth and is bound by it, leaving minor room for improvement.

The first pass is not entirely memory-bound because the DBMS additionally reads the tuples out of the table storage and has to select the correct columns. We note that the join writes twice the amount of data it reads as we only need to read 16 B, and add 8 B padding and 8 B hash value for writing out the tuples.

Following this first partition, we scan the relation once to build the histograms for our output partition sizes. In this pass, we are memory bound. Afterward, the second partitioning pass takes place, which is also almost memory-bound, followed by the join, which once again is memory bound. The short drop between the second partition and the final join is caused by freeing the results of our first partitioning as early as possible.

⁴<https://github.com/opcm/pcm>

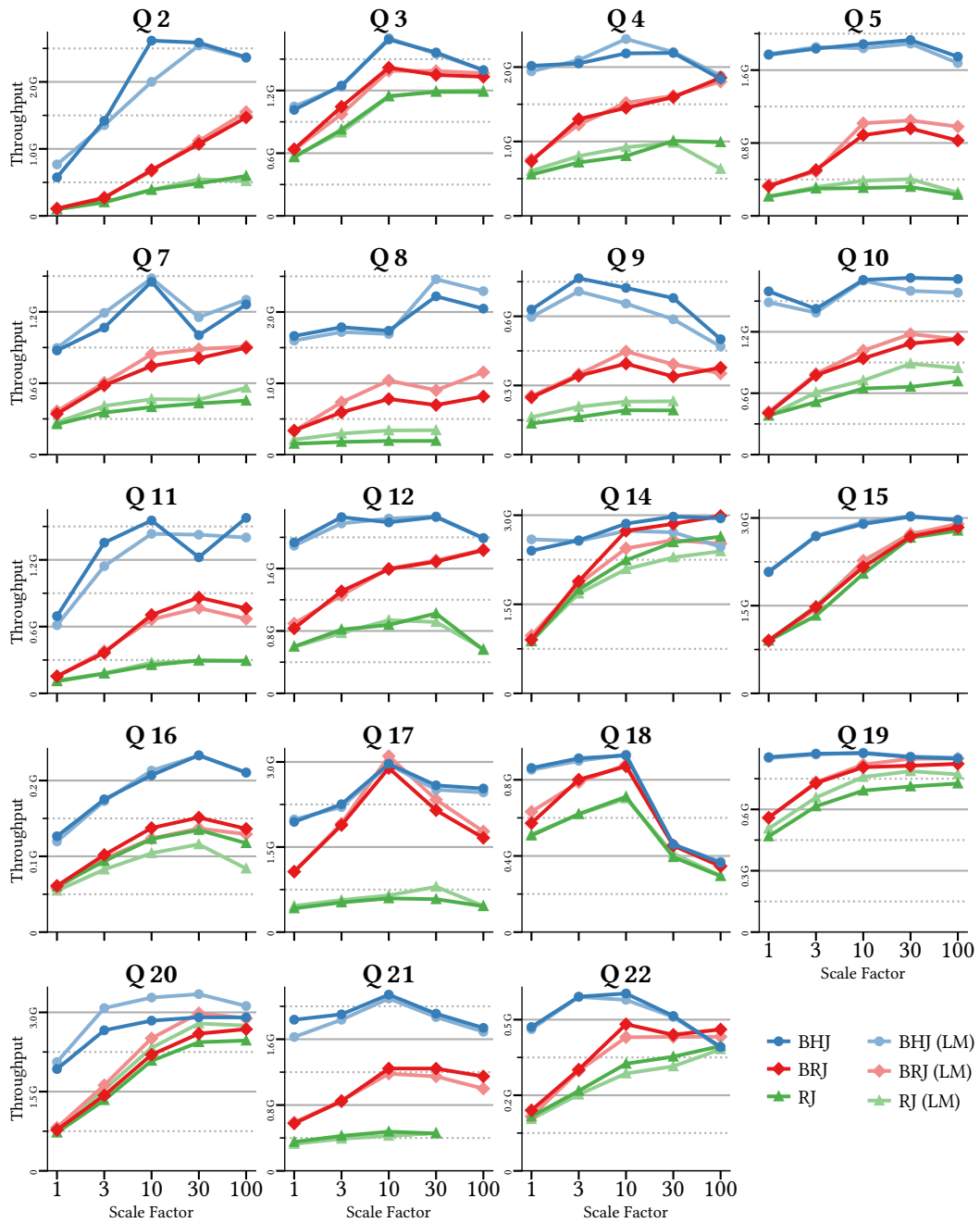


Figure 3.11: Throughput of all TPC-H queries containing joins with every join replaced by the one under testing⁸

3.5.3 TPC-H Evaluation

The TPC-H benchmark offers a variety of queries that put pressure on different parts of the RDBMS at varying scaling factors (SFs): i.e., string comparisons, large base table scans, or joins with different selectivities [29]. To address whether the RDBMS should use a radix join as the sole workhorse, we have compared the performance of our join implementations by replacing all joins in the query tree with the join under testing for different scaling factors.

Figure 3.11 shows the results of our experiments for relevant TPC-H queries as we vary the dataset size (i.e., scaling factor). We used processed tuples per second as a metric with the number of tuples being the sum of all tuples counted at the pipeline sources. For example, the number of tuples in “SELECT count(*) FROM a, b WHERE a.key = b.key;” is $tablescan + tablescan + groupby\ scan = size(a) + size(b) + 1$. *Queries 1, 6, and 13* were not included in our measurements since they do not use joins. Our system uses a groupjoin for Query 13, which combines join and group by [66, 134]. Deactivating group joins does not make a difference since aggregation wholly dominates the query.

We make the following key observations. First, the BHJ delivers the best overall performance, especially apparent for SFs under 30. Second, BRJ is faster than RJ for all queries because foreign keys mainly use filtered build sides (cf. Figure 3.2, [29]). Third, the BRJ outperforms the BHJ only in Query 22 for SF 30 and 100. Fourth, Late Materialization appears to be orthogonal to the question of whether to partition or not. Therefore, if one needs to choose to implement only one hash join in their system, the BHJ is the apparent implementation choice.

This conclusion confirms our hypothesis that replacing all joins is not desired because the radix join is most promising for selected workloads [14, 15, 181]. We continue our analysis in more detail for individual query plans to explain why BRJ and RJ cannot always replace the BHJ as the primary join.

End-to-End Query Performance

In this section, we analyze the selected TPC-H queries based on their query plan.⁵ Since the queries in TPC-H have different characteristics, we have split them into several groups and discuss the performance difference between BHJ, BRJ, and RJ based on the join sizes in SF 100.

Small Build Size (Q2, Q11): These queries contain only joins with a small build side, which fits in the caches. This is advantageous for the BHJ because there are no cache misses. *Query 2* contains nine different joins, whose build sides, even for SF 100,

⁵All generated query plans are similar to the ones reported by the Umbra Webinterface umbra-db.com/interface.

are smaller than the LLC. The 2 GB probe side causes materialization overhead, which is more significant for the RJ than for the BRJ.

In *Query 11*, the largest build side is 480 KB, so the global hash table fits in the L2 cache, making the partitioning phase redundant. The BRJ performs better than the RJ in both queries because it can avoid most partition overhead by pre-filtering the tuples.

Single Join Queries (Q4, Q12, Q14, Q19): For these queries, the number of pipelines is overseable, and the join mostly dominates the query runtime. *Query 4* contains one join of orders and lineitem that clearly dominates the query. The Bloom filter pays off since the join's build is pre-filtered. It can discard around 80% of unjoined tuples, for a predicate with 3% selectivity, and thus reduces the partitioning overhead. Even though its build side does not fit in the LLC for SFs larger than 10, the BHJ's performance remains constant, thanks to the buffers introduced by relaxed operator fusion. *Query 12* spends most of its time scanning the lineitem relation using it as the build side for a join with the orders relation. Once again, the bottom-most selection discards the majority (99.5%) of the tuples, but the resulting build side is 87 MB for SF 100, which is four times the LLC size. As before, the prefetching keeps the BHJ's performance stable, and the RJ cannot keep up with the BRJ. *Query 14* joins 1% of the tuples from lineitem with part which are 209 MB and 560 MB in size, respectively. As both sides are roughly equal in size, both BRJ and RJ perform well for a high enough SF. *Query 19* divides its runtime between filtering and joining the lineitem relation. The build side is only 2 MB in size, and fits in the LLC. The BHJ cannot significantly outperform the BRJ because the Bloom filter drops 90% of tuples before the partitioning phase.

Otherwise dominated Queries (Q3, Q10, Q15, Q16, Q17, Q18): In these queries, joins account for less than 40% of the total runtime, which limits the effect that the join implementation has on the overall performance. *Queries 15, 16, 17, and 18* are dominated by grouping of tuples. *Query 10* is dominated by scanning and selecting the base table, while *Query 3* is dominated by a group join. As a result, the differences in the join performance are minor for large SFs, as other operators dominate the query runtime. For small scale factors, however, the BHJ is superior.

Complex Queries (Q5, Q7, Q8, Q9, Q21, Q22): These queries contain various joins with different build and probe side sizes. We cannot explain the effect of the join performance solely based on the query plan and the total execution time. In the following sections we check if there might be a case to use the BRJ as a performance booster for each join.

Materialization Strategies: Late Materialization (LM) only helps when we substantially reduce the tuple width at selective joins. For example, in *Query 8*, LM reduces the build side in four out of seven joins. Or *Query 20*, where the result consists of two text columns, which are only present in the output. Materializing them late pays off,

reducing the probe side size by two-thirds. When using LM in *Query 14*, however, we only reduce the build size by 8 B. The random access for all build side tuples outweighs the positive effect.

Individual Join Comparison

The analysis in the previous section shows that most TPC-H queries perform multiple joins. Using just one join implementation for the whole query can lead to suboptimal performance. However, analyzing the impact for each join in a query plan in our system is challenging because all joins are part of pipelines (cf. Section 3.4.1), where all operators of a pipeline are fused to pass the tuples in registers and efficiently organize the code in tight loops.

Thus, we have examined all possible permutations of the join plan to compare BRJ and BHJ with TPC-H SF 100. To evaluate each join in the query plan (e.g., the second join), we computed the pairwise difference in performance when all other joins were fixed with one implementation, and we only varied the hash join algorithm used for that join. We show results for selected queries in Figure 3.12, where the x-axis denotes the join number within the query plan, and give an overview for all the joins in TPC-H in Figure 3.1, where we break down the measurements in build and probe side sizes.

One key observation is that most joins are not relevant for the total execution time. However, for some of the *expensive* joins, choosing the optimal implementation makes a big difference. For example, the execution time can be up to 60% slower or up to 30% faster when selecting the BRJ instead of the BHJ. Therefore, we focus the rest of our analysis on queries with multiple joins, where the join implementation choice has the most significant impact.

In *Query 5*, a single join dominates the runtime difference between the BRJ and BHJ. This join uses the unfiltered `lineitem` relation as the probe side and has a much smaller build side. Even though the build side does not fit in the LLC, the size difference between build and probe side is 1:117 and too big for the BRJ to pay off (cf. Figure 3.12). *Query 8* also uses the unfiltered `lineitem` as the probe side, which is 20 GB in size in the differentiating join. The build side is a 1 MB filtered relation. As a result, the hash table fits in the cache, and the BHJ is 60% faster in total execution time.

In *Queries 7 and 9*, the topmost two joins dominate the runtime difference. Each has a large build and probe side. RJ and BRJ still cannot outperform the BRJ, because the build tuple sizes are over 48 B, making partitioning too expensive to pay off. Prefetching in the BHJ also reduces cache misses more effectively.

Query 21 is dominated solely by joins, and each join has different characteristics, as shown in Figure 3.13. The query has a left-deep join tree, which prevents long pipelines.

⁸Due to the materialization overhead, the RJ cannot finish processing Q8, Q9, and Q21 for SF100

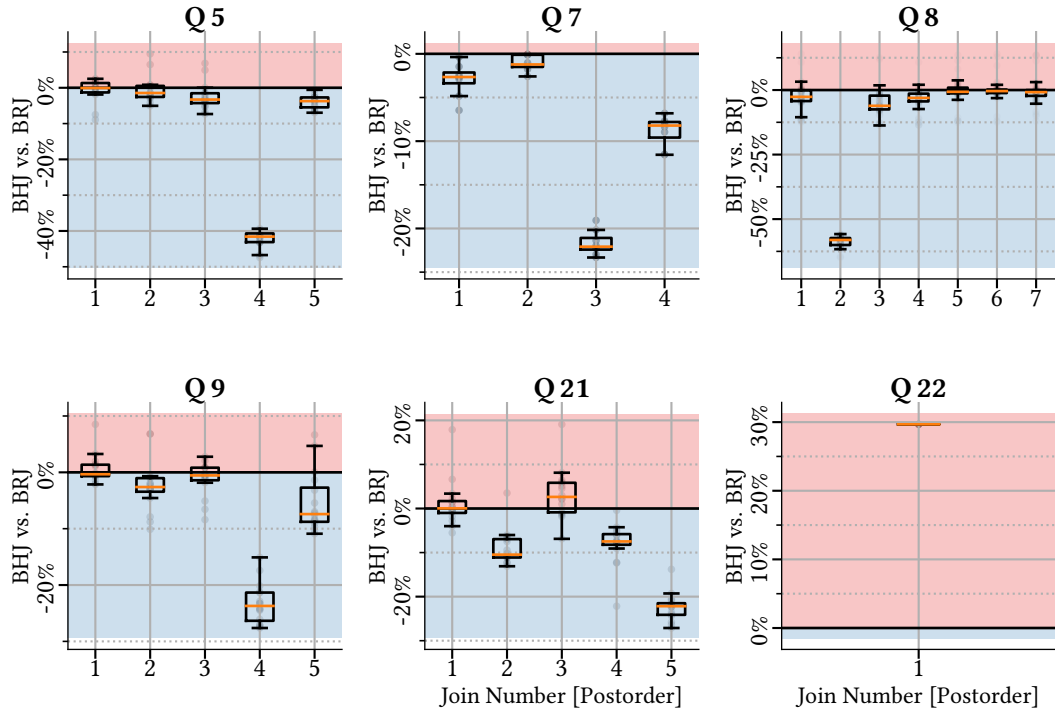


Figure 3.12: Relative impact per join for selected TPC-H queries (without Late Materialization)

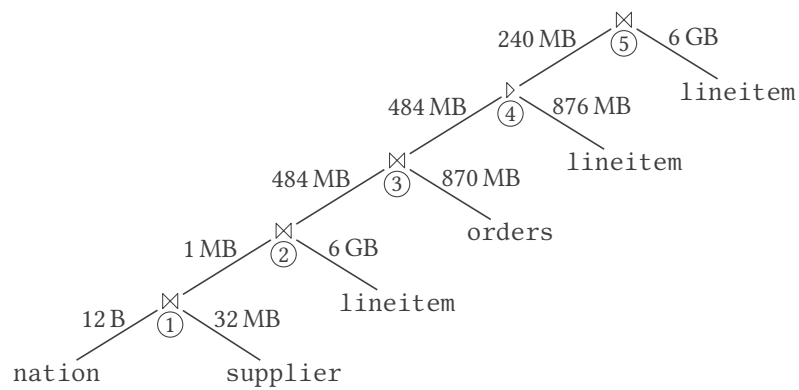


Figure 3.13: Q21 Join Tree annotated with build and probe size

① is negligible because of its size. For ②, the build side fits in the LLC. The Bloom filter can reduce the materialization overhead, so the BHJ is only 10% faster. ③ has narrow tuples and comparable sizes, so BRJ and BHJ perform equally. In ④ and ⑤, multiple factors lead to a suboptimal performance. The build side tuples are 33 B in size and the difference between build and probe size is not optimal. While Figure 3.12 shows that ③ is on average faster with BRJ, using BHJ for all leads to the overall fastest runtime.

Query 22 consists of two joins. One is a non-equi join, which cannot be handled by the hash join, so we do not enlist it in Figure 3.12. The anti-join reads the customer relation which is 155 MB in size as its build side and the unfiltered orders relation which is 1.8 GB as its probe side to evaluate a *not exists* predicate. Thus, each probe tuple is only 12 byte in size, including the hash value. Since small tuples work well for the BRJ, using the BRJ for this join improves the total query performance by 30% over the BHJ.

3.5.4 Isolating the effects of different factors

The analysis done so far has focused on the TPC-H benchmark, where join performance is concurrently affected by different factors. The combination of these factors leads to a completely different view on the RJ than in prior work (c.f. Section 3.5.2, [181]). In order to pin down the individual effects of each factor, we ran an extensive series of microbenchmarks. Combining all, we could isolate the cases where BRJ and RJ are superior to non-partitioned joins.

Effect of foreign key selectivity

One common pattern in all queries is that the BRJ outperformed the RJ due to selective foreign key joins (cf. Figure 3.2). In this experiment, we analyzed how varying selectivity affects each join's performance.

Our workload was based on workload A by Balkesen et al. [15], on which the radix join generally performs well (cf. Section 3.5.2). The build side remained unchanged for all selectivities. We modified the foreign key selectivity in the probe side while preserving its size to ensure that the number of processed tuples remained constant.

The results of the experiment are shown in Figure 3.14. We observe that both the BRJ and the BHJ are significantly affected by the varying selectivity. The BRJ is up to 50% faster than the RJ for low selectivities. However, when more than 50% of the foreign keys find a join partner, the RJ overtakes the BRJ because the computation time required to perform the filter lookup does not pay off – as it introduces up to one cache miss per lookup. We overcome this problem by sampling the probe side tuple

within our memory budget.

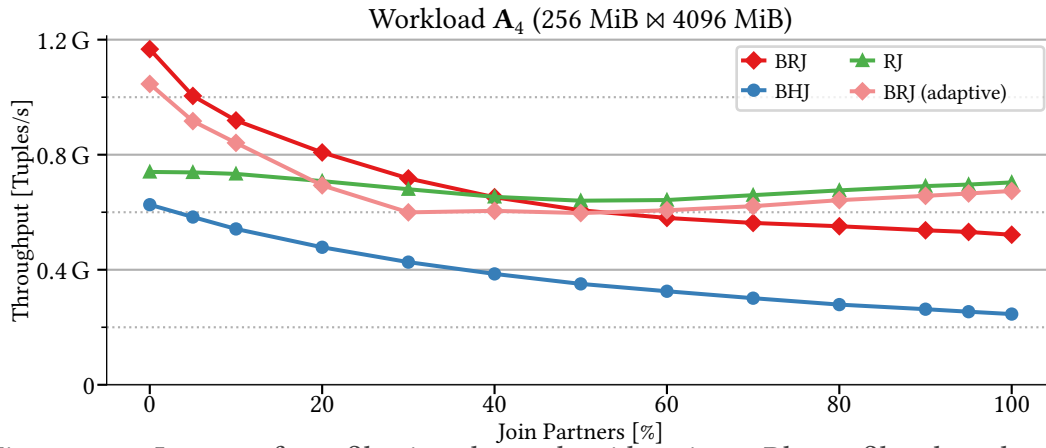


Figure 3.14: Impact of pre-filtering the probe side using a Bloom-filter based early probe

while probing the Bloom filter. This allows us to switch off the filter adaptively in case almost all tuples pass the filter, which introduces a minor overhead, mostly below 10%. We note, however, that TPC-H and real-world queries usually have selectivities below 25 % (cf. Figure 3.2, [29]). This experiment shows why the BRJ performs better than the RJ in TPC-H. We further note that the RJ is 10 to 40% faster than the BHJ for low selectivities, when all other parameters are near-optimal.

Effect of payload size

The payload size also influences join performance. Some joins have small payloads, but that is not always the case since the columns, e.g., may contain strings (cf. Figure 3.2). To isolate the effects that the payload size has on the performance of the RJ and BHJ, we set the foreign key selectivity to 100%.

Once again, we based our workload on the unskewed workload A by Balkesen et al. [15], where the radix join generally performs well (cf. Section 3.5.4). The build

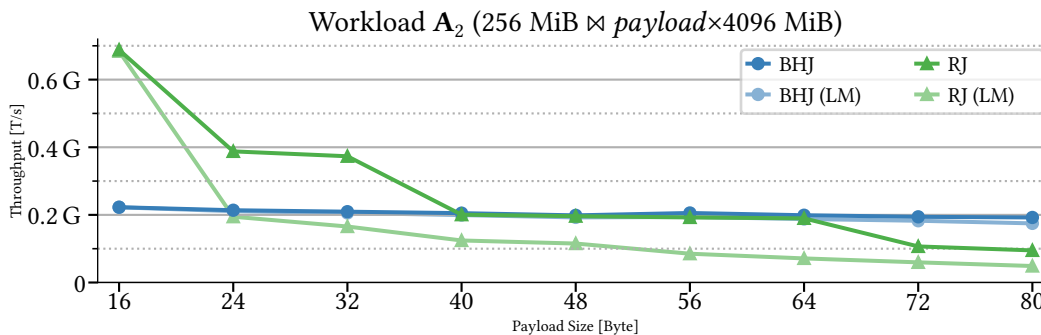


Figure 3.15: Impact of payload size on join performance

Table 3.3: Throughput [T/s] with and without Late Materialization

	LM	no LM	benefit
BHJ	452 M	453 M	±0 %
BRJ	656 M	487 M	+35%
RJ	341 M	153 M	+122%

side remained unchanged. We modified the probe tuple size by adding multiple 8 B wide columns with randomized integers. We used up to 8 payload columns, leading to a maximum payload size of 64 B. Together with the join key and its hash value, our tuples were at most 80 B wide.

Our queries are similar to the following with one payload:

```
SELECT sum(s.p1) FROM build r, probe s WHERE r.k = s.k;
```

This query materializes 32 B per tuple: 8 B for the payload, 8 B for the key, 8 B for its hash value, and 8 B padding. We show the results in Figure 3.15 and notice that the performance of the RJ is more affected by the payload size than the BHJ. The RJ performance degrades by a factor of 7, while the BHJ remains constant for five times larger tuples. Also, the use of SWWCBs is visible as the tuple sizes are padded to the next power of two. We do not use buffers for tuples larger than 64 B because padding would lead to higher performance losses than the benefits of non-temporal streaming.

LM lowers the performance, since the selectivity is at 100% and we have to additionally store the tuple id, leading to 24 B wide tuples. The RJ performs strictly worse due to cache misses introduced by random access after the join phase which could be addressed by radix decluster [127]. The BHJ is not affected by LM because there are no intermediate results.

The performance of the BHJ is memory-bound (i.e., affected primarily by the latency of random memory accesses). Hence the tuple size does not affect its performance significantly. The RJ, however, is bandwidth-bound. The materialization costs heavily influence its performance in the partitioning phase, which is directly dependent on the payload size (cf. Section 3.5.2). The RJ is up to three times faster than the BHJ for small tuples, but it completely loses the advantage once the tuple size exceeds 32 B.

Combined effect of payload size and selectivity

Both our previous benchmarks cannot individually show the benefits of LM. However, if we vary and analyze selectivity and payload size, we can see its benefits. We modified the workload with 5 % selectivity from Section 3.5.4 by adding columns to the probe side, like in Section 3.5.4. We used four 8 B columns which total 40 B including the

hash value. Using LM, we only had to materialize 24 B before and could fetch the remaining 24 B after the join.

Analyzing the results from Table 3.3, LM doubles RJ’s performance because it halves the necessary materialization. The thereby introduced random access has no negative consequences since only 5% of the tuples require it. Yet, it is still slower than the BRJ without LM, which follows the idea of sideways information passing [189] to prune most rows even before partitioning. However, LM gives the BRJ a significant boost by reducing the materialization, making it almost 50% faster than the BHJ. The BHJ does not materialize the intermediate result, so there is no benefit.

Effect of pipelining

When lining up multiple joins in a pipeline, the effects of both factors (selectivity and payload size) amplify each other. This is particularly bad for chaining RJs. Each RJ in the pipeline requires materialization and adds its column to the payload size, effectively enlarging the tuple size as the pipeline depth increases. This workload is a typical case for queries operating on a star schema where the central table connects various fact tables for additional information.

To evaluate the effects of the pipeline depth, we used the same workload as before, but instead of summing up the payloads, we used them as keys for fact tables, which resulted in a star-schema benchmark. Thus, we added multiple copies of our build side table containing randomly permuted rows. So we still achieved 100 % selectivity and could investigate the pipelining effect isolated. The optimizer had to use the central table every time because its keys connect the fact tables, finally resulting in a query plan with a single long pipeline (cf. BHJ in Figure 3.4).

We show the results in Figure 3.16, where we observe the throughput for each join in the pipeline. In the ideal case, the throughput is constant, which means that pipeline depth and join execution time do not correlate. This is indeed almost the case with the BHJ.

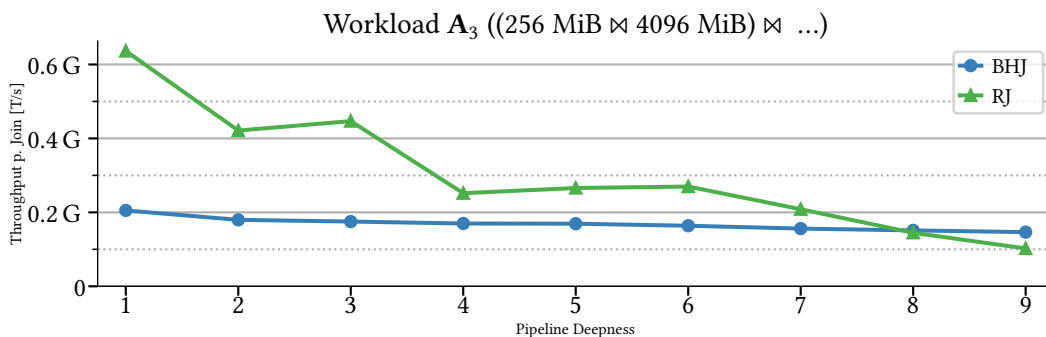


Figure 3.16: Impact of pipeline depth

The performance of the RJ, however, decreases proportionally to the length of the pipeline. Materialization overhead and memory bandwidth limitations add up, ultimately slowing down the join.

Effect of skew

To evaluate the effect of skew, we populated the foreign column in the probe relation with Zipf distributed data and varied the Zipf factor between a uniform distribution and $z = 2$, which resembles high skew. The same set-up was used by Balkesen et al. to evaluate the implementation of their PRJ (cf. Section 3.3.2) and their NPJ. The results of the experiment are shown in Figure 3.17.

We note that both the NPJ and BHJ benefit from an increase in skew as the workload exhibits better temporal cache locality and incurs less random memory accesses during the probe phase. Blanas et al. [24] already reported similar observations. For both radix joins, however, the skew has adverse effects. The partitioning of skewed data leads to heterogeneous partition sizes, which complicates the partition scheduling. This is especially visible when $z > 1$, meaning more than 50% of the tuples find their join partner in the first 20% of the build relation.

For workload **A**, BHJ outperforms RJ once the skew is higher than $z = 1$, and is more than five times faster for $z = 2$. For workload **B**, the intersection happens later for the NPJ and not at all for the BHJ since both relations are equally sized and have narrower tuples, both of which are more favorable to the radix joins. Comparing PRJ and our RJ, both show similar runtime characteristics. Our implementation is up to

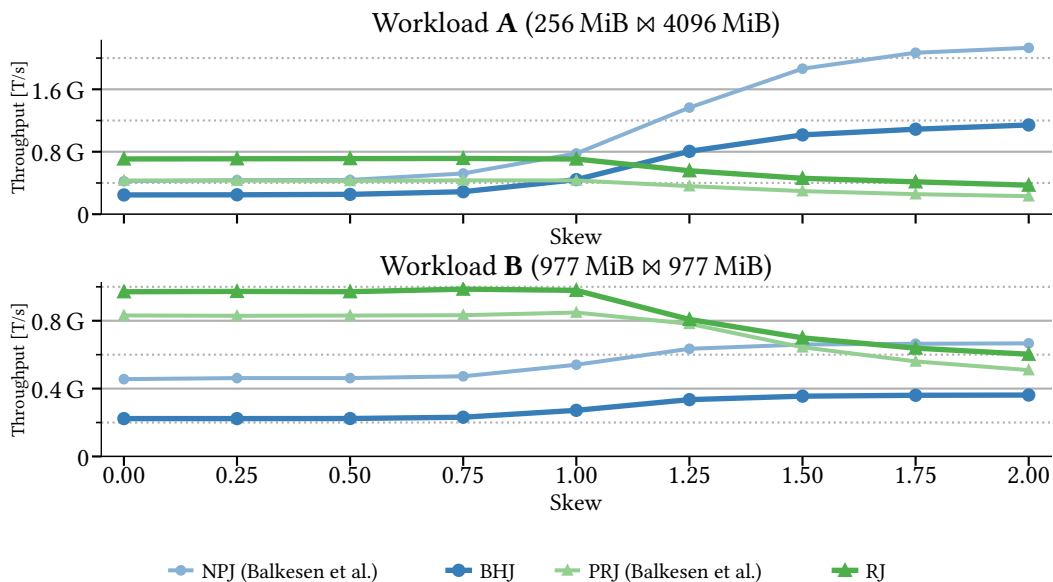


Figure 3.17: Impact of different Zipf factors

50 % faster because it parallelizes better, as we have already seen in Section 3.5.2. The BHJ profits from increased skew because it improves the cache locality. In contrast, the RJ loses performance for $z \geq 1$ since it throws partition sizes and scheduling out of balance.

Effect of build size

Prior work extensively studied this effect [13, 15, 24, 181]. As long as the build side fits into the LLC, the global hashtable does not suffer from cache misses, rendering partitioning useless. For larger hashtable sizes, prefetching reduces the cache misses for BHJ, while partitioning shows its strength for the BRJ.

We observed this behavior in the TPC-H measurements (c.f. Figure 3.11), where the BRJ only began to pay off in larger SFs. The in-depth join analysis presented in Figure 3.1 also shows that the LLC size is crucial: having a build side smaller than the LLC means there is no need for partitioning.

Effect of size difference

The difference in size between the build and the probe side has also been analyzed in prior work as we can see from the chosen datasets A and B, with size differences of 1:1 and 1:22. Schuh et al. also used a maximum difference of 1:10 [15, 181]. The reason is that a limited size difference ensures that the cost of materializing the partitions is in the same order of magnitude for both the build and the probe side.

We already observed the negative effect of a too-large size difference in the TPC-H measurements (c.f. Figure 3.1). When build and probe side are in the same order of magnitude, the RJ performs well and might outperform the BHJ (depending on the values of the other factors). The BRJ can operate on a broader range of workloads since pre-filtering decreases the materialization overhead. For example, the size difference in Query 22 is 1:11 and the BRJ leads to a speed-up of 30%. In contrast, for Join 4 in Query 5 the size difference is 1:100 and the BHJ is 40% faster.

3.6 Discussion

In this chapter, we have addressed one of the most important join questions of the last decade: *When does radix partitioning pay off?* To do that, we integrated a state-of-the-art radix partitioned hash join into a main-memory DBMS and compared it against an optimized non-partitioned hash join implementation. Given the results from prior work, our expectation was to use it to boost some expensive analytical queries (e.g., from the TPC-H workload).

Surprisingly, the benefits of the optimized radix join (with NUMA-awareness, SWWCBs, and non-temporal streaming instructions) are barely noticeable for any

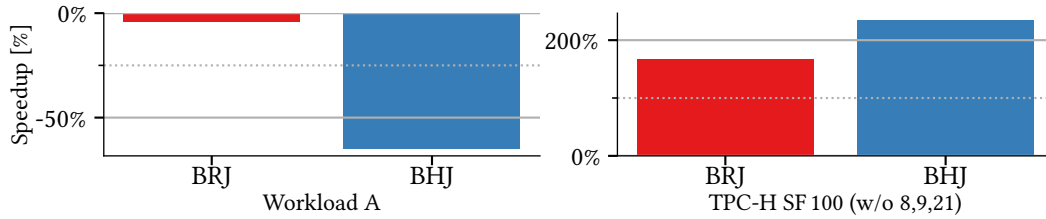


Figure 3.18: Speedup of different join implementations over the optimized radix join

join in TPC-H. After an in-depth inspection, we identified that partitioning (and materializing) tuples – which are not present in the join result – dominates its runtime, especially for selective joins. We tried Late Materialization to reduce tuple width, which sped up the RJ in some microbenchmarks but did not make a big difference in TPC-H. Lastly, we addressed this issue by implementing a Bloom filter in the probe side (BRJ). While this slightly slows down the join in the microbenchmarks, it is significantly faster for the TPC-H queries, as shown in Figure 3.18.

However, the non-partitioned hash join (BHJ) achieved comparable speed and a more stable performance than the BRJ for all queries, even with the BRJ optimized by Bloom filters. In fact, the BRJ is faster than the BHJ for SF 100 only for one join in TPC-H, and even then only by 30%. This shows a severe discrepancy with the insights obtained by prior work when the analysis was done only on microbenchmarks.

The second major contribution of our work comprises an extensive analysis of the performance of each individual TPC-H join (c.f. Figure 3.1) and isolating the effects of different workload factors with a series of microbenchmarks. The end goal was to synthesize the range of values for the key workload properties when using the radix join (and partitioning the data) actually brings benefits. Our findings are summarized in Table 3.4.

One key observation is that the RJ is very sensitive to any deviation from the near-optimal workload characteristics. While the BRJ delivers competitive performance for

⁹Late Materialization can handle large payloads when they occur with selectivity.

Factors	Workable	Beneficial
Selectivity	handled by Bloom filter	
Payload Size ⁹	$\leq 32B$	$\leq 16B$
Pipeline Depth	< 8 Joins	< 2 Joins
Skew (Zipf)	≤ 1	≤ 0.5
Build Size	$> LLC$	$\gg LLC$
Size Difference	$< \times 50$	$< \times 10$

Table 3.4: Workload Characteristics for Partitioned Joins

Factors	Prior Work	TPC-H	Real World [204]
Skew (Zipf)	0 – 2	none ¹⁰	yes
Payload Size	8 – 16 B	≈ 32 B	large (strings)
Pipeline Depth	1 Join	1 – 5 Joins	various
Selectivity	100%	low selectivity	low selectivity
Size Difference	1 – 25	mostly high	mostly high
Build Size	≫ <i>LLC</i>	mostly small	mostly small

Table 3.5: Workloads for Join Processing

a large range of queries (c.f. Figure 3.1), it seldom can reveal its full potential and bring performance improvements over the non-partitioned alternative. Theoretically, we can expect up to a 300% improvement by choosing the radix join. In reality, for some cases we even observe a performance drop because the required workload conditions are not met, e.g., the payload is not narrow enough. This makes it difficult for the optimizer to reliably predict the expected improvement from choosing the radix join over the hash join.

Putting the previously researched datasets and TPC-H into perspective, it becomes clear that past research took place on a relatively narrow range of data. We extended the applicability of the RJ to varying payload sizes and selectivities. While this makes it easier for practitioners to use it, it is still difficult to judge if the insights obtained from that evaluation are also applicable to their workloads. Although, TPC-H is synthetic, it still provides a broader range of queries and data properties (Table 3.5). TPC-DS did lead to similar insights. In the Join Order Benchmark [116], the RJ performed worse because it is string-processing heavy. Actual real-world data is even less suitable for the radix joins with its non-negligible data skew and high emphasis on string processing (and wider payloads).

We have shown that integrating the optimized radix join in an RDBMS is a non-trivial process and requires additional modifications to make it competitive for selective queries. Even then, choosing *when* to use it to gain a performance advantage requires many parameters to be satisfied and be accurately known by the optimizer at runtime. So unless the radix join is beneficial for other reasons, e.g., larger than main-memory working sets, we express reservations that implementing the radix join in a general-purpose production system justifies the added complexity.

¹⁰JCC-H [26] provides a more realistic drop-in replacement for TPC-H with skew. It puts even more pressure on the radix join.

CHAPTER 4

Sub-Operators as First-Class Entities

*Excerpts of this chapter have been published in [16].
With contributions from Jana Giceva.*

A wealth of technology has evolved around relational databases over decades that has been successfully tried and tested in many settings and use cases. Yet, the majority of it remains overlooked in the pursuit of performance (e.g., NoSQL) or new functionality (e.g., graph data or machine learning). In this chapter, we argue that a wide range of techniques readily available in databases are crucial to tackling the challenges the IT industry faces in terms of hardware trends management, growing workloads, and the overall complexity of a rapidly changing application and platform landscape.

However, to be truly useful, these techniques must be freed from the legacy component of database engines: relational operators. Therefore, we argue that to make databases more flexible as platforms and to extend their functionality to new data types and operations requires exposing a lower level of abstraction: instead of working with SQL it would be desirable for database engines to compile, optimize, and run a collection of *sub-operators* for manipulating and managing data, offering them as an external interface. In this chapter, we discuss the advantages of this, provide an initial list of such sub-operators, and show how they can be used in practice.

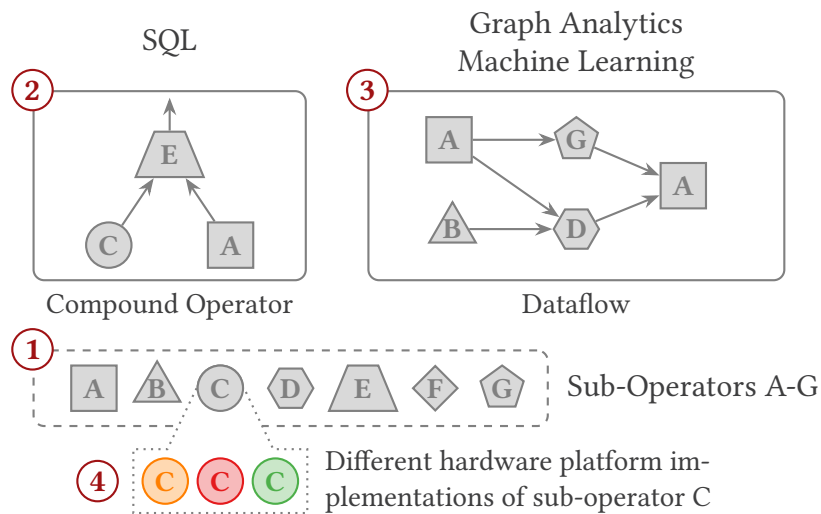


Figure 4.1: Sub-operators ① build more complex data operations ② or dataflows ③, where each sub-operator can be implemented on multiple hardware platforms ④.

4.1 Motivation

Databases have been a cornerstone of enterprise computing for decades. As is often pointed out, they offer what very few other systems, if any, provide: a powerful declarative language, a model and algebra to enable reasoning about programs, sophisticated compilation and optimization technologies, and a wealth of fundamental techniques to support very high throughput rates. All this while providing consistency, availability, and strong recoverability guarantees. Nevertheless, more and more users have been turning their backs on databases in the pursuit of flexibility and performance, willingly giving up the enumerated guarantees. For example, building directly upon intermediate formats like Apache Arrow has grown in popularity, offering more flexibility for storing and processing data. While this simplifies things in the short run, it makes management more complicated in the long run, for instance, when synchronizing data. The same holds for big data frameworks like Spark, which demonstrate the expressivity of offering finer granular operations for constructing various dataflows. This supports many use cases but lacks some advanced features, such as an optimizer. While this simplifies ingesting data, it increases the risk that your data lake evolves into a data swamp if not taken care of. We argue that there is no reason why traditional databases cannot support more flexible ways of accessing and working with data.

Currently, both big data processing and hardware advancements are driving the community to develop various techniques. These include domain-specific languages (DSLs) tailored to particular applications [37, 84], cross-compilation techniques to enable execution on different platforms [183, 195], automatic parallelization plat-

forms for running at large scale [88, 208], and connecting different frameworks for cross-optimization [151]. Some of these mirror developments in the database world: new compilation techniques [96, 111], new data types and languages for dealing with them [131], optimizations for multicore [216], designs for GPUs [80, 157] and FPGAs [147].

In this chapter, we argue that the most concrete starting points for such innovations are the concepts developed around database engines. Moreover, a great deal of existing technology can be reused, such as operator models [124], compilation techniques [101, 107, 137], composability and orthogonality of operators [55, 103], optimization and scheduling techniques [113], etc. However, the only way to enable more flexibility is to change the explicit abstraction level of the database engine interface. Thus, the system should also expose *sub-operators* and provide them as an intermediate representation to other applications and compilers (Figure 4.1).

By sub-operators, we mean logical functions that perform fundamental data transformations and management tasks. We call them sub-operators because instead of implementing a full relational operation (e.g., a join), they implement relatively basic functions, for example, hashing, filtering, sorting, scattering, or gathering data. Obviously, some of these are already used within database engines (for optimization or compilation [27, 52, 96, 103, 175]), and the literature is full of new ideas for sub-operators tailored to new hardware (from data exchange operators tailored to RDMA [106, 174] to FPGA-based partitioning [92]). By exposing an *interface* at the level of *sub-operators*, we can transform the database into a *language runtime engine capable of processing much more than just SQL*. This includes different dataflows, like machine learning, graph processing, or easier mapping of the operators onto hardware.

Our proposal produces clear benefits and provides a more elegant and more efficient solution to existing challenges than current ad-hoc proposals. For example, by building complex dataflows using sub-operators, we can reason about the workflow's logic decoupled from the hardware implementation details. Using sub-operators as common building blocks for different dataflows would simplify the maintenance of large codebases, especially when addressing the challenges of a diverse and rapidly evolving hardware landscape, as well as resource disaggregation in the cloud. Cross-compilation of hybrid programs to heterogeneous hardware platforms (CPU, GPU, FPGA) will be made easier by enabling alternative sub-operator implementations. Additionally, small as they are, computationally expensive but frequently used sub-operators can be integrated into the hardware circuit logic, thereby influencing the design of future hardware architectures. This especially pays off in cloud settings, where a holistic approach to hardware/software co-design is of particular interest.

Furthermore, databases can become extensible in a way they currently are not. While UDFs still need to be parts of an SQL query or table functions that mimic database functionality to customize the query freely, sub-operators offer more degrees of freedom, while reusing as much of the system as possible. They can be modeled

and analyzed more efficiently and thus incorporated into a query optimizer without compromising performance. This also makes them a more natural fit than UDFs, which are out of the optimizer's scope [48]. In such a setting, sub-operators resemble instructions in a processor, available for both the optimizer to re-arrange and the compiler to combine as needed. They transform the database engine from a virtual machine for SQL programs to a language runtime executing a complex Instruction Set Architecture (ISA) made up of sub-operators, catering to a heterogeneous range of dataflow workloads.

If needed, sub-operators make it also easy to interface common runtimes like Weld [151], which offers a runtime to cross-optimize between different frameworks. Each sub-operator can also be lowered to WeldIR after performing all optimizations in the database context. Also, integrating UDFs is trivial, as they can be integrated as sub-operator that performs the defined task on the dataflow. However, with sub-operators, more different tasks can be expressed inside the database system, making it superfluous in some cases to combine different libraries.

Finally, many of the ideas and concepts developed for conventional operators are directly applicable to sub-operators. This makes the database a compelling platform capable of optimized compilation, extensibility, concurrency on top of non-functional properties such as consistency, persistence, and recoverability.

4.2 Sub-operators as first class entities

Figure 4.2 presents an overview of the solution we propose. We envision a database engine that exposes a set of *sub-operators as an interface* and combines them into more complex dataflows. Example languages that can run on top are SQL operators and dataflow models used by graph processing and machine learning systems.

Using the sub-operators as an ISA, the database engine becomes more like a language runtime, executing a rich set of alternative sub-operator implementations (as instructions) on heterogeneous hardware platforms. The goal is to support not only relational (column and row) data stores but also graphs, key-value stores, and extensions for complex data types currently stored as blobs.

4.2.1 Sub-operators and interfaces

At present, our choice of sub-operators is based on analyzing prior work that captures basic data access and compute patterns of dataflows that can be mapped to modern hardware. We consider a sub-operator to be included in the ISA if it denotes an important management task or if it operates at the right level of data granularity. When an operator is too fine-grained, it becomes difficult to optimize, while if is

Name	Type	Category	Description	Examples
Scan	▶	Sequential Access	Scan materialized data into stream	Tablescan, Buffer scan
Materialize	◀	Sequential Access	Materialize data into buffer	Output, Print
Scatter	◀	Random Access	Write to different memory locations	Build hash table
Gather	▶	Random Access	Read from different memory locations	Probe hash table
Map	▶	Compute	Process stream with mapping function	Hash, UDFs, Filter
Fold	◀	Compute	Reduce streamed elements by combining	Accumulate, Prefix sum
Sort	◀	Compute	Materialize and sort data steam	Sort, Join
Loop	→	Control Flow	Pass data or state to next iteration	K-means, Gradient descent

Table 4.1: Example of possible sub-operators

Declarative Languages, DSLs (SQL, LINQ, HiveQL, Spark, ...)

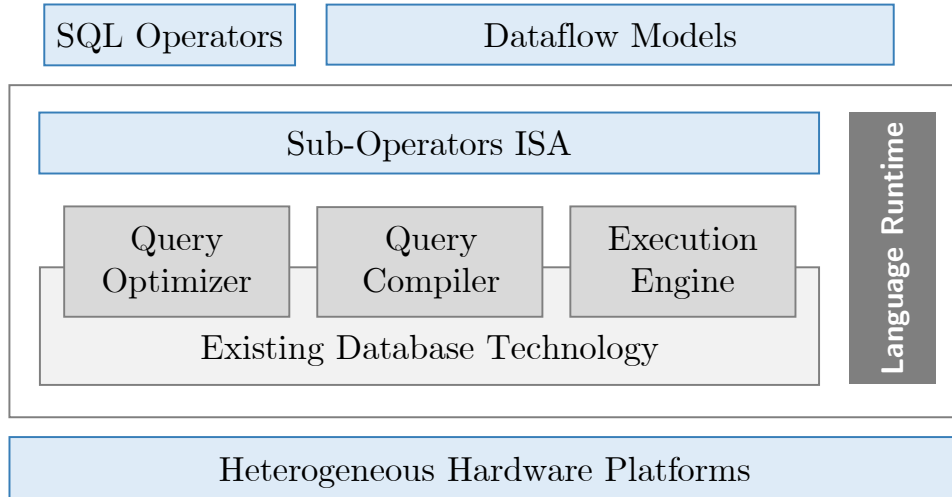


Figure 4.2: Overview of a sub-operator-based database engine

too coarse (e.g., an SQL join), it limits expressivity and the flexibility to benefit from hardware accelerators.

Table 4.1 shows and categorizes an example set of sub-operator types, which are sufficient to implement various dataflows. The set can be easily extended with other sub-operator types and concrete instances inspired either by prior work in relational databases (e.g., the exchange operator, range partitioning, string and bit manipulations, fuzzy string matching) or other forms of data processing like machine learning and graph-based analytics (e.g., complex statistical operations over array data).

Sequential Access *Scan* and *Materialize* ensure that the system can switch between streamed and buffered execution. It is crucial for working with materialized intermediate states or when distributing the compute pipelines to different hardware targets.

Random Access *Scatter* and *Gather* handle memory access to various locations. Combined, they can implement a join. *Scatter* takes a tuple stream and materializes it based on the scatter function, e.g., the tuple’s hash value. *Gather* fetches tuples from different memory locations based on an input stream and forwards the combined stream for further processing.

Compute *Map* and *Fold* provide support for functional programming primitives and typical Map-Reduce workloads. *Map*, for instance, processes a stream of tuples and applies a mapping function f to every element in the stream. It can return a varying amount of return values per tuple, including zero. One example function is predicate evaluation, but several different operations can be implemented, serving as building blocks for more complex dataflows. *Sort* is a handy sub-operator, which adds sorting

functionality to *Materialize*. It adds the property to the buffer that the data is sorted, which can be used in subsequent operators.

Control Flow Loop is different, as it is not part of a dataflow pipeline. It is necessary for controlling flow and to formulate iterative queries, often typical of incremental and converging workloads.

Depending on the use pattern, there may be compositions of sub-operators that are frequently used by a variety of dataflows and, thus, can be constructed as additional building blocks. For example, an efficient implementation of radix partitioning is often an integral part of relational operators (e.g., radix-join or aggregation).

Composing sub-operators

We use partitioning to explain how to combine sub-operators into a single, more complex operator. Partitioning consists of several phases, often involving two passes over the input data to enable efficient parallelization in a pipelined system [17]. The left-hand part of Figure 4.3 illustrates this. Phase 1 performs the first scan and computes the histogram of how the input tuples hash into the partitioning buckets. This phase is constructed using *map* (hash) and *scatter* (build histogram). In Phase 2, each thread calculates the prefix sum to determine each partition's span. To do this, it uses *scan* to read the partitions and *gather* to retrieve the histograms. Then, the *fold* sub-operator computes the prefix sums to determine the offset where each thread needs to write its share of tuples. The final phase performs the second pass over the input data and *scatters* the tuples to the precomputed locations.

Once constructed, the *new* partition sub-operator can be used as a building block for other relational operators or more complex dataflows. All existing efforts regarding

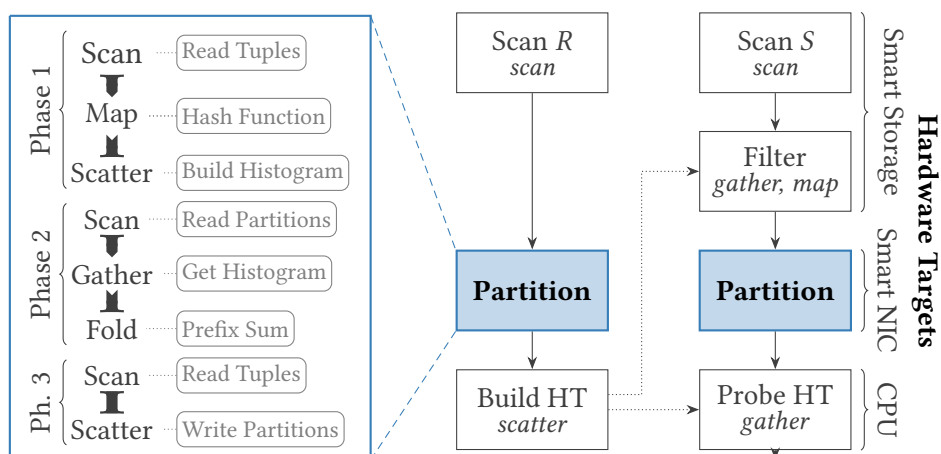


Figure 4.3: Composing a partitioner using sub-operators. The new *partition* sub-operator can be used in hash-joins.

hardware tuning and optimizing the implementation of various relational operators [99, 160], can be immediately applied to the sub-operators, automatically rendering them available to a wider set of operations. For instance, the benefits of software write combining, used to implement the radix-partitioning [207], can now be used for implementing one type of a *scatter* sub-operator. As a result, fast data scattering for many operations is available both in the relational domain and beyond.

Even more, sub-operators provide more intuitive mapping to data processing pipelines and are a natural way to begin automating the introduction of materialization points. We can use them to introduce mini-buffers, as demonstrated with relaxed operator fusion [132]. They also make it very easy to reason about distributing compute functions of a data pipeline to different targets. As shown on the right side of Figure 4.3, the data flow gradually transforms from reading from storage to compute, which is typical of data processing workloads. Close-to-storage workloads can be accelerated by smart storage, like computational storage [41, 61, 145], while partitioning is a good candidate for FPGA acceleration, and the final join logic works best on the CPU. This distributed use case is particularly attractive to every data system in a cloud context.

Also, the hash-join can be further augmented, as shown in Figure 4.3. For example, a semi-join reducer can be added to avoid materialization overhead in the join [17, 113] by plugging a filter just before partitioning the probe side to drop non-matching tuples early. Such flexibility also allows the database to react gracefully to changes in selectivity and adapt to the workload [55].

Finally, working with sub-operators, also enables us to efficiently compose hybrid relational operators. One example is the *hash teams* operator [94], which merges a hash-join and group-by aggregation to improve performance by performing several hash-based operations without repartitioning the intermediate results.

Interfacing sub-operators

More complex dataflows can be constructed using sub-operators as graph vertices. The edges of the dataflow represent data dependencies or how data moves from one sub-operator to another. When composing, it needs to be ensured that the input and output of the sub-operators are compatible. The input-output properties can be roughly classified as *buffered* or *streamed*. A streaming sub-operator (\blacktriangleright) directly accepts the input of its predecessor to further process each tuple immediately. A buffered operator (\blacktriangleleft) needs to process all tuples before further advancing, like *fold* and, thus, splits the dataflow at the materialization points into pipelines. The materialized buffers are stateful and encapsulate valuable data properties, such as sortedness, partitioned buffers, min/max statistics for pruning, and data distribution, which can help with additional logical optimizations.

All operators in a pipeline ($A \mapsto B \mapsto C$) can be compiled into a single function, which may be offloaded (run) as a compute kernel. This process is referred to as operator fusion and enables performant execution of query pipelines by keeping data in registers or hot caches. To increase performance, the optimizer can introduce minibuffers, for example to improve locality. Together with the aforementioned data properties, this exploits synergies to choose a faster implementation or avoid extra work, such as sorting data twice.

Common formats like Apache Parquet [9] or Arrow [8] can be used to ensure compatibility with other systems. The *scan* operator, for example, can use Parquet both for reading base tables and for in-memory communication. We can use Apache Arrow as an intermediate format for buffers to store data passed between the pipelines. The *materialize* operator serializes the data stream to allow efficient and convenient processing with existing libraries or user-defined code, similar to user-defined table functions.

4.2.2 Dataflows beyond standard SQL

Sub-operators offer a more expressive interface for constructing non-conventional operators and generic dataflow systems. One example would be *shared* operators, like a shared scan [201, 220] or a shared join [35], which combine reads of input relations, for example. They are thus effective at performing multiple queries at the same time [128]. It is also possible to combine subsequent relational operators, for example aggregation (group by) and join into a group join [66, 134]. This reuses the hash table for both probing and aggregating if join and aggregation operate on the same predicate.

However, most importantly, we aim to support all other complex dataflows, generated by higher-level declarative or domain-specific languages like HiveQL, LINQ, or Spark. Raven, for example, has previously demonstrated that in-DBMS machine learning can outperform dedicated frameworks [93]. Their system relies on a custom intermediate representation consisting of relational and linear algebra that allows for valuable cross-optimizations. Yet, they still need support for generating code for different hardware platforms and hence explore TVM [40] and Tensorflow [129].

It is along these lines that we propose lowering the data operations for various neighboring data processing domains (e.g., pagerank, k-means, connected components, graph connectivity) onto the sub-operator types: map, reduce, scatter and gather (to name only a few), or adding new ones (e.g., support for iterative and incremental computation).

To make things more concrete, Figure 4.4 shows how to compose one iteration of standard k-means using sub-operators. We first *scan* all points and hand them over to the *map* operator, which also accepts the current centroids as parameters. It processes each tuple and determines the closest centroid. The following *aggregation*

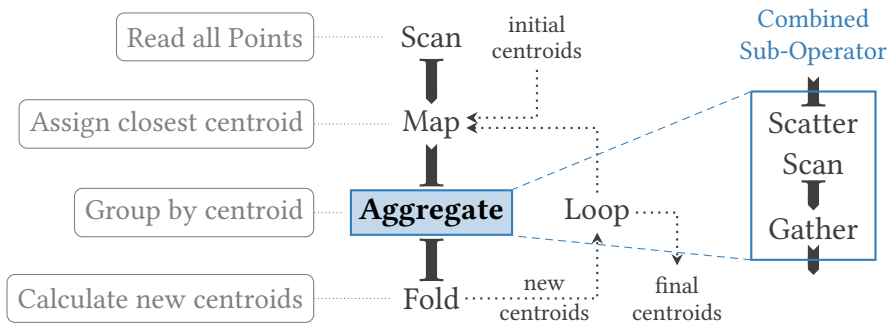


Figure 4.4: Schematic overview of one k-means iteration.

is composed of sub-operators and materializes the mapped point stream based on whichever centroid is closest. It restarts streaming the data once everything has been mapped and uses *fold* to calculate the new cluster centroids. If the centroids have changed since the previous iteration, *loop* passes the new ones as an argument to *map*, and the dataflow starts again by scanning the points. Otherwise, k-means has converged, and *loop* returns the centroids.

Upon closer examination of this idea, we notice that there are many key optimizations in these dataflow frameworks that can already benefit from existing techniques used in database engines.

Support for desired features such as differential computing has already been addressed by database systems. For example, memoization (caching and materializing intermediate results) within a program and even across concurrent programs can be implemented easily in a database engine, as the required techniques are already in use for queries. Prior work has explored their benefits when implementing memoization for streaming engines [53], or for data lineage and provenance [82]. Similar effort allows efficient algorithm re-computation when the input changes, needed by frameworks like Noria [73] and Naiad [130]. Finally, by allowing alternative sub-operator implementations, we can benefit from existing techniques for processing complex data types, such as images or documents.

4.2.3 Cross-platform compilation and execution

An important feature for modern systems is that they simplify maintenance of complex data processing code-bases for evolving hardware architectures and enabling easy cross-platform portability.

Offering alternative implementations tailored for different architectures allows platform-specific implementations of the sub-operators to be decoupled from the design of higher-level operations and data-processing algorithms. While this allows us to abstract from the specifics of hardware implementation, which simplifies reasoning when designing optimal dataflows, it also provides freedom for implementing

various flavors of sub-operators, each exploiting the full potential of the underlying architecture [91, 178, 197], deploying it to the cloud [88, 135], or even pushing the implementation down to the hardware circuit logic.

Furthermore, it can simplify reasoning for hybrid platform co-execution. Having multiple implementations of the same sub-operator allows us to execute portions of a dataflow computation on different platforms as shown in Figure 4.3. The actual deployment may depend on specific properties of the underlying resources, the data location, the sub-operator’s requirements, and many other factors.

TVM [40] already demonstrates the potential of such an approach. It is an end-to-end ML compiler that picks up ideas from Halide for image processing to separate compute and schedule [167]. Thus, it can optimize the required computations, like tensor operations, and schedule them to run on various hardware targets. Similarly, our sub-operators primarily describe the dataflow, allowing it to be flexible in scheduling the concrete implementation.

This idea extends beyond the compute resources of a single machine. With today’s trend for resource disaggregation and compute-capable devices (e.g., smart-NICs, programmable switches, computational storage), certain sub-operators (filtering, projection, partial sorting, partial aggregation) can be offloaded, either down to where the data sits [211], or the data can be processed as it moves (statistics, regular expression evaluation, partial aggregation, partitioning).

4.2.4 Hardware integration

Using sub-operators as common application kernels can also increase the influence that data-processing systems have on hardware design and implementation. Provided that the selected sub-operator instances are simple enough, the majority of them can be integrated into the hardware circuit logic. After all, specialization is one of the most effective ways of increasing performance, as successfully demonstrated by the SIMD instructions present in virtually all CPUs. SIMD instructions offer various primitives, for example, scatter/gather, for vectorized random memory access, which match the proposed sub-operators.

FPGAs have been shown to be an excellent platform for offloading data operations onto hardware logic, as well as for prototyping potential ASICs. Several enterprise systems are already using them accelerate data processing [164, 191] or encrypt it [10].

Following recent trends towards building heterogeneous architectures, more powerful co-processors (e.g., GPUs) are being placed the interconnect, to enable more effortless data transfer and co-processing. Intel’s research-oriented Xeon+FPGA architecture, in particular, has made the exploration and prototyping of costly data processing primitives on the FPGA even more appealing. For example, the work on FPGA-based histograms [87], while currently used to maintain accurate database statistics, can be repurposed as a sub-operator for data mining algorithms (e.g., for

cluster analysis). Similarly, approximated computations and various types of pre-computations (partial sorting or partial aggregation) implemented on an accelerator (FPGA, GPU, or programmable switches) can be used not only in SQL queries, but also for traditional soft computing algorithms, which are by design tolerant of imprecision, like in ML.

Such efforts will complement some of the work done by the architecture community on the design and implementation of various accelerators suitable for data-processing (e.g., Q100 [213], DRAM support for gather-scatter [185], processing-in-memory (PIM) architecture for graph analysis [2], or Oracle's DAX [155]).

Another take is Plasticine with Gorgon, which aims at having a coarse-grained accelerator (CGRA) for machine learning and database workloads in hardware [163, 202]. Their proposed checkerboard layout of reconfigurable materialization and computation units maps data flows in hardware to execute different flows in parallel batches. The materialization units have double buffers to ensure that they can ingest data while pushing out the data to the next flow in parallel.

Ideally, we can use the generality of the common sub-operators to influence future industry-scale architecture platforms. For instance, many operators can leverage on-chip circuits for automatic (hash and/or range) partitioning and routing of data across parallel entities (hardware threads or machines) [160]. One example of an enterprise chip design in which such a hardware-based partitioner could be integrated is Oracle's SPARC M8. It refined the DAX (data analytics accelerators) introduced by M7, which are specialized circuits on the memory controller used for the basic data processing operations of *selection*, *scanning*, and *decompression* as data moves between the DRAM and the LLC of the invoking core [3]. Rather than hiding such features behind a faster implementation of SQL, a database engine can encapsulate them as sub-operators and offer them as row primitives.

4.2.5 Impact on system design

Making sub-operators first-class entities of a data processing system also affects the design and implementation of the system stack. Figure 4.5 illustrates the affected components we discuss in the following sections.

The query optimizer receives the dataflow graph in a domain specific-language, usually derived from a SQL query or generated by the user-facing front-end of a dataflow engine. To make sure the optimizer can choose from the correct set of available sub-operators, they are passed along with their respective cost models to decide which of the sub-operators to offload. The Query Optimizer has to analyze the query at different granularities ahead of time and dynamically during runtime to find the best execution strategy as explained in Section 4.3.

The query optimizer passes a physical execution plan, which may consist of different alternative programs to the query compiler. It also knows all possible imple-

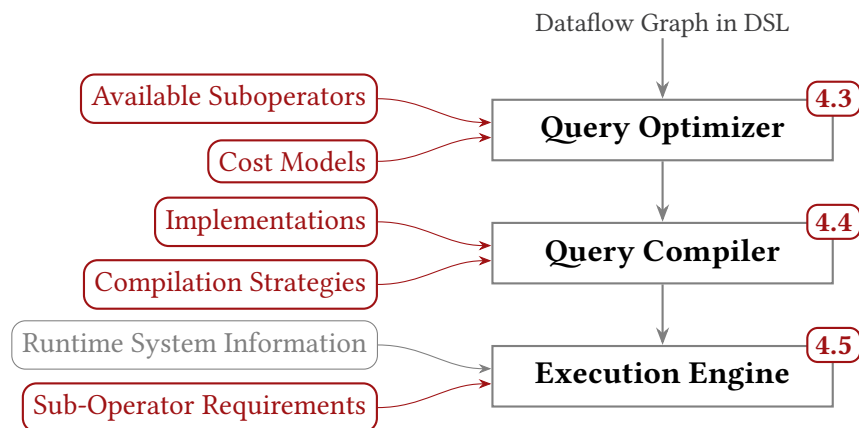


Figure 4.5: Overview of the sub-operator system stack.

mentations, e.g., for different accelerators or ISAs to run the sub-operators on, and combines the plan into executable chunks naturally formed by the data dependencies as pipelines. The compiler furthermore has to decide between different compilation strategies which, among others trade off throughput vs. latency. Section 4.4 gives an overview including a the challenges of generating code with different compilation strategies and architectures.

The execution engine keeps track of the runtime information to feed it back and optimize further executions. Thus, it can orchestrate where to offload compute and also change how the dataflow is shaped depending on the current workload as explained in Section 4.5.

4.3 Multi-Level Query Optimizer

One immediate challenge is that query optimization becomes more involved the more choices we have for executing a query. By adding sub-operators, we add a whole new level of customization to each query plan, which is why we propose optimizing statically in layers and dynamically during runtime.

High-Level Optimization: The first layer acts on higher-level operations before lowering to sub-operators. For example, by the time the database processes a SQL query, it has already parsed and optimized the query, e.g., decorrelating subqueries [141], and given hints, which physical operator to use [143]. This demonstrates how we still benefit from existing techniques.

Mid-Level Optimization: Our optimizer then receives a dataflow graph as input. Each relational operator is deconstructed into sub-operators, as outlined in Section 4.2.1. This opens up more optimization opportunities at the sub-operator layer, such as reasoning about which physical implementation to choose. This means that

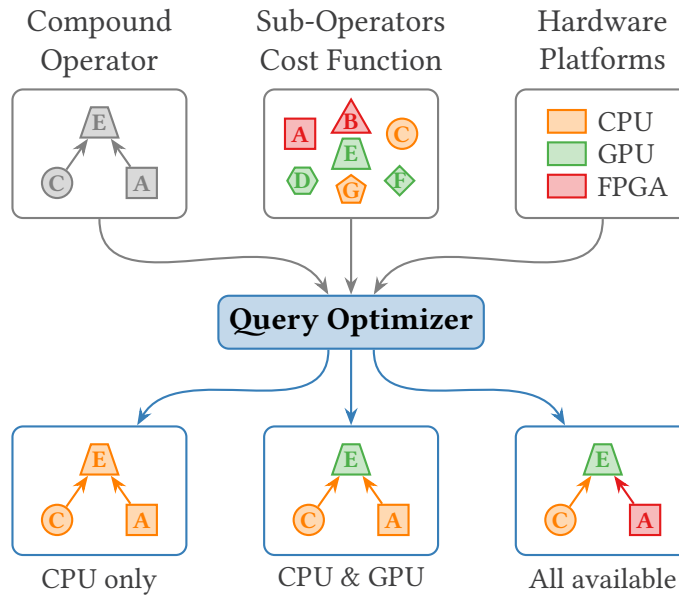


Figure 4.6: The query optimizer generates the optimal plan for the given target system using specialized sub-operators based on dataflow and cost functions.

we have to decide whether and where to offload computation, and as a direct result, how to split the dataflow between all available hardware resources. Relatively small sub-operators considerably simplify the analysis and derivation of their individual cost models compared to the cost functions for full SQL operators or UDFs.

Since sub-operators typically have a clear data access pattern and mostly consist of a single iteration over the data, building the cost models for each implementation can reuse a significant amount of prior work that has identified how to model the costs of the access patterns for different hardware platforms. We can target, for example, modern multicore machines [124], GPUs [81], or other heterogeneous architectures [206].

Low-Level Optimizations: Meta-frameworks like MLIR are used to lower the dataflow to target architectures [112]. Another alternative is to use a tailored emitter [75], which leads to different tradeoffs as detailed in Section 4.4. The optimizations can be applied after the sub-operators are assigned to a hardware target, consider the whole dataflow, and offer their own specific optimizations, such as constant propagation or auto-vectorization.

Dynamic Optimization: However, the opportunities of having so many compatible variants of the same sub-operator still poses a lot of challenges, Depending on where the data sits and how busy each of the disaggregated hardware resources is at the moment, offloading might be beneficial or not. Since we cannot foresee all these parameters while preparing the query, we also need runtime adaptivity. The idea is that, each sub-operator implementation alternative can be augmented with auxiliary

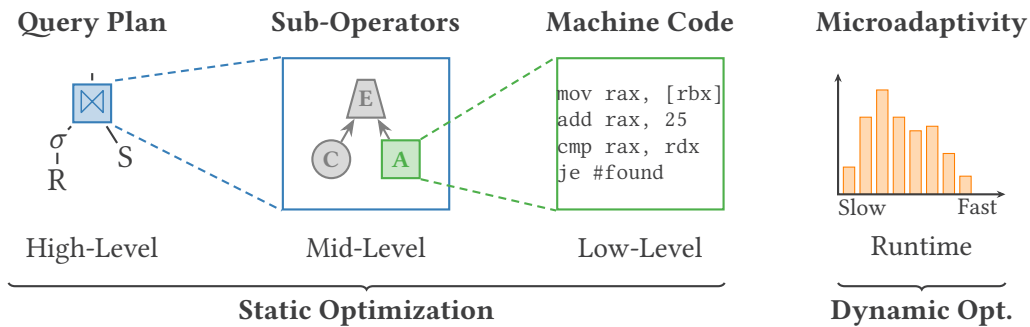


Figure 4.7: Optimization levels using sub-operators.

information, such as the cost of its execution on the particular hardware platform, or its resource footprint (i.e., the resource requirements for efficient execution). The latter is in particular useful for multi-query, parallel executions in a noisy environment (e.g., in the cloud) or microadaptivity, as we discuss in Section 4.5.

4.4 Query Compiler

The compiler takes the output from the optimizer and generates a program executable. In doing so, modern compilers already blur the boundaries between relational operators and operate on pipelines of sub-operators until a pipeline breaker is reached [96, 132]. Generating pipelines and pipeline breakers follows on naturally from our discussion on streaming vs. buffering sub-operators in Section 4.2.1.

Relational operators are already split into smaller units in the query compiler. For example, even if an operator consists of multiple stages that logically belong together (e.g., build and probe in a hash join), it is often physically separated into units as parts of different execution pipelines. Occasionally, even introducing extra-buffers, e.g., for cache locality, can speed up processing. [132] As a result, both compiler and execution engine do not operate internally with relational operators, but with their building components [113].

To further enhance the flexibility of data processing engines, a compiler could use ready-generated sub-operator implementations, either with different resource footprint(s) [20] or variants that match the desired hardware platform, as suggested by the optimizer. Ideally, the compiler directly uses the sub-operator implementations as a processor typically uses the ISA instructions.

These alternative sub-operator implementations can be either hand-crafted using existing optimization techniques for SQL operators or automatically generated. Frameworks like LLVM [111, 112, 183], Voila [77] or Lightweight Modular Staging (LMS) [175] support specialization and are already in use in systems like Tuple-

ware [49], HyPer [95], Umbra [140], and LingoDB [89, 90].

Finally, the mere idea of using sub-operators as an IR and explicitly exposing them as an external interface is highly compatible with recent proposals for meta-compiler frameworks like MLIR [112]. Combined with sub-operators as an intermediate representation, we consider this to be the only systematic way of developing database systems for heterogeneous hardware. A common intermediate representation enables our community to rely on proposals like MLIR for low-level optimizations or offloading to accelerators like FPGAs [183]. This intensifies the connection to the compiler community, and we can benefit from their progress and vice versa.

Umbra, for example, consists of a low-level instruction set called Umbra IR, which connects the different compilation backends with the SQL frontend and optimizer [96, 140]. Recently we looked into how to architect a query compiler to adapt to different hardware ISAs like moving from x86-64 to ARM64 or in general RISC architectures [75]. In the following, we look into how sub-operators can deal with both heterogeneity and HTAP workloads, which are omnipresent [75, 95, 140] by offering different compilation backends based on individual query characteristics and hardware.

4.4.1 CPU Heterogeneity: RISC on the rise

Heterogeneity is no longer only a matter when dealing with accelerators since CPUs with a RISC-ISA, mostly ARM-based (AArch64), are on the rise. This momentum is fueled by the stagnating performance increase of CISC x86-64 chips, while the application areas for ARM-based chips grow fast, which leads to the biggest disruptions in the processor market for two decades. Ten years ago, ARM-based chips were merely prevalent in mobile and low-end segments, while now successfully set foot into the consumer and server market [118, 192, 194].

This leads to the prognosis of rapidly increasing market shares in this decade, as shown in Figure 4.8b. RISC chips are furthermore increasing the weakening of traditional compute and data separation. The trend is going from a centralized high-performance machine to heterogeneous compute components, where ARM, and other RISC architectures, are already in use. Using this momentum, they plan to grow their revenue in traditional workloads and matrix workloads, characterized as real-time tasks, including AI inferencing (referred to as matrix workloads in Figure 4.8a).

While x86-64 was the only relevant architecture in the server and consumer market a decade ago, the ongoing development indicates that heterogeneity already increases at data centers giving a choice between ARM or x86 chips. Since x86-64 is a CISC instruction set, it offers many instructions with a simple memory model, which eases code generation. The developer can offload most responsibility of optimizing the IR instruction implementation to the processor because the CISC instructions often directly map to tasks described in the IR. ARM chips, however, offer a lot more flexibility in code generation, which makes the switch from x86-64 more significant

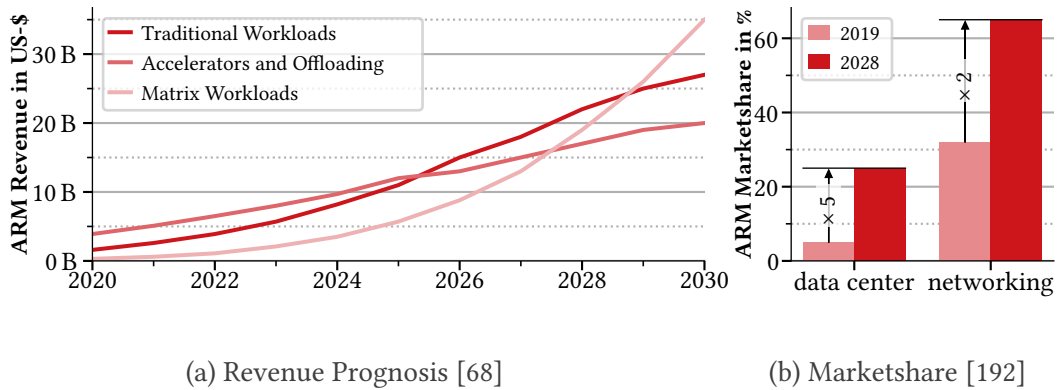


Figure 4.8: Prognosis of ARM usage in the server market.

than it sounds in the first place since both follow different IR design philosophies. A core philosophy of RISC chips is to have instructions more small-scoped so that they have to be combined. This on the one hand makes instruction selection more involved, but on the other makes code generation more nuanced and tunable. Thus, instead of mapping each higher-level IR instruction to a counterpart, multiple RISC instructions are combined to achieve the same [83].

While this sounds like a disadvantage initially, it does not have to be. The increased flexibility offers more room to tune the performance, especially when handling memory accesses that are more fine-granular [75]. However, the data processing system must use freedom with responsibility to avoid performance penalties. Furthermore, the situation gets more challenging since big cloud vendors like AWS (Graviton [12]), Alibaba (Yitian [217]), and Google (Ampere Altra [39]) use custom ARM-based processor designs. While these designs share most of their IR based on the ARM standard, each vendor can add custom instructions, e.g., accelerators or specialized vector instructions.

Having both x86 and different ARM cores readily available in the cloud makes considering heterogeneity important even when designing a CPU-centric dataflow system. This section outlines our proposal of designing a sub-operator-based database system from scratch and transfers the findings of porting our full-fledged DBMS Umbra to ARM [75] to sub-operators. It outlines design principles for compiling queries to utilize specific hardware properties like ARM’s fine-granular memory handling. The full experimental background can be found in [75]. Ultimately, combining domain knowledge, like the choice of sub-operators, and architecture awareness, like the processor’s ISA, is necessary to use the available resources best.

4.4.2 Compilation Strategies

When transforming the sub-operator plan to machine code as shown in Figure 4.7, the DMBS can use different strategies for code-generation [96]. Dataflow systems, e.g., Umbra [140], combine multiple strategies to choose the best-fitting approach depending on each query [102]. We use a similar set of properties as Gruber et al. [75] to evaluate and categorize these approaches. Based on the results, we recommend the most promising compilation techniques when implementing a sub-operator-centric dataflow system.

Properties

We rank the IR generation strategies based on three core properties focusing on the resulting performance, the development effort, and the specificity of the generated code.

Performance is one of the central metrics for code-generating dataflow systems and is directly visible to the end user. Code generation is mainly associated with the ahead-of-time compilation of programming languages like C or C++. In this case, compilation only takes place once. Thus translation and optimization are allowed to take longer because they are non-recurring and the time amortizes. In contrast, Just-In-Time (JIT) compilation focuses on continuous translation, scheduled by the remainder of the application [76].

While both techniques generate machine code, they are tuned for different properties. Generally, traditional compilers optimize for *high throughput*, while JIT compilers aim for *low latency* code generation. Thus, the quality of optimizations, the internally used IR, and the generation of machine code differ, which makes the performance relative to the actual use case. Dataflow systems need to set the focus both on *high throughput* and *low latency*. Otherwise, when focussing on throughput, compilation time will dominate short queries, like inserts or point lookups. Or vice versa, focusing on latency is not reasonable for long-running or recurring analytical queries since optimizations will quickly amortize.

Specificity expresses how well a dataflow engine can represent the algorithmic parts of a query. It focuses on the ability of code generation to utilize heterogeneous systems, not only stressing ARM support or fine-graded tuning like vectorization. Thus, it also covers co-processors or FPGA-based solutions like computational storage devices that are controlled by ARM processors. These require additional support for the specific architecture as well as hardware-specific control.

Domain Specificity describes how a particular strategy allows representing important algorithmic constructs for database systems (e.g., algorithmic operator details, memory accesses, etc.). Concretely, we rate the general possibility of expressing a given algorithm or construct in a given intermediate language. *Architecture Specificity*

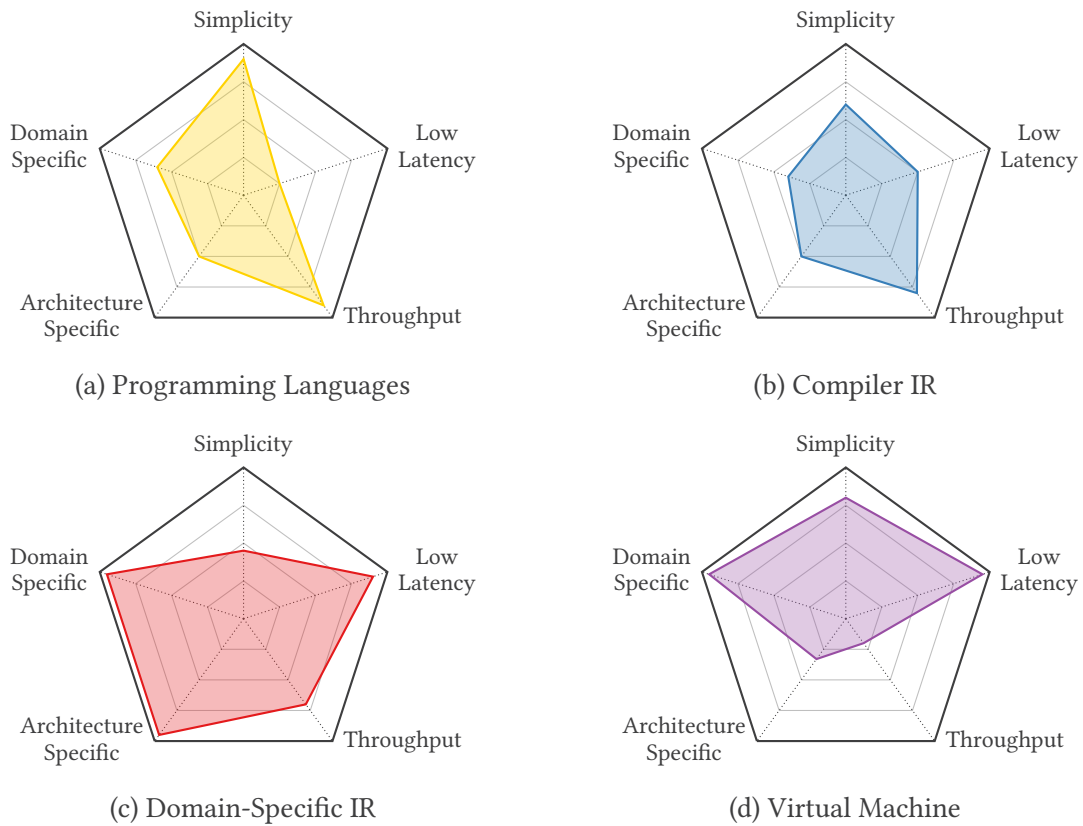


Figure 4.9: Qualitative overview of different query-compilation strategies' properties

assesses how well a given approach handles architecture-specific information (e.g., memory models, architectural constraints). The simplicity of such a representation allows a more straightforward description of algorithms, easing later code generation in contrast to unmaintainable constructs. Both sub-properties assess how tailored the optimized code is respecting the domain (dataflow) specifics and the underlying hardware architecture.

Simplicity rates the complexity of integrating a certain compilation strategy in a given dataflow system. Depending on the chosen approach, the *required knowledge* of the developer and the *integration complexity* vastly differs. A high amount of required knowledge may be a hurdle in initially integrating the compilation strategy. Optimizing for specific hardware properties is a complex, low-level task. Hence, simplicity combines the effort of integrating a strategy into the system and the architecture-related knowledge required for implementation.

Programming Languages

This category summarizes database systems that base their compilation on compiling programming languages to generate code for each query, mostly in C or C++ [52, 77, 215]. Using the same programming language for code generation and the generated code gives a clear advantage in simplicity, as it only requires profound knowledge of one programming language. In the end, only a call to an external compiler is required to transform the generated code into an optimized executable. Internal transformations are still part of the database system, but the final code generation, including instruction selection, is delegated to an external toolchain. While this allows the database developer to focus on algorithmics, not all domain-specific knowledge can be conveyed on the programming language level, like the higher-level construct of a query plan. The same holds for architecture awareness which is handled by the expressiveness of the language and the different compiler backends for the different architectures.

As the compiler toolchain takes on the heavy lifting in code generation, we get high throughput due to many fine-tuned optimization passes. However, this comes at the cost of high latency in query execution, making it well suited for either OLAP or streaming scenarios where the individual query compilation time becomes irrelevant [214, 215]. AWS Redshift combined programming languages with a high-performance compilation cache [11] to compile short-running queries efficiently by re-using cached fragments of the queries.

If simplicity and long-running queries are in focus, programming languages are the best choice for code generation in database systems since they relay most work to existing compiler infrastructure.

Compiler IRs

Building query programs with compiler IRs usually requires building the database engine around the compiler IR. This is more complex than an external call to compiled code but requires interaction with an already-developed module. The database system directly generates IR code, usually in SSA form, which it hands over to the compiler framework. HyPer is based on this idea by combining a C++ engine to generate and run LLVM code [137]. The generated LLVM code can then also call into pre-compiled C++ routines to tightly couple runtime and execution systems.

Generating IR based on a general-purpose IR reduces the simplicity since the developers have to know the underlying IR, which is usually closer to Assembly than a systems programming language. While this allows for more fine-granular control of the chosen instructions, it also increases the developer's responsibility. Due to the use of a general-purpose IR, it is not tailored to the domain-specific requirements of a data flow system. However, as a full-featured lowering, all algorithms can still

be expressed. Furthermore, the intermediate representation is mostly architecture agnostic and uses the compiler to tailor it to different architectures. Depending on how many optimization passes are selected for generating the code, latency, and performance vary.

Compared to programming languages, compiler IRs offer a better, more fine-granular, latency-throughput tradeoff at the cost of more implementation complexity. Compiler IRs offer a good balance and still offload most heavy lifting to existing toolchains.

Domain-Specific IR

In contrast to the previous strategies, domain-specific IRs also require the implementation of code generation and machine code backend. This needs in-depth knowledge about compiler construction and is more complex to integrate into the rest of the system, which reduces the simplicity. However, due to the in-depth knowledge, the domain-specific IRs with code generators can map dataflow operations best since they are designed around them.

Furthermore, by storing annotations for architecture-specific compilation, this approach generates the most architecture-aware code and can also hint at specialized instructions. Similar hints also help for high throughputs by compiling specifically for the dataflow use case. The direct emission of assembly instructions also leads to very low latencies, comparable to virtual machines.

Domain-specific IRs must be weighed against the effort of maintainability since they need to be designed, architected, and implemented. They give a good balance of the feature set leaning towards both performance and specificity, which makes them a great deal if the complexity is worth implementing them.

Virtual Machine

A virtual machine for dataflow execution implements all calls, usually in the same programming language as the remainder of the system is written. Thus, programmers only proficient in one programming language do not have to learn new abstractions, which makes them fairly easy to use in their simplest form. However, considerable work and knowledge are necessary to tune them, e.g., the register assignment.

The instruction set is by design tailored to the domain, but usually not aware of the architecture since the IR is not lowered to machine code. Still, specialized instructions for implementations can be called when executing the code on different architectures. Due to the co-design of IR and VM, latency is normally very low as few extra passes are necessary. This comes at the cost of runtime overhead since the code is not executed directly but in the VM. Thus, VMs a good starting point or especially

for short-running queries, as demonstrated by HyPer to mitigate lengthy compile times [102].

Conclusion

All approaches for query compilation have their strength and weaknesses and thus cater to different needs. So all approaches are integrated both into different research and industry data processing systems [52, 71, 75, 77, 95, 101, 132, 139, 140]. Figure 4.9 shows an overview of our defined properties for the query-translation strategies according to our analysis above.

When developing a purely analytical or stream-based database engine, latency is not the most important factor. Thus, emphasizing simplicity or throughput can be equally important, which makes programming languages a viable choice. On the other hand, when building a system with short to almost no transactions and a high volume of queries, latency is key. So, here virtual machines can shine with their almost negligible latency.

However, when combining the workloads into a single system, which can ideally run on different architectures, we need a more balanced approach. Tahboub et al. suggested that database developers use programming languages or existing compiler infrastructure (e.g., LLVM) for query compilation in database systems [196]. We do not fully agree with that since, compared to programming languages, compiler IRs are a step in the right direction but do not solve all challenges. They add complexity without fully mitigating issues like compilation latency or low expressiveness. We identified domain-specific IRs are the best choice for modern HTAP database systems, and Gruber et al. describe how domain-specific IRs should be designed for modern database systems [75]. It allows to address problems like latency and domain as well as architecture expressiveness. Toolchains like LLVM are a great blueprint and supporting technology for compiling databases, as Umbra shows [96]. The findings for the compilation backend also generalize to the whole sub-operator pipeline, as architecture awareness is an important factor in the overall system design (c.f., Section 4.4.4).

4.4.3 Performance Tradeoffs: Latency vs. Throughput

To support our claims with experiments, we use Umbra [140], which integrates all four backends [75, 96, 102]. This means we ensure that the intermediate representation for all backends is the same, e.g., uses the same query plan and UmbraIR code. To evaluate how well the system adapts to other hardware architectures, we test on an ARM device (*Apple Mac Mini with M1 processor*, 16GB unified main memory, *Arch Linux ARM*) and an x86 machine (*AMD Ryzen Pro 4750U*, 16 GB main memory running *Ubuntu 20.04.04*). We use *GCC 11.1* for compilation, and respectively *LLVM 13.0.1* for

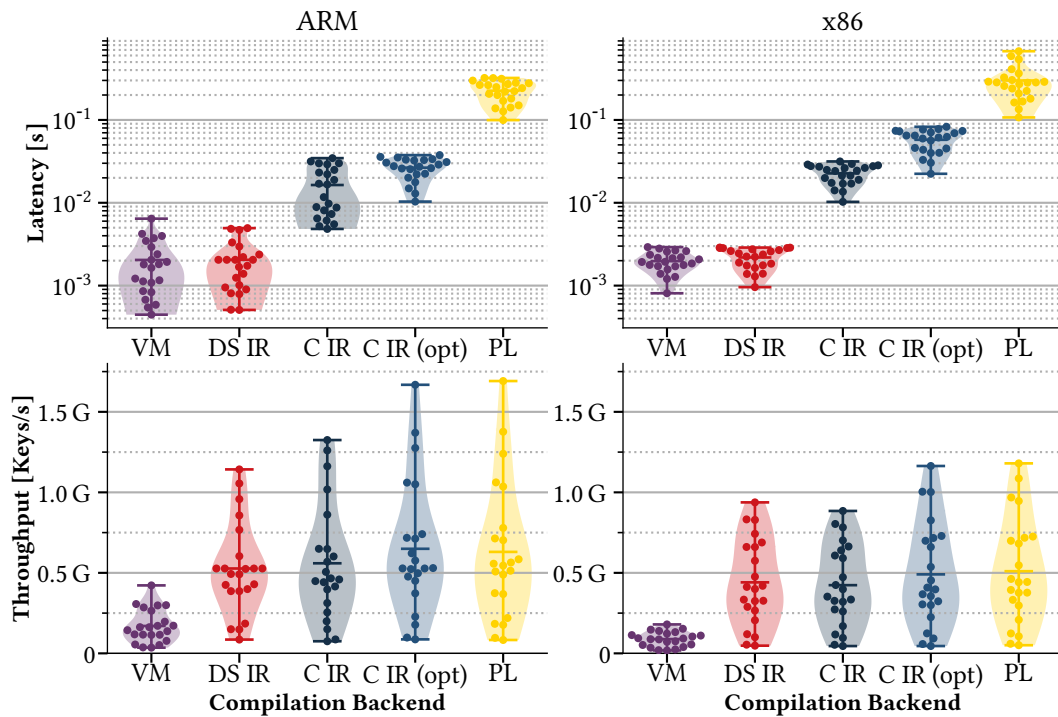


Figure 4.10: Comparison of different execution strategies in a database system.

JIT compilation in Compiler IR Backend. The benchmark is TPC-H, with a scale factor of 5. All tests are run single-threaded.

When comparing Figure 4.10 the different architectures, we see the same patterns emerging for latency and throughput. For latency, there are three different groups in the log-scale plot. The virtual machine and domain-specific backend, which do not have external dependencies and do not invoke a compiler, are the fastest and compile the queries in the milliseconds time range. Our compiler backend, which relies on parts of the LLVM compiler toolchain, comes in second place. The latency varies depending on how many optimizations are active but does not come close to using programming languages for code generation. Programming languages take the longest compile time since they have the additional overhead of parsing the textual program representation back into an AST and use more optimization passes.

However, the additional passes, which lead to an extensive compile time, pay off in throughput. Here, using a programming language as input is beneficial, since it uses the complete compiler toolchain which can find most optimization opportunities and, thus, results in the fastest code. However, using LLVM IR as input for compilation comes to a close second place. Thus, using a representation based on LLVM with an optimized set of flags is most promising when both throughput and latency are important, but there is a focus on throughput. The next-slower approach in terms of

throughput are domain-specific IRs or LLVM based compilation without optimization passes enables. Both roughly tie in throughput because disabling optimization passes resembles the approach taken by a domain-specific background with an emitter [75]. The emitter cannot perform overly involved code restructuring tasks in a single task and yields similar code as a compiler with most optimization passes disabled. However, due to extensive domain knowledge, the domain-specific emitter can outperform the compiler as it does on x86 [96].

Apart from that difference in performance, x86 and ARM show comparable performance numbers in our evaluation. Compilation on x86 is usually faster since each core's individual performance is higher and there is no differentiation between high-performance and energy-efficient cores. On Arm, we noticeably lose performance when compiling on an energy-efficient core. We also noticed similar performance behavior for both architectures in terms of throughput. The M1 performs better in general throughput, also partly because of its higher memory bandwidth, but the relative performance patterns stay the same.

Running on the virtual machine comes last with considerable distance on both architectures, making it unattractive for long-running queries. However, other factors like simplicity can still be a reason to focus on VMs, especially when only focusing on small, short-running transactions.

4.4.4 Architecture Conscious Sub-Operator Design

When developing the compiler for a new type of IR and focusing on performance, designing a domain-specific IR or using a compiler IR like LLVM are the most promising choices. With a compiler IR, you can build upon the foundation of an existing toolkit and leverage its potential for optimization without compromising too much on latency [102]. When deciding on a domain-specific IR, you can benefit from the architecture awareness and latency improvements, which provide fast execution times and low latency. Thus, domain-specific IR and compiler IRs offer the best tradeoffs for building a new query compiler. Toolchains like MLIR [112] offer a good foundation that is situated between both solutions since they build upon the shoulders of LLVM while offering the flexibility to define their own dialects and lowerings. Jungmair et al. use MLIR to develop their LingoDB System [89, 90], which combines MLIR lowerings with sub-operators.

The findings for the compilation backend generalize to the architecture of a sub-operator-based dataflow system, which has to be looked at as a holistic design in itself. Figure 4.11a shows the current design process of dataflow systems and their intermediate representations. The system design is based on the requirements of the dataflow plans and the constraints of the run-time system. The system's IRs primarily aim to offer the required feature set for translating the plans, which makes the implementation of the whole system principally top-down. This approach results



Figure 4.11: Design process of domain-specific and architecture-aware sub-operators for dataflow systems.

in high **domain expressiveness**, as for Umbra, but leads to issues when porting code generation to different ISAs.

Re-thinking the expressibility principle stated by Shaikhha et al. [186], we want to design a dataflow system **architecture-aware**. As shown in Figure 4.11b, the design process of the system has to be bottom-up and top-down simultaneously. The instructions sets of the target architectures are as important as the requirements of incoming data processing tasks since the target ISA is the lowest possible *domain-specific* language. Examining its feature and identifying potential restrictions results in a better system design. It allows adding information that benefits one architecture and does not hinder code generation for another.

Thus, the sub-operators form a translation layer between the high-level dataflow plans and multiple low-level execution targets. The additionally stored information benefits the multi-level optimizer that can both decide where to execute the operator and use the extensive knowledge for micro-adaptivity. Nonetheless, architectural constraints should not restrict the system design over a certain limit, e.g., by overly tying together the target ISA and the designed IR, since this would result in a non-portable design.

4.5 Execution engine

The query executor caches alternative implementations of the same sub-operator, and choose one of them dynamically during query execution, based on the preferences suggested by the optimizer.

If integrated within an (operating) system runtime, the query execution engine can also leverage information about current queue lengths and the utilization of various resources on heterogeneous hardware platforms (such as current GPU or FPGA use).

Such information is particularly important when executing concurrent workloads, and, hence, multiple dataflows. This, together with auxiliary information about the resource footprint and the requirements attached to the sub-operators can enable the execution engine to react better to runtime noise [72]. Similar to the *microadaptivity* technique, used by Vectorwise [166], this flexibility enables the system to adapt better to changing environments, which can result in both improved performance and better resource utilization.

Microadaptivity as well is crucial for choosing the optimal tradeoff when performing a query or also multiple queries at a time. Sub-operators ensure, that we have consistent interfaces as described in Section 4.2.1. Together with a runtime system that ensures that we can pick up execution after alternating the query plan [93, 200], this allows us to adapt to hardware utilization and improve the query plan by learning.

4.6 Related work

The benefits of working with sub-operators (or fragmenting traditional SQL operators into smaller components) are already known to our community; it is just that we have never properly formalized them or exposed the sub-operators as an interface.

For example, Dittrich et al. [55] propose splitting relational operators, like joins, into smaller fragments that allow finer, more granular performance tuning in the optimizer. Voila [77] uses a custom IR to chart the design space between vectorization and compilation, while Voodoo [157] shows that we can use an intermediate IR of database kernels to generate more efficient parallel executables for a variety of hardware platforms. Unsurprisingly, the ideas are also explored in other contexts. For instance, Love et al. [59] identify the most common *shuffle kernels* that can be used as building blocks for various graph algorithms. It is also an attractive approach for engines that support cross-platform execution. He et al. based their design of a hybrid CPU/GPU co-processing system GDB [81] on a set of data-parallel *primitives*, later used to implement common SQL operators. PyWren further demonstrates the elasticity and simplicity of serverless lambda functions as building blocks for maps [88]. Prior work also explored alternative methods of offloading parts of the operator computation onto a co-processor or accelerator [91, 156]. From an optimization perspective, our proposal shares a lot of challenges with workflow management systems that build dataflows from sub-operators that are backed by different variations, even though our focus is much closer to the hardware.

Novel framework proposals from the compiler community, such as TVM [40], LLHD [183], and MLIR [112] outline a more generic approach of lowering dataflow systems in a multi-level process of graph transformation and optimization through different granularities of intermediate representations before generating executables for various hardware targets, including accelerators. Jungmair et al. use MLIR to

build LingoDB, a database system with compiled performance based on different MLIR dialects as intermediate representations [89, 90].

Regarding the domain of hardware specialization, we have already referred to the DAX engines introduced in Oracle’s SPARC M7 [155] and refined in M8. Also, computational storage drives are having a revival [61]. Other examples are Google’s TPU, the Q100 data processing unit [213], the energy-efficient hardware partitioner [212], and Baidu’s data-processing accelerators [146, 149]. Reconfigurable hardware, like FPGAs, is an immediate choice for exploring operations that can be offloaded down to hardware. However, coarse-grained reconfigurable architectures (CGRA) [163, 202] are even closer to our concept of sub-operators. They are not only faster to re-program, but work at a coarser granularity with so-called parallel hardware primitives, which are powerful enough to express a variety of dataflows [162]. Templated-based FPGA framework designs customized for data-processing [122] are also worth considering.

We would like to emphasize that we propose building a language runtime using sub-operators as an ISA rather than locally extending the database to run programs, as was the case with stored procedures. An ISA, allowing execution of complex dataflows composed of sub-operators, can be used to augment existing efforts that revisit the interface between applications and databases [42]. In particular, if extended, the QBS [43] optimizer can use sub-operators to execute the application converted code beyond SQL-only queries. Along those lines, Weld [151] proposes a framework for jointly executing database workloads and machine learning tasks. However, instead of building on top of a database system, all systems lower to WeldIR which compiles them using LLVM.

4.7 Discussion

In this chapter, we present the benefits of lowering the explicit level of abstraction on which database engines traditionally operate by making *sub-operators* first-class entities. This enables a database engine to overcome the current limitations of the relational model and SQL and serves as a language runtime that executes an ISA of sub-operators. Such a change makes database engines more flexible platforms which execute various complex dataflows from a range of applications, so that they benefit from existing database technologies and non-functional properties, like consistency or recoverability.

Looking ahead, we believe that the proposed sub-operators are especially attractive in the context of today’s trends towards increased hardware specialization and resource disaggregation. In addition to accelerators, pushing compute functions either down to where the data sits (in smart storage) or as the data moves over the network (via smart NICs) is a promising way to address the widening gap of data deluge and the bandwidth capacities of today’s hardware. Thinking in terms of sub-operators is an

elegant and intuitive way of efficiently approaching the problem of executing dataflow pipelines in such deployment environments.

CHAPTER 5

Conclusion

The continuous increase in data processing workload and the growing heterogeneity of hardware have presented both challenges and opportunities for data processing engines. This thesis offers a comprehensive perspective on recent developments and devises strategies to address the challenges posed by the raised demand for data processing. We follow the path data takes through the data processing system, beginning with filtering during ingestion or retrieval, followed by data combination, and conclude with processing and compilation across different machines.

Initially, we tackle data ingestion and retrieval by presenting an extensive detailed benchmark for filters, including bloom and fingerprint filters. This benchmark identifies the optimal parameters for their performance and ideal optimization strategies every programmer should consider. We also explore use cases for both build and probe performance, enabling us to select the most suitable filter based on workload and algorithmic requirements.

Moving along the pipeline, we delve into join processing within the Umbra Database Management System. Given Umbra's efficient non-partitioned join, we examine its performance characteristics while integrating a fine-tuned radix-partitioned join. While this radix-partitioned join demonstrates competitive microbenchmark performance and the ability to handle generic SQL workloads, it falls short in standard benchmarks and most real-world applications.

Based on the initial results, we optimize the join with bloom filters in semi-join reducers and provide a comprehensive analysis of scenarios where partitioning is advantageous. By combining insights from our work on filters and partitioned joins, we find that on a single machine, partitioning often does not yield substantial benefits due to modern hardware's capacity to mitigate cache misses and stalls through simultaneous multithreading (SMT). This emphasizes the significance of runtime performance measurements instead of solely relying on performance counters for

optimization, where partitioning usually shows better memory bandwidth utilization and higher instruction-per-cycle measurements. So, combining microbenchmarks with close-to-real-world benchmarks and wallclock time with performance counters is the best combination to explore an algorithm's characteristics fully.

Finally, to harness the full potential of modern hardware, we propose sub-operators and look into heterogeneous execution. Our concept involves recombining the building blocks of diverse SQL and iterative operators into a new internal representation (IR) used within the DBMS. This chapter furthermore examines Umbra IR and how it gets transformed using various compilation backends. Ultimately, we advocate for systematic lowerings and a multi-level optimizer, enhancing both debugging capabilities and overall system understanding.

Outlook While this thesis greatly enhances the efficiency of data exploration and provides valuable insights for DBMS implementation, there is still untapped potential.

Working on and especially debugging and performance tuning code-generating systems inherently challenges the developer due to an extra layer of indirection compared to vectorizing systems. We performed benchmarks in various abstraction levels and with different tools to understand the impact of partitioning as detailed as possible inside the system. However, having a code-generating system can be exploited for our benefits in debugging by tooling that collects additional profiling data on the fly during the abstraction [21, 97]. This data presents an opportunity to tailor debugging experiences or explicitly show how queries are lowered to machine code using MLIR. This opportunity will pave the way for new tools that facilitate a deeper comprehension of the system and the benchmarks, ultimately improving its performance [29, 56].

The advancement of modern hardware and the evolving challenges posed by data processing will also continue to drive system design. With emerging technologies like CXL, interlink performance is high enough to consider disaggregated memory [7]. Collaborative efforts between hardware and software developers are crucial to achieving high throughput and cost-efficient data processing, especially in the context of environmental concerns. Standing on the shoulders of giants, with MLIR and clang, LingoDB proves that the proposal of sub-operators can be brought to life. [89, 90]

In conclusion, this thesis offers valuable insights for database development through a comprehensive exploration of data processing operators. As the data processing landscape evolves and technology advances, hardware and software developers must work closely together to achieve high performance while having an intuitive architecture. This will enable a more sustainable development by utilizing available resources best [115, 154] and re-using components wherever possible [16, 85] which leads to more cost-efficient data processing.

Bibliography

- [1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. “Materialization Strategies in a Column-Oriented DBMS”. In: *ICDE*. IEEE Computer Society, 2007, pp. 466–475.
- [2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoungh Choi. “A scalable processing-in-memory accelerator for parallel graph processing”. In: *ISCA*. ACM, 2015, pp. 105–117.
- [3] Kathirgamar Aingaran et al. “M7: Oracle’s Next-Generation Sparc Processor”. In: *IEEE Micro* 35.2 (2015), pp. 36–45.
- [4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. “Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems”. In: *Proc. VLDB Endow.* 5.10 (2012), pp. 1064–1075.
- [5] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. “The Evolution of LeanStore”. In: *BTW*. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 259–281.
- [6] Paulo Sérgio Almeida, Carlos Baquero, Nuno M. Preguiça, and David Hutchison. “Scalable Bloom Filters”. In: *Inf. Process. Lett.* 101.6 (2007), pp. 255–261.
- [7] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. “Programming Fully Disaggregated Systems”. In: *HotOS*. USENIX Association, 2023.
- [8] Apache Software Foundation. *Arrow — A cross-language development platform for in-memory data*. <https://arrow.apache.org>. 2019.
- [9] Apache Software Foundation. *Parquet — Columnar storage for the people*. <https://parquet.apache.org>. 2013.
- [10] Arvind Arasu et al. “Orthogonal Security with Cipherbase”. In: *CIDR*. www.cidrdb.org, 2013.
- [11] Nikos Armenatzoglou et al. “Amazon Redshift Re-invented”. In: *SIGMOD Conference*. ACM, 2022, pp. 2205–2217.

- [12] AWS. *AWS Graviton Technical Guide*. May 2023. URL: <https://github.com/aws/aws-graviton-getting-started/blob/main/README.md>.
- [13] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. “Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited”. In: *Proc. VLDB Endow.* 7.1 (2013), pp. 85–96.
- [14] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. “Main-Memory Hash Joins on Modern Processor Architectures”. In: *IEEE Trans. Knowl. Data Eng.* 27.7 (2015), pp. 1754–1766.
- [15] 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*. IEEE Computer Society, 2013, pp. 362–373.
- [16] Maximilian Bandle and Jana Giceva. “Database Technology for the Masses: Sub-Operators as First-Class Entities”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2483–2490.
- [17] Maximilian Bandle, Jana Giceva, and Thomas Neumann. “To Partition, or Not to Partition, That is the Join Question in a Real System”. In: *SIGMOD Conference*. ACM, 2021, pp. 168–180.
- [18] Ronald Barber et al. “Memory-Efficient Hash Joins”. In: *Proc. VLDB Endow.* 8.4 (2014), pp. 353–364.
- [19] Rudolf Bayer and Edward M. McCreight. “Organization and Maintenance of Large Ordered Indexes”. In: *SIGFIDET Workshop*. ACM, 1970, pp. 107–141.
- [20] Steven Keith Begley, Zhen He, and Yi-Ping Phoebe Chen. “MCJoin: a memory-constrained join for column-store main-memory databases”. In: *SIGMOD Conference*. ACM, 2012, pp. 121–132.
- [21] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. “Profiling dataflow systems on multiple abstraction levels”. In: *EuroSys*. ACM, 2021, pp. 474–489.
- [22] Michael A. Bender et al. “Don’t Thrash: How to Cache Your Hash on Flash”. In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1627–1637.
- [23] Altan Birler, Bernhard Radke, and Thomas Neumann. “Concurrent online sampling for all, for free”. In: *DaMoN*. ACM, 2020, 5:1–5:8.
- [24] Spyros Blanas, Yinan Li, and Jignesh M. Patel. “Design and evaluation of main memory hash join algorithms for multi-core CPUs”. In: *SIGMOD Conference*. ACM, 2011, pp. 37–48.
- [25] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (1970), pp. 422–426.

- [26] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. “JCC-H: Adding Join Crossing Correlations with Skew to TPC-H”. In: *TPCTC*. Vol. 10661. Lecture Notes in Computer Science. Springer, 2017, pp. 103–119.
- [27] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. “Breaking the memory wall in MonetDB”. In: *Commun. ACM* 51.12 (2008), pp. 77–85.
- [28] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. “Database Architecture Optimized for the New Bottleneck: Memory Access”. In: *VLDB*. Morgan Kaufmann, 1999, pp. 54–65.
- [29] Peter A. Boncz, Thomas Neumann, and Orri Erling. “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”. In: *TPCTC*. Vol. 8391. Lecture Notes in Computer Science. Springer, 2013, pp. 61–76.
- [30] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. “An Improved Construction for Counting Bloom Filters”. In: *ESA*. Vol. 4168. Lecture Notes in Computer Science. Springer, 2006, pp. 684–695.
- [31] Prosenjit Bose et al. “On the false-positive rate of Bloom filters”. In: *Inf. Process. Lett.* 108.4 (2008), pp. 210–213.
- [32] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. “Simple and Space-Efficient Minimal Perfect Hash Functions”. In: *WADS*. Vol. 4619. Lecture Notes in Computer Science. Springer, 2007, pp. 139–150.
- [33] Alexander Dodd Breslow and Nuwan Jayasena. “Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity”. In: *Proc. VLDB Endow.* 11.9 (2018), pp. 1041–1055.
- [34] Andrei Z. Broder and Michael Mitzenmacher. “Survey: Network Applications of Bloom Filters: A Survey”. In: *Internet Math.* 1.4 (2003), pp. 485–509.
- [35] George Candea, Neoklis Polyzotis, and Radek Vingralek. “A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses”. In: *Proc. VLDB Endow.* 2.1 (2009), pp. 277–288.
- [36] Pedro Celis, Per-Åke Larson, and J. Ian Munro. “Robin Hood Hashing (Preliminary Report)”. In: *FOCS*. IEEE Computer Society, 1985, pp. 281–288.
- [37] Hassan Chafi et al. “A domain-specific approach to heterogeneous parallelism”. In: *PPOPP*. ACM, 2011, pp. 35–46.
- [38] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A Structured English Query Language”. In: *SIGMOD Workshop, Vol. 1*. ACM, 1974, pp. 249–264.

- [39] Subra Chandramouli and Jamie Kinney. *Expanding the Tau VM family with Arm-based processors*. July 2022. URL: <https://cloud.google.com/blog/products/compute/tau-t2a-is-first-compute-engine-vm-on-an-arm-chip>.
- [40] Tianqi Chen et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: *OSDI*. USENIX Association, 2018, pp. 578–594.
- [41] Xubin Chen et al. “KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression”. In: *DaMoN*. ACM, 2021, 3:1–3:10.
- [42] Alvin Cheung. “Rethinking the Application-Database Interface”. PhD thesis. Cambridge, USA: Massachusetts Institute of Technology, 2015.
- [43] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. “Optimizing database-backed applications with query synthesis”. In: *PLDI*. ACM, 2013, pp. 3–14.
- [44] Raul F. Chong, Clara Liu, Sylvia F. Qi, and Dwaine Snow. *Understanding DB2 learning visually with example*. IBM Press u.a., 2005.
- [45] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (1970), pp. 377–387.
- [46] Saar Cohen and Yossi Matias. “Spectral Bloom Filters”. In: *SIGMOD Conference*. ACM, 2003, pp. 241–252.
- [47] Douglas Comer. “The Ubiquitous B-Tree”. In: *ACM Comput. Surv.* 11.2 (1979), pp. 121–137.
- [48] Andrew Crotty et al. “An Architecture for Compiling UDF-centric Workflows”. In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1466–1477.
- [49] Andrew Crotty et al. “Tupeware: ”Big” Data, Big Analytics, Small Clusters”. In: *CIDR*. www.cidrdb.org, 2015.
- [50] Benoit Dageville et al. “The Snowflake Elastic Data Warehouse”. In: *SIGMOD Conference*. ACM, 2016, pp. 215–226.
- [51] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. “Monkey: Optimal Navigable Key-Value Store”. In: *SIGMOD Conference*. ACM, 2017, pp. 79–94.
- [52] Cristian Diaconu et al. “Hekaton: SQL server’s memory-optimized OLTP engine”. In: *SIGMOD Conference*. ACM, 2013, pp. 1243–1254.
- [53] Yanlei Diao, Daniela Florescu, Donald Kossmann, Michael J. Carey, and Michael J. Franklin. “Implementing Memoization in a Streaming XQuery Processor”. In: *XSym*. Vol. 3186. Lecture Notes in Computer Science. Springer, 2004, pp. 35–50.

- [54] Martin Dietzfelbinger and Stefan Walzer. “Dense Peelable Random Uniform Hypergraphs”. In: *CoRR* abs/1907.04749 (2019).
- [55] Jens Dittrich and Joris Nix. “The Case for Deep Query Optimisation”. In: *CIDR*. www.cidrdb.org, 2020.
- [56] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. “Quantifying TPC-H Choke Points and Their Optimizations”. In: *Proc. VLDB Endow.* 13.8 (2020), pp. 1206–1220.
- [57] Dominik Durner, Viktor Leis, and Thomas Neumann. “JSON Tiles: Fast Analytics on Semi-Structured Data”. In: *SIGMOD Conference*. ACM, 2021, pp. 445–458.
- [58] Dominik Durner and Thomas Neumann. “No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System”. In: *ICDE*. IEEE, 2019, pp. 734–745.
- [59] Eric Love. “Ressort: An Auto-Tuning Framework for Parallel Shuffle Kernels”. MA thesis. Berkeley, California, USA: University of California, Berkeley, 2016.
- [60] Facebook. *RocksDB*. 2012. URL: <https://github.com/facebook/rocksdb>.
- [61] Faezeh Faghieh, Zsolt István, and Florin Dinu. “A Short Study of Recent Smart Storage Solutions for OLAP: Lessons and Opportunities”. In: *ADMS@VLDB*. 2022, pp. 58–66.
- [62] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. “Cuckoo Filter: Practically Better Than Bloom”. In: *CoNEXT*. ACM, 2014, pp. 75–88.
- [63] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. “Summary cache: a scalable wide-area web cache sharing protocol”. In: *IEEE/ACM Trans. Netw.* 8.3 (2000), pp. 281–293.
- [64] Jian Fang, Jinho Lee, H. Peter Hofstee, and Jan Hidders. “Analyzing In-Memory Hash Join: Granularity Matters”. In: *ADMS@VLDB*. 2017, pp. 18–25.
- [65] Philipp Fent, Guido Moerkotte, and Thomas Neumann. “Asymptotically Better Query Optimization Using Indexed Algebra”. In: *Proc. VLDB Endow.* 16.11 (2023), pp. 3018–3030.
- [66] Philipp Fent and Thomas Neumann. “A Practical Approach to Groupjoin and Nested Aggregates”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2383–2396.
- [67] Domenico Ficara, Stefano Giordano, Gregorio Procissi, and Fabio Vitucci. “MultiLayer Compressed Counting Bloom Filters”. In: *INFOCOM*. IEEE, 2008, pp. 311–315.

- [68] David Floyer. *Arm Yourself: Heterogeneous Compute Ushers in 150x Higher Performance*. Wikibon Research. 2020. URL: <https://wikibon.com/arm-yourself-heterogeneous-compute/> (visited on 01/14/2022).
- [69] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. “Adopting Worst-Case Optimal Joins in Relational Database Systems”. In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1891–1904.
- [70] Michael J. Freitag and Thomas Neumann. “Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates”. In: *CIDR*. www.cidrdb.org, 2019.
- [71] Henning Funke, Jan Mühlig, and Jens Teubner. “Efficient generation of machine code for query compilers”. In: *DaMoN*. ACM, 2020, 6:1–6:7.
- [72] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. “Deployment of Query Plans on Multicores”. In: *Proc. VLDB Endow.* 8.3 (2014), pp. 233–244.
- [73] Jon Gjengset et al. “Noria: dynamic, partially-stateful data-flow for high-performance web applications”. In: *OSDI*. USENIX Association, 2018, pp. 213–231.
- [74] Thomas Mueller Graf and Daniel Lemire. “Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters”. In: *CoRR* abs/1912.08258 (2019).
- [75] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. “Bringing Compiling Databases to RISC Architectures”. In: *Proc. VLDB Endow.* 16.6 (2023), pp. 1222–1234.
- [76] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Cerial J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. 2nd. Springer Publishing Company, Incorporated, 2012. ISBN: 1461446988.
- [77] Tim Gubner and Peter A. Boncz. “Charting the Design Space of Query Execution using VOILA”. In: vol. 14. 6. 2021, pp. 1067–1079.
- [78] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. “The Dynamic Bloom Filters”. In: *IEEE Trans. Knowl. Data Eng.* 22.1 (2010), pp. 120–133.
- [79] Gabriel Haas, Michael Haubenschild, and Viktor Leis. “Exploiting Directly-Attached NVMe Arrays in DBMS”. In: *CIDR*. www.cidrdb.org, 2020.
- [80] Pawan Harish and P. J. Narayanan. “Accelerating Large Graph Algorithms on the GPU Using CUDA”. In: *HiPC*. Vol. 4873. Lecture Notes in Computer Science. Springer, 2007, pp. 197–208.
- [81] Bingsheng He et al. “Relational query coprocessing on graphics processors”. In: *ACM Trans. Database Syst.* 34.4 (2009), 21:1–21:39.

- [82] Thomas Heinis and Gustavo Alonso. “Efficient lineage tracking for scientific workflows”. In: *SIGMOD Conference*. ACM, 2008, pp. 1007–1018.
- [83] John L. Hennessy and David A. Patterson. *A New Golden Age for Computer Architecture*. Turing Lecture. 2018. URL: https://iscaconf.org/isca2018/turing_lecture.html.
- [84] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. “Green-Marl: a DSL for easy and efficient graph analysis”. In: *ASPLOS*. ACM, 2012, pp. 349–362.
- [85] Cunchen Hu et al. “Skadi: Building a Distributed Runtime for Data Systems in Disaggregated Data Centers”. In: *HotOS*. ACM, 2023, pp. 94–102.
- [86] IDC; Seagate; Statista estimates. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025 (in zettabytes)*. Statista. June 2021. URL: <http://www.statista.com/statistics/871513/worldwide-data-created/>.
- [87] Zsolt István, Louis Woods, and Gustavo Alonso. “Histograms as a side effect of data movement for big data”. In: *SIGMOD Conference*. ACM, 2014, pp. 1567–1578.
- [88] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. “Occupy the cloud: distributed computing for the 99%”. In: *SoCC*. ACM, 2017, pp. 445–451.
- [89] Michael Jungmair and Jana Giceva. “Declarative Sub-Operators for Universal Data Processing”. In: *Proc. VLDB Endow*. 16.11 (2023), pp. 3461–3474.
- [90] Michael Jungmair, André Kohn, and Jana Giceva. “Designing an Open Framework for Query Optimization and Compilation”. In: *Proc. VLDB Endow*. 15.11 (2022), pp. 2389–2401.
- [91] Tim Kaldewey, Guy M. Lohman, René Müller, and Peter Benjamin Volk. “GPU join processing revisited”. In: *DaMoN*. ACM, 2012, pp. 55–62.
- [92] Kaan Kara, Jana Giceva, and Gustavo Alonso. “FPGA-based Data Partitioning”. In: *SIGMOD Conference*. ACM, 2017, pp. 433–445.
- [93] Konstantinos Karanasos et al. “Extending Relational Query Processing with ML Inference”. In: *CIDR*. www.cidrdb.org, 2020.
- [94] Alfons Kemper, Donald Kossmann, and Christian Wiesner. “Generalised Hash Teams for Join and Group-by”. In: *VLDB*. Morgan Kaufmann, 1999, pp. 30–41.
- [95] Alfons Kemper and Thomas Neumann. “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots”. In: *ICDE*. IEEE Computer Society, 2011, pp. 195–206.

- [96] Timo Kersten, Viktor Leis, and Thomas Neumann. “Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra”. In: *VLDB J.* 30.5 (2021), pp. 883–905.
- [97] Timo Kersten and Thomas Neumann. “On another level: how to debug compiling query engines”. In: *DBTest@SIGMOD*. ACM, 2020, 2:1–2:6.
- [98] Omar Khattab, Mohammad Hammoud, and Omar Shekfeh. “PolyHJ: A Polymorphic Main-Memory Hash Join Paradigm for Multi-Core Machines”. In: *CIKM*. ACM, 2018, pp. 1323–1332.
- [99] Changkyu Kim et al. “Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs”. In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1378–1389.
- [100] Adam Kirsch and Michael Mitzenmacher. “Less hashing, same performance: Building a better Bloom filter”. In: *Random Struct. Algorithms* 33.2 (2008), pp. 187–218.
- [101] Yannis Klonatos, Christoph Koch, Tiark Rumpf, and Hassan Chafi. “Building Efficient Query Engines in a High-Level Language”. In: *Proc. VLDB Endow.* 7.10 (2014), pp. 853–864.
- [102] André Kohn, Viktor Leis, and Thomas Neumann. “Adaptive Execution of Compiled Queries”. In: *ICDE*. IEEE Computer Society, 2018, pp. 197–208.
- [103] André Kohn, Viktor Leis, and Thomas Neumann. “Building Advanced SQL Analytics From Low-Level Plan Operators”. In: *SIGMOD Conference*. ACM, 2021, pp. 1001–1013.
- [104] André Kohn, Viktor Leis, and Thomas Neumann. “Making Compiling Query Engines Practical”. In: *IEEE Trans. Knowl. Data Eng.* 33.2 (2021), pp. 597–612.
- [105] Marcel Kornacker et al. “Impala: A Modern, Open-Source SQL Engine for Hadoop”. In: *CIDR*. www.cidrdb.org, 2015.
- [106] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, and Gustavo Alonso. *Modularis: Modular Data Analytics for Hardware, Software, and Platform Heterogeneity*. 2020.
- [107] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. “Generating code for holistic query evaluation”. In: *ICDE*. IEEE Computer Society, 2010, pp. 613–624.
- [108] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. “Massively Parallel NUMA-Aware Hash Joins”. In: *IMDM@VLDB (Revised Selected Papers)*. Vol. 8921. Lecture Notes in Computer Science. Springer, 2013, pp. 3–14.

- [109] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. “Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput”. In: *Proc. VLDB Endow.* 12.5 (2019), pp. 502–515.
- [110] Harald Lang et al. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation”. In: *SIGMOD Conference*. ACM, 2016, pp. 311–326.
- [111] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *CGO*. IEEE Computer Society, 2004, pp. 75–88.
- [112] Chris Lattner et al. “MLIR: A Compiler Infrastructure for the End of Moore’s Law”. In: *CoRR* abs/2002.11054 (2020).
- [113] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. “Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age”. In: *SIGMOD Conference*. ACM, 2014, pp. 743–754.
- [114] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. “LeanStore: In-Memory Data Management beyond Main Memory”. In: *ICDE*. IEEE Computer Society, 2018, pp. 185–196.
- [115] Viktor Leis and Maximilian Kuschewski. “Towards Cost-Optimal Query Processing in the Cloud”. In: *Proc. VLDB Endow.* 14.9 (2021), pp. 1606–1612.
- [116] Viktor Leis et al. “How Good Are Query Optimizers, Really?”. In: *Proc. VLDB Endow.* 9.3 (2015), pp. 204–215.
- [117] Henry M. Levy. *The IBM System/38*. Capability-Based Computer Systems, 1984. ISBN: 0932376223.
- [118] Mark Liu. *ARM-based Server Penetration Rate to Reach 22% by 2025 with Cloud Data Centers Leading the Way, Says TrendForce*. URL: <https://www.trendforce.com/presscenter/news/19700101-11178.html> (visited on 03/29/2022).
- [119] Jianyuan Lu et al. “Ultra-Fast Bloom Filters using SIMD techniques”. In: *IWQoS*. IEEE, 2017, pp. 1–6.
- [120] Chen Luo and Michael J. Carey. “LSM-based storage techniques: a survey”. In: *VLDB J.* 29.1 (2020), pp. 393–418.
- [121] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. “Optimizing Bloom Filter: Challenges, Solutions, and Comparisons”. In: *IEEE Commun. Surv. Tutorials* 21.2 (2019), pp. 1912–1949.
- [122] Divya Mahajan et al. “TABLA: A unified template-based framework for accelerating statistical machine learning”. In: *HPCA*. IEEE Computer Society, 2016, pp. 14–26.

- [123] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. “Many-query join: efficient shared execution of relational joins on modern hardware”. In: *VLDB J.* 27.5 (2018), pp. 669–692.
- [124] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. “Generic Database Cost Models for Hierarchical Memory Systems”. In: *VLDB*. Morgan Kaufmann, 2002, pp. 191–202.
- [125] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. “Optimizing Main-Memory Join on Modern Hardware”. In: *IEEE Trans. Knowl. Data Eng.* 14.4 (2002), pp. 709–730.
- [126] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. “What Happens During a Join? Dissecting CPU and Memory Optimization Effects”. In: *VLDB*. Morgan Kaufmann, 2000, pp. 339–350.
- [127] Stefan Manegold, Peter A. Boncz, and Niels Nes. “Cache-Conscious Radix-Decluster Projections”. In: *VLDB*. Morgan Kaufmann, 2004, pp. 684–695.
- [128] Renato Marroquín, Ingo Müller, Darko Makreshanski, and Gustavo Alonso. “Pay One, Get Hundreds for Free: Reducing Cloud Costs through Shared Query Execution”. In: *SoCC*. ACM, 2018, pp. 439–450.
- [129] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: <https://www.tensorflow.org/>.
- [130] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. “Differential Dataflow”. In: *CIDR*. www.cidrdb.org, 2013.
- [131] Erik Meijer, Brian Beckman, and Gavin M. Bierman. “LINQ: reconciling object, relations and XML in the .NET framework”. In: *SIGMOD Conference*. ACM, 2006, p. 706.
- [132] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. “Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last”. In: *Proc. VLDB Endow.* 11.1 (2017), pp. 1–13.
- [133] Michael Mitzenmacher. “Compressed bloom filters”. In: *PODC*. ACM, 2001, pp. 144–150.
- [134] Guido Moerkotte and Thomas Neumann. “Accelerating Queries with Group-By and Join by Groupjoin”. In: *Proc. VLDB Endow.* 4.11 (2011), pp. 843–851.
- [135] Ingo Müller, Renato Marroquín, and Gustavo Alonso. “Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure”. In: *SIGMOD Conference*. ACM, 2020, pp. 115–130.

- [136] Thomas Neumann. *Comparing Join Implementations*. <http://databasearchitects.blogspot.com/2016/04/comparing-join-implementations.html>. Apr. 2016.
- [137] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550.
- [138] Thomas Neumann. “Engineering High-Performance Database Engines”. In: *Proc. VLDB Endow.* 7.13 (2014), pp. 1734–1741.
- [139] Thomas Neumann. “Evolution of a Compiling Query Engine”. In: *Proc. VLDB Endow.* 14.12 (2021), pp. 3207–3210.
- [140] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *CIDR*. www.cidrdb.org, 2020.
- [141] Thomas Neumann and Alfons Kemper. “Unnesting Arbitrary Queries”. In: *BTW*. Vol. P-241. LNI. GI, 2015, pp. 383–402.
- [142] Thomas Neumann and Viktor Leis. “Compiling Database Queries into Machine Code”. In: *IEEE Data Eng. Bull.* 37.1 (2014), pp. 3–11.
- [143] Thomas Neumann, Viktor Leis, and Alfons Kemper. “The Complete Story of Joins (in HyPer)”. In: *BTW*. Vol. P-265. LNI. GI, 2017, pp. 31–50.
- [144] Thomas Neumann and Bernhard Radke. “Adaptive Optimization of Very Large Join Queries”. In: *SIGMOD Conference*. ACM, 2018, pp. 677–692.
- [145] NGD Systems. *Newport Platform*. <https://www.ngdsystems.com/>. 2021.
- [146] Nicole Hemsoth. *An Early Look at Baidu’s Custom AI and Analytics Processor*. <https://www.nextplatform.com/2017/08/22/first-look-baidus-custom-ai-analytics-processor/>. 2017.
- [147] Eriko Nurvitadhi et al. “GraphGen: An FPGA Framework for Vertex-Centric Graph Computation”. In: *FCCM*. IEEE Computer Society, 2014, pp. 25–28.
- [148] Patrick E. O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O’Neil. “The Log-Structured Merge-Tree (LSM-Tree)”. In: *Acta Informatica* 33.4 (1996), pp. 351–385.
- [149] Jian Ouyang et al. “SDA: Software-Defined Accelerator for general-purpose big data analysis system”. In: *Hot Chips Symposium*. IEEE, 2016, pp. 1–23.
- [150] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo hashing”. In: *J. Algorithms* 51.2 (2004), pp. 122–144.
- [151] Shoumik Palkar et al. “A Common Runtime for High Performance Data Analysis”. In: *CIDR*. www.cidrdb.org, 2017.

- [152] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. “A General-Purpose Counting Filter: Making Every Bit Count”. In: *SIGMOD Conference*. ACM, 2017, pp. 775–787.
- [153] Andrew Pavlo. *Parallel Join Algorithms (Hashing)*. <https://15721.courses.cs.cmu.edu/spring2023/slides/11-hashjoins.pdf>. Feb. 2023.
- [154] Rui Pereira et al. “Ranking programming languages by energy efficiency”. In: *Sci. Comput. Program.* 205 (2021), p. 102609.
- [155] Stephen Phillips. “M7: Next generation SPARC”. In: *Hot Chips Symposium*. IEEE, 2014, pp. 1–27.
- [156] Holger Pirk, Stefan Manegold, and Martin L. Kersten. “Waste not... Efficient co-processing of relational data”. In: *ICDE*. IEEE Computer Society, 2014, pp. 508–519.
- [157] Holger Pirk, Oscar R. Moll, Matei Zaharia, and Sam Madden. “Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware”. In: *Proc. VLDB Endow.* 9.14 (2016), pp. 1707–1718.
- [158] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. “Joins on high-bandwidth memory: a new level in the memory hierarchy”. In: *VLDB J.* 29.2-3 (2020), pp. 797–817.
- [159] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. “Rethinking SIMD Vectorization for In-Memory Databases”. In: *SIGMOD Conference*. ACM, 2015, pp. 1493–1508.
- [160] Orestis Polychroniou and Kenneth A. Ross. “A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort”. In: *SIGMOD Conference*. ACM, 2014, pp. 755–766.
- [161] Orestis Polychroniou and Kenneth A. Ross. “Vectorized Bloom filters for advanced SIMD processors”. In: *DaMoN*. ACM, 2014, 6:1–6:6.
- [162] Raghu Prabhakar et al. “Generating Configurable Hardware from Parallel Patterns”. In: *ASPLOS*. ACM, 2016, pp. 651–665.
- [163] Raghu Prabhakar et al. “Plasticine: A Reconfigurable Architecture For Parallel Paterns”. In: *ISCA*. ACM, 2017, pp. 389–402.
- [164] Andrew Putnam et al. “A reconfigurable fabric for accelerating large-scale datacenter services”. In: *ISCA*. IEEE Computer Society, 2014, pp. 13–24.
- [165] Felix Putze, Peter Sanders, and Johannes Singler. “Cache-, hash-, and space-efficient bloom filters”. In: *ACM J. Exp. Algorithmics* 14 (2009).
- [166] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. “Micro adaptivity in Vectorwise”. In: *SIGMOD Conference*. ACM, 2013, pp. 1231–1242.

- [167] Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *PLDI*. ACM, 2013, pp. 519–530.
- [168] Maximilian Reif and Thomas Neumann. “A Scalable and Generic Approach to Range Joins”. In: *Proc. VLDB Endow.* 15.11 (2022), pp. 3018–3030.
- [169] David Reinsel, John Gantz, and John Rydning. *The Digitization of the World – From Edge to Core*. IDC White paper. Nov. 2018. URL: <https://resources.moredirect.com/white-papers/idc-report-the-digitization-of-the-world-from-edge-to-core>.
- [170] Alexander van Renen and Viktor Leis. “Cloud Analytics Benchmark”. In: *Proc. VLDB Endow.* 16.6 (2023), pp. 1413–1425.
- [171] Alice Rey, Michael Freitag, and Thomas Neumann. “Seamless Integration of Parquet Files into Data Processing”. In: *BTW*. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 235–258.
- [172] Stefan Richter, Victor Alvarez, and Jens Dittrich. “A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing”. In: *Proc. VLDB Endow.* 9.3 (2015), pp. 96–107.
- [173] Adrian Riedl, Philipp Fent, Maximilian Bandle, and Thomas Neumann. “Exploiting Code Generation for Efficient LIKE Pattern Matching”. In: *ADMS@VLDB*. 2023.
- [174] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. “High-Speed Query Processing over High-Speed Networks”. In: *Proc. VLDB Endow.* 9.4 (2015), pp. 228–239.
- [175] Tiark Rompf and Martin Odersky. “Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs”. In: *GPCE*. ACM, 2010, pp. 127–136.
- [176] Christian Esteve Rothenberg, Carlos Alberto Braz Macapuna, Maurício F. Magalhães, Fábio Luciano Verdi, and Alexander Wiesmaier. “In-packet Bloom filters: Design and networking applications”. In: *Comput. Networks* 55.6 (2011), pp. 1364–1378.
- [177] Christian Esteve Rothenberg, Carlos Alberto Braz Macapuna, Fábio Luciano Verdi, and Maurício F. Magalhães. “The deletable Bloom filter: a new member of the Bloom family”. In: *IEEE Commun. Lett.* 14.6 (2010), pp. 557–559.
- [178] Nadathur Satish et al. “Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort”. In: *SIGMOD Conference*. ACM, 2010, pp. 351–362.

- [179] Tobias Schmidt, Maximilian Bandle, and Jana Giceva. “A four-dimensional Analysis of Partitioned Approximate Filters”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2355–2368.
- [180] Tobias Schmidt, Philipp Fent, and Thomas Neumann. “Efficiently Compiling Dynamic Code for Adaptive Query Processing”. In: *ADMS@VLDB.* 2022, pp. 11–22.
- [181] Stefan Schuh, Xiao Chen, and Jens Dittrich. “An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory”. In: *SIGMOD Conference.* ACM, 2016, pp. 1961–1976.
- [182] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. “On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning”. In: *Proc. VLDB Endow.* 8.9 (2015), pp. 934–937.
- [183] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. “LLHD: a multi-level intermediate representation for hardware description languages”. In: *PLDI.* ACM, 2020, pp. 258–271.
- [184] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. “ArrayQL for Linear Algebra within Umbra”. In: *SSDBM.* ACM, 2021, pp. 193–196.
- [185] Vivek Seshadri et al. “Gather-scatter DRAM: in-DRAM address translation to improve the spatial locality of non-unit strided accesses”. In: *MICRO.* ACM, 2015, pp. 267–280.
- [186] Amir Shaikhha et al. “How to Architect a Query Compiler”. In: *SIGMOD Conference.* ACM, 2016, pp. 1907–1922.
- [187] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. “Cache Conscious Algorithms for Relational Query Processing”. In: *VLDB.* Morgan Kaufmann, 1994, pp. 510–521.
- [188] Alexander Shraer et al. “CloudKit: Structured Storage for Mobile Applications”. In: *Proc. VLDB Endow.* 11.5 (2018), pp. 540–552.
- [189] Lakshmikant Shrinivas et al. “Materialization strategies in the Vertica analytic database: Lessons learned”. In: *ICDE.* IEEE Computer Society, 2013, pp. 1196–1207.
- [190] Moritz Sichert and Thomas Neumann. “User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases”. In: *Proc. VLDB Endow.* 15.5 (2022), pp. 1119–1131.
- [191] Malcolm Singh and Ben Leonhardi. “Introduction to the IBM Netezza warehouse appliance”. In: *CASCON.* IBM / ACM, 2011, pp. 385–386.

- [192] Softbank Group. *Annual Report – ARM Business Strategy*. Statista. Aug. 2020. URL: https://group.softbank/system/files/pdf/ir/financials/annual_reports/annual-report_fy2020_01_en.pdf.
- [193] SQLite Consortium. *SQLite: Most Widely Deployed and Used Database Engine*. <https://www.sqlite.org/mostdeployed.html>. Jan. 2022.
- [194] Andreas Stiller. “ARMs langer Marsch in die Serverwelt. Neuer Anlauf”. In: *iX* 1 (2022), pp. 60–65.
- [195] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Comput. Sci. Eng.* 12.3 (2010), pp. 66–73.
- [196] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. “How to Architect a Query Compiler, Revisited”. In: *SIGMOD Conference*. ACM, 2018, pp. 307–322.
- [197] Jens Teubner and René Müller. “How soccer players would do stream joins”. In: *SIGMOD Conference*. ACM, 2011, pp. 625–636.
- [198] Transaction Processing Performance Council (TPC). *TPC BENCHMARKTM DS (Decision Support) – Standard Specification Revision 3.2.0*. 2015–2022.
- [199] Transaction Processing Performance Council (TPC). *TPC BENCHMARKTM H (Decision Support) – Standard Specification Revision 3.0.1*. 1993–2022.
- [200] Immanuel Trummer et al. “SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning”. In: *SIGMOD Conference*. ACM, 2019, pp. 1153–1170.
- [201] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. “Predictable Performance for Unpredictable Workloads”. In: *Proc. VLDB Endow.* 2.1 (2009), pp. 706–717.
- [202] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. “Gorgon: Accelerating Machine Learning from Relational Data”. In: *ISCA*. IEEE, 2020, pp. 309–321.
- [203] Berthold Vöcking. “How asymmetry helps load balancing”. In: *J. ACM* 50.4 (2003), pp. 568–589.
- [204] Adrian Vogelsgesang et al. “Get Real: How Benchmarks Fail to Represent the Real World”. In: *DBTest@SIGMOD*. ACM, 2018, 1:1–1:6.
- [205] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. “Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters”. In: *Proc. VLDB Endow.* 13.2 (2019), pp. 197–210.
- [206] Ze-ke Wang, Bingsheng He, and Wei Zhang. “A study of data partitioning on OpenCL-based FPGAs”. In: *FPL*. IEEE, 2015, pp. 1–8.

- [207] Jan Wassenberg and Peter Sanders. “Engineering a Multi-core Radix Sort”. In: *Euro-Par (2)*. Vol. 6853. Lecture Notes in Computer Science. Springer, 2011, pp. 160–169.
- [208] Tom White. *Hadoop: The Definitive Guide*. O’Reilly Media, Inc., 2009.
- [209] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. “On-Demand State Separation for Cloud Data Warehousing”. In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2966–2979.
- [210] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. “Meet Me Halfway: Split Maintenance of Continuous Views”. In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2620–2633.
- [211] Louis Woods, Zsolt István, and Gustavo Alonso. “Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading”. In: *Proc. VLDB Endow.* 7.11 (2014), pp. 963–974.
- [212] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. “Navigating big data with high-throughput, energy-efficient data partitioning”. In: *ISCA*. ACM, 2013, pp. 249–260.
- [213] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. “Q100: the architecture and design of a database processing unit”. In: *ASPLOS*. ACM, 2014, pp. 255–268.
- [214] Steffen Zeuch et al. “Analyzing Efficient Stream Processing on Modern Hardware”. In: *Proc. VLDB Endow.* 12.5 (2019), pp. 516–530.
- [215] Steffen Zeuch et al. “The NebulaStream Platform for Data and Application Management in the Internet of Things”. In: *CIDR*. www.cidrdb.org, 2020.
- [216] Ce Zhang and Christopher Ré. “DimmWitted: A Study of Main-Memory Statistical Analytics”. In: *Proc. VLDB Endow.* 7.12 (2014), pp. 1283–1294.
- [217] Jeff Zhang. *Alibaba Cloud Unveils New Server Chips to Optimize Cloud Computing Services*. Oct. 2021. URL: <https://www.alibabacloud.com/blog/598159>.
- [218] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. “Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities”. In: *CIDR*. www.cidrdb.org, 2019.
- [219] Marcin Zukowski, Sándor Héman, and Peter A. Boncz. “Architecture-conscious hashing”. In: *DaMoN*. ACM, 2006, p. 6.
- [220] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. “Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS”. In: *VLDB*. ACM, 2007, pp. 723–734.