Technische Universität München

TUM School of Computation, Information and Technology

Adaptive Optimizations for Databases

Christoph Maximilian Anneser

Technische Universität München

TUM School of Computation, Information and Technology

Adaptive Optimizations for Databases

Christoph Maximilian Anneser

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr. Thomas Neumann

Prüfende der Dissertation:
1. Prof. Alfons Kemper, Ph.D.
2. Prof. Dr. Jana Giceva Makreshanska
3. Prof. Dr. Maximilian E. Schüle

Die Dissertation wurde am 18.12.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 25.04.2024 angenommen.

Abstract

The increasing demand for big data analytics highlights the importance of database systems. The concurrent shift towards resource disaggregation, multi-tenancy cloud databases, and hardware diversification and specialization requires re-architecting databases for adaptive optimizations. This dissertation contributes three novel adaptive optimization approaches concerning index structures, query optimization, and leveraging fully disaggregated systems.

Index structures substantially contribute to the memory footprint of databases. First, we propose Adaptive Hybrid Indexes that compress rarely accessed nodes while optimizing frequently accessed nodes for performance. Adaptive Hybrid Indexes significantly reduce the memory footprint while incurring almost no performance overhead under skewed workloads. Second, we address the challenges in rule-based query optimization, where statically applied rewrite rules can occasionally degrade performance. We introduce a new framework that uses a machine-learned cost model to adaptively find and turn off these rules per query. Experiments show that our framework significantly reduces dashboard applications' tail latencies for a petabyte-scale PrestoDB deployment at Meta. Third, resource disaggregation and hardware specialization make it challenging for database developers to leverage modern hardware. We propose a blueprint for a new programming model for fully disaggregated systems that raises the memory abstraction level and defers the task and memory mapping to hardware devices to execution time. A runtime system then adaptively co-optimizes data and compute placement.

Zusammenfassung

Die steigende Nachfrage nach Analysen großer Datenmengen unterstreicht die Bedeutung von Datenbanksystemen. Die gleichzeitigen Veränderungen hin zur Trennung von Hardwareressourcen, mehrbenutzerfähigen Cloud-Datenbanken und die Diversifizierung und Spezialisierung der Hardware erfordert eine Neuarchitektur von Datenbanksystemen für adaptive Optimierungen. Diese Dissertation stellt drei neuartige adaptive Optimierungsansätze in Bezug auf Indexstrukturen, Anfrageoptimierung und die Ausnutzung von Systemen mit vollständig getrennten Hardwareressourcen vor.

Indexstrukturen tragen wesentlich zum Speicherbedarf von Datenbanken bei. Als Erstes schlagen wir daher adaptive hybride Indexstrukturen vor, die selten zugegriffene Knoten komprimieren und gleichzeitig häufig zugegriffene Knoten für schnellere Zugriffe optimieren. Adaptive hybride Indexstrukturen reduzieren den Speicherbedarf deutlich und büßen bei ungleich verteilter Arbeitslast kaum Performanz ein. Zweitens gehen wir auf die Herausforderungen bei der regelbasierten Anfrageoptimierung ein, bei der statisch angewandte Regeln die Leistung des Anfrageplans gelegentlich auch verschlechtern können. Wir führen ein neues Framework ein, das ein maschinell erlerntes Kostenmodell verwendet, um diese Regeln adaptiv pro Abfrage zu finden und auszuschalten. Unsere Experimente zeigen, dass das Framework die Ladezeiten von Metas Dashboard-Anwendungen, die PrestoDB verwenden, um Daten im Petabyte Bereich zu analysieren, erheblich verringern kann. Drittens machen es die Trennung der Hardwareressourcen und die Spezialisierung der Hardware für Datenbankentwickler schwierig, das Potenzial moderner Hardware vollständig auszunutzen. Wir schlagen einen Entwurf für ein neues Programmiermodell für Systeme mit vollständig getrennten Hardwareressourcen vor, der die Speicherabstraktionsebene anhebt und die Zuordnung von Tasks und Speicher zu Hardware erst zur Ausführungszeit festlegt. Ein Laufzeitsystem optimiert dann die gemeinsame Platzierung von Daten und Berechnungen.

Acknowledgments

First, I would like to thank the members of my thesis committee, Prof. Alfons Kemper, Prof. Jana Giceva, Prof. Maximilian E. Schüle, and Prof. Thomas Neumann.

I am deeply thankful to my advisor, Prof. Alfons Kemper, for his invaluable guidance. He provided me with the opportunity to pursue a Ph.D. in his group and encouraged me to explore my research ideas, for which I am very grateful.

I want to thank Prof. Thomas Neumann for his excellent courses on database implementation and lab work on compiling query engines.

I extend my gratitude to Prof. Jana Giceva for the inspiring discussions and innovative ideas. I very much appreciate her for helping me become a better researcher.

I thank Prof. Andreas Kipf, my mentor and first point of contact, for his guidance and for our collaborations on several publications.

During my Ph.D., I had the privilege of interning at Intel. I want to thank my hosts, Dave Cohen and Nesime Tatbul, for their exceptional support and for making the internship possible.

Special thanks go to my co-authors Prof. Huanchen Zhang, Prof. Ryan Marcus, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, Zhenggang Xu, Prithviraj Pandian, and Nikolay Laptev.

Thank you also to all my colleagues at the Technical University of Munich for the exciting research discussions and all the other fun activities.

Finally, I want to thank my parents, siblings, and friends for all their support and love during this journey. This thesis would not have been possible without you.

Preface

This doctoral thesis is based on the following peer-reviewed core publications, which are included in the appendix of this document. Throughout this thesis, excerpts and findings from these publications are integrated and presented without additional labeling.

P1 Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. "Adaptive Hybrid Indexes". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* ACM, 2022, pp. 1626–1639

P2 Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. "AutoSteer: Learned Query Optimization for Any SQL Database". In: *PVLDB* 16.12 (2023), pp. 3515–3527

P3 Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. "Programming Fully Disaggregated Systems". In: *HotOS*. ACM, 2023, pp. 188–195

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

Artem Kroviakov, Petr Kurapov, Christoph Anneser, and Jana Giceva. "Heterogeneous Intra-Pipeline Device-Parallel Aggregations". In: *DaMoN*. ACM, 2024, pp. 1–10

Leon Windheuser, Christoph Anneser, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. "Adaptive Compression for Databases". In: *EDBT*. OpenProceedings.org, 2024, pp. 143–149

Christoph Anneser, Mario Petruccelli, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, Ryan Marcus, and Alfons Kemper. "QO-Insight: Inspecting Steered Query Optimizers". In: *PVLDB* 16.12 (2023), pp. 3922–3925

Christian Winter, Andreas Kipf, Christoph Anneser, Eleni Tzirita Zacharatou, Thomas Neumann, and Alfons Kemper. "GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons". In: *EDBT*. OpenProceedings.org, 2021, pp. 169–180

Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, and Alfons Kemper. "The Case for Hybrid Succinct Data Structures". In: *EDBT*. 2020, pp. 391–394

Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins". In: *EDBT*. OpenProceedings.org, 2020, pp. 347–358

Although I am the first author and main contributor to the core publications of this thesis, I use the first person plural to recognize the contributions of my co-authors.

Contents

CHAPTER 1
Introduction

1.1 Increasing Demand for Data Analytics

Data explosion. The world's digitization is accelerating at an unprecedented pace. In 2015, a whitepaper projected that the global datasphere would reach 33 Zettabytes in 2018 and will grow to over 175 by 2025 [145]. Recognizing the immense (economic) potential of big data, Clive Humby declared as early as 2006 that '*data is the new oil*' [79]. However, to uncover the potential of big data, data mining aims at identifying relationships and correlations within the datasets [87, Chapter 17]. These help to better understand the data, evaluate decisions, and predict their consequences more reliably.
Democratizing data analytics. With the decline in computing and memory costs since the early 2000s, data mining of large datasets – often referred to as data analytics – as well as plenty of commercial and open-source software solutions have been made increasingly accessible and affordable to a broader range of users [144, Chapter 1]. This has facilitated more sophisticated analyses of large datasets across various domains [94, 169]. In today's digital landscape, numerous industries rely on fast analytics of large datasets. Examples include online marketplaces utilizing shopping basket analysis and customer segmentation for personalized advertising [144] and healthcare analytics systems employing big data to decrease costs and implement personalized treatments [50].
From data to decision. However, the pathway from raw data to informed business decisions is far from straightforward [169]. Before data can inform decision-makers, multiple Extraction-Transformation-Loading (ETL) steps must be performed. Only a few database systems like HyPer directly support complex data analytics tasks being performed *within* the database system [78, 150, 152]. Instead, in most cases, data must be first exported from various sources in different formats. These formats span from unstructured data in text files, semi-structured data in CSV and JSON files to relational data persisted in database systems [144]. Then, data pipelines integrate data cleansing, preprocessing, and validation steps [158] to facilitate later analyses through data analytics tools [116, 122].

Optimizing data analytics. Data analytics applications have not only high data rate requirements but also high computational needs, which results in high energy consumption and imposes challenges for data centers [110]. Therefore, optimizing the performance and resource efficiency of data-intensive applications is an important goal to achieve environmentally sustainable 'green' data analytics [179]. Application optimizations can be primarily differentiated into two categories: *static* and *adaptive optimizations* [97].

Static optimizations are performed *before* a program is executed, typically during the design, implementation, or compilation phase. Since static optimizations are independent of input data and do not have access to runtime information, they do not introduce runtime overhead for decision-making. Static optimizations mostly concern algorithms and data structures, e.g., by reducing their asymptotical runtime and improving data locality. Also, Ahead-Of-Time (AOT) compilers play a crucial role in this optimization stage, employing various techniques when generating efficient machine code [117].

Adaptive optimizations (also referred to as *dynamic optimizations*) are performed during a program's execution and optimize the program by adjusting to runtime conditions and user inputs [97]. These optimizations can take real-time information into account, including the current system load, the available hardware resources, the workloads, and datasets, among others. Examples of adaptive optimizations include *Just-In-Time (JIT)* compilation [17], where individual parts of the source code can be optimized based on their access frequencies. Another example is adaptive query optimization in databases [54, 71, 75], which includes the reordering of operators during query execution when cardinality estimates turn out to be wrong [104].

Besides their large potential for improving performance and resource efficiency, adaptive optimizations also introduce new challenges. For example, decision-making at runtime incurs computational overhead, which can also outweigh the optimization's improvements. Additionally, adaptive optimizations can make systems more complex, unpredictable, and difficult to test.

1.2 Adaptive Optimizations for Databases

Database systems are the underpinning of many data analytics applications and are responsible for data storage, retrieval, and management, all of which are performance-critical tasks. Therefore, optimizing database systems also enhances the performance of data analytics. While both static and adaptive optimizations are pivotal to improving the performance of database systems, recent trends emphasize the need for adaptive optimizations:

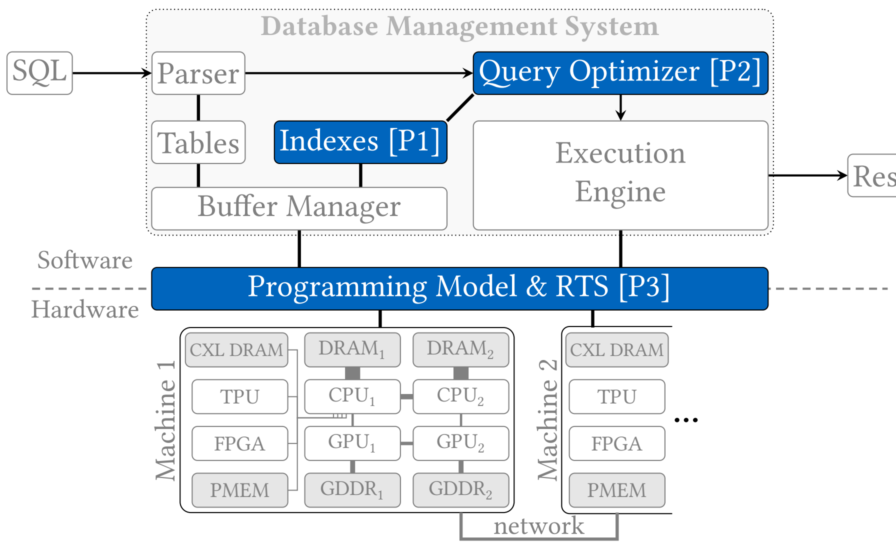


Figure 1.1: The publications of this thesis in the context of a database system.

Multi-tier systems. The diversification of memory and storage devices motivates multi-tier systems, which consist of various memory and storage layers. Since each layer has different capacity, performance, and cost characteristics and dataset sizes increase at an unprecedented pace while most workloads access only few data frequently [35, 180], data placement within these systems requires adaptive optimization accounting for data access types and frequencies, and resource constraints at runtime [172, 190].

Resource disaggregation and multi-tenancy. The shift toward resource disaggregation, where multiple machines share their compute, memory, and storage resources over high-speed networks has facilitated the implementation of cloud-based multi-tenant data warehouses and analytics. Cloud databases process a wide spectrum of queries with CPU times ranging from milliseconds to years and data sizes span from kilobytes to petabytes, as a recent analysis by van Renen and Leis shows [141]. To achieve high performance and efficient resource utilization, databases must dynamically adapt to fluctuating workloads, e.g., switching to distributed processing of very large queries and adding compute nodes during peak workloads.

Contributions. As highlighted by these trends, adaptive optimizations for database systems are becoming increasingly important. This thesis contributes three novel adaptive optimization approaches, which are visualized in the context of a database system in Figure 1.1. The first publication (P1) investigates adaptive optimizations to reduce the storage overhead of index structures. The second publication (P2) addresses the challenges in rule-based query optimization, where statically applied rules occasionally degrade performance. The third publication (P3) envisions a new programming model that facilitates the adaptive co-optimization of dataflow systems' task and data placement in fully disaggregated systems. The following sections briefly introduce the publications and their associated research questions.

1.2.1 Index Structures

Many database systems rely on order-preserving index structures like B- [23], UB- [22], and B+-trees [26] to answer queries with highly selective predicates. Their efficiency and reliability are the results of years of engineering and optimization. Nevertheless, these index structures often utilize a single encoding for all nodes, which cannot account for the specific distribution and access patterns of the data they organize, thereby missing opportunities for adaptive optimizations. For example, while the default node encodings in B+-trees *trade space for performance*, index structures often account for more than half of the database's memory footprint [185], presenting a significant compression potential.

Furthermore, since workloads and key accesses are often skewed [35, 45, 180], so-called *Adaptive Hybrid Indexes* could leverage multiple node encodings with different space-performance trade-offs *within one index structure* to dynamically adapt to workloads: applying more compressed encodings to rarely accessed nodes could reduce the index structure's memory footprint while not harming the index performance. These observations motivate the following research question:

> ***Research Question 1:*** *Can Adaptive Hybrid Indexes optimize the performance-memory trade-off under skewed workloads and outperform conventional index structures when space and performance are considered equally important?*

The first publication in this thesis introduces a novel framework that enables the implementation of Adaptive Hybrid Indexes. Furthermore, it applies the framework to two popular index structures — B+-trees and prefix trees.

Sections concerning publication 1. Section 2.1 delineates the research methodology, Section 3.1 presents the related work, and appendix P1 includes a summary and the full text.

1.2.2 Query Optimization

Query optimization is essential for enhancing query performance in database systems. Based on the abstract syntax tree the parser generated from the SQL query, the query optimizer tries to find an efficient query plan. Besides cost-based optimizers, which generate multiple plans and select the cheapest one, *rule-based query optimizers* apply rewrite rules to transform *one* query plan. These rewrite rules consist of a criterion and a rewrite action, and the optimizer recursively searches for subplans that match the rules' criteria, where it applies the rewrite actions. However, in some cases, rewrite rules can also degrade the performance of a query plan [12, 111, 118, 178, 188]. To address this problem, previous work explored the use of *hint-sets* that *enable or disable* rewrite rules [12, 111, 118, 188]. Applying hint-sets to queries, which is often referred to as *steering*, can result in alternative query plans. These plans are then evaluated using cost or machine-learned models to predict the best plan.

While previous steering approaches can achieve significant performance improvements, they have several limitations. For example, they are tailored to specific database systems and do not efficiently explore the search space of all potential hint-sets. These limitations motivate the following research question:

> ***Research Question 2:*** *Can one framework steer rule-based query optimization across database systems to automatically identify query-aware hint-sets and thereby discover faster query plans?*

The second publication in this thesis introduces a novel framework called AutoSteer, which implements the steering approach *outside* the database system and uses an adaptive greedy search to find query-aware hint-sets. We test AutoSteer with five open-source database systems and use real-world and public benchmarks to evaluate its performance.

Sections concerning publication 2. Section 2.2 explains the research methodology, Section 3.2 presents the related work, and appendix P2 includes a summary and the full text.

1.2.3 Fully Disaggregated Systems

Driven by the end of Moore's law, data centers have shifted to disaggregated architectures with specialized compute and memory devices in recent years [76]. As illustrated in the lower part of Figure 1.1, disaggregated systems connect multiple machines over fast networks and share their memory and compute devices [157]. Furthermore, advancements like cache-coherent interconnects, such as Compute Express Link™ (CXL™) [46], and the latest 4th generation of Intel™ Xeon™ scalable processors [80] enable new data and compute placement options. However, leveraging the full potential of these systems becomes increasingly complex and is the developer's burden, since traditional programming models are not designed for fully disaggregated systems where data movement is the dominating cost factor [93, 135]. Fully leveraging the capabilities of disaggregated systems is essential to achieve cost-effective and sustainable data analytics. However, this requires adaptive optimizations taking the workloads, the available hardware, and their utilization into account. These observations lead to the third research question:

> ***Research Question 3:*** *Can high-performance data-intensive applications like database systems be developed and optimized for fully disaggregated systems sustainably?*

The third publication in this dissertation proposes design principles for a new programming model that facilitates the sustainable development of data-intensive applications for fully disaggregated systems.

Sections concerning publication 3. Section 2.3 outlines the research methodology, Section 3.3 presents the related work, and appendix P3 provides a summary and the full text.

1.3 Outline

This thesis is organized into the following sections:

- **Chapter 2** explains the research methodologies that were used to answer the research questions.
- **Chapter 3** presents the most important related work of each publication and the related work on adaptive query optimization and self-driving database systems, two other important fields where adaptive optimizations have been applied successfully.
- **Chapter 4** concludes the thesis by summarizing the main results and discussing opportunities for future work.
- **Appendices P1 to P3** present the full publications, prefaced by synopses that summarize their contributions.

CHAPTER 2

Research Methodology

This chapter explains the research methodologies used to address the research questions presented in Chapter 1. To provide a comprehensive understanding of the applied research methodologies, the following sections outline them for every publication separately.

2.1 Adaptive Hybrid Indexes

Motivation. As discussed in Chapter 1, database systems rely on order-preserving index structures, such as trees, to accelerate queries with selective predicates. Given that a single table can often have multiple indexes on different columns, these index structures contribute significantly to the database system's overall memory footprint [185]. Consequently, index structures have a large potential to reduce the storage overhead of database systems.

Many conventional index structures, including B- [23], UB- [22], and B+-trees [26] use node encodings that facilitate various operations efficiently, e.g., inserts, updates, deletes, reads, and scans. Therefore, we call them *universal encodings*. These encodings, however, often trade space for query performance, negatively affecting the database system's memory footprint. The same observation also applies to entire classes of index structures that are either optimized for performance *or* space. For example, succinct tree index structures reduce the space consumption to the theoretical limit by *implicitly* encoding node relationships in bitmaps, but also negatively impact query performance [186]. On the contrary, traditional pointer-based data structures use 64-bit *per* indirection on a modern machine. While this allows for differentiation among 2^{64} elements, it comes at the expense of a higher memory footprint.

At the same time, recent studies have shown that real-world workloads often exhibit skewed access patterns: only a few key ranges are frequently accessed, whereas most keys are rarely or not accessed [35].

Therefore, in publication P1, we propose *Adaptive Hybrid Indexes* that use lightweight tracking of the workload's access patterns to identify hot and cold data. We differentiate two types of Adaptive Hybrid Indexes:

Type 1: Multiple node encodings with different space and performance characteristics are utilized *within one index.*

Type 2: *Two or more index structures* are combined into one logical index, e.g., a performance-optimized and a succinct index structure.

At execution time, the tracked accesses inform adaptive decisions like (1) changing the node encodings or (2) migrating nodes between the indexes.

These observations lead to the first research question:

> ***Research Question 1:*** *Can Adaptive Hybrid Indexes optimize the performance-memory trade-off under skewed workloads and outperform conventional index structures when space and performance are considered equally important?*

Before introducing hypotheses that can be empirically validated, the trade-offs between an index structure's space utilization and its query latency must be made measurable. Therefore, we adopt the following cost function introduced by Zhang et al. [187]:

$$C = P^r \cdot S \tag{2.1}$$

In this equation, P denotes the latency of index operations, thereby serving as an indicator of performance. S stands for the space occupied by the index. The parameter r controls the balance between these two aspects. Specifically, for $0 \leq r < 1$, the equation prioritizes minimizing space over enhancing performance. Conversely, when $r > 1$, the focus shifts to maximizing performance at the cost of increased space usage. Space and performance are considered equally important throughout this thesis and we set $r = 1$. To use this cost function in the evaluation, the raw values for space and performance – i.e., latency and memory consumption – are scaled to the range between 0 and 1 using max normalization. This step is necessary since the number of elements inserted into the index and the type of workload being processed can substantially influence both the space and performance metrics, thereby affecting the final cost.

We propose two hypotheses to answer Research Question 1. The first hypothesis addresses the Hybrid B+-tree, which is a Type 1 Adaptive Hybrid Index introduced in publication P1:

> ***Hypothesis 1.1:*** *An adaptive hybrid B+-tree having different leaf node encodings can achieve a lower cost C than a conventional B+-tree with only one encoding under skewed workloads.*

The second hypothesis concerns Adaptive Hybrid Indexes of Type 2:

> ***Hypothesis 1.2:*** *A hybrid index that combines a performance-optimized with a space-optimized prefix tree can balance the trade-offs between memory footprint and query performance, resulting in a lower cost C than either type of prefix tree alone under skewed workloads.*

Outline. Section 2.1.1 presents the scientific methods used to address the research question and test the two hypotheses. Section 2.1.2 provides implementation details and Section 2.1.3 presents the evaluation.

2.1.1 Scientific Method

We develop two novel index structures to empirically validate the two hypotheses: the Hybrid B+-tree (Type 1) and the Hybrid Trie (Type 2). The following sections outline their foundational principles and discuss the mechanisms enabling their workload adaptivity at runtime.

2.1.1.1 Hybrid B+-tree

Traditional B+-tree implementations employ a universal encoding for all leaf nodes, which we refer to as *Gapped*. Figure 2.1 visualizes the conceptual overview of a node in the Gapped encoding: each node has a fixed size[1] with a static number of slots, irrespective of the number of key-value pairs the node contains. This encoding not only enables efficient inserts in the case that the node is not yet full, but it also defers merge operations in case an entry is removed from the node. Thus, the Gapped encoding prioritizes query performance, such as fast inserts and updates, while requiring more memory than necessary to store the existing entries. As pointed out by Alhomssi and Leis [5], this memory inefficiency yields a space utilization below 70% in classical B-trees. To address this issue, we extend the STX B+-tree [26] with two encodings:

The **Packed encoding** applies the same storage principles as the Gapped encoding but does *not* retain empty slots. Therefore, the node size is no longer static but depends on the number of items the node contains. Figure 2.1 illustrates the Packed encoding. While this encoding improves space utilization, insert and delete operations suffer. For example, nodes in the Packed encoding must always be reorganized to make space for new entries or shrink after removing an item. Because of this trade-off, the Packed encoding is particularly well-suited for read-heavy workloads with only a few or no updates.

[1]This thesis focuses on B+-trees storing fixed-size data types, such as numerics or integers.

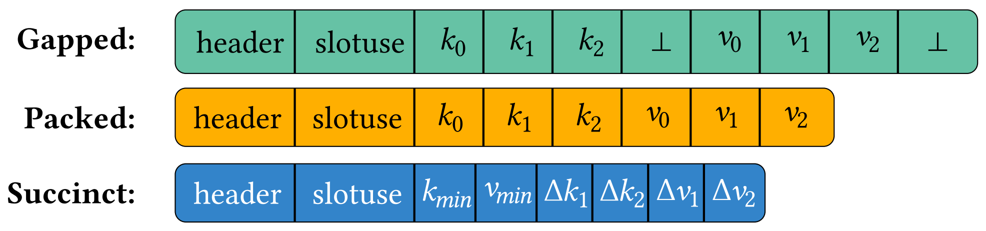


Figure 2.1: Leaf node encodings of the Hybrid B+-tree.

The **Succinct encoding** optimizes memory efficiency further. It extracts the minimum key k_{min} and value v_{min} from all items in the node and stores them separately. For all keys and values, we store only the differences to the minimum key (Δk) or value (Δv). Furthermore, the Succinct encoding leverages *bit packing* to reduce space consumption by only occupying the minimum number of bits required to represent the key or value. However, this representation requires more computational steps like additional shift operations to retrieve the keys and values. Figure 2.1 shows the detailed memory layout of a node stored in the Succinct encoding.

Performance implications. After introducing the different leaf node encodings, we execute multiple micro-benchmarks to evaluate their space utilization and performance. Initially, we assess the lookup efficiency of B+-trees employing one of the three leaf node encodings. We use the Open Street Map Cells dataset [125] and generate uniformly distributed lookups. We conducted the experiment on Testbed 1 (refer to Section 2.1.3 for more details of the experiment setups), and we measured the leaf node accesses' performance, excluding the inner node traversal. The results are shown in Table 1 of P1 and indicate that both the Packed and Succinct encodings significantly reduce the memory footprint compared to the Gapped encoding.

Enabling workload adaptivity. However, to fully validate Hypothesis 1.1, the Hybrid B+-tree must achieve a lower cost C than a conventional B+-tree with a static encoding for all leaf nodes under skewed workloads. To this end, the Hybrid B+-tree must become workload-aware and adaptively select suitable leaf node encodings. Therefore, information is required on both the number of accesses and the access types at runtime. There are two options to collect this information:

- In a *decentralized* tracking approach, each leaf node independently keeps track of its accesses. Access counters for reads and updates are directly stored in the leaf node's header.

- In a *centralized* tracking approach, a dedicated data structure, such as a

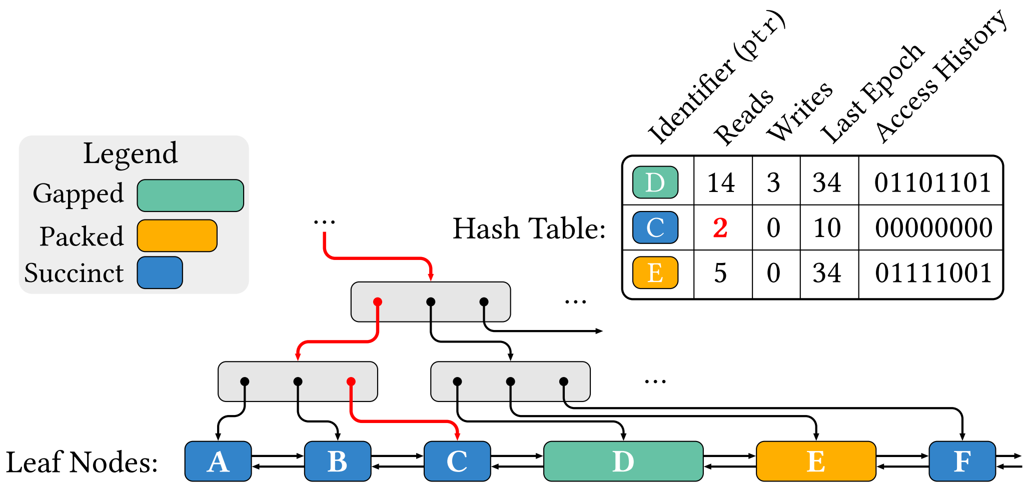


Figure 2.2: Hybrid B+-tree with different leaf node encodings. A hash table tracks the access statistics.

hash table, maintains distinct counters for reads and updates per accessed leaf node.

While the decentralized approach would require intrusive modifications to all leaf nodes – including those that never get accessed – the centralized tracking approach only tracks accessed nodes, thereby saving space under skewed workloads. Due to these advantages, we adopt the centralized approach for the Hybrid B+-tree to collect the access information in a hash table. Figure 2.2 visualizes the centralized tracking approach. The Hybrid B+-tree is shown at the bottom, leveraging the different leaf node encodings introduced above. To the top right, a hash table tracks the access information for the leaf nodes. It uses the leaf node's memory address as the key and maintains separate counters for reads and writes. This example illustrates a read access to node C . The according read counter is incremented in the hash table. The two other hash table entries *Last Epoch* and *Access History* will be explained later.

To confirm Hypothesis 1.1, the overall cost *C* of the Hybrid B+-tree must be lower than the cost of conventional, static B+-trees that use one node encoding for all leaf nodes. However, tracking all leaf node accesses yields substantial performance overhead. To explore the tracking overhead, we conduct an experiment on Testbed 1 using the same dataset as before. Figure 5 of P1 shows the results: Tracking all accesses (*skip length* = 1) incurs a substantial overhead of 60% compared to the conventional B+-tree without tracking.

Sampling node accesses. To address this issue, we adopt a lightweight sampling technique proposed by Vitter et al. [171], which defines a parameter called *skip length*, specifying the number of accesses ignored between two successive tracked accesses. For every index operation, the skip length is decremented by one. When the skip length reaches zero, the access is tracked in the hash table and the skip length is subsequently reset to its default value. As shown in Figure 5 of P1, a *skip length* of 2 reduces the tracking overhead to 40%. By sampling only every 20th access, the overhead drops to 1.6%.

While a higher *skip length* effectively reduces the tracking overhead, it also delays the hybrid index's ability to detect and adapt to changing workload patterns. Therefore, adaptively adjusting the *skip length* at runtime can help improve its performance and adaptability. For instance, if the workload patterns are stable and the node encodings do not change, increasing the *skip length* could further reduce the tracking overhead. Conversely, if the node encodings frequently change, the need for precise tracking information could justify a higher sampling rate and the resulting tracking overhead.

Sample size. In addition to the sampling rate, another key parameter to consider is the sample size, which determines how many accesses must be tracked before deciding on suitable leaf node encodings. While smaller samples can be collected more quickly and require fewer resources, they may result in suboptimal encoding decisions.

Given that computing environments have limited memory resources, we use a memory budget to constrain the problem scope. The memory budget sets an upper limit within which the Hybrid B+-tree can optimize its leaf nodes. The next task is to identify how many compressed nodes (e.g., Succinct or Packed) k can be optimized – such as by using the Gapped encoding – without exceeding the memory budget. Considering the average node sizes from the prior experiment and the number of Succinct, Packed, and Gapped leaf nodes in the Hybrid B+-tree, one can derive an approximation for k.

Once k is determined, a sufficient sample size s must be found. We use an approach proposed by Pietracaprina et al. [134] that calculates the minimum sample size s that is required to identify the top-k nodes at a reliability δ and a classification error rate ϵ. We then conduct experiments to derive reasonable values for ϵ and δ across various workloads, as shown in Figure 2 of P1. Setting $\epsilon = \delta = 0.05$ yields a good trade-off between sample size and accuracy. Based on Equation (1) of P1, the sample size can be dynamically adjusted.

Designing a generic framework for Adaptive Hybrid Indexes. Building on the previous findings, we conceptualize and develop a new framework that seamlessly combines sampling and node access tracking. The resulting framework leverages the implementation of Adaptive Hybrid Indexes. Figure 4 of P1 visualizes the framework on a conceptual level. It comprises two phases:

Sampling Phase: During this phase, node accesses are tracked in the hash table. Once the sample size is reached, the framework transitions to the adaptation phase.

Adaptation Phase: In this phase, the framework uses the aggregated access statistics to find the most frequently accessed nodes and migrates them to performance-optimized encodings, such as Gapped or Packed, while ensuring that the memory bound is not exceeded.

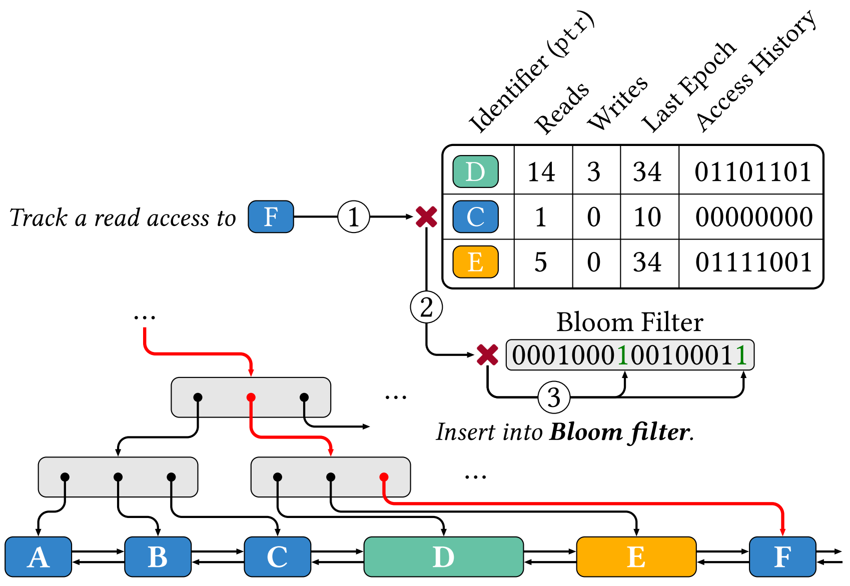
Furthermore, the framework utilizes a *global epoch*, which is incremented in every adaptation phase. Each node in the hash table records the epoch of its last access (cf. Figure 2.2). When a node is accessed and its recorded epoch is less than the global epoch, the read and write counters for that node are reset to zero and its epoch is updated to the global epoch. Additionally, each hash table entry maintains the access history – a bitmap that tracks the last epochs in which the node was accessed. This information helps to mitigate oscillatory effects that could otherwise result in frequent node encoding changes.
Improving performance with bloom filters. In addition to the hash table, we use a bloom filter to enhance the tracking framework's performance. A bloom filter is a hash-based data structure that enables fast, approximate membership tests [27]. This optimization prevents infrequently accessed nodes from being inserted into the hash table, which is computationally more expensive than inserting a node into the bloom filter. Figure 2.3 illustrates the tracking process of an initially untracked node $\boxed{\text{F}}$.

First, the framework queries the hash table to verify the node's existence ①. Since the node has not yet been added to the hash table, the system proceeds to check the bloom filter ②. Since the node is also absent from the bloom filter, it is inserted there ③.

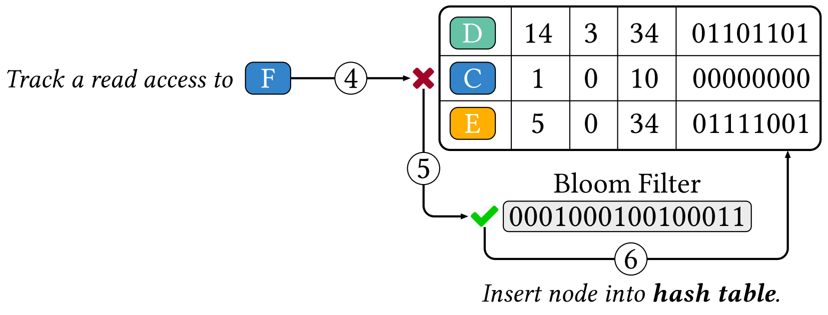
Figure 2.3b illustrates a subsequent read access to the same node $\boxed{\text{F}}$. The framework, once again, queries the hash table ④, which reports that the node has not yet been inserted. However, this time, the bloom filter indicates that the node has been previously registered ⑤, followed by the node's insertion into the hash table ⑥.

Lastly, Figure 2.3c demonstrates a third read access to node $\boxed{\text{F}}$. This time, the hash table already contains an entry for node ⑦ and its read counter is incremented by one ⑧.

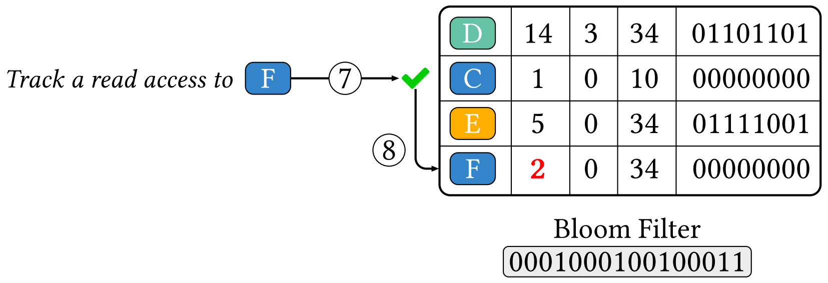
As part of the above-introduced micro-benchmark, we also evaluate the impact of the bloom filter. Figure 5 of P1 shows that the bloom filter significantly reduces the relative tracking overhead compared to the Hybrid B+-tree without the bloom filter.



(a) **Initial State**: Node not yet inserted into hash table or bloom filter.



(b) **Hash Table Insert**: Node exists in the bloom filter and is added to the hash table.



(c) **Hash Table Update**: Node exists in the hash table and its read counter is incremented.

Figure 2.3: Tracking multiple read accesses to a leaf node in the Hybrid B+-tree.

2.1.1.2 Hybrid Trie

After designing the Hybrid B+-tree, we use the adaptation framework to implement the Adaptive Hybrid Trie. Tries, also called prefix trees, organize keys hierarchically, with each tree level storing a part of the key. Hybrid Trie is a Type 2 hybrid index that combines two well-known prefix trees: the Adaptive Radix Tree (ART) [99] and the Fast Succinct Trie (FST) [187].

ART is a high-performance, state-of-the-art, pointer-based index that serves as the default index structure in HyPer [88]. Although ART employs four node encodings to optimize space utilization depending on the number of elements stored in a node, its pointer-based structure consumes more memory than the information-theoretical minimum requires.

On the other end of the spectrum, FST focuses on optimizing memory usage, employing bitmaps instead of pointers to navigate to subsequent nodes, albeit at the expense of higher computational overhead. FST leverages two encodings:

> **Dense encoding**: Nodes *implicitly* store the existing items within a fixed-size bit array. For example, if each level represents 8 bits of a key, then the bit array would have a size of $2^8 = 256$ per node. An index in this bitmap is set to 1 if the corresponding item exists and 0 otherwise.
>
> **Sparse encoding**: Nodes *explicitly* store the existing items, providing a more compact representation if nodes are sparsely populated.

Both encodings require an additional bitmap to indicate whether a specific entry represents the terminal of a key or the item is a prefix of at least one key continuing in the next level. Zhang et al. [187] provide more details on the node traversal in FST. Table 2 of P1 shows that ART has a significantly higher space consumption but achieves better lookup performance than FST-dense and FST-sparse.

These observations motivate a level-wise combination of the two index structures. Because of its better performance, ART represents the upper levels, where all index operations start, while FST stores the lower levels. Figure 10 of P1 illustrates this idea. We leverage the existing adaptation framework to monitor the accesses to the ART nodes. This data enables runtime decisions on whether nodes should be encoded in the performance-optimized ART or stored in the space-efficient FST. Considering skewed workloads, where only a few keys are accessed frequently, combining ART's performance advantages with FST's space efficiency could achieve a lower cost C compared to using either ART or FST exclusively. Should the Adaptive Hybrid Trie outperform ART and FST regarding the overall cost C under skewed workloads, this would be an empirical validation for Hypothesis 1.2.

2.1.2 Implementation Overview

The tracking framework is implemented as a header-only C++ library and compiled using GCC 9.3.0 with optimization flags `-O3` and `-march=native`. All experimental evaluations are conducted on Testbed 1 (cf. Section 2.1.3).

An exhaustive set of test cases asserts the correctness of the implemented index structures.

Performance optimization strategies are incorporated at various levels of the framework. For example, the skip length is realized through thread-local counters to mitigate thread contention and improve parallel processing. On the single-threaded side, the hash table responsible for access tracking employs an optimized hopscotch hash map [33]. For the multi-threaded experiments concerning the Hybrid B+-tree, a concurrent Cuckoo hash map [32, 105] tracks the node accesses. Furthermore, atomic data types, like `std::atomic_uint`, are used to implement efficient read and write counters, obviating the need for mutex-based synchronization. For better multi-threaded performance, Optimistic Lock Coupling (OLC) techniques are employed [101], providing a more efficient alternative to basic lock coupling.

The Hybrid B+-tree implementation is built upon the well-established STX-B+-tree [26]. Our approach is not tightly bound to this specific B+-tree but should also work with other implementations. A more detailed discussion on the implementation can be found in Section 4 of P1.

2.1.3 Evaluation

The following machine is used for all experiments:

- **Testbed 1**: 16-core AMD Ryzen 9 3950X CPU @ 3.5GHz equipped with 64GB DDR4-2667 RAM. The operating system is Ubuntu 21.04.

Evaluating Hypothesis 1.1. Let us start with the first hypothesis:

> "*An adaptive hybrid B+-tree having different leaf node encodings can achieve a lower cost C than a conventional B+-tree with only one encoding under skewed workloads.*"

To validate the hypothesis, we must show that a skewed workload exists for which the Adaptive Hybrid B+-tree achieves a lower cost C than a conventional B+-tree that employs one static leaf node encoding. Figure 13 of P1 investigates the trade-off between space and performance for the Adaptive Hybrid B+-tree and its competitors for two skewed workloads. Here, we focus on workload W1.3, whose read, scan, and insert operations follow a log-normal distribution.

Table 2.1: Results from Figure 13 of P1 (W1.3) and the max-normalized scores for performance P and space S as well as the computed cost $C = P \times S$.

Competitor	**Latency [ns]**	**Size [GB]**	P	S	C
AHI-BTree	291.8	2.36	66.4	27.2	1808.5
Succinct	439.2	2.3	100.0	26.5	2652.8
Packed	260.4	6.15	59.3	70.9	4205.7
Gapped	246.2	8.67	56.1	100.0	5605.6

Table 2.2: Results from Figure 19 of P1 (W6.1) and the max-normalized scores for performance P and space S as well as the computed cost $C = P \times S$.

Competitor	**Latency [ns]**	**Size [MB]**	P	S	C
AHI-Trie	381.0	374.5	36.5	37.0	1350.0
FST	1045.1	322.6	100.0	31.9	3190.0
ART	250.6	1011.3	24.0	100.0	2397.9

All index structures store 8-byte keys from the OSM dataset [125] and 8-byte tuple identifiers as values.

Table 2.1 shows each competitor index's average latency and size, along with the normalized values of performance P and space S. The last column shows the cost according to Equation (2.1). The adaptive Hybrid B+-tree achieves a lower cost $C = 1808.5$ than the conventional B+-trees that either use the Gapped (5605.6), Packed (4205.7), or Succinct (2652.8) encodings for all leaf nodes. This finding validates Hypothesis 1.1.

Evaluating Hypothesis 1.2. The second hypothesis addresses the Hybrid Trie:

> "*A hybrid index that combines a performance-optimized with a space-optimized prefix tree can balance the trade-offs between memory footprint and query performance, resulting in a lower cost C than either type of prefix tree alone under skewed workloads.*"

To validate the hypothesis, we must find a skewed workload where the Adaptive Hybrid Trie (AHI-Trie) yields a lower cost C than the Fast Succinct Trie (FST) [187] and Adaptive Radix Tree (ART) [99]. Figure 19 of P1 investigates the space and performance trade-off for AHI-Trie and its competitors FST and ART. It also shows a pre-trained, static variant of AHI-Trie that is tuned for the most frequently accessed keys (assuming a priori knowledge of the workload). The indexes store 33 million email addresses in host-reversed order and process workload W6.1, which comprises read accesses that follow a Zipfian distribution.

Table 2.2 summarizes the results and shows the calculated scores for P, S and $C = P \times S$. As AHI-Trie achieves a lower cost of 1350.0 compared to FST (3190.0) and ART (2397.9), Hypothesis 1.2 is successfully validated.
Conclusions. The proposed framework facilitates the implementation of the Hybrid B+-tree and the Hybrid Trie, which are two examples of Adaptive Hybrid Indexes. The experiments confirm Hypothesis 1.1 and Hypothesis 1.2 and demonstrate that Adaptive Hybrid Indexes can achieve lower costs C compared to their non-adaptive counterparts under skewed workloads. These findings positively answer Research Question 1.

2.2 Steering Query Optimizers

Motivation. SQL is the de facto standard for retrieving and manipulating data in database systems. It is a declarative language that allows users to specify *what* data should be retrieved, leaving it up to the database to determine *how* to efficiently retrieve it. After accepting an SQL statement, the database system first parses it into an Abstract Syntax Tree (AST). Next, the query optimizer searches for an efficient execution plan [83]. Query optimization is crucial in enhancing a database's performance and has been the subject of extensive research efforts, including cardinality estimation [90, 91, 119, 120], query rewriting [192], and join ordering [113].

Query optimizers primarily follow two paradigms: cost-based or rule-based optimization. *Cost-based query optimizers* use cost functions and statistics to estimate the cost of different query plans and pick the cheapest one. *Rule-based query optimizers* use *rewrite rules* to transform *one* query plan. Each rewrite rule consists of a *criterion* and a *rewrite action.* For every rewrite rule, the optimizer searches the query plan for subtrees that satisfy the criterion and applies the rewrite action to transform it.

Hybrid query optimizers combine cost-based and rule-based optimization strategies and are prevalent in both open-source and commercial databases, including PrestoDB [156], SCOPE [189], SparkSQL/Catalyst [13], Greenplum/Orca [159], and Apache Calcite [24].
Challenges in rule-based query optimization. Rule-based query optimizers implement many rewrite rules. For example, PrestoDB has more than 170 rewrite rules [136]. Some of these rules are tailored for specific scenarios, making assumptions on or having been tested with a certain number of worker nodes, network latencies, and dataset sizes. For instance, PrestoDB's *HashGenerationOptimizer* accelerates query processing on larger datasets, but it can significantly slow down query processing on smaller datasets [10]. As a result,

the rule's effectiveness depends on multiple parameters and can also negatively impact performance [111, 118, 188].

Database knobs. Many database systems have configurable *knobs* to address these issues, allowing their users to selectively activate or deactivate rewrite rules per query, a process known as *steering*. For instance, PostgreSQL users can turn off index scans, which becomes useful when the query optimizer overestimates a predicate's selectivity.

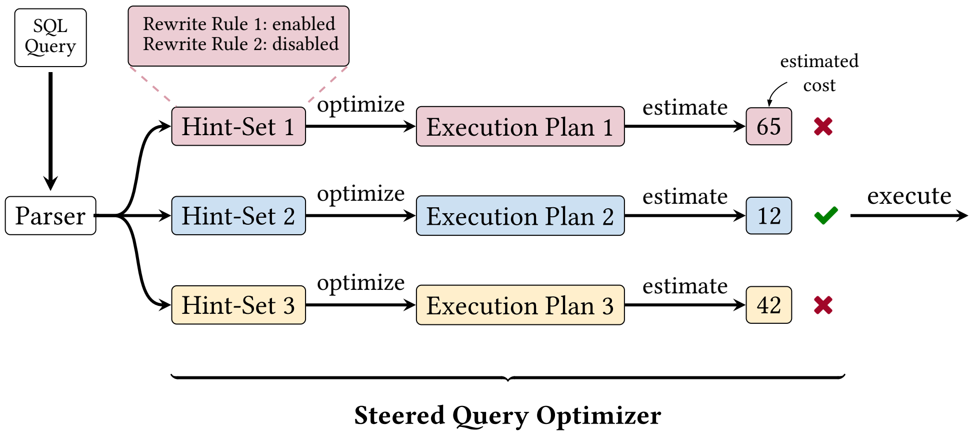


Figure 2.4: A steered query optimizer with three predefined hint-sets.

Prior work, such as the Bandit Optimizer (Bao) [111] and its adoptions for Microsoft Scope [118, 188], Vertica, Azure Synapse, and RedShift [12], have investigated automated *steering* approaches for rule-based query optimizers. Figure 2.4 illustrates Bao's approach: predefined *hint-sets* specify which rules are turned on or off. Then, Bao uses these hint-sets to steer the query optimizer and generates an optimized plan for every hint-set. Finally, a machine-learned model estimates the execution times of these plans, allowing Bao to pick the cheapest one.

Limitations of state-of-the-art steered optimizers. Steered query optimizers have already achieved significant performance improvements, but they have several limitations:

- **Integration complexity:** Previous steering approaches are tightly coupled to specific database systems and are thus not generalizable to other database systems.
- **Required expertise:** The effective use of these strategies requires expert-level knowledge of a database system and its query optimizer to predefine useful hint-sets.

- **Static hint-sets:** Given a large number of configurable knobs in database systems, the limitation of predefining static and query-agnostic hint-sets becomes apparent: it significantly narrows the search space, thereby overlooking many effective hint-sets. Additionally, not all of these static hint-sets will impact a given query, but they are still evaluated by the steered optimizer, leading to inefficiencies.

These limitations motivate the following research question:

> ***Research Question 2:*** *Can one framework steer rule-based query optimization across database systems to automatically identify query-aware hint-sets and thereby discover faster query plans?*

We introduce two hypotheses to address the different aspects of this question.

> ***Hypothesis 2.1:*** *Rule-based query optimizers can be steered exclusively through SQL and explain statements, thereby obviating the need for system-level integrations.*

The second hypothesis addresses the steering framework's performance improvements:

> ***Hypothesis 2.2:*** *The steering framework can identify alternative execution plans that reduce the query execution time for established database benchmarks.*

Outline. Section 2.2.1 outlines the scientific methods and research process employed to address the research question and the first hypothesis. Section 2.2.2 delves into implementation details, while Section 2.2.3 focuses on the quantitative evaluation to test the second hypothesis.

2.2.1 Scientific Method

While publication P2 elaborates on the technical details and results of the steering framework, this section summarizes the research methodology and explains how the research questions and hypotheses are addressed.

2.2.1.1 Conceptual Approach and Idea

In contrast to previous steering approaches like Bao [111] that use static, *query-agnostic* hint-sets, our goal is to create adaptive, query-aware hint-sets. Therefore, we must first identify those *rewrite rules* that impact a query's optimization.

Identifying effective rewrite rules. To reduce the large search space of hint-sets, Negi et al. [118] proposed the concept of *query spans.* A query span includes only the *effective* rewrite rules for which toggling it on or off results in a different query plan. However, interdependencies among rewrite rules complicate calculating the query span. Instead, Section 3.2 of P2 explores two heuristics to *approximate* query spans. While identifying effective rewrite rules is straightforward *within* a rule-based query optimizer (cf. Listing 3 of P2), identifying them *outside* the database system would broaden the approach's applicability across other systems, thereby addressing Hypothesis 2.1.

Approximating query spans using `EXPLAIN`. Although not part of the SQL standard [81], `EXPLAIN` statements are supported across a wide range of relational database management systems – including PostgreSQL [61], MySQL [60], DuckDB [58], SparkSQL [64], PrestoDB [62], Snowflake [63], Hyper [59], Amazon Redshift [57] – to help users understand the query processing. Invoking `EXPLAIN <statement>` typically yields the optimized query plan *without* executing it. Selectively disabling rewrite rules and then comparing the explained query plans allows identifying the effective rewrite rules *outside* the database system. The proposed steering framework in publication P2 implements this idea. However, it is important to note that this approach has limitations, as it cannot find all effective rewrite rules, particularly those affecting the query plan only at an algorithmic level. Such fine-granular changes are often *not* included in the explained query plans.

Adaptive hint-sets. Next, we conduct an experiment to evaluate the query span approximation algorithms. Considering queries e6a, 8d, and 16d from the extended Join Order Benchmark (JOB) [100, 113], Figure 9 of P2 shows that only 13 to 15 rewrite rules effectively contribute to the query plan's optimization. However, while 15 effective rewrite rules per query reduce the search space, there are still 2^{15} potential hint-sets, which are too many to explore. Detailed experiments across various databases in Section 4 of P2 lead to two observations:

(1) Most beneficial hint-sets, i.e., hint-sets resulting in a speedup over the original query plan, are small and consist of fewer than four rewrite rules.

(2) Most beneficial hint-sets consist of smaller hint-sets, which are also beneficial.

Observation (2) implies that in most cases, larger beneficial hint-sets could be constructed from smaller beneficial hint-sets bottom-up. Therefore, we implement a greedy algorithm (cf. Listing 1 of P2) that comprises three phases:

Phase 1: Execute the query using the default plan to establish a baseline.

Phase 2: Based on the query span, use the effective rewrite rules to generate singleton hint-sets and collect their execution metrics.

Phase 3: Iteratively construct larger hint-sets by selectively combining beneficial hint-sets identified in the previous steps. Execute the larger hint-sets only when they yield a new, previously unexplored query plan.

This algorithm generates query-aware hint-sets, thereby addressing Research Question 2: *"[...] automatically identify query-aware hint-sets [...]"*.

Predict query plan execution times. Like Bao's reinforcement learning-based approach, the query plans generated through the adaptive hint-set search can serve as training data for a learned cost model. The greedy hint-set search can then use the cost model to estimate the execution times of the generated query plans, thereby eliminating the need for their actual execution. Due to the inference overhead, this approach is more beneficial for long-running and analytical than short-running queries.

Automatically steering query optimizers with AutoSteer. Building on the previous ideas and experimental findings, we propose a new framework called AutoSteer. This framework automatically steers rule-based query optimizers of SQL database systems that expose binary knobs and support `EXPLAIN` statements for detailed query plan analysis. AutoSteer combines the approximation of query spans, the adaptive hint-set search, and the prediction of query plan execution times based on learned cost models.

Connecting AutoSteer to a database. As illustrated in Figure 2 of P2, only two database-specific files are required to connect AutoSteer to a new database system:

Knobs: This file contains the names of the binary knobs belonging to the query optimizer's rewrite rules that the database system exposes.

Connector: A connector for AutoSteer and the database system, implementing knob toggling, query explanation, and query execution.

We can also integrate AutoSteer directly into a database system's query optimizer for better performance. However, such integrated connectors require more engineering effort. If the framework is used with a custom, integrated connector, we refer to the framework as AutoSteer-**C**. Otherwise, when it uses the generic connector, we refer to it as AutoSteer-**G**. According to Research Question 2, this thesis focuses on AutoSteer-G.

AutoSteer's integration effort. We integrated AutoSteer-G into five open-source database systems and evaluated it with TPC-H [167], TPC-DS [166], and the extended Join Order Benchmark [100, 111]. We use the Lines of Code (LOC) metric to quantify the integration effort, excluding comments, import and log

Table 2.3: Comparison of AutoSteer-G's Database Connectors.

Database System	Lines of Code	Tunable Knobs
PostgreSQL	49	20
PrestoDB	53	177
SparkSQL	68	49
DuckDB	34	14
MySQL	55	22

statements, and blank lines. Table 2.3 summarizes the findings. For AutoSteer-G's integration with PrestoDB, adaptations were made to the database's source code [136] to expose the 177 knobs. Remarkably, each connector is implemented in less than 100 lines of Python code, demonstrating the minimal programming effort needed for AutoSteer-G's integration. Furthermore, the evaluation in Section 4 of P2 shows that AutoSteer-G, which uses only SQL and EXPLAIN statements to steer rule-based query optimization in five database systems, successfully identifies query-aware hint-sets that significantly improve the performance and thereby confirms Hypothesis 2.1.

2.2.2 Implementation Overview

AutoSteer-G. AutoSteer-G and all of its algorithms are implemented in Python 3.10. We use SQLite 3 to persist all generated data, such as the query spans, the hint-sets, the query plans, and the measurements. A comprehensive set of test cases covers AutoSteer's key functionalities.

The inference of query plan execution times is based on the Bao-for-PostgreSQL prototype, which uses Tree Convolutional Neural Networks to predict the query plans' execution times [19]. AutoSteer-G's source code and generic connectors are open-source and available under the MIT license [15].

AutoSteer-C. AutoSteer's system-level integration for PrestoDB is based on version 0.274 and uses Java 8. The query span approximation is implemented directly within PrestoDB's rule-based query optimizer. Listing 3 of P2 illustrates this idea in pseudocode.

2.2.3 Evaluation

Section 2.2.3.1 presents the testbeds that were used in the experimental evaluation. Following this, Section 2.2.3.2 provides an overview of the experiments that address Hypothesis 2.2.

2.2.3.1 Testbeds

The experiments in publication P2 were conducted using the following testbeds:

Testbed 2.1: PrestoDB is deployed on a Kubernetes cluster with one coordinator and four worker nodes. Each node is equipped with a dual-socket Intel® Xeon® Platinum 8280 CPU, featuring 2×28 cores at 2.7 GHz. These nodes have 256 GB of memory and an Intel® DC S3500 SSD for data storage. Each SSD holds a copy of the datasets used in the experiments. All nodes are connected with a 1 Gbit Ethernet network. All queries are executed in isolation and with warm caches.

Testbed 2.2: PrestoDB is deployed on a large-scale cluster consisting of hundreds of compute nodes at Meta.

Testbed 2.3: PostgreSQL 13 is installed on a machine with a 16-core AMD Ryzen 3950X@3.5 GHz. It has 96GB of DDR4-2667 memory. Only those hint-sets that yield new query plans are considered. All queries are executed with warm caches.

Testbed 2.4: SparkSQL v3.2.2 is configured as it is internally used at Intel and runs on a single machine with a dual-socket Intel® Xeon® Platinum 8280 CPU with 2×28 cores at 2.7 GHz and 256 GB of memory. All datasets reside in memory using `tmpfs`.

2.2.3.2 Experiments

To empirically evaluate Hypothesis 2.2, we conduct several experiments using AutoSteer-G for PostgreSQL and SparkSQL.

- **AutoSteer-G for PostgreSQL** is evaluated using the extended Join Order Benchmark [100, 111] and executed on Testbed 2.3. The adaptive hint-set search generates alternative query execution plans that reduce JOB's execution time by up to 33.5% when always choosing the best known hint-set. Further details can be found in Section 4.5 of P2.
- **AutoSteer-G for SparkSQL** is evaluated using TPC-DS [166] and runs on Testbed 2.4. The best hint-sets that AutoSteer finds achieves relative run time improvements of 44.3%. Further details can be found in Section 4.6 of P2.

While the primary focus of this thesis is on AutoSteer-G, additional evidence supporting Hypothesis 2.2 is obtained from AutoSteer-C for PrestoDB.

- **AutoSteer-C for PrestoDB** is evaluated through the extended Join Order [100, 111] and the Stack Benchmark [160] on Testbed 2.1 (cf. Table 4 of P2). The best known hint-sets that AutoSteer discovers reduce the overall execution time by 30.25% for JOB and by 42.38% for Stack. AutoSteer's inference mode uses the pre-trained TCNN to predict the execution time of query plans and achieves improvements of up to 27.93% for JOB and 31.54% for Stack.

Conclusions. The experiments show that AutoSteer-G identifies alternative query plans that reduce the end-to-end benchmark execution times, thereby confirming Hypothesis 2.2. The presented qualitative and quantitative analyses validate Hypothesis 2.1 and Hypothesis 2.2, thereby providing a positive answer to Research Question 2.

2.3 Programming Fully Disaggregated Systems

The research methodology for publication P3 diverges from the two previous publications because of its theoretical and visionary nature. It comprises two steps:

1. A comprehensive literature review identifies and discusses recent trends and advancements in data center hardware and disaggregated systems, specifically considering the implications on the development of data-intensive applications. Section 2.3.1 discusses these trends in more detail and explains how they motivate Research Question 3.
2. Building upon the literature review, Section 2.3.2 presents the foundational concepts of a new programming model and a runtime system for fully disaggregated systems. We show each concept's relevance and how it contributes to answering Research Question 3.

2.3.1 Key Trends in Large-Scale Data-Center Computing

This section presents six important trends identified by thoroughly reviewing current literature and related work.

Trend 1: Data explosion. The data volume's exponential growth described in Chapter 1 has significant implications, particularly for companies focusing on large-scale data analytics. For instance, at Meta, dashboard applications must process petabytes of data from various sources, running on clusters with hundreds of compute nodes [10]. Given the growing data volumes and the demand for fast analytics, developing scalable and efficient data-intensive applications has become an important challenge.

Trend 2: Disaggregated memory. In traditional computer architectures, memory and compute resources are tightly coupled. To reliably run multiple applications on a single machine, they are assigned static, fixed memory and compute resources to not risk the other applications' stability. However, this approach requires the overprovisioning of resources to applications to reliably serve peak workloads, even though the resource demands can significantly fluctuate over their lifetimes, which results in a low memory utilization with average effective use between 50–65% [103, 164]. Considering that memory is an expensive resource that constitutes up to 50% of Azure's server costs [48] and 40% of Meta's rack costs [114], enhancing memory utilization reduces operational costs. Therefore, modern data centers adopt disaggregated architectures that separate memory [4, 46, 56, 137, 142] and compute resources [14, 28, 34, 37, 51, 84, 89, 138] and share them across multiple machines via high-speed networks [18, 135]. Disaggregation creates flexible and efficient memory and compute pools that optimize resource utilization and enhance the applications' scalability.
Trend 3: Data movement. While memory disaggregation addresses the problem of low utilization and availability, recent studies have shown that the dominant cost driver in data centers has become data movement [93, 135]. Optimized data placement strategies that minimize data movement are required and have a large potential to reduce the operational costs of data center applications.
Trend 4: Cache-coherent interconnects. Beyond pooling memory and compute resources across *multiple machines*, new cache-coherent interconnect (CCI) standards like Compute Express Link™ (CXL™) [46] enable memory and compute resource pooling *within single machines*. CXL is based on PCIe 5.0 and enables memory expansions and cache-coherent memory sharing across multiple devices. CXL has already been adopted by data center processors, such as AMD's 4th generation EPYC™ processor [6] and the 4th generation of Intel™ Xeon™ scalable processors [80]. While technologies like CXL allow for more flexible data and compute placement, they also contribute to the system's complexity and require advanced optimization strategies to exploit their potential.
Trend 5: Diverse memory types. Adopting CCIs like CXL facilitates the development of new PICe 5.0 memory devices, which further enrich and contribute to the memory device pool shown in Table 1 of P3. For example, the first CXL memory expansion modules have already been announced [115, 146]. However, to optimize data placement, developers must be aware of the memory devices' properties, including their latencies, bandwidths, supported access types, and persistency guarantees, as they substantially impact the performance.
Trend 6: Optimization complexity. Given these trends, optimizing applications for a heterogeneous hardware environment with accelerators and different memory types becomes increasingly complex. For instance, developing a hardware-efficient external merge sort algorithm for a system involving DRAM,

PMEM, and SSD is a non-trivial endeavor even for expert developers [173]. To fully exploit the potential of such a memory-tiered system requires the usage of specialized libraries like the Storage Performance Development Kit (SPDK) [162] to move data from one device to another more efficiently without involving the host CPU [173]. However, such optimizations are non-trivial and increase the algorithm's complexity significantly. Since the utilization of compute and memory devices is not known at development time, such applications must be optimized *adaptively at runtime*.

Programming model for disaggregated systems. These trends show that hardware capabilities evolve and CCI standards enable new data placement strategies, but leveraging these advancements is becoming increasingly complex and currently the application developer's burden. Existing programming models and frameworks such as MapReduce [52] or Spark [184] already support developers in focusing on the application logic, but they are often too restricted and specialized to a specific use case. Furthermore, *each* of these frameworks must implement new optimization strategies to leverage the full potential of disaggregated systems. Therefore, the identified trends and the lack of suitable programming models collectively motivate the following research question:

> ***Research Question 3:*** *Can high-performance data-intensive applications like database systems be developed and optimized for fully disaggregated systems sustainably?*

2.3.2 Programming Model Design

This section explains the key design principles of a new programming model for fully disaggregated systems. We show how every principle specifically addresses Research Question 3.

2.3.2.1 Positioning the Programming Model

To answer Research Question 3, we propose a new programming model to support existing frameworks in leveraging the capabilities of fully disaggregated systems more *sustainably*. Figure 2.5 provides a high-level overview of the programming model. Data-intensive applications A are often implemented using frameworks like MapReduce [52], Spark [184], Tensorflow [1], and PyTorch [129], which utilize declarative languages that allow developers to specify *what* needs to be accomplished rather than *how* it should be executed [149]. Such programs are usually first parsed into a logical plan B, which is then transformed into a physical plan C. Our programming model's API D takes over at this stage. It accepts the physical plan in the form of a *task-based directed*

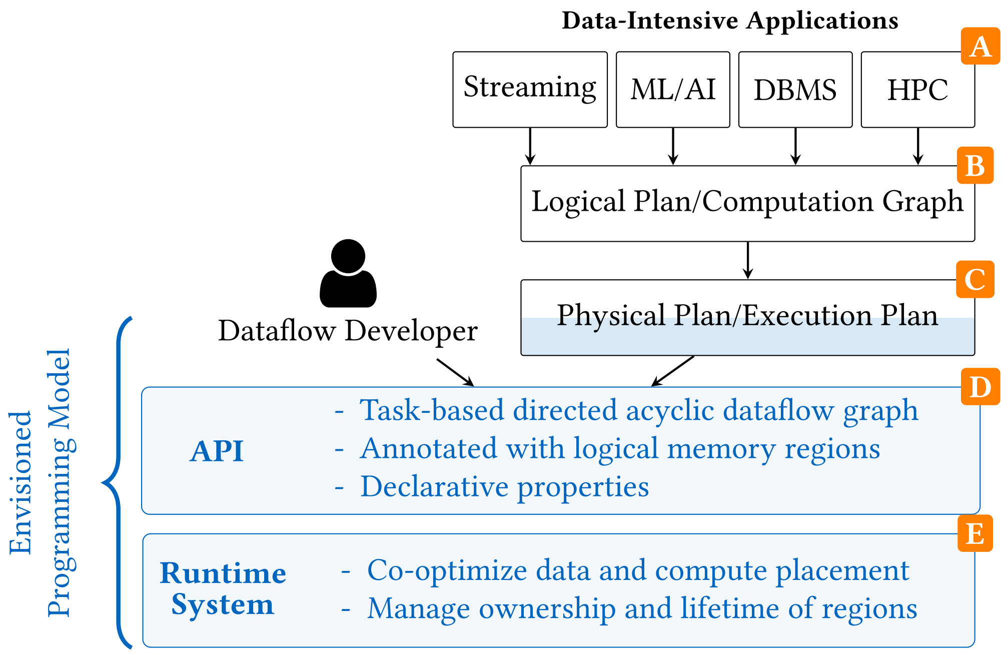

Data-Intensive Applications
A
Streaming
ML/AI
DBMS
HPC
B
Logical Plan/Computation Graph
C
Physical Plan/Execution Plan
Dataflow Developer
Envisioned Programming Model
D
API
- Task-based directed acyclic dataflow graph
- Annotated with logical memory regions
- Declarative properties
E
Runtime System
- Co-optimize data and compute placement
- Manage ownership and lifetime of regions

Figure 2.5: High-level overview of the envisioned programming model.

acyclic graph (DAG) as input. The DAG's nodes represent computational tasks and its arrows denote data flowing from one task to another. Further optimizations, such as data and compute placement, are performed at the application's *execution time* by the programming model's runtime system E. Additional information can be annotated in the physical plan in a declarative manner, details which will be discussed further in the following.

2.3.2.2 Foundational Design Principles

Abstraction through memory regions. As highlighted by trends 4, 5, and 6, different memory types and cache-coherent interconnects enable new data placement options but also make the optimization more complex. Moreover, in cloud-based disaggregated systems, the available devices and their utilization are unknown during the application's development. Furthermore, trend 3 shows that data movement dominates the costs of data centers. The envisioned programming model implements a memory-centric design by raising the memory abstraction level and adopting the concept of *memory regions* [68, 70, 165]. Figure 2.6 visualizes this idea: rather than specifying a particular memory device, dataflow developers define memory regions ① and attach the required memory properties like low latency or high bandwith ②. This idea has recently been adopted in Intel's Unified Memory Framework [124]. As discussed later in this section, memory regions could also be annotated with other attributes like confidentiality ④. At execution time, the runtime system has more information to better co-optimize data and compute placement and map logical memory regions to suitable devices ③.

Addressing Research Question 3*: The programming model uses memory regions to raise the abstraction level, which makes the application's development more sustainable as it becomes independent of the physical memory devices available at runtime.*

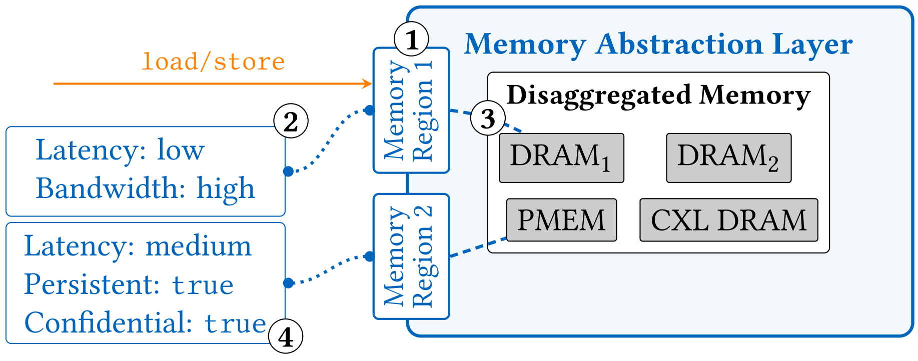


Figure 2.6: Logical memory regions abstract from physical memory devices.

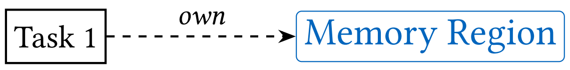
Typed memory regions. Dataflow applications commonly utilize memory for three primary objectives: synchronization, exchanging data, and thread-local computations. Predefining typed memory regions for each objective lets the programming model cover most memory usages and simplifies the application's development. Additionally, predefined memory regions facilitate compile-time checks, like ensuring that different tasks do not concurrently access thread-local memory regions. Table 3 of P3 shows how these three predefined memory regions can be utilized across different application types.

Memory region ownership. As outlined by trend 4, technologies like CXL allow for cache-coherent access to multiple compute devices, raising new questions concerning *memory ownership*, for example, 'who is responsible for cleaning up allocated memory in case of unexpected errors?'

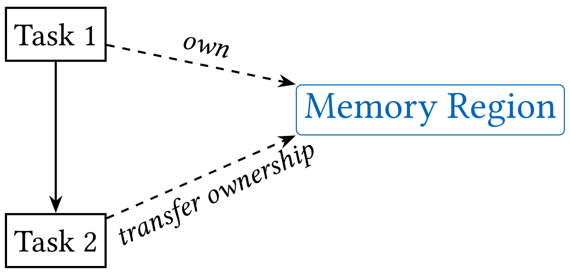
Memory ownership is a well-established concept in modern low-level programming languages like C++11 and Rust, facilitated through smart pointers. Building upon its memory-centric design with explicitly defined memory regions, these concepts can be integrated into the programming model.

Figure 2.7 introduces the key ideas of memory region ownership. Like a unique pointer, a task can exclusively own memory regions (cf. Figure 2.7a). Upon task completion, uniquely owned memory regions get deallocated. The ownership of these memory regions can also be transferred between tasks without the need for physical data movement, as illustrated in Figure 2.7b. Essentially, the memory region is reassigned, similar to *moving* a unique pointer in C++. Additionally, multiple tasks can *share* a memory region for synchronization or message passing (cf. Figure 2.7c).

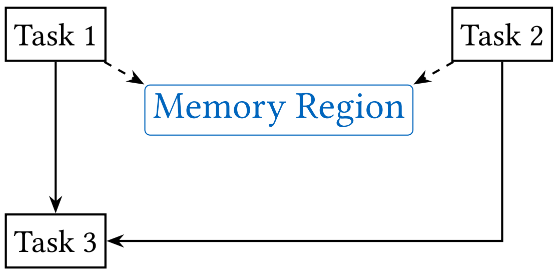
Addressing Research Question 3: *Typed memory regions with explicit ownership can prevent various memory errors and support the applicaton's development. Data placement in disaggregated systems can be optimized based on the memory region's lifetimes.*



(a) Unique ownership of a memory region.



(b) Transferring the ownership of a memory region.



(c) Shared ownership of a memory region.

Figure 2.7: Different ownership types of memory regions.

Declarative annotations. Many data-intensive applications deployed in data centers share common requirements, including data confidentiality and security, persistence and materialization of (intermediate) results, and acceleration on specialized compute devices. Instead of requiring each application to implement these functionalities and optimize them for disaggregated systems independently, the envisioned programming model makes them reusable across applications. To achieve this, developers could attach declarative properties to the DAG's tasks and memory regions. For example, declaring a memory region as *confidential* triggers the runtime system to encrypt the region's data to prevent unauthorized access ④. Declarative annotations simplify the development of data-intensive applications on disaggregated systems and allow developers to focus on the application logic instead of re-implementing complex and error-prone core functionalities.

***Addressing Research Question 3**: Annotating common requirements declaratively to tasks and memory regions simplifies the development and optimization of data-intensive applications for disaggregated systems.*

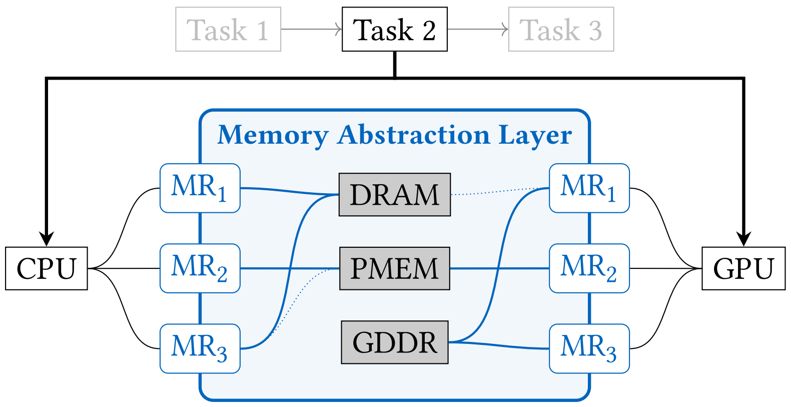


Figure 2.8: Optimizing data and compute placement on fully disaggregated systems is interdependent.

2.3.2.3 Runtime System

The implementation of the programming model requires a complex runtime system (RTS) intersecting multiple layers of the system stack. This section focuses on the RTS's key challenges concerning data and compute placement.
Data placement. The RTS maps the memory regions to specific memory devices. However, this challenge is particularly complex due to the variety of memory devices distributed across multiple machines in modern data centers and their concurrent usage by different applications. Thus, the RTS must optimize the memory region's mapping while considering various factors, including device utilization, the expected size of memory regions, and their required properties. Additionally, the RTS must also reduce data movement, which has become the dominating cost factor in modern data centers (cf. trend 3).
Task placement. Data and compute-intensive tasks can often benefit significantly from offloading to specialized co-processors. Specifically, previous research efforts investigated how FPGAs [36, 65, 86, 128], GPUs [29, 30, 151, 153, 154] and TPUs [77] can be used to accelerate database workloads. While modern data center CPUs come with various built-in accelerators, such as for streaming

or encryption [80], choosing the right processing unit for a task is crucial for optimizing energy efficiency and performance. Since the available accelerators and their utilization are not known at development time, it is the RTS's task to cost-efficiently map computational tasks to the most suitable devices.
Co-optimizing data and compute placement. The presented challenges in data and compute placement are interdependent and must be co-optimized for optimal system performance and efficient resource utilization. The need for such co-optimization provides plenty of opportunities for further research, including effective cost models for fully disaggregated systems. Figure 2.8 visualizes this problem for a task that a CPU or a GPU could process, showing that the optimal memory device also depends on the processing device the task runs on. For example, when a task runs on the GPU, memory regions will preferably be mapped to GDDR as it provides a lower latency and a higher bandwidth than DRAM. The opposite is the case for a task running on the CPU.

***Addressing Research Question 3**: Given that important information, such as device availability and utilization, is only known at the application's execution time, the proposed programming model delegates the co-optimization of data and compute placement to the runtime system.*

For a more comprehensive discussion of the runtime system, its design and functionality, and further challenges, please refer to Section 3 of P3. However, since these details are not directly linked to Research Question 3, they are *not* discussed here.

2.3.3 Conclusions

We identified six important trends in large-scale data center computing and introduced the design concepts of a novel programming model that facilitates the implementation of dataflow applications and their optimization for fully disaggregated systems. The main idea is to use logical memory regions to make the application development independent of the memory devices that are available at runtime. The memory regions are dynamically mapped to physical devices at execution time, which is facilitated through a runtime system that adaptively co-optimizes data and compute placement. The proposed design principles collectively answer Research Question 3.

CHAPTER 3

Related Work

This chapter provides an overview of the work on adaptive optimizations in database systems. First, we present the most important related work of the publications of this thesis. Section 3.1 focuses on Adaptive Hybrid Indexes. Section 3.2 investigates related work on learned query optimization and Section 3.3 examines work related to the programming of disaggregated systems. Beyond our publications, adaptivity plays a crucial role in many other parts of database systems: Section 3.4 focuses on adaptive optimizations that interleave query optimization and execution. Furthermore, Section 3.5 discusses self-driving database systems that can adapt to changing workloads.

3.1 Adaptive Hybrid Indexes

A more comprehensive overview of the related work on Adaptive Hybrid Indexes can be found in Section 6 of P1.

Hot/cold clustering is a data management strategy that differentiates frequently (hot) from rarely (cold) accessed data. This approach enhances application performance by moving cold data to slower storage while keeping hot data in faster main memory. Funke et al. [67] implemented this strategy within hybrid transactional/analytical processing systems, utilizing workload skewness to efficiently categorize pages into *hot*, *cold*, and *frozen*. Levandoski et al. [102] sample accesses at a record level, exploring different offline analysis techniques to accurately estimate the record's access frequencies.

Succinct trees use space close to the information-theoretic lower bound but still enable efficient read operations. Instead of using fixed-size memory addresses (pointers) to store node relationships *explicitly*, these are encoded *implicitly*. One such encoding is the ***L**evel-**O**rdered **U**nary **D**egree **S**equence* (LOUDS), introduced by Jacobson in 1988, which encodes ordinal trees in bit vectors [82]. This encoding method uses zero bits to mark the end of nodes, while the number of 1 bits represents the number of child nodes. Figure 3.1 illustrates the LOUDS encoding at the bottom for a simple binary tree shown at the top. While LOUDS significantly

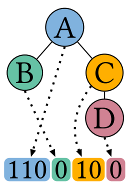


Figure 3.1: LOUDS Encoding.

reduces memory usage, it increases the number of instructions for traversal, leading to slower performance compared to pointer-based data structures [7]. To mitigate this performance problem, Zhang et al. developed the Fast Succinct Trie, which combines two LOUDS-based encodings. LOUDS-Dense optimizes the access performance for the frequently accessed but smaller upper levels, whereas LOUDS-Sparse reduces space consumption for the lower levels, thereby balancing space efficiency and performance [187].

Hybrid index structures. As discussed in Section 2.1, there are two types of hybrid index structures. To facilitate the implementation of Type 1 hybrid indexes, Zhang et al. introduced the dual-stage reference architecture [186]. The dynamic stage prioritizes performance and primarily stores frequently accessed data; the static stage focuses on space efficiency and compactly stores the rarely-accessed majority of the data. Additionally, they propose compaction rules to minimize the memory footprint of conventional index structures such as B+-trees, radix trees, and skip lists. Dual-stage hybrid indexes periodically merge cooling data and updates from the dynamic into the static stage.

The B^2-tree combines trie- and comparison-based search strategies to optimize string indexing in database systems [147]. Its design is similar to a B+-tree with interlinked leaf nodes that improve scan performance. It uses two different node types: *decision nodes* are similar to traditional B+-tree nodes and link to child nodes, whereas *span nodes* are organized as search tries that store the keys' common prefixes to reduce the tree's height and cache misses. The B^2-tree is a Type 1 hybrid index that uses different node encodings to adapt to the stored data. However, it does *not* adapt to the workload's node accesses.

In 2023, Zhou et al. proposed 2-Tree [191], a reference architecture to improve performance under skewed workloads for indexes that are larger than the available memory. The idea is to utilize two indexes: one index structure stores the frequently accessed records, while a separate index stores rarely accessed data. Their experiments show that 2-Tree increases memory utilization while it reduces the number of page evictions.

The B^{link}-hash is a new Type 2 hybrid index and a B+-tree variant optimized for time-series data [38]. Similar to the Hybrid B+-tree, it employs two leaf node encodings. For rarely updated nodes, B^{link}-hash uses the conventional B+-tree node encoding, which is less efficient for inserting monotonically increasing data due to high contention. Contrary, hash nodes enhance the concurrent insert operations by distributing them across multiple buckets.

The Scalable Adaptive Learned Index (SALI) uses decentralized, lightweight probabilistic models to improve the performance of concurrent workloads [69]. The probabilistic models facilitate more informed decision-making, like retraining nodes or changing their encodings.

DiffLex improves memory usage by storing rarely accessed keys in dense arrays, thus conserving memory space [47]. Recently inserted keys are stored in a performance-optimized delta array, similar to the dual-stage hybrid index [186]. Once the delta array is full, it is merged into the primary array. Additionally, DiffLex enhances scalability on multi-node platforms by replicating hot keys across NUMA nodes.

Morphtree [106] implements different leaf node encodings that are optimized for read- or write-dominated workloads, similar to the Hybrid B+-tree. Leaf node encodings are chosen adaptively at runtime based on the sampled accesses.

3.2 Learned Query Optimization

For a more exhaustive overview of the work related to learned query optimization, please refer to Section 2 of P2.

In recent years, machine-learning-based approaches have been proposed to improve traditional query optimizers. These range from improved cardinality estimation [90, 91, 120, 163, 174, 182] to join ordering [113, 183]. Furthermore, fully learned drop-in replacements for query optimizers have been proposed, like Neo in 2019 [112] and Balsa in 2022 [181]. However, these approaches require long training times and can lead to significant performance regression.

In contrast to fully learned query optimizers, *steered query optimizers* enhance existing query optimizers by machine-learned hints to steer the optimization process (refer to Section 2.2.1.1 for more details). A prominent example is the Bandit Optimizer (Bao) from 2021 [111], which has seen successful industrial adoption in Microsoft Scope [118, 188] and other database systems like Vertica, Amazon Redshift, and Microsoft Azure Synapse [12]. These approaches use predefined static hint-sets to generate multiple alternative query plans. A machine-learned tree convolutional neural network (TCNN) predicts the execution times of these query plans. Instead of always selecting the best plan, Bao balances exploitation and exploration using Thompson sampling.

FastGres from 2023 leverages context-aware learned models to *predict hint-sets* for incoming analytical SQL queries in PostgreSQL [178], which is in contrast to Bao's approach of predefining the hint-sets.

Lero, proposed in 2023, is a learning-to-rank query optimizer [193]. Instead of predicting query plan latencies, it leverages a pairwise TCNN-based plan comparator model to rank the different query plans and select the cheapest one.

3.3 Programming Fully Disaggregated Systems

A more comprehensive overview of the work related to the programming of fully disaggregated systems can be found in P3.

Memory regions. Besides the two popular memory management strategies allocation-deallocation (like `malloc/free` in `C`) and garbage collection, region-based memory management organizes memory into discrete regions [68]. This method allocates objects with similar lifetimes within the same region and facilitates collective deallocation, significantly reducing the deallocation time of nested index structures like trees and graphs. Region-based memory has been shown to surpass traditional memory management techniques in performance for dataflow applications under certain workloads [68]. Broom leverages memory regions to enhance performance in garbage-collected distributed dataflow systems [70]. Since the dataflow operator's allocated objects usually persist for no longer than the operator itself, these objects are grouped into the same memory region and deallocated all at once when the operator finishes.

Programming models. The end of Moore's law and the rise of multi-core processors, specialized accelerator devices, and the disaggregation of memory and compute resources have complicated the implementation of dataflow applications. In response, several programming languages and models have been proposed to simplify application development. One example is IBM's X10, an open-source parallel and object-oriented programming language initiated in 2004 [39]. X10 specifically addresses the challenges of parallel and high-performance non-uniform cluster computing. It introduces partitioned global address spaces, which divide memory among nodes to enhance efficient data access. X10's *places* allow the programmer to define *where* data and tasks are placed. However, these placements are fixed and cannot be changed at runtime.

Following this, in 2010, the SMPS programming model dynamically detects task dependencies at runtime and automatically extracts parallelism [133]. Legion is another example of a parallel programming model. It uses logical regions that describe the data organization [21]. In addition to SSMP, Legion supports arbitrary partitioning of logical regions.

3.4 Adaptive Query Optimization and Execution

Traditional query optimization. Traditional cost-based approaches for query optimization and execution date back to System R in 1979 [155] and adhere to the *'optimize-then-execute'* paradigm, which distinctly separates query optimization and execution. Despite its widespread adoption over the years [44, 72, 73, 74, 98, 159, 168], this paradigm imposes several challenges. One is the accumulation of

inaccuracies in estimating join cardinalities and predicate selectivities, especially with a growing number of joined relations and predicates. These inaccuracies can lead to suboptimal plans significantly impacting query performance [100]. To address these challenges, different approaches combine query optimization and execution and transform them into an adaptive process that adjusts the plan upon detecting estimation errors [54, 71, 75].
Adaptive join order. In contrast to System R's static approach, the Ingres database system's query decomposition scheme uses a greedy algorithm that adaptively chooses the next smallest relation [161]. Ingres' approach of dynamic re-optimization has also recently found its way into modern big data management systems [132]. In addition, even more flexible approaches have been proposed. For example, the *eddy* operator dynamically optimizes the query plan on a per-tuple basis, known as *row-routing* [16, 53, 104, 139]. In row-routing approaches, a runtime optimizer actively monitors and adjusts to the operator selectivities observed during execution and independently routes individual rows or batches through a series of operators.
Adaptive operators. For long-running analytical queries, the available memory resources may fluctuate during execution since resources are shared among multiple concurrently executed queries. This presents a significant challenge for memory-intensive operations like sorting. Memory-adaptive sort [126] and hash join operators [127] have been proposed to efficiently utilize additional memory buffers while gracefully degrading under memory constraints. In distributed database systems, *locality-sensitive operators* can dynamically determine whether they should execute locally or broadcast to other nodes during execution [143].
Adaptive compilation. Also, adaptive optimizations are crucial to enhance query performance in compilation-based database systems, such as HyPer [88] and Umbra [121]. For instance, the compilation overhead for short-running queries can exceed the execution time. The *adaptive execution framework* uses a bytecode interpreter for LLVM IR for short-running queries, thereby avoiding the expensive compilation overhead [95]. Based on the adaptive execution framework, Schmidt et al. proposed Dynamic Blocks to enable intra-query optimization without the need for recompilation [148]. In this approach, several query plans are explored and the most efficient one is used to process the remaining query.

3.5 Self-Driving Database System

Self-driving database systems are designed to autonomously adjust to workload changes and have attracted researchers and industrial professionals for decades. The sustained interest and the enormous potential of self-driving database systems is well-documented [25, 42, 43, 175]. While most approaches are not fully autonomous, they are rather decision-support tools aiding database administrators (DBA) in specific tasks. Pavlo et al. introduce a new taxonomy that categorizes autonomous database systems into six levels, from manual systems requiring significant human intervention to fully autonomous, self-driving systems [131]. Next, we provide a non-exhaustive overview of the existing approaches.

Physical database design. IBM's DB2 database system implements several methods to achieve more autonomy. The DB2 Partition Advisor analyzes SQL statements and automatically determines the most efficient data partitioning strategy to minimize the overall workload cost [140]. DB2's Design Advisor recommends suitable indexes [170], materialized views, or multidimensional clusterings of tables [194]. Parallel to IBM's efforts, the AutoAdmin project at Microsoft Research started in 1996 and focuses on automating physical database design [2, 42], including indexes [40, 41] and materialized views [3].

Index selection. Index structures can significantly improve the performance of query processing. However, indexes also degrade the performance of write- and update-heavy workloads [66]. Additionally, limited memory resources prevent the implementation of all potential indexes, but they require a careful selection, which is known as the *index selection problem* [66]. In 1998, Chaudhuri and Narasayya proposed the hypothetical '*what-if*' indexes, which are used to estimate the query costs if the index would exist [41]. Also, commercial tools from Microsoft [40], IBM DB2 [170], and Oracle [49] assist DBAs in the selection of suitable indexes.

Several machine learning-based approaches have addressed the index selection problem in recent years. In 2015, Basu et al. proposed a cost model oblivious approach based on reinforcement learning [20]. In 2019, machine-learned models used query execution statistics to recommend indexes [55]. Furthermore, in 2020, active learning automated index selection in Microsoft Azure, demonstrating the practicability in a cloud database environment [108].

Self-driving database systems. In recent years, self-driving databases with higher levels of autonomy have emerged. They surpass mere recommendation tools for single aspects like index selection. One example is the Peloton database system [130]. Peloton continuously monitors query execution and resource utilization and uses them for workload classification and ML/AI-based forecasting. Its *control framework* oversees the system and the workloads and derives appropriate actions. These actions range from data migration and partitioning to configuration adjustments and modifications of the storage layout. Furthermore, Peloton dynamically adjusts the number of compute nodes to adapt to changing workloads.

Further advancements in query forecasting [107] and enhanced data collection methodologies [31] have been pivotal in the development of NoisePage, the successor of Peloton [123, 131]. Recognizing the complexity of DBMSs, NoisePage adopts a novel approach by decomposing the system into distinct operating units, all of which are then modeled separately [109].

CHAPTER 4
Conclusions

This thesis introduced novel frameworks for index structures and query optimization and proposed the main design principles for a new programming model for fully disaggregated systems. Our contributions demonstrate the significant potential of adaptive optimizations in database systems.

Adaptive Hybrid Indexes

We presented a novel framework that facilitates the implementation of workload-aware Adaptive Hybrid Indexes. The Hybrid Trie combines the Adaptive Radix Tree (ART) [99] and the slower but more space-efficient Fast Succinct Tree (FST) [187]. ART encodes the more frequently accessed upper levels, while the FST stores the rarely accessed majority of the data. Hybrid Trie adapts to changing workloads by sampling node accesses and then migrating frequently accessed nodes to ART and cold nodes to FST.

The Hybrid B+-tree uses three leaf node encodings, each having different implications on space and query performance. Sampled node accesses are used to adaptively change the leaf nodes' encodings to minimize space consumption and maximize performance.

The experimental evaluation shows that our framework can significantly compress index structures while causing negligible performance overhead, outperforming traditional indexes when space and query performance are considered equally important.

Future work. There are several opportunities to further improve Adaptive Hybrid Indexes. One major limitation is the contention the centralized access tracking introduces, particularly when multiple threads try to update the statistics for a single node concurrently. Therefore, decentralized access tracking – similar to the idea proposed in Scalable Adaptive Learned Indexes [69] – could improve the performance of concurrent workloads. Furthermore, graph workloads often exhibit skewed access patterns. Our framework could facilitate the implementation of hybrid graph structures that improve performance and reduce memory consumption [35].

Learned Query Optimization

Our second publication introduced AutoSteer, a novel framework that automatically steers rule-based query optimizers. AutoSteer builds upon the Bandit Optimizer (Bao) [111] but requires minimal integration effort and human expertise as it connects to database systems only via SQL and EXPLAIN statements. Therefore, it can be applied effortlessly to other database systems and it achieves excellent generalizability. However, one limitation of this approach is that it does not reliably identify all effective rewrite rules since not all changes made by the rewrite rules are discoverable through explained query plans.

We integrated AutoSteer successfully into five widely used, open-source database systems. Moreover, we tested AutoSteer with synthetic benchmarks and real-world workloads from a petabyte-scale PrestoDB deployment at Meta, which significantly reduced tail latencies of dashboard queries. Furthermore, our experiments have shown that AutoSteer's query span approximation algorithm and the adaptive greedy hint-set search discover alternative query plans more efficiently than previous approaches.

Future work. Instead of solely optimizing execution time, future research could explore multi-objective cost functions, considering other important metrics like memory usage and processing time. Additionally, AutoSteer can be an invaluable tool for query optimization experts since it systematically detects problematic rewrite rules and provides concrete example queries for which the rule degrades the performance. Such insights can help to enhance rule-based query optimizers.

Programming Fully Disaggregated systems

We identified six prevailing trends influencing the design and implementation of data-intensive applications and their optimization for fully disaggregated systems. Based on these trends, we introduced the key design principles of a new programming model that adopts a memory-centric approach based on logical memory regions. Building upon execution plans from data-intensive applications like database systems, streaming, or ML/AI applications as input broadens our programming model's applicability and facilitates its adoption. The programming model delegates decisions on data and task placement to the runtime system, which adaptively co-optimizes them at execution time.

Future work. The proposed programming model opens up many questions and challenges. For example, future research could investigate the implementation of the memory abstraction layer and how the logical memory regions could be integrated into general-purpose programming languages. A first step towards memory allocation across different devices based on declarative annotations could be Intel's Unified Memory Framework [124]. Additionally, the runtime system crosses multiple layers of the system stack and raises questions like 'Who is responsible for managing the utilization of the disaggregated resources'? Furthermore, leveraging different compute devices to accelerate database workloads introduces new challenges. A layered compilation stack based on MLIR, as adopted in LingoDB, could be a promising approach [85]. Section 3 of P3 sets up the stage for future work and deeper investigation into these challenges.

PUBLICATION P1

Adaptive Hybrid Indexes

P1.1 Synopsis

Database systems use index structures to improve the performance of queries with highly selective predicates. Index lookups are significantly cheaper than scanning the entire relation when only a few tuples qualify. However, index structures also contribute substantially to the memory footprints of database systems. Sometimes, they comprise half of the database system's memory usage [185]. As data volumes grow faster than memory capacities, more compact index representations can fit larger datasets into memory. However, compressed index structures not only reduce the memory footprint but also incur runtime overhead due to the additional decompression steps.

This publication introduces Adaptive Hybrid Indexes, a new framework that uses lightweight tracking of the workload's queries to identify hot and cold data. Based on the workload's access patterns, Adaptive Hybrid Indexes migrate frequently accessed nodes to performance-optimized encodings, while rarely accessed nodes are compressed. We differentiate two types of hybrid indexes:

Type 1: Multiple node encodings with different space-performance trade-offs are utilized within one index.

Type 2: Two or more index structures are combined into one logical index, e.g., a performance-optimized and a succinct index.

To show our approach's effectiveness, we integrate the framework into B+-trees and prefix trees, which are widely used index structures in database systems. The Hybrid B+-tree (Type 1) uses different leaf node encodings, each being optimized for different access types. The Hybrid Trie (Type 2) combines the Adaptive Radix Tree [99] and the Fast Succinct Trie [187] into one index.

We conduct experiments to evaluate specific components and parameters of the framework. Comprehensive end-to-end tests based on established public benchmarks and real-world workloads allow us to assess the overall implications on performance and space, demonstrating that Adaptive Hybrid Indexes can significantly reduce space consumption while hardly affecting performance.

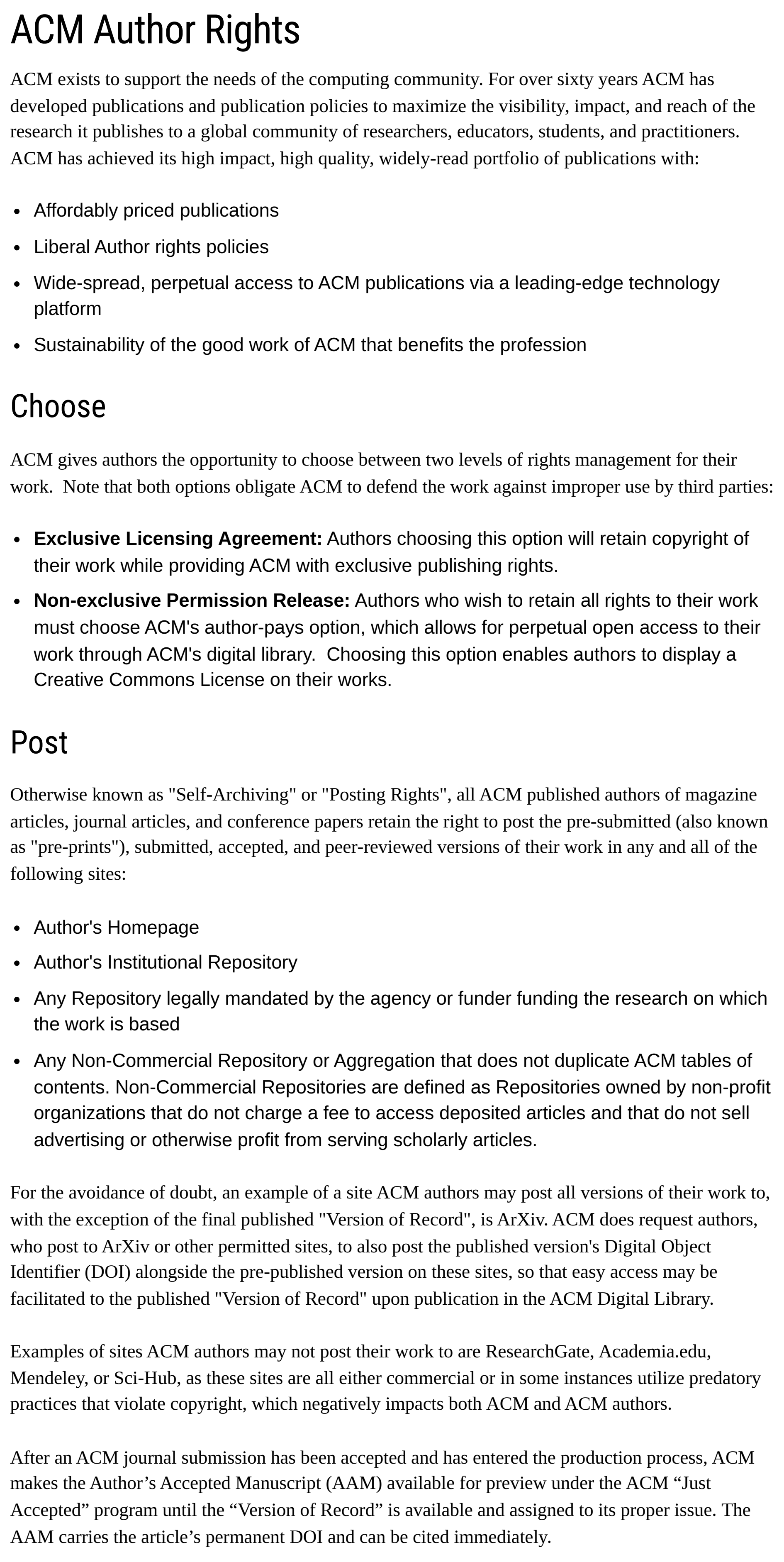
P1.2 Contributions and Publication Details

Author Contributions. Christoph Anneser developed the ideas and the concepts of Adaptive Hybrid Indexes. Furthermore, he implemented the framework and integrated it into B+-trees and prefix trees. In addition, he conducted the experiments and authored substantial parts of the paper.

Reference. Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. "Adaptive Hybrid Indexes". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* ACM, 2022, pp. 1626–1639.

DOI. https://doi.org/10.1145/3514221.3526121

https://authors.acm.org/author-resources/author-rights. Last accessed: 2024/06/10.



ACM Author Rights

ACM exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

Choose

ACM gives authors the opportunity to choose between two levels of rights management for their work. Note that both options obligate ACM to defend the work against improper use by third parties:

- **Exclusive Licensing Agreement:** Authors choosing this option will retain copyright of their work while providing ACM with exclusive publishing rights.
- **Non-exclusive Permission Release:** Authors who wish to retain all rights to their work must choose ACM's author-pays option, which allows for perpetual open access to their work through ACM's digital library. Choosing this option enables authors to display a Creative Commons License on their works.

Post

Otherwise known as "Self-Archiving" or "Posting Rights", all ACM published authors of magazine articles, journal articles, and conference papers retain the right to post the pre-submitted (also known as "pre-prints"), submitted, accepted, and peer-reviewed versions of their work in any and all of the following sites:

- Author's Homepage
- Author's Institutional Repository
- Any Repository legally mandated by the agency or funder funding the research on which the work is based
- Any Non-Commercial Repository or Aggregation that does not duplicate ACM tables of contents. Non-Commercial Repositories are defined as Repositories owned by non-profit organizations that do not charge a fee to access deposited articles and that do not sell advertising or otherwise profit from serving scholarly articles.

For the avoidance of doubt, an example of a site ACM authors may post all versions of their work to, with the exception of the final published "Version of Record", is ArXiv. ACM does request authors, who post to ArXiv or other permitted sites, to also post the published version's Digital Object Identifier (DOI) alongside the pre-published version on these sites, so that easy access may be facilitated to the published "Version of Record" upon publication in the ACM Digital Library.

Examples of sites ACM authors may not post their work to are ResearchGate, Academia.edu, Mendeley, or Sci-Hub, as these sites are all either commercial or in some instances utilize predatory practices that violate copyright, which negatively impacts both ACM and ACM authors.

After an ACM journal submission has been accepted and has entered the production process, ACM makes the Author's Accepted Manuscript (AAM) available for preview under the ACM "Just Accepted" program until the "Version of Record" is available and assigned to its proper issue. The AAM carries the article's permanent DOI and can be cited immediately.

https://authors.acm.org/author-resources/author-rights. Last accessed: 2024/06/10.

Distribute

Authors can post an Author-Izer link enabling free downloads of the Definitive Version of the work permanently maintained in the ACM Digital Library.

- On the Author's own Home Page or
- In the Author's Institutional Repository.

Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is notthe editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

Create

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.

Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"
- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

Retain

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
- Ownership and control of third-party permissions to use of software published by ACM

Adaptive Hybrid Indexes

Christoph Anneser
Technical University of Munich
anneser@in.tum.de

Andreas Kipf
Massachusetts Institute of Technology
kipf@mit.edu

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

ABSTRACT

While index structures are crucial components in high-performance query processing systems, they occupy a large fraction of the available memory. Recently-proposed compact indexes reduce this space overhead and thus speed up queries by allowing the database to keep larger working sets in memory. These compact indexes, however, are slower than performance-optimized in-memory indexes because they adopt encodings that trade performance for memory efficiency. Applying different encodings within a single index might allow optimizing both dimensions at the same time – however, it is not clear which encodings should be applied to which index parts at *build-time*.

To take advantage of multiple encodings in *one* index structure, we present a new framework forming the basis of *workload-adaptive hybrid indexes* which moves encoding decisions to *run-time* instead. By sampling incoming queries adaptively, it tracks accesses to index parts and keeps fine-grained statistics which are used for space- and performance-optimized encoding migrations. We evaluated our framework using B+-trees and tries, and examine the adaptation process and space/performance trade-off for real-world and synthetic workloads. For skewed workloads, our framework can reduce the space by up to 82% while retaining more than 90% of the original performance.

CCS CONCEPTS

• **Information systems** → **Data access methods**; **Data layout**.

KEYWORDS

Space-efficient Index; Adaptive Index; Hybrid Index

ACM Reference Format:
Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526121

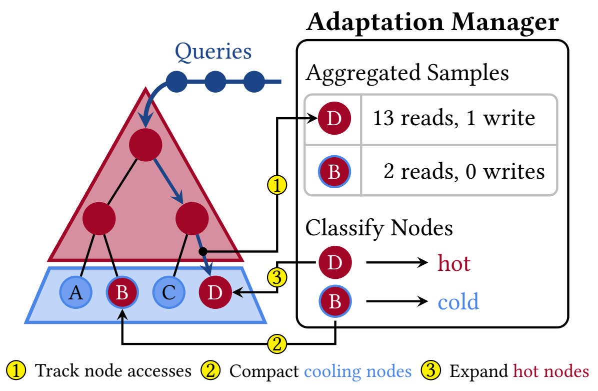


Figure 1: Our sampling-based workload adaptation supports hybrid index structures in choosing the most suitable encoding for each part based on fine-grained access statistics at run-time. It supports user-defined settings such as an upper memory budget and it keeps sampling-related overhead limited by following an adaptive cost-optimized approach.

1 INTRODUCTION

Back in 2006, Jim Gray stated that memory is the new disk and disk is the new tape [5]. This also applies to modern database systems that store the entire data in random access memory (RAM) to allow real-time analyses for trading companies and financial services, for example. They need to process large datasets efficiently to react to new developments and updates within a few milliseconds.

While the DRAM-prices have been stable during the last six to seven years, the data collected by sensors, smartphones, social media platforms, IoT-devices, and digital market-places *increases at a high rate* resulting in data overflows [54], and storing all data in memory becomes infeasible in many cases. However, as in-memory database systems become more and more popular for performance-critical businesses, AWS offers RAM instances that are optimized for in-memory database systems [1]. These instances are equipped with in-memory capacities of up to 24 TB, but the hourly cost of such an instance is more than $120.

To achieve high-performance query-processing for real-time analyses, index structures such as B-trees, tries, and hash tables are widely used by DBMSs. Because there might be multiple indexes per table, especially in OLTP DBMSs, the storage overhead for indexes can be significant. In many cases, more than half of the available memory of a DBMS can be attributed to index structures [54].

Over the last decades, multiple approaches have been developed to represent traditional data structures using more compact encodings [39]. For example, succinct representations avoid storing unnecessary pointers in a data structure by calculating the nodes' position offsets directly [55]. While succinct indexes require significantly less memory compared to performance-optimized state-of-the-art indexes, most of them are slower in point lookups and scans, and they do not support updates efficiently.

To overcome the disadvantages of memory-efficient but static indexes, Zhang et al. [53] proposed *hybrid index* which is a dual-stage architecture combining a regular dynamic index and the memory-efficient read-only one into a single logical index. The dynamic component in hybrid index absorbs all updates and periodically merges all the delta into the more compressed static component. This approach, however, imposes overhead in the expensive merge process because different node encodings are separated into two stand-alone data structures.

"The RUM conjecture" states that we cannot have all three of read, update, and space optimized for a data structure [11]. For example, succinct data structures achieve close to theoretically optimal space, but they sacrifice read performance and updatability. Most index structures used in today's DBMSs are designed for fast reads and updates, and therefore, often at the expense of the memory overhead.

However, we found that the RUM conjecture could have less effect when the workloads are skewed. Unlike standard benchmarks such as TPC-H where the data is uniformly distributed, we observe heavy skews in real-world workloads. The skew appears in multiple dimensions such as in query patterns, keys, and access-space [14].

We, therefore, propose to leverage the skewed workload patterns to determine node layouts at a fine-granularity based on their access frequencies sampled *adaptively at run-time* so that we can reduce the memory overhead of an index while sacrificing minimum performance (cf. Figure 1). More precisely, given an unbounded stream of index queries where the keys follow an unknown distribution, our approach adjusts the layout for each node adaptively so that "hot" nodes are encoded using performance-optimized formats while "cold" nodes are highly compressed.

The framework we proposed in this paper is divided into two phases: during the first phase, we sample and aggregate accesses to different parts in an index ①. In the second phase, we run a heuristic-based classification to identify hot and cold parts. Based on the access statistics and the most recent classifications, we compact cold parts ② and expand hot parts ③ adaptively using different encodings to achieve a better performance-space trade-off. Furthermore, our framework separates all index-related code from the sampling and classification logic so that it can be easily integrated into existing indexes and systems.

The evaluation in Section 5 shows that our framework can successfully identify the hot and cold parts of an index at a fine granularity and then adjust their encodings adaptively. For skewed workloads, our workload-adaptive hybrid indexes reduce the memory overhead by up to 82% while retaining more than 90% of the performance compared to the original state-of-the-art indexes.

We make the following contributions:

1. A novel framework that helps indexes choose different encodings adaptively based on our lightweight workload sampling to make better performance-memory trade-offs.
2. An alternative offline training for hybrid indexes based on historic or predicted workloads.
3. Applied the framework to two widely-used index structures: B+- and prefix trees.
4. An in-depth evaluation using both real-world and synthetic workloads.

In the following, we first present an overview of sampling-based classification approaches in Section 2. These preliminary approaches are internally used by our approach, which is introduced in Section 3. We provide detailed insights into our approach at an algorithmic level and experimentally evaluate the used parameters. In Section 4, we integrate our approach into a B+-tree and a prefix tree. For both indexes, we present an in-depth evaluation in Section 5. We provide an overview of the related work in Section 6. Ultimately, we draw conclusions and outline possible future work in Section 7. Further experiments based on other datasets and workloads can be found in the online appendix, which is available at https://www.hybrid-index.online.

2 PRELIMINARIES

Many existing works leverage skew in the context of database systems (e.g., [8, 29, 33, 42, 45]). The main idea is to keep frequently accessed data in DRAM to improve overall system performance [29]. For example, Levandoski et al. proposed different *offline* algorithms to efficiently identify hot tuples in Microsoft's in-memory database Hekaton so that the cold ones can be swapped out to disk [33]. To speed up classification, they uniformly sample 10% of all record accesses and accept a *memory hit rate loss* of 2.5% compared to evaluating all record accesses. Depending on the context such as the available memory and the working set (data which is actively used), they *rephrase* the hot-cold-classification as a top-k frequent item detection problem: the k records with the highest estimated frequency are classified as *hot* and are thus kept in memory.

Existing optimizations to the top-k algorithms [15, 16, 37, 38] often assume predefined sizes for the data samples. Therefore, it requires us to determine an appropriate sample size first before we can identify frequently accessed items at *run-time*. While a smaller sample size can lead to higher classification errors, larger sample size brings extra overhead in collecting and analyzing the samples.

To keep classification errors limited, we make use of error-bounded top-k approximations which we formally define as follows. Let $\mathcal{I}$ be a set of items and let $\mathcal{D}$ be a multiset defined as a 2-tuple $\langle \mathcal{I}, m_{\mathcal{D}} \rangle$ with a function $m_{\mathcal{D}} : \mathcal{I} \rightarrow \mathbb{N}_0$ describing the multiplicity of each item in $\mathcal{D}$. Let $\mathcal{S} = \langle \mathcal{I}, m_{\mathcal{S}} \rangle$. We call $\mathcal{S}$ a *sample* of $\mathcal{D}$, iff $\forall x \in \mathcal{I} : m_{\mathcal{S}}(x) \leq m_{\mathcal{D}}(x)$. We define $f_{\mathcal{X}} : \mathcal{I} \rightarrow [0, 1]$ to be a function mapping items to their relative frequencies within an arbitrary multiset $\mathcal{X}$, with $f_{\mathcal{X}}(y) = m_{\mathcal{X}}(y) / \sum_{x \in \mathcal{I}} m_{\mathcal{X}}(x)$, and let $f^k_{\mathcal{X}}$ be the k$^{\text{th}}$ largest frequency within $\mathcal{X}$ for $1 \leq k \leq |\mathcal{I}|$.
According to [43], we define the set of top-k frequent items as follows:

$$TOPK(\mathcal{D}, \mathcal{I}, k) = \{(x, f_{\mathcal{D}}(x)) \mid x \in \mathcal{I} \wedge f_{\mathcal{D}}(x) \geq f^k_{\mathcal{D}}\}$$

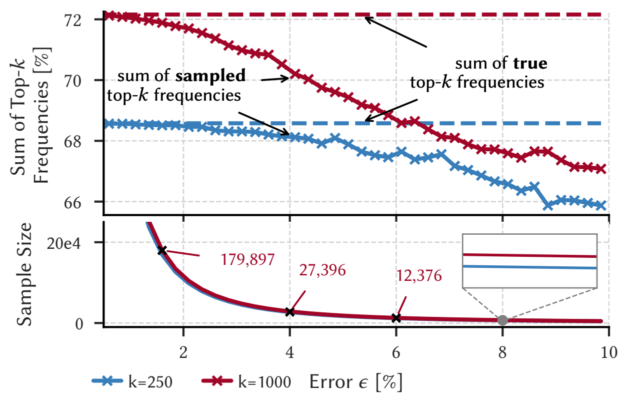


Figure 2: Sample sizes according to Equation (1) for error-bounded top-k analyzes for 1M items. Dashed lines denote the sum of the true top-k frequencies whereas solid lines show the sum of the sampled top-k item frequencies. The workload is generated using a Lognormal distribution. While $\epsilon < 5\%$ does not yield considerable precision gains, it yields larger sample sizes. Experiments using other distributions show similar results and can be found in the online appendix.

An ϵ-approximation to $TOPK(\mathcal{D}, \mathcal{I}, k)$ is a set W of k pairs (x, f) such that $x \in \mathcal{I}, f \in [0, 1]$, and for which the following holds:

$$\forall(x,f) \in W : f_{\mathcal{D}}(x) \geq f^k_{\mathcal{D}}(x) - \epsilon$$
$$\forall(x,f) \notin W : f_{\mathcal{D}}(x) < f^k_{\mathcal{D}}(x) + \epsilon$$
$$\forall(x,f) \in W : |f - f^k_{\mathcal{D}}(x)| \leq \epsilon$$

We use the equation introduced in [43] to calculate the required sample size $|\mathcal{S}|$ for an ϵ-approximation at a probability of $1 - \delta$ with $\delta \in (0, 1)$.

$$|\mathcal{S}| = \frac{2}{\epsilon^2} \ln \frac{2n + k(n-k)}{\delta}, \text{ with } n = |\mathcal{I}| \qquad (1)$$

In Figure 2, we visualize the classification precision and the required sample sizes for varying error rates ϵ. In the upper plot, we compare the sum of the top-k frequencies based on the entire dataset (dashed line) to the one based on the sample (solid line). The lower plot shows the required sample size based on Equation (1), where we observe fast-growing samples for decreasing ϵ. We conducted this experiment for other distributions as well (see online appendix) and found that $\epsilon = \delta = 0.05$ results in an overall frequency decrease of at most 2.5% for $k \leq 1000$, which provides a reasonable trade-off between sample size and accuracy for our application.

In the following section, we present our workload-adaptive approach which internally uses Equation (1) to calculate sufficient sample sizes for error-bounded top-k analyses. The result is used to adjust the node encodings adaptively in hybrid index structures.

3 ADAPTIVE HYBRID INDEXES

Many DBMS indexes are designed to equally support all possible access types such as lookups, updates, inserts, or deletes, by using *universal encodings*. An example of a universal encoding can be found in traditional B+-tree implementations (e.g. Postgres [48] and Umbra [40]): all leaf nodes use the same encoding where a fixed number of key and value slots are pre-allocated to allow efficient reads *and* writes. While such encodings simplify the implementation, on the one hand, they often result in space overhead. In some cases, this overhead might also result in indexes larger than main memory, leading to significant performance degradations due to paging. Therefore, it is attractive to minimize the space overhead of an index to allow pure memory residency.

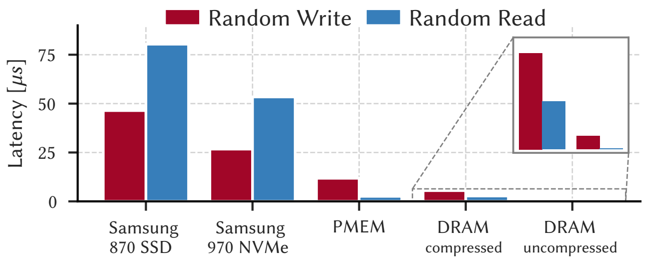


Figure 3: Read and write latencies to LZ4-compressed and uncompressed B+-tree leaf nodes having an average occupancy of 70% and being stored on different storage devices. The experiments were carried out on an Intel Xeon 6212U CPU (24 cores) equipped with 192GB DRAM and 768GB Intel Optane persistent memory. Before each leaf node access, we drop the caches to get IO-related latencies more accurately.

If data does not fit in memory, buffer managers or more lightweight approaches such as LeanStore [30] will manage the page replacements. Despite recent advances in SSDs and NVMe-devices, I/O operations are still multiple orders of magnitude slower than memory accesses [52]. In Figure 3, we experimentally compare lookup and insert operations on uncompressed and compressed B+-tree leaf nodes. With the leaf nodes having an average occupancy of 70%, LZ4-based compression allowed to reduce the storage overhead by up to 47%. On-the-fly (de-)compression of in-memory nodes is faster by multiple orders of magnitude compared to (de-)compression of disk-resident nodes but much slower compared to uncompressed in-memory nodes. Therefore, applying more compact or even compressed encodings to *rarely accessed parts* might improve the overall latency of indexes by reducing storage overhead and preventing expensive I/O operations.

However, the main problem of index structures with different encoding schemes is that the actual workload is *not available beforehand* – instead, indexes get optimized for all possible access types *at development time* by applying universal encodings. Using different encodings, therefore, requires us to get more fine-grained information at *run-time*.

There are two different approaches to collect the required information. In a *decentralized* scheme, we would store tracking information in the index structure itself. For example, we could add an information unit (IU) that contains the last access time, the number of reads, writes, etc. for each index part. Such intrusive changes, however, add space overhead to *all* parts of the index – even to the never-accessed ones. Instead, we propose a new *centralized* approach, which stores IUs for accessed parts only. We combine it with lightweight sampling to reduce tracking-related overhead.

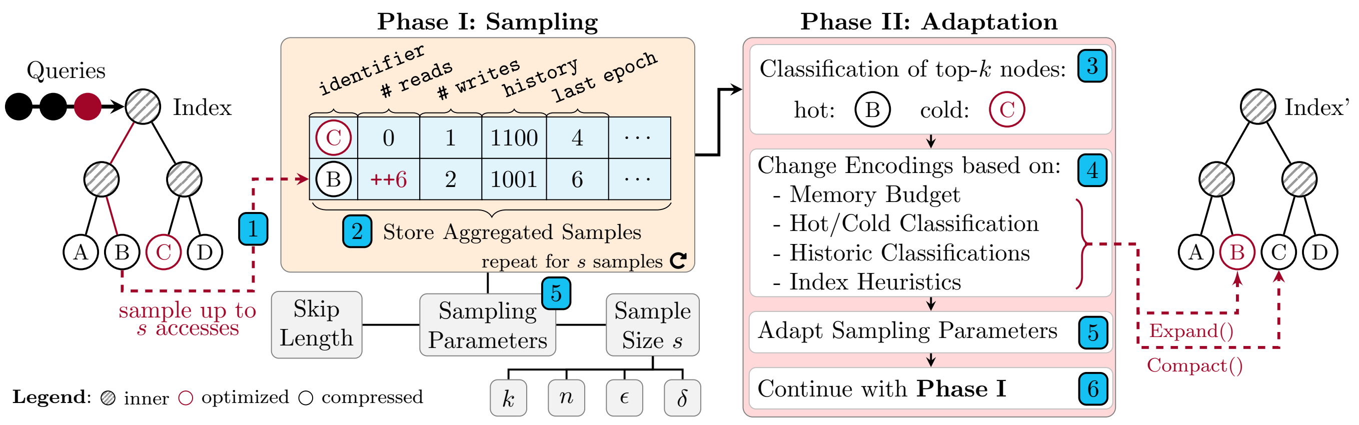


Figure 4: Basic overview of our workload-sampling approach applied to the example of a tree-like index structure.

In Figure 4, we show a conceptual overview of our approach applied to a tree having both a performance-optimized and a space-optimized encoding for the leaf nodes which is similar to our proposed Hybrid B+-tree in Section 4.1 (for implementation details, please refer to Section 3.1). To identify an appropriate encoding for each leaf node based on its access frequency, our approach has a sampling and an adaptation phase.

Phase I – Sampling: As the index processes incoming queries, it traverses the internal structure from top to bottom. Based on a predefined, adaptive skip length sk, every skth leaf node access gets sampled $\boxed{1}$. For each sample, the leaf node and the access type (cf. Figure 4 *read* access to node Ⓑ) get passed to the *adaptation manager* and stored in aggregated form $\boxed{2}$. To consider the most recent accesses of the *current* sampling phase only, we enumerate the sampling phases by increasing epoch numbers and let aggregates store the epoch in which they were accessed last. In Section 3.1.3, we discuss the usage of epochs in more detail. When the required sample size s has been reached, the approach continues to the *adaptation phase*.

Phase II – Adaptation: Based on the aggregations, all samples get classified as either *hot* or *cold* $\boxed{3}$. As an abbreviation, let $\mathcal{N}_e$ denote the set of nodes accessed during epoch e. Therefore, we run a top-k classification on all nodes in $\mathcal{N}_c$ with c being the current epoch, while nodes that were not sampled during c are considered to be cold. In this context, we set k to the number of *theoretically expandable nodes* based on the index size and the memory budget. Affording n additional bytes, we can keep up to $8n$ of the last classification results as historic information to support future encoding decisions.

Next, the adaptation manager determines *promising* encoding changes based on the available memory, the current and the historic classifications, and heuristics $\boxed{4}$. These heuristics are index-dependent as they take encoding migration costs and performance gains into account (cf. Section 3.1.4).

For example, as in Figure 4, node Ⓑ, which has been classified as *hot*, gets expanded from the compressed to the performance-optimized encoding, whereas node Ⓒ, which is no longer hot, gets compacted the other way around.

Sample-based Classification: As tracking all of the internal node accesses will cause severe performance overhead (cf. Section 3.1.3), the adaptation manager considers a sampled subset of queries only. In this context, we introduce two relevant sampling parameters: the *skip length* and the *sample size*. The skip length corresponds to the number of skipped queries between two samples, while the sample size defines the number of sampled accesses before starting Phase II. Based on the skip length sk, the costs for sampling one access get amortized over sk queries: larger skip lengths will reduce the costs per query and vice versa. However, larger skips will also increase the time until optimizing frequently accessed nodes. The adaptation manager sets the skip length adaptively at run-time: frequent encoding migrations will lead to *smaller* skip lengths so that the hybrid index can quickly adapt to workload changes. For more details, please refer to Section 3.1.3.

To bound sampling errors, we introduced Equation (1) in Section 2 to get the sample sizes for error-bounded top-k approximations, with n being the number of leaf nodes, ϵ denoting the classification error, and δ representing its reliability. Based on the results in Figure 2, we set $\epsilon = \delta = 5\%$ as the default values because they provide a reasonable trade-off between sample size and precision. To set k, we approximate the number of nodes that could be expanded without exceeding the memory budget. Assuming a tree has n_c compressed and n_u uncompressed nodes, where compressed/uncompressed nodes use m_c/m_u bytes each on average. For a memory budget mb, we can approximate $k = (mb - (n_c \cdot m_c + n_u \cdot m_u))/(m_u - m_c)$.

At the end of each adaptation phase, the adaptation manager changes both parameters skip length and sample size adaptively $\boxed{5}$.

We next present the architecture, the unified interface, and the different strategies for sampling-based classifications in detail. We then discuss supported user-defined parameters and show how to use heuristics to determine nodes for potential encoding migrations.

3.1 Architecture and Interface

While Figure 4 shows the concepts of adaptive hybrid indexes, this section describes the concrete steps to implement them. We present the framework's architecture and its unified interface in more detail. In the rest of this paper, we refer to the controlling instance of the workload adaptation framework as *adaptation manager*.

3.1.1 Tracking Granularity and Encodings. Before making a hybrid index adaptive, we must determine the *tracking granularity*: the basic unit (e.g., key, node, bucket) where we collect statistics and apply encoding migrations. We then design different encodings for this basic unit. Each encoding comes with different trade-offs in read/write performance and space efficiency. Based on the different encoding characteristics, we implement heuristics that map a basic unit to an encoding by taking its sampled access information and the available resources into account. We describe the usage of such heuristics in more detail in Section 3.1.4.

We must also provide unique *identifiers* for the basic units. In most cases (e.g., B-tree nodes) we simply use pointers. For others such as succinct index representations, we use position offsets.

In our approach, each hybrid index keeps its adaptation manager as a member variable (cf. Listing 1, line 50). This instance is then used for all workload adaptation-related tasks such as monitoring the space consumption and storing the fine-grained access information, which separates index- from sampling-related code.

3.1.2 Interface. As a next step, we identify those functions which access or modify the index at the predefined granularity, such as lookups, inserts, or iterator increments and dereferencing operators. Each of the relevant functions first checks whether the current access is considered to be a sample (Listing 1, lines 38 and 42). Depending on the return value we conditionally pass the accessed part (here `cur_node`) alongside the access type to the adaptation manager using the `Track`-function (lines 39 and 43).

To enable the adaptation manager to change the encoding of a tracked part, we implement a callback function that handles the migration logic between different encodings (line 49). As an example, for a B+-tree with compressed and uncompressed leaf nodes, we provide a callback function implementing the compression and decompression of leaf nodes.

3.1.3 Phase I: Sampling Phase. During the sampling phase, the hybrid index invokes the adaptation manager to track a sampled subset of basic units and corresponding access types. For each unit, the adaptation manager maintains individual access statistics (cf. Listing 1, line 6), where accesses are grouped by access type (read, insert, update, and delete).

To consider only the sampled accesses of the *current* phase, the adaptation manager maintains a global epoch counter (cf. line 28), and each access statistic stores the epoch of last access (cf. line 6). Before updating access statistics, we first check if its epoch matches the global epoch. In the case of different epochs, we first reset the aggregate counters and set the local to the global epoch before registering new accesses. Storing the epoch of the last access further adds new relevant information when deciding which node encodings should be changed.

To efficiently map identifiers to their access statistics (cf. line 25), we use a high-performance hop-scotch hash map for single-threaded execution [6], and a concurrent cuckoo-based hash map for parallel workloads (cf. Listing 1, line 25) [34]. It allows concurrent readers and writers while it retains high throughput under contention.

As an optimization, we install a *bloom filter* in front of the hash table to prevent cold nodes from being tracked accidentally (cf. line 26). Before an identifier gets tracked in the hash map, it must be added to the filter first. Only in the case the identifier is already

```
1  // AdaptationManager.hpp
2  template <class Index, typename Identifier, typename Context>
3  class AdaptationManager {
4   public:
5    enum AccessType { READ, WRITE, UPDATE, DELETE };
6    enum AccessStats { size_t reads, size_t writes, BitSet
   ↪  last_classifications, Epoch last_epoch, ... };
7    explicit AdaptationManager(Index *index);
8    bool IsSample() {
9      if (--skip_length == 0) {
10         skip_length = global_skip_length_.load(); // synchronized
11         return true;
12     }
13     return false;
14   };
15   template <typename... Args>
16   void Track(Identifier&, AccessType&, Args&& ...);
17   void UpdateContext(Identifier&, Context&);
18  private:
19   void Classify(); // Classify nodes as hot and cold
20   void Adapt(); // Start encoding migrations
21   atomic<size_t> global_skip_length_; // Adaptive parameter
22   static thread_local size_t skip_length;
23   atomic<size_t> global_sample_size_; // Adaptive parameter
24   static thread_local size_t sample_size;
25   HashMap<Identifier, pair<AccessStats, Context>> samples_;
26   BloomFilter<Identifier> filter_;
27   Index *index_;
28   Epoch current_epoch_;
29 };
30 // HybridIndex.hpp
31 #include "AdaptationManager.hpp"
32 template <typename K, typename V>
33 class HybridIndex {
34   struct Node {...}
35   friend class AdaptationManager;
36  public:
37   V Lookup(const K& k) { // leave out lookup logic
38     if (adapt_manager_.IsSample())
39         adapt_manager_.Track(node, READ)
40   }
41   bool Insert(K& k, V& v) { // leave out insert logic
42     if (adapt_manager_.IsSample())
43         adapt_manager_.Track(node, INSERT)
44   }
45  private:
46   // Callback functions invoked by adapt_manager_
47   size_t GetUsedMemory();
48   Encoding EvaluateHeuristic(const AccessStats&);
49   void Encode(Node*, EncodingSchema& /*target*/, Node*
   ↪  /*parent*/);
50   AdaptationManager<HybridIndex<K,V>, Node*, /*Parent-*/ Node*>
   ↪  adapt_manager_;
51 };
```

Listing 1: Simplified draft of the workload sampling interface. We left out constructors, index-dependent lookup- and insert-function logic, and thread-local sampling maps.

contained in the filter, it gets added to the hash map. We reset the filter after each sampling phase. The bloom filter is configured to use 10 bits per item and its capacity is set to half of the sample size.

To reduce the tracking-related overhead, (e.g. hashing the identifiers, accessing and modifying the aggregated sampling statistics) we do not consider *all* node accesses, but a sampled subset only. As our approach aims to classify index parts into categories such as hot and cold, the chance to miss frequently accessed parts decreases inversely proportional with increasing access frequencies.

While sampling reduces the *tracking-related* overhead, it introduces *new* overhead to decide which accesses get sampled. Instead

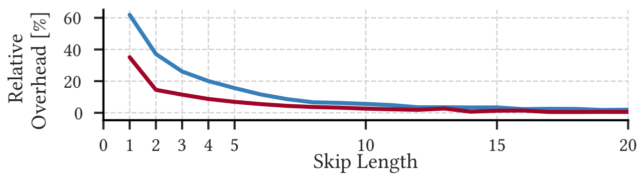


Figure 5: Relative sampling overhead for different skip lengths using the Hybrid B+-tree (cf. Section 4.1). The well-known STX-B+-tree represents the baseline. The blue line additionally samples leaf nodes accesses and collects individual tracking information. It shows the relative overhead of the workload sampling. The red line shows the performance overhead for the filter-based optimization. Further experiments can be found in the online appendix.

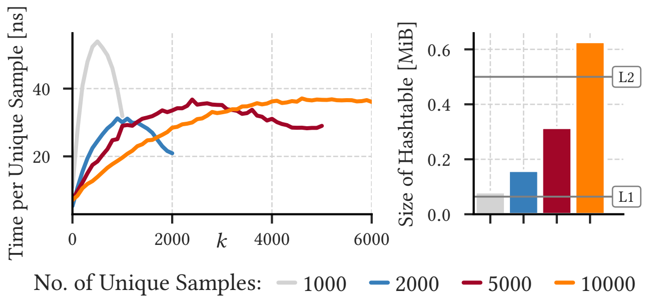


Figure 6: The left plot shows the classification overhead per sample for different sample sizes and different values for k. In the right subplot, we show the size of the hash map that stores the individual samples and the access statistics.

of probabilistically deciding for each access whether it gets sampled or not, Vitter et al. suggest minimizing sampling-related overhead by defining so-called *skip lengths*: a skip length defines how many accesses are skipped between two samples [49]. We experimentally evaluated the sampling overhead for different skip lengths using the OSM dataset and a log-normal distributed workload in Figure 5. The STX-B+-tree is the baseline in this experiment, and the tree represented by the blue line additionally collects access statistics. For a skip length of 0, which means that all accesses get sampled, we observe significant overhead of up to 61.9% compared to the baseline. However, the sampling overhead quickly decrease for larger skip lengths, e.g. 1.6% for a skip length of 20. The red line shows the overhead after adding the additional bloom filter. The filter prevents cold nodes from being tracked and reduces sampling-related overhead significantly. While this experiment shows results for the log-normal workload, other workloads show similar overhead.

We can further reduce contention by defining *one skip per thread* (line 24): decrementing the thread-local skip (line 9) does not require synchronization. Only in case the skip becomes zero, we reset the skip to the global skip using an atomic load instruction (line 10).

In some cases we need a way to store additional *context information* alongside the identifier to allow for efficient encoding migrations. For example, we could integrate the adaptation manager to identify hot and cold leaf nodes in a B+-tree. Whenever we expand or compact a node, its parent must efficiently be made available to change the corresponding child-node pointer. Based on *variadic template arguments* and *perfect forwarding*, hybrid indexes can efficiently pass arbitrary context information to the adaptation manager without requiring changes of our framework.

As context information might change over time (e.g. parent changes because of node splits), our framework allows to propagate context changes to the adaptation manager (cf. line 17).

3.1.4 Phase II: Adaptation Phase. When the sample reaches the predefined size, the adaptation manager terminates the sampling and passes over to the adaptation phase, which consists of three steps (cf. 3 - 5 in Figure 4). First, the top-k frequent nodes of the last sampling phase get labeled as hot, the rest is considered to be cold. Second, based on the classification and the index heuristic function, the adaptation manager determines the most suitable encoding for each tracked node and applies the appropriate encoding migrations. Last, the adaptation manager adapts the parameters skip length and sample size before the next sampling phase starts.

Classification: We implement the top-k analysis using a priority queue based on a binary heap having a capacity of k. In our experiment, we use the sum of the read and write access counters as default priority. However, we could also assign custom weights to the different access counters. Then, we traverse the hash map and insert those aggregated samples whose epoch matches the global epoch, and label them as hot. Nodes, which were not accessed during the last sampling phase will not be inserted into the priority queue at all – instead, we can directly classify them as cold. When nodes are displaced from the priority queue, they are marked cold again. Therefore, we find the top-k frequent items in a single pass and classify all nodes accordingly. This algorithm runs in $O(u(1 + log(k)))$ with u being the number of unique samples, and the space to store the priority queue is $O(k)$.

We experimentally evaluated the classification performance for different numbers of unique samples and different values for k in Figure 6. For $k \approx s/2$, the sum of heap inserts and removals reaches its maximum, while it decreases for smaller and larger k, explaining the different latencies. Assuming a classification latency of 60ns per sample and a skip length of $sk = 20$, the classification overhead per query can be amortized to 60ns/20 = 3ns.

Identifying hot nodes based on sampling will also introduce inaccuracies, therefore, we further back up future encoding decisions by keeping the most recent n classifications. In our example implementation, we use one additional byte to keep the last eight classifications. This information can be used in the heuristics to further improve encoding decisions.

Heuristics: After the classification, the adaptation manager might optionally change the encoding for tracked parts. Making *optimal* encoding-decisions is not possible, as there will be sampling inaccuracies and future queries and accesses are not known beforehand. However, we can *react* to current workload developments based on the sampled access statistics. Therefore, *context-sensitive heuristic functions* (CSHF) support the workload manager to decide which encoding migrations *might* improve the performance. As shown in Figure 7, a CSHF is similar to a decision tree that takes sampled access statistics and other context information into account and returns an encoding. Branches represent *decisions*, while leaf nodes contain the *suggested encoding*. Furthermore, the CSHF can decide to stop tracking of specific nodes, e.g. if they are cold or were not sampled for a longer time. Each hybrid index will implement its own, tailored CSHF as it must take the different encodings and space-performance implications into account (cf. line 48).

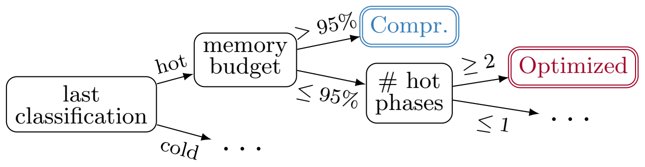


Figure 7: Heuristic functions propose target encodings depending on a variety of factors such as the current and the last classifications, system resources, or the last access.

Next, the adaptation manager evaluates the CSHF for all tracked items and if required, migrates the basic units to their suggested encoding. Therefore, the hybrid index implements a callback function (cf. line 49) to handle the migration between different encodings.

Sampling Parameters: Providing an equation or a metric that adequately considers all parameters is challenging. Instead, we identified three important parameters *sample size*, *skip length*, and *encoding migration costs* and provided experimental validation for each. Based on these experiments, we let the adaptation manager set the parameters adaptively at runtime. E.g., the adaptation manager will calculate the new *sample size* based on Equation (1) as well as a new *skip length*. As we discussed earlier, a smaller skip length allows hybrid indexes to adapt more quickly to workload changes, but it imposes more sampling overhead. And we use the number of node encoding changes in the current adaptation phase to approximate workload stability. For example, if the migrated nodes make up less than 10% of the sampled accesses, the skip length will increase to reduce the sampling overhead. Contrary, if the share exceeds 30%, we decrease the skip length and therefore increase the sampling frequency. In our example implementation, the adaptation manager will adaptively set the skip length within the range [50, 500]. Additionally, the adaptation manager could randomize *sk* in a limited range to cope with query patterns.

3.1.5 Concurrency. While hybrid indexes work best under skewed workloads, concurrency requires contention and synchronization to be kept at a minimum. We compare and evaluate two approaches: (1) **GS**: All worker threads (WT) track samples in a global cuckoo hash map which is optimized for concurrent readers and writers [34]. During adaptation, the map gets locked globally to process each sample. (2) **TLS**: All WTs track the samples in thread-local maps, which get merged once the target sample size is reached. In both approaches, one WT runs the adaptation, while the remaining WTs continue with the sampling phase. While GS optimizes space efficiency, TLS allocates more memory for thread-local sampling.

3.1.6 Optional Memory Budget. Our framework allows to set either an *absolute* or a *relative* memory budget. While absolute budgets are suited for read-only workloads, relative budgets allow us to define an average ratio of bits per item and therefore provide more flexibility with inserts and deletes. In other words, a memory budget

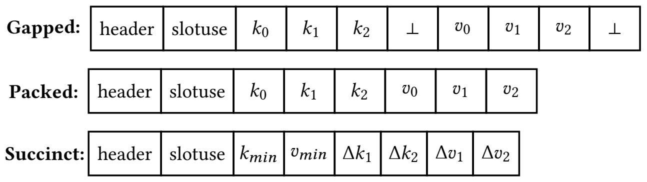


Figure 8: Three different leaf node encodings are used in our Hybrid B+-tree implementation. The Gapped encoding stores a fixed number of slots with possible empty slots (⊥) at the end. The Packed encoding stores keys and values densely packed, and the Succinct layout further employs frame-of-reference encoding.

lets us define a compression ratio in which the index structure can be adaptively optimized. During execution, the adaptation manager optimizes the index while keeping its size below the upper bound.

3.2 Trained Hybrid Indexes

In some contexts, dataset and workload remain stable for a longer time, or a workload prediction might be available beforehand. E.g., self-driving database systems as proposed by Pavlo et al. in 2017 [41] build indexes based on predicted workload patterns. Therefore, more space-efficient hybrid indexes could make use of such fine-grained predictions for training. Furthermore, *online* workload adaptation imposes additional overhead to collect, aggregate, and classify samples, so it might be desirable to train hybrid indexes based on previous workloads in these cases beforehand.

Therefore, our framework also implements an offline solution for hybrid indexes: given a predicted or a historic workload, the adaptation manager analyzes the access patterns and ranks the nodes according to their access frequencies. Starting with the most promising nodes, the adaptation manager optimizes the nodes until all nodes are optimized or the memory budget is reached.

4 EXAMPLE IMPLEMENTATIONS

In this section, we apply our approach to two state-of-the-art index structures: B+trees and radix trees.

4.1 Hybrid B+-Tree

Although invented for disk-based database systems, B+-trees are still the most widely-used indexes, even for in-memory DBMSs [53]. Most implementations make use of two encoding schemes: one for inner and another one for leaf nodes. While both node types have a fixed number of slots, long-running systems with millions of insert and delete queries lead to unused slots. While these empty slots do allow for efficient inserts and deletes, they also often result in space utilization below 70% [8].

4.1.1 Tracking Granularity and Encodings. We first determine a suitable tracking granularity. For the B+-tree, *leaf nodes* make a good candidate as they make up the largest part of the overall data structure and maintain all keys and values. Figure 8 introduces three leaf node encodings that trade-off space and performance differently.

Gapped is the traditional, universal encoding for B+-tree leaf nodes. Those nodes support all access types efficiently by accepting free slots (⊥), but they require a fixed amount of memory. The more space-efficient *Packed* layout allocates memory for the *used* slots only. This *packed* representation supports efficient read, update, and delete operations (using tombstones), but does not support efficient inserts. It stores the number of elements and two arrays for keys and values. As an alternative, the Succinct encoding trades performance for space efficiency more aggressively. For a leaf node, it combines frame-of-reference (FOR) encoding with bit packing to store the keys and values in a compressed fashion. The first (and smallest) key/value is stored separately. While this node layout still allows for random access, it requires additional instructions and bitwise operations to access keys and values. This results in higher access latencies compared to the gapped and packed encoding schemes for smaller indexes, however, for larger indexes exceeding the caches, the succinct layout will cause fewer cache misses which *might* outweigh the additional CPU costs. In Table 1, we provide a performance analysis for the different layouts. Gapped and packed nodes achieve significantly higher throughput, whereas succinct nodes require 73% less space on average compared to gapped nodes. Furthermore, special allocators and memory pools can optimize the allocation of differently sized nodes to prevent heap fragmentation.

4.1.2 Encoding Migrations. In Figure 9, we experimentally evaluate the migration costs between the different node encodings for two index sizes. The left index consumes around 10MB and entirely fits into the L3 cache, while the right index needs around 1GB. For both indexes, we observe significant overhead for switching between the *succinct* and one of the other encodings: in these cases, the migration will modify the physical key and value representation and comes at the cost of additional instructions. In contrast, migrating between packed and gapped node layouts is cheaper as these migrations use a system call to copy keys and values.

4.1.3 Tracking Leaf Nodes. Tracking leaf nodes during reads and inserts is straightforward as the leaf and its context is directly available. Scans, however, require minor, structural changes: each inner node has a link to its right sibling, and iterators keep a pointer to the current parent. In the case of a sampled leaf node access through an iterator, the parent can be efficiently retrieved.

4.1.4 Handling Updates. Inserts and deletes will cause split and merged nodes. In case a leaf node gets a new parent, this information must be propagated to the tracking framework: we pass the leaf node and its new parent to the adaptation manager, that will update the context information of the *actually tracked* leaf nodes only.

Table 1: Different leaf node encodings storing 64-bit key-value pairs of the OSM dataset (cf. Section 5.1) and their performance implications on uniform lookups for a node occupancy of 70%.

Leaf Node Encoding	Average Size	Lookup Latency	Instruc.	LLC Misses	Branch Misses
Gapped	4096B	56ns	85	2.1	4.44
Packed	2872B	57ns	84	1.4	4.46
Succinct	1076B	125ns	341	1.1	6.69

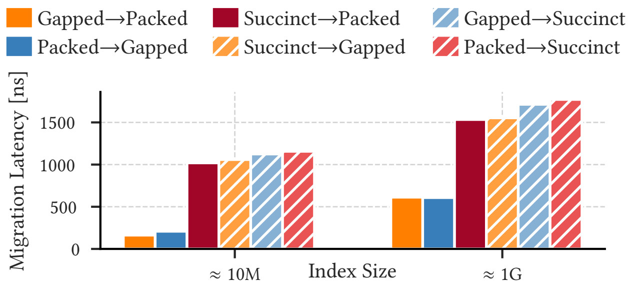


Figure 9: Evaluation of the migration costs between different leaf node encodings for two selected index sizes. The used CPU is equipped with an L3 cache size of 64MB.

4.1.5 Concurrency Control. We synchronize Hybrid B+-tree using Optimistic Lock Coupling (OLC) as described in [32]. Each node stores a lock and an atomic version counter. Compared to lock coupling, OLC scales significantly better on multi-core systems, because it minimizes the number of acquired locks.

4.2 Hybrid Trie: ART and FST

Tries are pointer-based index structures and are mainly used to index variable-length keys. Especially on modern hardware, research has shown that tries achieve high performance [10, 13, 31, 35, 50, 55]. Compared to B-trees, tries do not store entire keys on each level, but they index key suffixes, also referred to as *labels*, instead, where each level stores the next $k > 0$ bits of the key.

The Adaptive Radix Tree (ART) was introduced in 2013 and represents the default index structure in HyPer [31]. It allows to dynamically choose between four differently sized node types based on the number of labels stored within a node. While each node type has different implications on lookup performance [19], the node type is chosen based on the indexed keys only and does not depend on the actual workload. ART requires k to be 8 and therefore limits the maximum fanout to $2^8 = 256$.

Besides ART, there is another state-of-the-art trie called Fast Succinct Trie (FST) which has been introduced by Zhang et al. in 2018 [55]. Compared to ART, FST does not store child pointers to traverse the index structure, but instead, it *computes* the position of the next node based on two bitmaps, one storing the existing labels and another one maintaining the information whether a path terminates. While FST does not impose any restrictions on k, we assume $k = 8$ in the following for the sake of simplicity. Furthermore, FST uses two different encoding schemes for upper and lower levels: The upper, more frequently accessed parts are encoded using a more performance-optimized and space-demanding encoding, referred to as *FST-dense*, where each node stores the key-labels implicitly by using 2^k bits per node, which allows for fast random access within each node. The lower levels use the *FST-sparse* encoding which stores existing *labels* explicitly. This *might* reduce the space usage[1] but also requires an explicit search within the nodes and therefore more computations for sparse-encoded nodes.

[1]The sparse encoding requires less space compared to the dense encoding when the average number of stored labels l within the nodes is smaller than $256/8 = 32$.

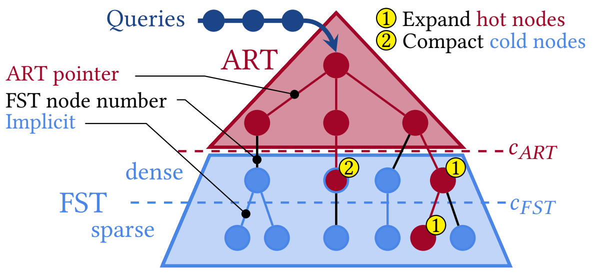


Figure 10: Our workload-adaptive Hybrid Trie. It combines the Adaptive Radix Tree and the Fast Succinct Trie level-wise at build-time. Combined with our sampling framework, it supports branch-wise refinements at run-time.

In Table 2, we compare the sizes of ART and the two FST encodings for the prefix-random dataset (cf. Section 5.1). While ART allows for faster lookups, it requires significantly more memory. FST requires additional instructions to compute the position of the next node and value resulting in decreased performance. For FST-sparse, we measured more cache misses than for ART, which can be explained with the efficient path compression in ART: while FST-sparse requires less memory, its lookups will traverse more nodes on average and therefore cause more cache misses.

Based on the work of Anneser et al. in 2020 [9], we introduce Hybrid Trie: an adaptive, level-wise combination of ART and FST.

4.2.1 Tracking Granularity and Encodings. Hybrid Trie combines ART and FST level-wise: levels 0 to c_{ART} (incl.) are represented by ART, levels between c_{ART} and c_{FST} (incl.) are represented by FST-dense, and the remaining levels are encoded using FST-sparse, as illustrated in Figure 10. We chose the level-wise combination with ART being at the top based on the fact that all queries start at the root node and ART achieves significantly higher throughput.

To allow for more fine-grained control and branch-wise expansions *beyond the cutoff level* c_{ART}, we extend Hybrid Trie with vertical refinements. While ART uses pointer tagging to differentiate *pointers* and inlined *TIDs*, we use an extra bit to further differentiate the case of inlined FST node numbers. The tagged pointers can then be used as unique identifiers by the adaptation manager.

4.2.2 Interface and Callbacks. Since FST is a static index supporting only lookups and range scans, we do *not* handle inserts as they would require a complete *rebuild* of FST each time. To support efficient inserts, we experimented with storing multiple FSTs (one per "cold" subtree) instead of a single, global one. However, as each FST adds some storage overhead (for header information and auxiliary data structures), this approach did not pay off. We hence leave inserts for future work.

Next, we identify the functions accessing nodes below c_{ART}. In Listing 2, we show how the simplified lookup code integrates with the sampling framework. To enable fast node migrations, we provide additional context for each tracked identifier: we store its parent, the key label within the parent, and the FST node number.

The callback function `Encode(...)` implements the migration logic between ART and FST nodes. Compacting ART nodes to the FST representations (cf. ② in Figure 10) requires deleting the expanded node and replacing the tagged identifier within the parent

```
1   V Lookup(const K& key) {
2     const bool isSample = adapt_manager_->IsSample();
3     Node* node = root_;
4     while (node != nullptr && isARTPointer(node)) {
5       node = findChild(node, key[level++]);
6       if (isSample && level > c_art)
7         adapt_manager_->Track(node, READ, ...);
8     }
9     if (isFSTNode(node))
10      return fst_->Lookup(getFSTNode(node), key, level);
11    return getValue(node);
12  }
```

Listing 2: Simplified lookup code for Hybrid Trie. The highlighted lines handle the required calls to the workload sampling framework. In line 7, we dropped additional arguments such as the parent identifier and the key-part at the current level. This function is intentionally not declared `const` as it may modify the internal structure.

Table 2: Space and performance metrics for different trie indexes measured for the prefix-random dataset and workload (cf. Section 5.1).

Index	Size	Per Lookup-Query			
		Latency	Instruc.	LLC misses	Branch misses
ART	274MB	81ns	177	8.49	0.03
FST-dense	116MB	206ns	675	6.33	1.82
FST-sparse	104MB	576ns	4337	9.2	9.64

node with the FST node number. Expanding FST nodes to ART nodes (cf. ①) requires us to determine the appropriate ART node type based on the number of labels within the node. In both cases, we retain the historic access statistics in the workload tracking.

We experimentally evaluated the latencies for migrating nodes between ART and FST. Migrations from FST to ART cause overhead of up to ≈5000ns on average (assuming a node occupancy of 50%): labels stored within the FST node must first be collected and then inserted into the new ART node. Migrating the other way around takes up to ≈100ns only, as it does not involve the construction of a new node, but the deletion of the existing ART node and its replacement in the parent node with the FST node number.

5 EVALUATION

We conduct all experiments on a 16-core AMD Ryzen 9 3950X CPU @ 3.5GHz equipped with 64GB DDR4-2667 RAM and compile the C++ code with GCC 9.3.0, using the flags `O3` and `march=native`. Please note that the CPU overhead for sampling, compacting, and expanding nodes are already included in the shown performance.

5.1 Datasets and Workloads

The *OSM*-dataset [26, 36] comprises 400M uniformly sampled Open Street Map locations represented as 64-bit S2-cell-identifiers [4]. We further use the RocksDB tool `dbbench` to generate 64-bit user-ids

Table 3: Operation counts and distribution types for each workload. The synthetic workloads W1.1-W2 were generated by us, W3 is a realistic workload generated using RocksDB's dbbench [14], and W4 has been generated using YCSB [17]. The scan length is uniformly distributed within [10, 50], and for W4 [100, 250].

	Reads	Scans	Inserts
W1.1	49% Zipfian	49% Zipfian	2% Zipfian
W1.2	49% Normal	49% Normal	2% Zipfian
W1.3	49% Lognormal	49% Lognormal	2% Lognormal
W2	56% Lognormal 24% Uniform	20% Lognormal	
W3 [14]	100% prefix-rand.		
W4 YCSB	75% Zipfian	25% Zipfian	
W5.1		20% Zipfian	80% Zipfian
W5.2	20% Zipfian	80% Zipfian	
W6.1	100% Zipfian		
W6.2		100% Zipfian	

and anonymized workloads that contain common patterns seen at Facebook [3, 14]. Besides fixed-size keys, we use a dataset of 33M unique email addresses (host-reversed, e.g. foo.com@) drawn from a real-world dataset (average length = 22 bytes, max length = 49 bytes). Further optimizations such as key compression are orthogonal to our approach: Adaptive indexes choose the most promising internal node encodings adaptively at run-time, while key compression, e.g. [56], is performed at key granularity.

Based on the datasets, we generate different workloads. Figure 11 visualizes the cumulative distribution functions (CDF) of the workloads W1.1 - W1.3 applied to the OSM dataset.

In Table 3, we show the number of operations and the used distribution for each workload and query type. We use the following *relative* distributions to decide on *record selections* and *scan lengths*: Zipf with $\alpha \in [1, 1.5]$ and N being the number of keys, Normal with μ=0.5 and σ=0.03, Lognormal with μ=0 and σ=0.1, and Uniform.

Additionally, we used RocksDB's dbbench to generate a more realistic workload based on the *prefix-random* configuration described by Cao et al. in 2020 [14]. They analyzed RocksDB workloads at Facebook and extracted common characteristics, which are re-generated by dbbench. Furthermore, they found a correlation between key prefixes and lookup frequencies: while most keys are not accessed at all, there are some *hot* key prefix ranges which are

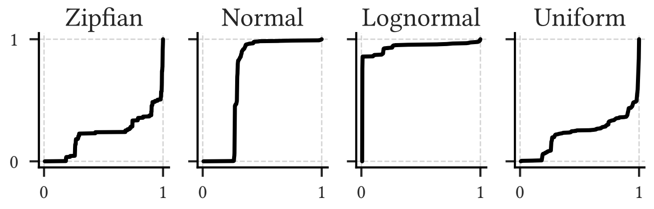


Figure 11: Cumulative distribution functions (CDFs) of the workloads (from left to right) W1.1, W1.2, W1.3, and a uniform distribution on the OSM dataset.

accessed frequently. We evaluate this workload using ART and FST. We use a custom read-only YCSB configuration with a hot set size of 1% of the dataset (cf. W4).

We further use the Yahoo! Cloud Service Benchmark (YCSB) [17] to generate a dataset of 200M key-value pairs (16 bytes each) and workload W4 with 200M Zipfian-distributed queries. Workloads W5.1 and W5.2 let us investigate the performance of adaptive indexes for write-dominated workloads, while workload W6 evaluates point lookups and scans on the mail dataset using Hybrid Trie.

5.2 Hybrid B+-tree

For the following evaluation of the Hybrid B+-tree, we assume an average leaf node occupancy of 70%. We refer to the adaptive Hybrid B+-tree as AHI-BTree.

In Figure 12, we use the OSM dataset and the workloads W1.1 - W1.3 to show the performance *developing over time* and the average space consumption for the adaptive and pre-trained Hybrid B+-tree and compare them to the *succinct, packed,* and *gapped* tree variants which do apply a single encoding to all of their leaf nodes. For each workload phase, we observe a short period of time in which the latencies of the adaptive index decrease, and then stabilize at a lower level. During this time, the workload adaptation detects frequently accessed nodes and migrates them to performance-optimized encodings. In contrast to the first and last phases, the second phase W1.2 is less skewed, which is the reason for the increased latencies. The adaptive tree achieves 85%, 99%, and 84% of the throughput of the performance-optimized Gapped tree on average per workload phase. At the same time, the adaptive tree reduces the memory footprint (2.36GB) by up to 72% compared to the Gapped tree (8.66GB).

To better understand the space-performance trade-off, we use the cost function $C = P^r S$ defined by Zhang et al. in 2018 [55], with P representing the performance (latency) and S representing the index size. The exponent r defines the relative importance between P and S: $0 \leq r < 1$ considers space to be more important, while $r > 1$ trades performance for space.

Figure 13 visualizes C for $r = 1$ (space and performance are equally important) by using blue curves and shows the *average* performance and *last measured* index size. Indexes on the same curve are considered to be "indifferent" in the space-performance trade-off. According to C, the succinct, adaptive, and pre-trained variants provide a better space-performance trade-off than the gapped and packed variants. For the highly skewed Lognormal workload W1.3, the adaptive tree achieves the *best* trade-off.

In Figure 14, we investigate to what extent the adaptive tree can leverage differently skewed workloads. We generate the workloads based on W1.1 for parameter $\alpha \in (0, 1.6]$. For $\alpha = 1$, our adaptive tree reduces the index size by 71%/59% while it increases query latency by 17%/7% wrt. the Gapped/Packed trees. With decreasing α, AHI-BTree cannot retain the high performance improvements: the access frequency of the 10K most frequent nodes (out of 2M nodes) decreases from 67% for $\alpha = 1$ to 45%/28%/11% for $\alpha = 0.9/0.8/0.7$. For this experiment, the break-even point is at $\alpha \approx 0.6$: for $\alpha < 0.6$, sampling overhead outweighs performance improvements due to node expansions, and for $\alpha > 0.6$, the contrary is the case.
Despite the sampling overhead, we observe no considerable performance decreases for AHI-BTree wrt. the Succinct tree (3% higher

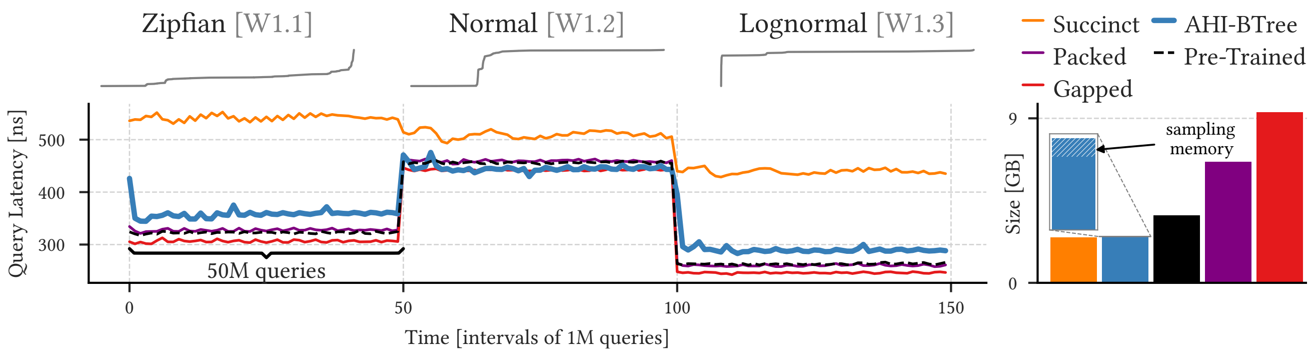


Figure 12: Query latency evolving over time for the OSM-dataset and three selected workloads using the Hybrid B+-tree and its baselines. Each workload phase comprises 50M queries and at the top, we show the corresponding CDF. For all phases, we observe a performance increase over time for the adaptive tree (AHI-BTree). The bar chart to the right shows the overall index structure sizes. The sampling framework takes up to 2.53MB, which is 0.1% of the adaptive index size of 2.36GB.

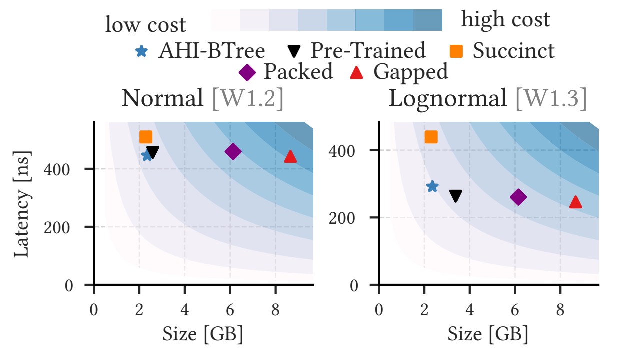


Figure 13: Average latencies and index sizes for B+-trees having different leaf encodings for the OSM dataset and workloads W1.2 and W1.3. The blue curves show a cost function that considers performance and space as equally important. Points on the same curve are considered to be *indifferent*.

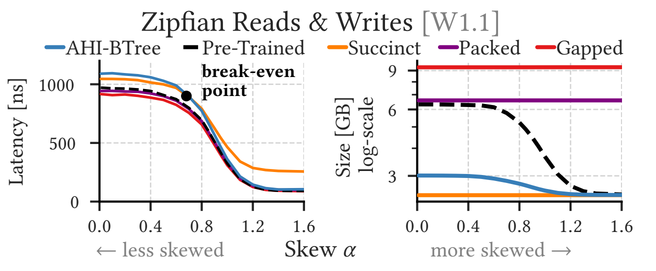


Figure 14: Average latencies and sizes of the Hybrid B+-tree indexing the OSM dataset for workload W1.1 for varying α.

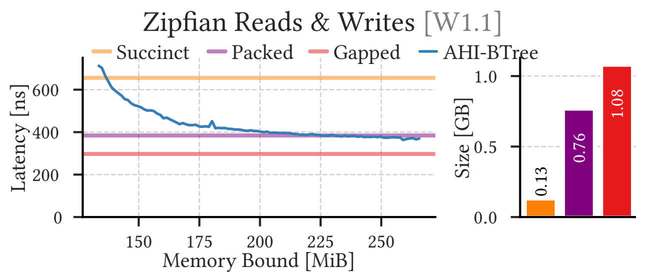


Figure 15: Latency and size of Hybrid B+-tree indexing 50M consecutive 64-bit keys for different memory budgets.

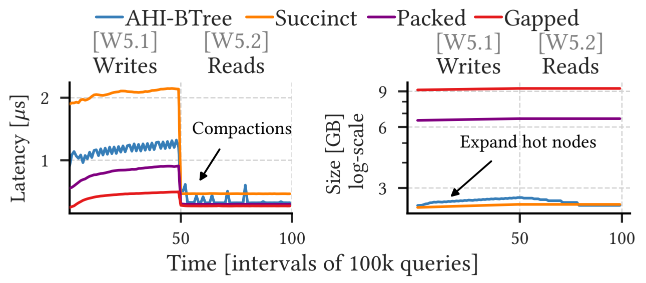


Figure 16: Latencies and index sizes for the Hybrid B+-tree running workload W5 on the OSM dataset. W5.1 is write-dominated and W5.2 is scan-dominated. We run both workloads consecutively.

latency) under less skewed workloads ($\alpha = 0.01$).
AHI-BTree eagerly migrates Succinct nodes to the Gapped encoding on inserts and defers their compaction until they are cold again. For $\alpha = 0.01$, inserts affect 26% of all nodes and the adaptive tree allocates 46% more memory compared to the Succinct tree.

Figure 15 shows the impact of the memory budget on the performance of AHI-BTree. With increasing budgets, AHI-BTree can expand more nodes to the performance-optimized encodings. As the most frequently accessed nodes get optimized first, the performance improvements per additional MB are larger for smaller memory budgets under skewed workloads.

Figure 16 shows the performance for AHI-BTree running workload W5. With Succinct nodes being optimized for read accesses only, insert operations during the write-intensive workload W5.1 require expensive changes to the node structure. While AHI-BTree uses the Succinct encoding as default for cold nodes, it eagerly migrates nodes to the Gapped encoding on inserts. At the beginning

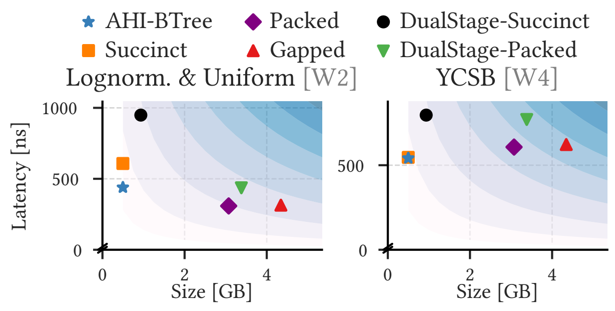


Figure 17: We compare the space and performance of our Hybrid B+-tree to the Dual-Stage Hybrid B+-Tree described in [53]. During the benchmark, the dynamic stage contains the latest inserted keys (5% of all data). The dataset contains 200M consecutive 64-bit keys and 64-bit TIDs.

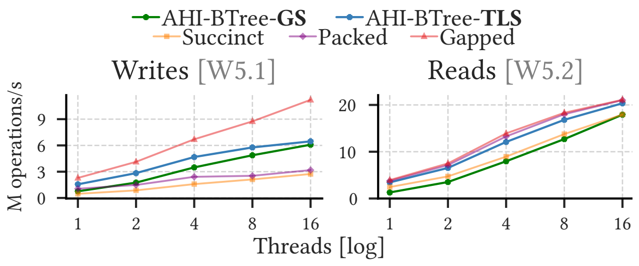


Figure 18: Average throughput for the two concurrent workload adaptations based on Global Sampling (GS) and Thread-Local Sampling (TLS) applied to the Hybrid B+-tree. We run both workloads W5.1 and W5.2 *separately* using different numbers of worker threads.

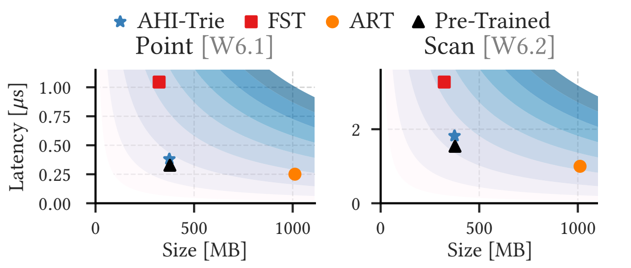


Figure 19: Space and performance for point lookups (W6.1) and scans (W6.2) on 33M unique email addresses.

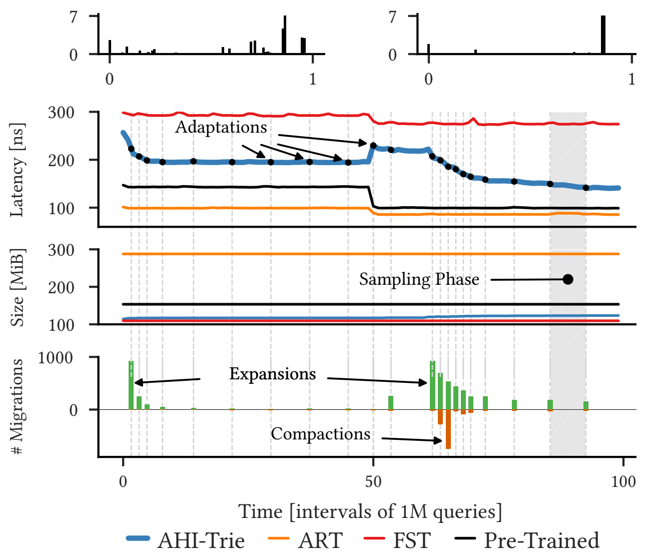


Figure 20: Latencies, index sizes, and encoding migrations shown over time for the prefix-random workload W3 and a dataset of 172M user ids. We split the workload into two phases, each containing different hot prefix ranges visualized by the two histograms at the top. The vertical dashed lines show the time of the adaptations. Sampling phases take place between two subsequent adaptations.

of W5.2, previously expanded nodes, which are rarely accessed in W5.2, get compacted again to reduce the memory footprint.

In Figure 17, we compare our approach to the *Dual-Stage (DS) framework* proposed by Zhang et al. in 2016 [53]. DS consists of a dynamic stage for recently modified data and a static stage for the remaining data. Inserts, deletes, and updates modify the dynamic stage, whereas reads first check the dynamic stage, and if the key was not found, they continue the lookup in the static stage. We can see that our approach outperforms DS in both dimensions, space and performance. Based on the access statistics, it allows for more fine-grained encoding decisions and can therefore leverage skew to a higher extent. Contrary, DS keeps only recently inserted or modified items in performance-optimized structures *independent* of the workload skew. As described in [53], we add the LevelDB [2] bloom filter to DS to further speed up lookups as it allows to *skip* the dynamic stage in most cases when the key does not exist there.

In Figure 18, we compare the performance of the two concurrent adaptation approaches (cf. Section 3.1.5) and apply them to the Hybrid B+-tree to run workloads W5.1 and W5.2 on the OSM dataset. We pin each thread to one logical core. For both workloads, thread-local sampling (TLS) achieves higher throughputs compared to global sampling (GS). GS locks the entire map during the adaptation phases and table resizing operations, which leads to high contention that severely degrades performance while executing the read-only workload W5.2. The skewed inserts in W5.1, however, already incur high contention and the performance gains due to node expansions outweigh the sampling overhead of both approaches. Compared to single-threaded workload adaptation, the shared and the thread-local maps allocate up to 10x more memory (up to 1.5%/2.4% of the index) to reduce sampling-related contention.

5.3 Hybrid Trie

In Figure 19, we consider index size and performance of FST, ART, as well as the trained and the adaptive Hybrid Tries (AHI-Trie) for point lookups and scans on 33M email addresses. While FST stores one character per level, ART nodes inline up to eight *common prefix* characters. This reduces the tree height from 49 to 32 levels and improves performance. For Hybrid Trie, ART stores the upper 9 levels which contain 5.23% of all nodes (≈ 4.28% of the size).

Table 4: Overview of the Lines of Code (LOC) per lookup and insert functions of our two workload adaptive hybrid indexes compared to their non-adaptive counterparts.

	Lookup		Insert	
Index	Logic	Tracking	Logic	Tracking
B+-tree	13	-	100	-
AHI-BTree	15	+1	119	+5
ART/FST	18/46	-	-	-
AHI-Trie (ART/FST)	25/47	+3/+0	-	-

We compare the performance and size of the adaptive and pre-trained Hybrid Trie to ART and FST in Figure 20 for the prefix-random workload W3 and a dataset of 172M 64-bit user ids. We split the workload into two phases and assigned the different prefix ranges (defined by the 44 most significant key bits) randomly to one of the phases. The sampling and adaptation phases for the adaptive trie are highlighted: black dots indicate adaptation phases, while sampling phases take place between two adaptations. The duration of a sampling phase is the product of skip length and sample size. As the sample size does not significantly change in this example (not visualized), the sampling phase duration is mainly determined by the changing skip lengths.

At the beginning of each phase, we observe an increased number of encoding migrations. For phase 1, there are expansions only, as all nodes below c_{ART} are stored in FST. However, during phase 2, nodes frequently accessed in phase 1, are considered to be cold now, and, after a short delay, get compacted again.

After the workload manager identified and expanded/compacted the hot/cold nodes, it increases the skip length to lower sampling-related overhead (cf. Section 3.1.3). This increase results in a larger distance between two consecutive adaptation phases. In contrast, when the workload manager detects an increased number of migrations, it decreases the skip length to allow faster adaptations.

5.4 Code Complexity

To give a rough overview of the required changes and the additional code complexity, we use the metric Lines of Code (LoC) – without considering comments, locks, and empty lines. In Table 4, we denote the lookup and insert functions of the original indexes and compare them to our workload-adaptive variants. We differentiate LoC into the actual logic (e.g., traversing the B+-tree) and the workload-tracking-related code (e.g., adding a sample to the adaptation manager). It can be seen that the tracking-related overhead is limited to at most 3/5 additional lines for lookups/inserts while coping with different encodings adds also extra complexity. An additional function handles the encoding migrations (140 lines for the Hybrid B+-tree, 51/70 lines for expansions/compactions in Hybrid Trie). Subclasses further encapsulate the communication between index and adaptation manager. These consist of 107/88 LoC for Hybrid B+-tree/Hybrid Trie.

6 RELATED WORK

Previous research proposed different strategies to reduce storage overhead and to leverage skew in DBMSs. Back in 2012, Funke et al. introduced an *online* compaction of hybrid in-memory OLTP/OLAP DBMSs based on a hot/cold clustering [21]. In this approach, the access frequencies get tracked at a VM page level. In contrast to this, Levandoski et al. monitor sampled accesses at a record level and write them to a log-file which is evaluated *offline* at a later point in time [33]. Both approaches primarily aim to move cold data to secondary storage devices to free memory capacities.

In 2016, Zhang et al. propose several *compaction rules* to reduce the memory footprints of in-memory DBMSs by reducing the space overhead of index structures such as B+-trees, radix trees, and skip lists [53]. In contrast to previous work, these techniques aim for full in-memory indexing as opposed to migrating cold data to disk: frequently accessed parts get stored using performance-optimized structures, whereas cold data gets compacted, but *remains* in memory. Our experiment in Figure 3 confirms that despite the most recent advances of SSD and NVMe disks, random I/O is still multiple orders of magnitude slower than on-the-fly in-memory decompression. While the introduced compaction rules might create *immutable indexes*, this problem is mitigated by their proposed dual-stage architecture: the dynamic stage contains the deltas created by inserts which are periodically merged into the compacted index.

Contrary, our approach applies different encodings within one single-stage index based on fine-grained access statistics. It does not require index developers to define complicated or expensive merge routines. Nevertheless, implementing different encodings and migration functions might also increase the code complexity. Besides the complexity, our adaptation framework has shown that it can leverage skewed workloads to a higher extent.

Succinct data structures such as FST [55] (which we use in Hybrid Trie) or its alternatives [12, 23, 44] also trade performance for memory efficiency. Succinct [7] and BlowFish [25] are two examples of data systems that use succinct data structures (in this case compressed suffix arrays [24]) for reduced space utilization and improved query performance through fitting more data in memory.

Learned indexes [18, 20, 22, 27, 28, 46, 47] also aim to reduce index size while retraining or even increasing lookup performance over traditional structures. Yet, we argue that the idea of learned indexes is orthogonal to our approach. For example, the Learned Index with Precise Precisions (LIPP) provides tight precision guarantees for all key ranges [51]. Combined with our approach, we could detect hot/cold ranges and increase/lower their precision bounds to reduce LIPP's size without affecting query performance.

7 CONCLUSIONS

We have presented an adaptive workload sampling approach that allows for switching between different node encodings at run-time and applied it to B+-trees and tries. We have shown that it provides significant space benefits without severely impacting performance under skewed workloads while causing negligible overhead under uniform workloads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was funded by the German Research Foundation (DFG) within the SPP2037 under grant no. Ke 401/22-2.

REFERENCES

[1] AWS EC2 Instances. https://aws.amazon.com/en/ec2/instance-types/high-memory [accessed 2021-03-01].
[2] LevelDB. https://github.com/google/leveldb [accessed 2021-03-01].
[3] RocksDB. https://rocksdb.org/ [accessed 2021-03-01].
[4] S2 Geometry Library. https://s2geometry.io/ [accessed 2021-03-01].
[5] Tape is dead, Disk is tape, Flash is disk, RAM locality is king. http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt [accessed 2021-03-01].
[6] C++ HopscotchMap. https://github.com/Tessil/hopscotch-map [accessed 2021-03-01].
[7] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling Queries on Compressed Data. In *NSDI*. USENIX Association, 337–350.
[8] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*. 26–37.
[9] Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, and Alfons Kemper. 2020. The Case for Hybrid Succinct Data Structures. In *EDBT*. 391–394.
[10] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *ACSC*. 97–105.
[11] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *EDBT*. 461–466.
[12] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2005. Representing Trees of Higher Degree. *Algorithmica* 43, 4 (Nov. 2005), 275–292.
[13] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*. 227–246.
[14] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX*. 209–223.
[15] Kun-Ta Chuang, Jiun-Long Huang, and Ming-Syan Chen. 2008. Mining top-k frequent patterns in the presence of the memory constraint. *The VLDB Journal* 17, 5 (Aug. 2008), 1321–1344. https://doi.org/10.1007/s00778-007-0078-6
[16] Edith Cohen, Nadav Grossaug, and Haim Kaplan. 2006. Processing Top k Queries from Samples. In *CoNEXT*.
[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154. https://doi.org/10.1145/1807128.1807152
[18] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. 969–984. https://doi.org/10.1145/3318464.3389711
[19] Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. 2020. START—Self-Tuning Adaptive Radix Tree. In *ICDEW*. IEEE, 147–153. https://doi.org/10.1109/ICDEW49219.2020.00015
[20] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135
[21] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *VLDB* 5, 11 (2012). https://doi.org/10.14778/2350229.2350258
[22] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*. ACM, 1189–1206. https://doi.org/10.1145/3299869.3319860
[23] Roberto Grossi and Giuseppe Ottaviano. 2014. Fast Compressed Tries through Path Decompositions. *ACM J. Exp. Algorithmics* 19, 1 (2014). https://doi.org/10.1145/2656332
[24] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.* 35, 2 (2005), 378–407. https://doi.org/10.1137/S0097539702402354
[25] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2016. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *NSDI*. USENIX Association, 485–500.
[26] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (Dec. 2019). http://arxiv.org/abs/1911.13014
[27] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *aiDM*. 1–5.
[28] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. ACM, 489–504. https://doi.org/10.1145/3183713.3196909
[29] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. ACM, 311–326.
[30] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. IEEE, 185–196. https://doi.org/10.1109/ICDE.2018.00026
[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, Vol. 13. 38–49.
[32] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *DaMoN*. 1–8.
[33] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying Hot and Cold Data in Main-Memory Databases. In *ICDE*. IEEE, 26–37.
[34] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *EuroSys*. 1–14.
[35] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*. 183–196.
[36] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13. https://doi.org/10.14778/3421424.3421425
[37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*. Springer, 398–412.
[38] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. 2006. Continuous Monitoring of Top-k Queries over Sliding Windows. In *SIGMOD*. 635–646.
[39] Gonzalo Navarro. 2016. *Compact Data Structures - A Practical Approach*. Cambridge University Press.
[40] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.
[41] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR*.
[42] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*. 61–72.
[43] Andrea Pietracaprina, Matteo Riondato, Eli Upfal, and Fabio Vandin. 2010. Mining top-K frequent itemsets through progressive sampling. *Data Mining and Knowledge Discovery* 21, 2 (2010), 310–326.
[44] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. *ACM Trans. Algorithms* 3, 4 (2007), 43.
[45] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-Join: Adaptive Skew Handling for Distributed Joins over High-Speed Networks. In *ICDE*. IEEE, 1194–1205.
[46] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the Last Mile: Efficient Learned String Indexing. *3rd International Workshop on Applied AI for Database Systems and Applications* (2021).
[47] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. *3rd International Workshop on Applied AI for Database Systems and Applications* (2021).
[48] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. 1990. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 125–142.
[49] Jeffrey S Vitter. 1985. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.
[50] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. 473–488.
[51] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *VLDB* 14, 8 (2021), 1276–1288.
[52] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance Analysis of NVMe SSDs and their Implication on Real World Databases. In *SYSTOR*. 1–11.
[53] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*. ACM, 1567–1581.
[54] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *TKDE* 27, 7 (2015), 1920–1948.
[55] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD*. 323–336.
[56] Huanchen Zhang, Xiaoxuan Liu, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *SIGMOD*. 1601–1615.

PUBLICATION P2

AutoSteer: Learned Query Optimization for Any SQL Database

P2.1 Synopsis

Rule-based query optimizers use rewrite rules that consist of a criterion and a rewrite action to optimize query plans. For every rewrite rule, the optimizer recursively searches the query plan for subtrees that satisfy the rule's criterion and applies the rewrite action. While rule-based query optimizers often define hundreds of rewrite rules, some are tailored for specific scenarios, making assumptions on or having been tested only with specific parameters, like the number of worker nodes and network latencies. As a result, the rule's effectiveness depends on multiple parameters and it can also degrade the query plan's performance [111, 118, 188].

To address this issue, database systems expose configurable knobs that allow users to toggle rewrite rules per query, a process known as *steering*. Recent work explored how query optimizers can be steered *automatically*. They use so-called hint-sets, which define the state of the database knobs, to generate multiple query plans and then use cost or machine-learned models to predict their execution time [111, 118, 188].

However, these approaches have several limitations. First, the number of hint-sets grows exponentially with the number of knobs. Since many database systems expose hundreds of knobs, predefining hint-sets or randomly selecting them at query optimization time significantly restricts the search space. Second, previous methods require deep integration into the database system's query optimizer, which limits their applicability to other systems.

This publication introduces AutoSteer, a novel framework that steers query optimizers automatically without requiring deep integration into database systems. Applying AutoSteer to a new system requires only a *text file* that contains the names of all the rule's knobs and a *connector* that establishes the connection with the database system. As a result of this design, AutoSteer shows excellent generalizability across database systems. We successfully applied AutoSteer to five open-source database systems: PrestoDB, PostgreSQL, DuckDB, SparkSQL, and MySQL.

P2.2 Contributions and Publication Details

Author Contributions. Christoph Anneser realized the conceptualization and development of AutoSteer, along with the implementation of the five database connectors. Additionally, he conducted all evaluations, except for the experiments conducted at Meta. He authored substantial parts of the manuscript.

Reference. Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. "AutoSteer: Learned Query Optimization for Any SQL Database". In: *PVLDB* 16.12 (2023), pp. 3515–3527.

DOI. https://doi.org/10.14778/3611540.3611544

https://authors.acm.org/author-resources/author-rights. Last accessed: 2024/06/10.

ACM Author Rights

ACM exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

Choose

ACM gives authors the opportunity to choose between two levels of rights management for their work. Note that both options obligate ACM to defend the work against improper use by third parties:

- **Exclusive Licensing Agreement:** Authors choosing this option will retain copyright of their work while providing ACM with exclusive publishing rights.
- **Non-exclusive Permission Release:** Authors who wish to retain all rights to their work must choose ACM's author-pays option, which allows for perpetual open access to their work through ACM's digital library. Choosing this option enables authors to display a Creative Commons License on their works.

Post

Otherwise known as "Self-Archiving" or "Posting Rights", all ACM published authors of magazine articles, journal articles, and conference papers retain the right to post the pre-submitted (also known as "pre-prints"), submitted, accepted, and peer-reviewed versions of their work in any and all of the following sites:

- Author's Homepage
- Author's Institutional Repository
- Any Repository legally mandated by the agency or funder funding the research on which the work is based
- Any Non-Commercial Repository or Aggregation that does not duplicate ACM tables of contents. Non-Commercial Repositories are defined as Repositories owned by non-profit organizations that do not charge a fee to access deposited articles and that do not sell advertising or otherwise profit from serving scholarly articles.

For the avoidance of doubt, an example of a site ACM authors may post all versions of their work to, with the exception of the final published "Version of Record", is ArXiv. ACM does request authors, who post to ArXiv or other permitted sites, to also post the published version's Digital Object Identifier (DOI) alongside the pre-published version on these sites, so that easy access may be facilitated to the published "Version of Record" upon publication in the ACM Digital Library.

Examples of sites ACM authors may not post their work to are ResearchGate, Academia.edu, Mendeley, or Sci-Hub, as these sites are all either commercial or in some instances utilize predatory practices that violate copyright, which negatively impacts both ACM and ACM authors.

After an ACM journal submission has been accepted and has entered the production process, ACM makes the Author’s Accepted Manuscript (AAM) available for preview under the ACM “Just Accepted” program until the “Version of Record” is available and assigned to its proper issue. The AAM carries the article’s permanent DOI and can be cited immediately.

https://authors.acm.org/author-resources/author-rights. Last accessed: 2024/06/10.

Distribute

Authors can post an Author-Izer link enabling free downloads of the Definitive Version of the work permanently maintained in the ACM Digital Library.

- On the Author's own Home Page or
- In the Author's Institutional Repository.

Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is notthe editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

Create

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.

Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"
- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

Retain

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
- Ownership and control of third-party permissions to use of software published by ACM

AutoSteer: Learned Query Optimization for Any SQL Database

Christoph Anneser[1]
Technical University of Munich
anneser@in.tum.de

Nesime Tatbul
Intel Labs and MIT
tatbul@csail.mit.edu

David Cohen
Intel
david.e.cohen@intel.com

Zhenggang Xu
Meta
zhenggang@fb.com

Prithviraj Pandian
Meta
prithvip@fb.com

Nikolay Laptev
Meta
nlaptev@fb.com

Ryan Marcus
University of Pennsylvania
rcmarcus@seas.upenn.edu

ABSTRACT

This paper presents AutoSteer, a learning-based solution that automatically drives query optimization in any SQL database that exposes tunable optimizer knobs. AutoSteer builds on the Bandit optimizer (Bao) and extends it with new capabilities (e.g., automated hint-set discovery) to minimize integration effort and facilitate usability in both monolithic and disaggregated SQL systems. We successfully applied AutoSteer on PostgreSQL, PrestoDB, SparkSQL, MySQL, and DuckDB – five popular open-source database engines with diverse query optimizers. We then conducted a detailed experimental evaluation with public benchmarks (JOB, Stackoverflow, TPC-DS) and a production workload from Meta's PrestoDB deployments. Our evaluation shows that AutoSteer can not only outperform these engines' native query optimizers (e.g., up to 40% improvements for PrestoDB) but can also match the performance of Bao-for-PostgreSQL with reduced human supervision and increased adaptivity, as it replaces Bao's static, expert-picked hint-sets with those that are automatically discovered. We also provide an open-source implementation of AutoSteer together with a visual tool for interactive use by query optimization experts.

PVLDB Reference Format:
Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. AutoSteer: Learned Query Optimization for Any SQL Database. PVLDB, 16(12): 3515 - 3527, 2023.
doi:10.14778/3611540.3611544

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/IntelLabs/Auto-Steer.

1 INTRODUCTION

Our research community has been making rapid strides in applying modern machine learning (ML) techniques to tackle longstanding problems in databases [6, 24, 48]. Learned query optimization lies at the forefront of this progress [51]. Various techniques from query-driven and data-driven to their combinations have been proposed [19, 20, 23] – not only to improve core query optimization tasks such as cardinality estimation [22, 23, 31, 32, 37, 39, 43], join order enumeration [29], or query rewriting [50], but also to build end-to-end query optimizers replacing [28, 42] or enhancing [27, 30, 44, 47] traditional ones. The practicality and robustness of these techniques are critical when applying them in industrial settings [47].

The so-called "steering approach" of Bao (Bandit optimizer) has been a successful example of a practical solution due to its emphasis on shortening training times, adaptivity to dynamic workloads, and ability to integrate with traditional optimizers [27]. Given a pre-determined collection of "*hint-sets*" (a hint-set indicates which query rewrite rules (RRs) should be considered in query optimization), Bao learns to steer an already existing query optimizer by helping it choose the right hint-set to use for every incoming query. This way, potential planning mistakes of traditional query optimizers can be avoided. As Bao's initial success continues to drive wider adoption in increasingly more sophisticated deployment and workload settings [3, 47], it also brings new challenges to the surface. We tackle two such challenges in this paper:

Integration effort: Adopting Bao to a new database system requires coming up with the right collection of hint-sets. In the original approach developed for PostgreSQL [1], a static collection of 48 hint-sets is manually selected based on deep knowledge of the underlying PostgreSQL optimizer [5], after which Bao independently learns to choose among these hint-sets on a per-query basis. Unfortunately, manually engineering feature hint-sets can be quite

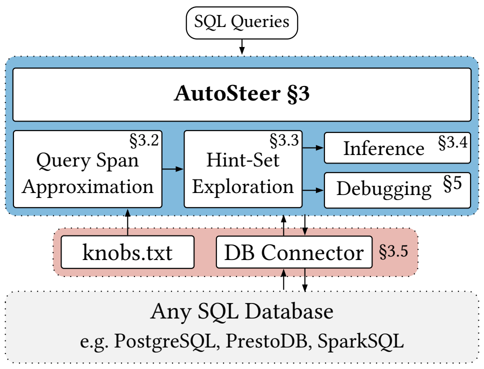


Figure 1: AutoSteer is a framework for steering query optimizers of SQL databases autonomously. For each query, we search for effective rewrite rules and store them in the query span. Then, we use a greedy algorithm to explore alternative query plans efficiently. The results can be used to train predictive models or to debug existing query optimizers.

[1]Work done while at Intel.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611544

challenging, as first noted by Negi et al. [30]. It is especially hard to hand-select hint-sets for those systems with a high number of possible hints to explore (e.g., Microsoft's query engine SCOPE has 256 rewrite rules, leading to 2^{256} possibilities to consider [30]). While it is possible to handcraft effective hint-sets for almost every database system [7–10], the knobs available in a particular system are not only different from all the other systems but are also implemented at different granularities. For example, PostgreSQL, PrestoDB, and MySQL expose these knobs at a session level [7–9], while SQLServer extends the structured query language to embed the knobs directly within the queries [10]. Therefore, integrating Bao into a new system requires deep insights and a solid understanding of the query optimizer to determine a promising collection of hint-sets. This impedes a generic application of Bao to new database systems and their optimizers. Instead, we need a more systematic approach that, given a SQL database, automates the hint-set selection process as well as minimizing the overall expert knowledge and manual engineering effort involved in integrating Bao.

Use in disaggregated settings: Database systems have been evolving away from their traditional monolithic architectures (e.g., federated, connector-based, coordinator/worker-style, data lake, and lakehouse querying systems [4, 13, 26, 34, 35, 40, 45, 49]). In such disaggregated settings with loosely-coupled database components, query optimizers must operate in complex and dynamic environments, often with limited access to accurate statistics and metadata [16, 21, 26]. Therefore, they often lack reliable cost models and rely on rules or heuristics for optimization. As such, an ML-based approach that can be easily integrated and automatically self-adapts could bring significant improvements to query performance [2, 41].

In this paper, we present *AutoSteer*, a new "plug-and-play" query optimization solution that builds on and extends Bao with new capabilities so it can be easily integrated and used with any SQL database system that exposes tunable optimization knobs. As illustrated in Figure 1, given a list of knobs as input (*knobs.txt*) and interacting with the database through SQL and explain statements (*DB Connector*), our solution uses a greedy algorithm to systematically explore promising hint-sets. This approach takes advantage of the notion of "query spans" [30] together with the compositional structure of advantageous hint combinations. This is in contrast to manually generated static hint-sets [27] and other previous approaches that rely on leveraging cost models or random sampling [30, 47]. Furthermore, AutoSteer generates hint-sets dynamically on a per-query basis, which maximizes workload adaptivity.

Furthermore, AutoSteer also provides an interactive usage mode to support human database experts in debugging and improving existing optimizers. For example, AutoSteer automatically discovers new hint-sets, generates alternative query plans, and evaluates the performance of the generated plans. This approach can assist query optimizer experts in gaining a deeper understanding of the cases where specific rewrite rules have a negative impact.

Overall, our extensions to Bao significantly expand the practical applicability of steering optimizers [27, 30, 47]. We provide experimental evidence from AutoSteer's use in five open-source databases. We further tested our solution using a real PrestoDB workload deployed at Meta, showing that it can also be effectively used in large-scale industrial settings.

Contributions. The key contributions of this paper include:

- We introduce AutoSteer, a practical learning-based framework to steer existing Cascades-style query optimizers.
- AutoSteer builds on and extends Bao with a novel hint-set discovery approach, which helps to generalize the technology behind Bao for a wide variety of SQL systems.
- We provide an open-source prototype implementation of AutoSteer[2] together with an interactive tool[3] to help human experts better understand AutoSteer's results. Our prototype has "plug and play" support for PrestoDB, PostgreSQL, SparkSQL, MySQL, and DuckDB and is extensible to other SQL engines.
- We demonstrate AutoSteer's practicality, generality, and effectiveness in query performance improvement via an extensive experimental evaluation based on several public benchmarks and a production workload tested inside Meta.
- Based on the experience gained along the course of this project, we share a few key insights which we believe can inform future work in query optimizer development.

2 RELATED WORK

Traditional Query Optimization. Our work primarily focuses on improving query optimization in SQL databases with traditional, Cascades-style query optimizers. First proposed in the 1990s [17], Cascades is an extensible query optimization framework that has been widely used in many industrial-scale as well as open-source database systems (e.g., PrestoDB [35], SCOPE [49], SparkSQL/Catalyst [13], Greenplum/Orca [36], Apache Calcite [14]). The Cascades framework follows a unified approach to logical/physical query planning by supporting both rule- and cost-based optimization, which is achieved by a set of transformation (logical) and implementation (physical) rules that are applied to the query plan. While rewrite rules drive logical planning, physical planning requires a reliable model to estimate the costs of query plan alternatives. Cost models, in turn, rely on the availability of accurate, up-to-date statistics and cardinality estimates [18]. Since mistakes happen, most industrial systems provide various workarounds to minimize the impact of such mistakes in their production deployments. For example, most of these systems support query hinting mechanisms as a tool to guide the optimizer's choices in exploring the plan search space more effectively [7–10, 15, 33]. Furthermore, in big data systems with federated architectures, such as PrestoDB [35], SCOPE [49], or SparkSQL [13], rule-based optimization is more heavily used in lack of the required statistics and cost models, which are harder to maintain in their larger scale and more heterogeneous production environments [16, 26].

Learned Query Optimization. Unsolved challenges of traditional query optimization have been investigated by several novel approaches that leverage recent advances in ML [51]. While there are too many to enumerate here [6], we believe it is sufficient to give a few representative examples. ML has been applied to improve both key components of a query optimizer such as the cardinality estimator [22, 23, 31, 32, 37, 39, 43] and the query planner [29, 44, 50], as well as the query optimizer itself as a whole [28, 42]. As an example of the use of ML in query planning, the LearnedRewrite approach

[2]https://github.com/IntelLabs/Auto-Steer
[3]https://github.com/christophanneser/QO-Insight

recently proposed by Zhou et al. uses Monte Carlo tree search to find better orders in which rewrite rules should be applied, reducing both query optimization and execution time in PostgreSQL's query optimizer [50]. As another example, HybridQO proposed by Yu et al. can produce better join orders by combining cost- and learning-based optimization and leveraging an optimizer's hint functionality in candidate plan generation [44]. In contrast, end-to-end learned query optimizers such as Neo [28] and Balsa [42] are designed as more performant drop-in replacements for traditional ones. While Neo bootstraps itself by learning from an expert optimizer (e.g., PostgreSQL's query optimizer), Balsa leverages a simulation-based approach. Both approaches have been shown to outperform open-source and commercial query optimizers, but only under certain workload assumptions (e.g., static datasets and schemas) and at the expense of long training times, which prohibits their frequent retraining. This motivated the more practical approach taken in Bao [27], which aims at steering traditional optimizers towards making better plan choices instead of entirely replacing them.

Steering Query Optimizers with Bao. The **Ba**ndit **o**ptimizer (Bao) learns to assist an already existing optimizer by providing it with *hints*, indicating which rewrite rules (RRs) should be turned off during query optimization [27]. Providing hints to a database system does not involve intrusive changes, as most database systems already expose optimizer knobs or flags that can be configured. Database experts and administrators can use these knobs to enable or disable specific RRs.

Bao was first applied to PostgreSQL, where it leveraged $n = 48$ different hint-sets to generate n (not necessarily different) query execution plans (QEP). Each hint-set disables a subset of the rewrite rules and can be seen as a *simpler* version of the default PostgreSQL query optimizer. In the second step, a tree convolutional neural network (TCNN) predicts the cost (e.g., the query latency or the CPU time) of each QEP. Based on the generated plan alternatives and their predicted costs, Bao decides which QEP should be executed. Instead of always choosing the plan with the best-predicted performance, Bao uses Thompson sampling to balance the exploration of new, alternative QEPs and the exploitation of plans already known to be efficient. Next, PostgreSQL executes the selected plan and records the execution time. Finally, both plan and execution times are added to Bao's experience, which is used to periodically re-train the model. After PostgreSQL, Bao has also been successfully applied to several commercial and open-source database systems, including Vertica, Microsoft Azure Synapse (SQL Server), and Amazon RedShift [3].

To assess Bao's industrial promise, Negi et al. explored how to apply Bao at the scale of Microsoft's SCOPE workloads [30]. SCOPE is Microsoft's internal query processing system for big data workloads, which primarily uses a rule-based query optimizer, though it can also support cost-based optimization through its cost model [49]. Negi et al.'s work on "*Bao-for-SCOPE*" introduced a number of key concepts which we also leverage in our work: *rule categories*, *rule configurations*, *rule signatures*, and *job/query spans*. There are four rule categories: required, off-by-default, on-by-default, and implementation. While required rules must be turned on to generate valid query plans, only rules from the other categories can be turned off to generate new query plans. A rule configuration is a bit vector specifying which rules are turned on and off during query optimization. Rule signatures track which rules have effectively contributed to the final query plan during the optimization. In addition to the rule signature, a job/query span contains only non-required rules. Based on these definitions, Negi et al. introduced a randomized configuration search to generate M rule configurations that produce possibly yet unknown QEPs.

In follow-up work, Zhang et al. built QO-Advisor to prepare Bao-for-SCOPE for actual production deployments at Microsoft [47]. This required adding support to deal with various operational challenges, such as the steering overhead, unexpected performance regressions, and the need for debugging. To reduce steering overhead, expensive tasks such as query span approximation and alternative plan exploration are done offline, as well as utilizing the cost model provided by SCOPE where possible.

All previous adoptions of Bao described above were done as custom integrations, each time targeting a particular system considering its specific architecture and workloads. Our approach fundamentally differs from these due to its *focus on generality*, i.e., making Bao more easily applicable in any SQL database system. The key enabler for this has been our *automated hint-set discovery approach*, which not only removes the need for manually designing system-specific hint-sets, but also makes them more flexible to use under changing workload conditions. Unlike previous Bao extensions [30, 47], AutoSteer is publicly available[2] to enable use across a wide range of SQL systems.

3 AUTOSTEER

In this paper, we present AutoSteer – a practical framework for adding Bao-style, steering-based learned query optimization capability to any SQL database that: (i) has a Cascades-style, rule-based query optimizer and (ii) exposes binary knobs to configure its rules.

Given a database system (DBMS) with an existing rule-based query optimizer, we aim to find semantically equivalent query plan alternatives that execute faster than the default plan generated by that system's native query optimizer. We follow the same general learned query optimization framework as in Bao [27]: Several hand-selected static hint-sets define which rewrite rules of the DBMS are turned on and off during optimization. Bao then leverages these different hint-sets to generate alternative query plans and picks the cheapest plan for execution using its TCNN-based neural prediction model. This approach is shown to be effective in finding query plans that are better than the ones that the underlying query optimizer can find, but there are two fundamental limitations:

(1) Database experts must manually identify a good collection of static hint-sets from scratch for each system Bao is to be integrated. Such an approach requires a deep understanding of that system and its optimizer.
(2) Scaling the number of hint-sets comes at the cost of additional optimization overhead. Bao always considers the same predetermined collection of hint-sets, since they are chosen in advance and independent of the actual query workloads, whether they impact a query's optimization or not.

In the following, we introduce AutoSteer as a new approach that overcomes these limitations. AutoSteer's key focus is on practicality. It builds on and extends Bao to make it more easily and adaptively applicable in SQL databases, no matter how simple or sophisticated

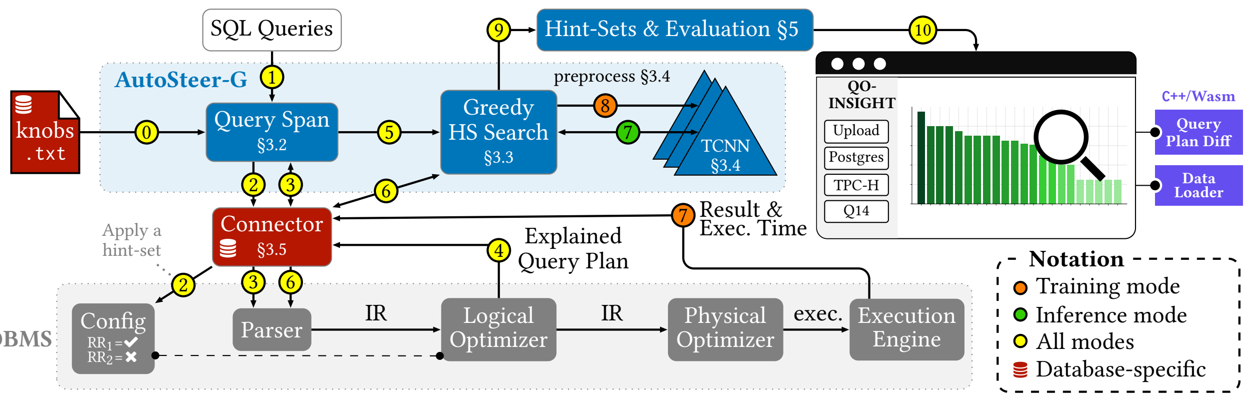


Figure 2: This figure illustrates AutoSteer-G, which communicates with the DBMS through a connector that is completely external to the DBMS. DBMS components are gray, and AutoSteer's components are blue. AutoSteer has two different execution modes: (1) generate training data and build the learned model (Training), (2) optimize queries at run time using the model (Inference). AutoSteer's results can also be interactively explored in QO-Insight [12] for debugging and analysis.

their optimizers are. As such, it generalizes recent industrial efforts on integrating Bao to specific systems [3, 30, 47], thereby facilitating broader adoption of this novel technology in a larger number and variety of DBMSs in the ecosystem. To maximize practicality, AutoSteer has been designed to support several usage options in terms of its: (i) DBMS integration level (custom vs. generic), (ii) execution mode (training vs. inference), and (iii) interaction mode (steering vs. debugging). We will describe these options in detail as part of the following subsections.

3.1 Architectural Overview

In Figure 3, we sketch two alternative ways of integrating AutoSteer into an existing DBMS: **(1) AutoSteer-Generic** leverages an *external connector* whose communication is purely based on SQL and explain statements; **(2) AutoSteer-Custom** implements a *connector* which is directly integrated into the DBMS optimizer. In this subsection, we assume AutoSteer-G, and provide further details on these two integration options in Section 3.5.

In Figure 2, we illustrate a typical SQL query optimization pipeline in a DBMS and show our AutoSteer-G solution in action. First, we provide a text file containing the knobs exposed by the optimizer to AutoSteer ⓪. Based on these knobs, AutoSteer will automatically explore and discover hint-sets without additional user input. Furthermore, instead of sending them to the DBMS, users and applications submit all queries to AutoSteer ①.

For each query, AutoSteer approximates a 'query span' by turning off rewrite rules (RR) systematically ②. Query spans track those RRs that *actually rewrite* the query plan. E.g., when such RRs are turned off, the optimizer would generate an alternative plan [30] (cf. to Section 3.2 for more details). As we are assuming AutoSteer-G in

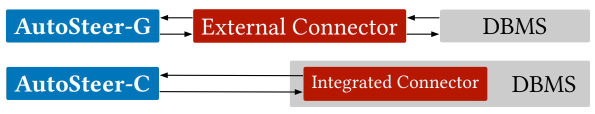


Figure 3: Integration Options: AutoSteer-G and -C.

Figure 2, we use an external connector to approximate query spans: First, it configures the DBMS's RRs through its exposed knobs and then lets the DBMS explain the query plan (③ & ④). We call the RRs *effective* if the plan changes and add them to the query span.

Based on the query span ⑤, AutoSteer searches for alternative query plans using a greedy hint-set exploration strategy, which is explained in more detail in Section 3.3. For a query span with *n* effective RRs, the algorithm first creates *n* hint-sets, with each hint-set disabling *one* of the query span's RRs. In AutoSteer-G, as a next step, we send the query and *each* hint-set to the external connector ⑥ and let the DBMS explain the resulting plans ④.

For the following steps, we differentiate two execution modes:
(1) Training mode: In the training mode, we *execute* the query plans from step ⑥ and track their execution times ⑦. Then, the aforementioned greedy search explores the search space of the beneficial hint-sets[4] by iteratively combining smaller beneficial hint-sets to create larger hint-sets, which might be beneficial as well. Later, we leverage the query execution plans (QEPs) and their run times to train a tree convolutional deep neural network ⑧.
(2) Inference mode: In contrast to the training mode, where QEPs have been *executed* to find out their *actual* run times, in the inference mode, we leverage the pre-trained tree convolutional neural network (TCNN)[5] to *predict* plan execution times ⑦. As before, when a query is submitted, we use the greedy search to find promising hint-sets more efficiently by pruning those hint-sets expected to perform poorly. Once the greedy search finishes, we sort all explored hint-sets by their predicted execution times and use contextual bandit to pick one hint-set to steer the query.

Finally, AutoSteer supports two user interaction modes:
(1) Steering mode: AutoSteer steers query execution at run time and uses the pre-trained TCNN to predict the execution time of the hint-sets (steps ① – ⑦, as described above).

[4]We call a hint-set beneficial iff it reduces the execution time wrt. the default plan.
[5]The DBMS could also be leveraged here if it provides a reliable cost model. In our experiments with PostgreSQL, however, we observed that the learned model leads to choosing better hint-sets and QEPs than the cost model.

(2) Debugging mode: AutoSteer exports the generated and evaluated hint-sets alongside the queries ⑨. These results can then be interactively explored in QO-Insight [12] ⑩.

3.2 Query Spans

In Bao, hint-sets define which rewrite rules (RRs) are turned on and off, and they are used to generate alternative query plans [27]. We cannot consider all the exponentially many hint-sets as database systems usually implement several tens to hundreds of RRs. However, creating a fixed number of *valuable* hint-sets, as suggested in [27], limits the search space and requires a deep understanding of the system's query optimizer and the workloads. Furthermore, Bao considers the same hint-sets for all queries regardless of whether the hint-sets turn off rules impacting the plan.

Instead, Negi et al. consider *effective* rules only (rules that *actually* rewrite the plan) and therefore introduce the concept of query spans [30]. A query span belongs to exactly one query and contains all the *non-required* rules that can *potentially* modify the query plan during its optimization. A rule r is non-required if the optimizer can generate a valid query plan without r. Calculating the *true* query span is challenging, as rules might have unknown dependencies on other rules. For example, turning off a set of rules could result in a different intermediate query plan that causes other alternative rules to become active.

Batch Approximation. Negi et al. [30] use a heuristics-based approximation of query spans instead. They leverage the SCOPE system for their work, whose query optimizer already tracks the *effective* RRs. Then, they turn off *all* effective RRs in one *batch*, and the process repeats until it does not detect other alternative rules.

Iterative Approximation. Alternatively, we use a more fine-grained, *iterative* approach: We iteratively turn off *one* effective rule (and its dependencies) at a time and check if other rules become effective. While this approach requires the query optimizer to run more often, it tracks rule dependencies more accurately. Later, we can utilize these dependencies during the exploration of hint-sets.

AutoSteer's *integration level* also impacts the query span approximation and the detected rules. When a connector is directly integrated into the query optimizer, it can track all rules programmatically in *one pass*. However, an external connector will not detect those RRs that change the query plan at an algorithmic level because such changes are usually not included in the explained query plan. Of course, the *more effective RRs* AutoSteer finds the *more* and potentially better *hint-sets* it can generate later. We evaluate the impact of the integration level for PrestoDB in Section 4.4.

While PrestoDB's optimizer implements 170 RRs, our experiments with AutoSteer-C and the iterative query span approximation for the 137 JOB queries show that only a few rules ($\leq$ 20) effectively contribute to the query plans. This observation reduces the theoretical search space of hint-sets from 2^{170} to 2^{20}. As 2^{20} configurations are still too many to explore, we propose a greedy exploration approach, as described next.

3.3 Dynamic Exploration of Hint-Sets

For a given SQL query, AutoSteer's goal is to find the most beneficial hint-sets that steer the optimizer toward better query plans. Although there are 2^n potential hint-sets for a query span with n non-required effective RRs, in our experiments in Section 4 with several different systems and workloads, we observed that only a few hint-sets are *beneficial* in practice. AutoSteer aims to find those few beneficial hint-sets as efficiently as possible.

Negi et al. propose an algorithm that *randomly* generates M hint-sets first and then filters for the most promising query plans according to SCOPE's cost model [30]. However, this approach requires an accurate cost model. Otherwise, we must *execute* the plans to determine whether they are beneficial. Furthermore, our experimental findings in Section 4.5 *suggest*:

(1) Most beneficial hint-sets *consist of smaller, beneficial hint-sets.*
(2) Most beneficial hint-sets are *small* (fewer than four knobs).

Based on these two empirical observations, we introduce a more structured and efficient way to explore the hint-sets. Our proposed algorithm, outlined in pseudocode in Listing 1, utilizes a greedy approach consisting of three building blocks:

🟩 First, we use the empty hint-set **{}** to execute the *default plan*, serving us as a baseline (i.e., the native optimizer's optimized plan) in line 4. In `results`, we map the hint-sets to their resulting query plans and execution times. The function `exec` executes the given query and hint-set, and returns the query plan and the execution time. When `infer=true`, AutoSteer does not execute the plan but instead uses a pre-trained model to predict its run time.

🟦 In the second block starting in line 7, we leverage the query span's effective RRs to generate *singleton* hint-sets, which turn off *exactly one* rule. Then, we let the DBMS execute these and track their resulting query plans and execution times in lines 11 and 12 if they perform better than the default plan. Please note that the greedy search is easily extendable. E.g., we could consider only those hint-sets whose improvements exceed a certain threshold.

```
            1  def explore_hint_sets(query, query_span, infer):
Baseline    2    results = dict() # Hint-set → (QP, exec. time)
            3    # 1. Execute baseline ({} is the empty hint-set)
            4    results[{}] = exec(query, {}, infer)
            5    singleton_hint_sets = []
Gen.        6    # 2. Run query with one rule turned off at a time
Singletons  7    for rule in query_span.effective_rules:
            8      QP, exec_time = exec(query, {rule}, infer)
            9      # Keep track of beneficial hint-sets only
           10      results[{rule}] = {QP, exec_time}
           11      if exec_time < results[{}].exec_time: # Beneficial?
           12        singleton_hint_sets.push({rule})
Greedy     13    # 3. Run a bottom-up greedy search
Exploration 14   hint_sets = copy(singleton_hint_sets)
           15    while not hint_sets.is_empty():
           16      hs = hint_sets.pop()
           17      # Generate larger hint-sets
           18      combined_hs = combine(hs, singleton_hint_sets)
           19      for new_hs in combined_hs:
           20        QP, exec_time = exec(query, new_hs, infer)
           21        results[new_hs] = {QP, exec_time}
           22        if exec_time < results[{}].exec_time: # Beneficial?
           23          hint_sets.push(new_hs)
           24    return results
```

Listing 1: Pseudocode of AutoSteer's greedy hint-set search.

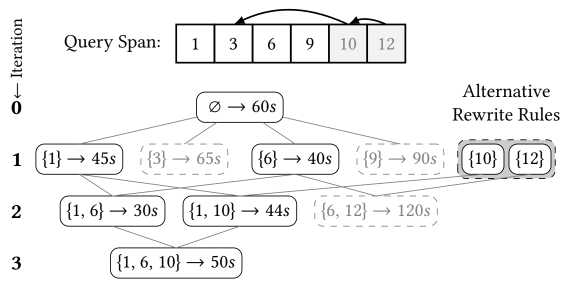


Figure 4: Example for AutoSteer's greedy exploration.

🟥 Third, the bottom-up greedy hint-set exploration loops over the previously seen beneficial hint-sets (line 15). We extract one hint-set at a time from the queue of beneficial hint-sets (line 16) and generate all other combinations with other singleton hint-sets in line 18. Note that we do not show the handling of alternative rules and tracking of the best-performing hint-sets (which is necessary for the inference mode) due to space limitations.

Figure 4 visualizes the algorithm for an example query span with effective RRs 1, 3, 6, and 9, and alternative RRs 10 and 12. Outgoing arrows denote rule dependencies of alternative rules. E.g., if rule c has been identified as an alternative to rule a, there is an edge $c \longrightarrow a$. We use the empty hint-set in the first iteration to execute the default plan. We then generate alternative plans using the hint-sets 1, 3, 6, and 9 and track their execution times. Hint-sets resulting in query plans with execution times exceeding the default plan execution time (3 and 9) are discarded and not considered in subsequent iterations. We also consider alternative rules in the following iterations while generating larger hint-sets.

3.4 Inference Mode using TCNNs

When AutoSteer executes in its inference mode (cf. step 7), it uses a learned model to make predictions about plan execution times. We borrow this part from Bao which uses a tree convolutional neural network (TCNN) for its predictive model [27]. Before query plans can be used with TCNNs, we must preprocess and featurize them. However, this step will slightly differ between DBMSs as databases have their custom query plan formats and operator types (e.g., PostgreSQL supports index scans, but PrestoDB does not). In addition to the preprocessing of PostgreSQL plans in [27], we implemented the preprocessing of PrestoDB's query plans and made the code publicly available.[2] We use, however, the same configuration for training as in [27]. We refer the reader to [28] for a deeper investigation into tree convolution applied to query plans.

3.5 Generic vs. Custom Integration

Generic Integration. As was already illustrated in Section 3.1, AutoSteer-G leverages an *external connector* whose communication is purely based on SQL and explain statements. This option is appealing due to its low programming effort. We thus far implemented external connectors for PostgreSQL, PrestoDB, SparkSQL, MySQL, and DuckDB, in less than 100 lines of code each. We show an example implementation of such an external database connector for PostgreSQL and PrestoDB in Listing 2. Since databases have their own custom APIs for exposing knobs or explaining query plans,

```
class PostgreSQLConnector(ExternalDBConnector):
  def __init__(url: str):
    self.conn = ... # setup PostgreSQL connection
  def set_knob(knob: str, enable: bool) -> void:
    self.conn.exec(f"SET {knob} TO \
        {'ON' if enable else 'OFF'}")
  def explain(query: str) -> dict:
    return self.execute(f'EXPLAIN {query}')
class PrestoDBConnector(ExternalDBConnector):
  def __init__(url: str):
    self.conn = ... # setup PrestoDB connection
  def set_knob(knob: str, enable: bool) -> void:
    self.conn.exec(f'SET SESSION {knob} = {enable}')
  def explain(query: str) -> dict:
    return self.execute(f'EXPLAIN JSON {query}')
```

Listing 2: External connectors for PostgreSQL and PrestoDB.

```
1  class QueryOptimizer:
2    def optimize(root_node) -> QuerySpan:
3      query_span = QuerySpan()
4      for rewrite_rule in self.rewrite_rules:
5        rewrite_rule.apply(root_node, query_span)
6      return query_span
7  class RewriteRule:
8    def apply(node, query_span):
9      if self.condition(node):
10       query_span.add(rule_id)
11       rewrite(node)
12     for child in node.child_nodes:
13       apply(child, query_span)
```

Listing 3: Tracking query spans in AutoSteer-C for PrestoDB.

these connectors implement functions to toggle knobs, explaining, and executing queries. As can be seen in this example, our external connectors only slightly vary in syntax from one system to another.

Custom Integration. As an alternative, AutoSteer-C's connector is directly integrated into the database's optimizer (i.e., similar to previous DBMS-specific applications of Bao [3, 30, 47]). While the implementation of an integrated connector involves more programming effort and requires a deeper understanding of the DBMS's optimizer, it can also make AutoSteer more efficient to execute. For example, AutoSteer-C would allow tracking effective RRs in a *single pass* and find RRs that cannot be detected by comparing the explained query plans. In contrast, AutoSteer-G would run *multiple* explain statements in the number of exposed knobs. Furthermore, AutoSteer-C reduces the run time overhead of optimizing queries in the inference mode by more efficiently interacting with the DBMS.

Listing 3 sketches how PrestoDB's query optimizer can be extended for AutoSteer-C to track query spans during optimization directly. First, PrestoDB parses the SQL statement into a logical query plan and then invokes the query optimizer on the root node in line 2. The query optimizer sequentially executes the RRs' apply function and passes references of the root node and the query span in lines 4 and 5. The modified RR directly adds itself to the query span once its conditions is fulfilled and it is applied to the query plan. Then, similar to explain statements, we would extend SQL's grammar to run the in-database query span approximation. Consequently, the custom integration avoids AutoSteer-G's overhead of running multiple explain statements.

Table 1: Benchmarks and workloads.

Benchmark	Dataset Size	Number of Queries
JOB [25]	7.2 GB	137
Stack [27]	100 GB	100
TPC-DS [38]	1/10/100 GB	100
Meta	>1 PB	>3000

Table 2: List of experiments.

Section	Experiment	Workload	Setup
§4.2	AutoSteer-C for PrestoDB	JOB, Stack	1
§4.3	AutoSteer-C for PrestoDB	Meta	2
§4.4	AutoSteer-C vs. -G for PrestoDB	JOB	1
§4.5	AutoSteer-G vs. Bao for PostgreSQL	JOB	3
§4.6	AutoSteer-G for SparkSQL	TPC-DS	4
§4.7	AutoSteer Coding Effort	N/A	N/A

We implemented both an external (see Listing 2) and an integrated connector for PrestoDB, and provide an empirical comparison in Section 4.4 and a more general discussion on coding effort in Section 4.7. In general, both of these integration options will be useful in practice. For example, we envision AutoSteer-G to be used for rapid proof-of-concept prototyping to show the feasibility and to approximate potential performance gains on a DBMS, after which AutoSteer-C is implemented for use in production deployments where its run time efficiency would matter more.

4 EVALUATION

To evaluate AutoSteer, we applied it to five different SQL databases: PrestoDB, PostgreSQL, SparkSQL, MySQL, and DuckDB. We report our experimental findings with the former three in this section and provide a summary of our experience with the latter two as part of Section 5. The high-level goals of our experimental study are:

- Show AutoSteer's generality and practicality by testing its effectiveness on a variety of open-source systems and benchmarks commonly used by database researchers and practitioners.
- Validate AutoSteer's effectiveness when applied to real-world workloads from large-scale deployments in industrial settings.
- Evaluate how AutoSteer's automatically generated hint-sets fare against the expert-selected hint-sets of original Bao and the randomized approach used in its SCOPE adaptation [30].
- Quantify the productivity and performance tradeoffs of using AutoSteer in its custom and generic integration levels.

4.1 Experimental Setup

Benchmarks and Workloads. Table 1 shows the workloads we used in our experiments. Three of these are public benchmarks heavily used by the database and query optimization communities (JOB w/o FK indexes [25], Stack [27], and TPC-DS [38]), and the fourth is a real-world workload from large-scale PrestoDB deployments at Meta. These workloads cover a range of scales regarding dataset sizes (GBs-PBs) and the number of queries (100s-1000s).

Hardware and Software Setups. We used multiple different hardware/software setups for our experiments:

Setup 1: As sketched in Figure 5, we deploy PrestoDB on a 5-node Kubernetes cluster. All nodes have a dual-socket Intel® Xeon® Platinum 8280 CPU with 2 × 28 cores at 2.7 GHz, 256 GB memory,

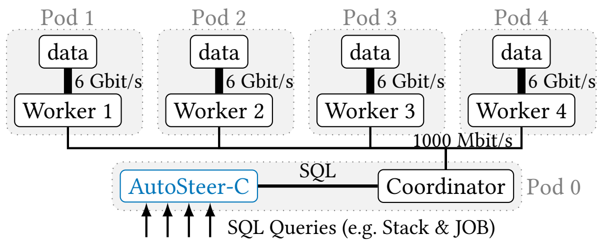


Figure 5: In Setup 1, we run PrestoDB on a 5-pod K8s cluster. AutoSteer intercepts queries and automatically explores alternative plans. Datasets are cached on worker-local SSDs.

and an Intel® DC S3500 SSD attached, which stores a copy of the datasets to reduce transfer times between nodes. The compute nodes are connected with 1000 MbE. We run all queries in isolation and with warm caches.

Setup 2: This setup corresponds to our real-world workload experiments with PrestoDB conducted at Meta. This involved executing a large interactive dashboarding workload scanning petabytes of data against a large PrestoDB cluster with hundreds of nodes. We tested more than 3000 queries that run every day at Meta.

Setup 3: We run PostgreSQL 13 on a 16-core AMD Ryzen 3950X@3.5 GHz machine with 96GB DDR4-2667 memory. We only execute hint-sets yielding new query plans and use warm caches.

Setup 4: We configured SparkSQL v3.2.2 as it is internally used at Intel and deployed it on a single machine equipped with a dual-socket Intel® Xeon® Platinum 8280 CPU with 2×28 cores at 2.7 GHz and 256 GB memory. All datasets were stored in memory.

To account for runtime variances in Setups 1, 3, and 4, we executed the query plans generated by each hint-set multiple times and compared their median execution times.

Overview of Experiments. Table 2 provides an overview of the conducted experiments together with the experimental workloads and setups used for each. In terms of its usage options, we explicitly state if AutoSteer was used in the custom (*AutoSteer-C*) vs. generic (*AutoSteer-G*) integration level in each of the following subsections. For each experiment, we state whether we used the *Training* or the *Inference* execution mode. We set the interaction mode to *Steering* for all of our experiments.

4.2 AutoSteer-C for PrestoDB

Does AutoSteer-C find better plans than PrestoDB? In Figure 6, we compare AutoSteer-C to PrestoDB. We executed all 137 JOB queries on the PrestoDB cluster (Setup 1), and we show the relative performance changes for a uniform sample. Then, we sort the queries by their relative performance improvements achieved by the best known[6] query plan generated by AutoSteer-C's training mode and plot them in ascending order. Here, we consider only *alternative* plans that differ from the default query plan. Our approach finds a better alternative execution plan for most queries (green bars). For this selection, there are only four queries for which the best known alternative plan performed worse than the default plan generated by PrestoDB (28a, 5c, 25a, 21a). However, those queries have short execution times of ≤ 4 seconds. In contrast, by

[6]The term "best known hint-set" refers to the hint-set that leads to the fastest execution plan among *all plans explored* by either AutoSteer or its competitors.

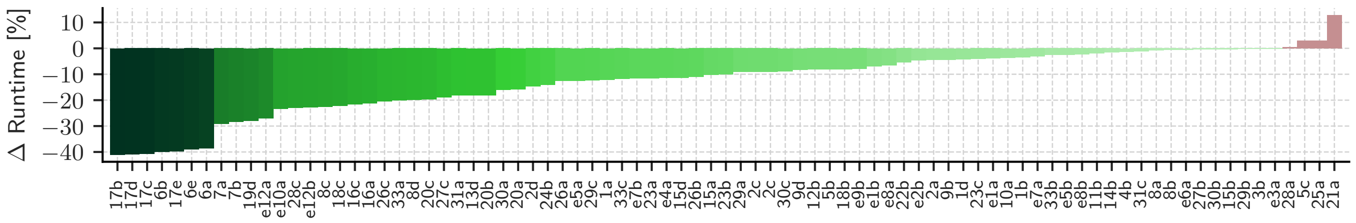


Figure 6: The relative run time changes (lower is better) of the *best known alternative (non-default) query plan* found by AutoSteer-C's training mode compared to PrestoDB's default plan. For space reasons, we consider a uniform sample of the JOB queries executed in Setup 1. We used the greedy algorithm described in Section 3 to explore beneficial hint-sets.

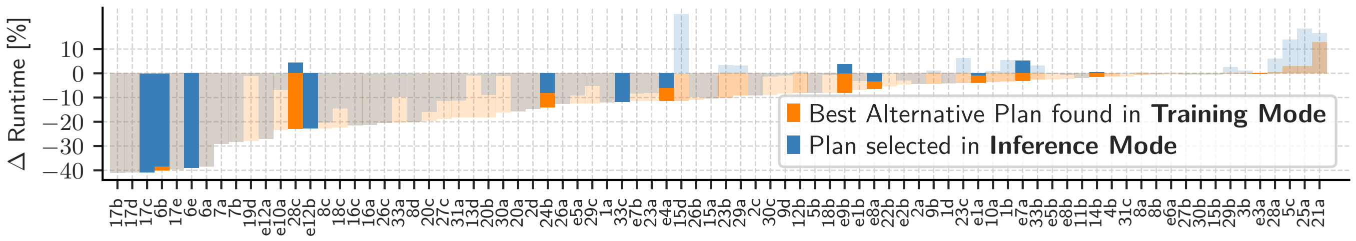


Figure 7: The relative run time savings of the best known plan (🟧) found by AutoSteer-C's training mode compared to PrestoDB. For the inference mode, a pre-trained TCNN predicts plan run times and selects the plan with the best-predicted performance (🟦). Solid colors represent unseen queries from the test set, and transparent colors represent those from the training set.

Table 3: The top-5 hint-sets that yield the largest performance gains wrt. to PrestoDB's default plan. We ran JOB queries in isolation (Setup 1). The last column shows the number of JOB queries for which this hint-set produced the fastest plan.

	Run Time Changes [%]		
Hint-Set	**Average**	**Worst Case**	**# Best HS**
HashGenOptimizer	-30.35%	+12.82	75
HashGenOptimizer, UnaliasSymbolRefs	-38.29%	+0.07	25
PickTabLayoutForPred	-5.75%	+0.03	13
UnaliasSymbolRefs	-8.99%	-0.44	9
PruneTabScanCols	-8.63%	+3.01	8

turning off the *HashGenOptimizer*, the execution time of query 17b decreases by more than 40% (127 − 75 = 52 seconds).

What are the top hint-sets AutoSteer-C generates? Table 3 presents the top-5 hint-sets discovered by AutoSteer-C for PrestoDB. The second column shows the average run time reduction achieved by each hint-set when it generated the fastest query plan. Out of all 137 JOB queries, the third column shows the impact on the performance in the worst case. The last column shows the number of queries for which the hint-set produced the best known plan. The top hint-set disables *HashGenOptimizer* and yields the fastest execution plan for 75 JOB queries. As described in [35], HashGenOptimizer adds local projections to compute hash codes early during execution, increasing the cost of downstream shuffles and filling up buffer memory. Moreover, as our experiments show, these shuffles' overhead will outweigh the parallelism's performance gains. While most JOB queries (75/137) benefit from disabling HashGenOptimizer, a few will regress by up to 12.8%. In Section 5, we discuss how experts could leverage AutoSteer's insights to improve a specific rule conceptually.

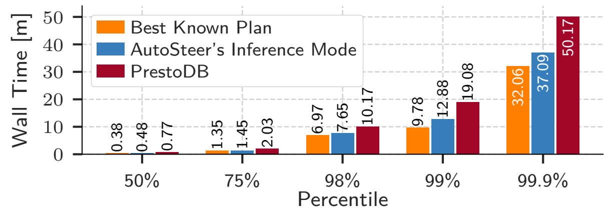
Table 4: AutoSteer's relative and absolute run time improvements compared to PrestoDB's default plans on average using Setup 1. The results belong to the queries from the test set.

	JOB	**Stack**
Rel. Improv. (Best Known Hint-Set)	30.25%	**42.38%**
Rel. Improv. (Inference Mode)	27.93%	**31.54%**
Abs. Improv. (Best Known Hint-Set)	**305s**	284s
Abs. Improv. (Inference Mode)	**282s**	211s

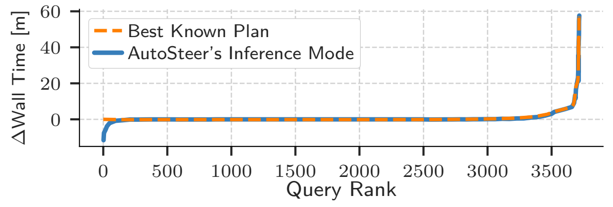
Can AutoSteer-C's Inference Mode improve PrestoDB? To answer that question, we fit a tree convolutional neural network (TCNN) that we later use to infer the query plans' execution times. First, we use AutoSteer's training mode to explore hint-sets for the JOB and Stack benchmarks in Setup 1. We split the 237 (100 Stack and 137 JOB) queries into training and test sets at an 80/20 ratio. Next, we train the TCNN in a supervised fashion on the training set and choose the same configurations as suggested in [27]. Then, the TCNN predicts the run time for each query plan.

Figure 7 compares the best-performing plans found in training mode to those selected in AutoSteer's inference mode for JOB. The orange bars show the relative improvements of the best known plan wrt. PrestoDBs default query plan. Blue bars show the relative improvements of the plans selected by AutoSteer's inference mode. For most queries, the TCNN chooses a hint-set that improves the execution time compared to PrestoDB's default plan. However, for a few queries, such as e9b and 15d, the TCNN chooses hint-sets that negatively impact the execution time. Overall, the inference mode generalizes well to unseen queries from the test set.

Table 4 considers AutoSteer's overall impact on the JOB and Stack benchmarks in Setup 1. We compare the average relative and absolute performance improvements of the hint-sets found



(a) AutoSteer's inference mode using the top hint-set reduces tail query latency closer to performance of the best known plans.



(b) Distribution of raw query time changes for AutoSteer with the top hint-set. A few queries regress, but tail latency improves.

Figure 8: Experiments with a production workload at Meta.

by AutoSteer-C when choosing either the best hint-sets known from the training mode or the selected hint-sets in the inference mode. For Setup 1 and the Stack queries, the best plans discovered in AutoSteer's training mode reduce the run time by up to 42%. AutoSteer's inference mode reduces the relative run time by up to 31%. The absolute, overall JOB execution time decreases by 305 seconds when selecting the best known hint-set. AutoSteer's inference mode reduces the absolute run time by 282 seconds.

RESULT SUMMARY. *AutoSteer-C can generate better query plans than PrestoDB's native query optimizer in both of its execution modes (Training and Inference), with zero help from a human expert in hint-set selection. It finds "HashGenOptimizer" to be the top hint-set.*

4.3 AutoSteer-C for PrestoDB: Meta Workload

To validate AutoSteer's effectiveness in real-world scenarios, we tested our approach on a large-scale dashboard application deployed at Meta (Setup 2). The dashboard application runs on PrestoDB and executes thousands of queries every day over petabytes of data. Since dashboard views commonly consist of many widgets (queries) and the dashboard is only helpful once a large portion of queries are completed, we focus our analysis on tail latency.

We first run a selection of hint-sets generated by AutoSteer on the workload. To minimize computation time, we leverage the most promising hint-set known from the experiments described in Section 4.2 (i.e., disabling *HashGenOptimizer* as shown in Table 3). First, we run AutoSteer's training mode to generate alternative query plans and track their execution times. Then, we use that training data to fit a TCNN, which is later used by AutoSteer's inference mode. Figure 8a shows the tail latencies achieved from an "optimal oracle" that always chooses the best query plan known from AutoSteer's training mode, best-predicted plan from AutoSteer's inference mode, and the default plan from Meta's production PrestoDB configuration, respectively. With the single hint-set discovered by our approach, we observe a noticeable reduction in the tail latency.

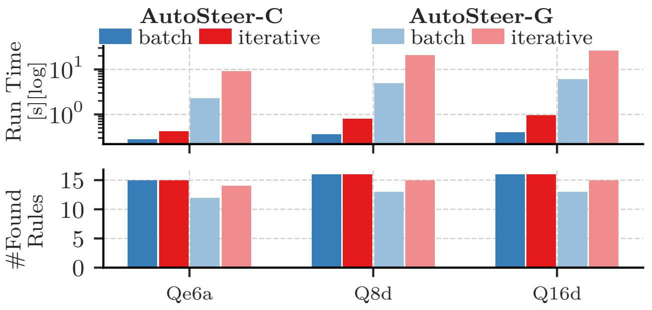


Figure 9: Time elapsed and number of rules found for different query span approximation variants in AutoSteer for PrestoDB. We consider three random queries from JOB [25].

While our predicted approach does not quite reach the performance of the best known plan, it comes close in most cases.

Any change to a query optimizer might result in a regression. We plot the query performance changes for our approach (AutoSteer) and the best known plan in Figure 8b. The "long tail" is significantly decreased by our approach, although a few regressions (in the order of 10 minutes) do occur. Since the slowest queries determine this application's performance, these regressions are negligible, and the overall performance is improved.

RESULT SUMMARY. *AutoSteer can help reduce tail query latency of a PB-scale interactive dashboard application running on a PrestoDB cluster at Meta. Even with one top hint-set discovered by AutoSteer ("HashGenOptimizer"), about 20% reduction in 99% tail latency can be achieved over PrestoDB's native query optimizer.*

4.4 Approximating Query Spans in PrestoDB

In this section, we evaluate the impacts of AutoSteer's integration level (Section 3.5) as well as its two approximation heuristics *batch* and *iterative* (Section 3.2) on query span approximation performance. We measure the performance considering two dimensions: the execution time and the number of effective RRs found.

How long does query span approximation take for the different variants? In the upper part of Figure 9, we compare the execution times of the query span approximations based on AutoSteer-C and AutoSteer-G for PrestoDB on three selected JOB queries. We further differentiate between the *batch* and the *iterative* approximation heuristics. AutoSteer-C, whose connector is *directly integrated* into PrestoDB's optimizer, tracks effective RRs during the optimization phase, which allows AutoSteer-C to achieve better performance and makes it almost an order of magnitude faster than AutoSteer-G's external connector. In contrast, the external connector runs one explain statement for each of the exposed knobs (170).

The *batch* approximation requires the query optimizer to run the fewest iterations, therefore, completes faster than the *iterative* approximation, which additionally tracks dependencies of the alternative RRs. The rule dependencies, however, help reduce the search space in the following hint-set exploration (cf. Section 3.3).

Given the significant overhead of AutoSteer-G using the iterative heuristic, this approach is not suitable for short-running and transactional queries, but it might amortize for long-running and analytical queries. Furthermore, it could help database experts in quickly implementing a first proof-of-concept revealing the potential performance improvements, where the query span approximation time would not be as critical.

How many rules do the different query span approximation variants detect? In the lower part of Figure 9, we plot the number of detected RRs for each approach. The experiments show that AutoSteer-C for PrestoDB finds the same number of RRs independent of the used approximation heuristic. AutoSteer-G detects fewer rules (especially in the batch mode) because some of the RRs change operator details at an algorithmic level which are *not* included in the explained query plan that is used by the generic integration level. For instance, the rules *SetFlatteningOptimizer*, *PickTableLayoutWithoutPredicate*, and *ApplyConnectorOptimization* affect the execution plan for JOB query 8d, but their changes are transparent in the explained plans. When using the iterative heuristic, the degradation in the number of RRs is not as noticeable.

RESULT SUMMARY. *AutoSteer-C with batch heuristic is most efficient in finding the most number of rules during query span approximation. Despite being slower, AutoSteer-G can also find a significant majority of the rules and, as such, can be a suitable option to use for workloads with long-running queries and initial prototyping.*

4.5 AutoSteer-G for PostgreSQL

We use Setup 3 to compare AutoSteer-G to (1) the original Bao-for-PostgreSQL [27] and (2) the randomized hint-set search used in its SCOPE adaptation [30, 47]. We analyze what hint-sets these approaches explore and their impact on query performance.

Does AutoSteer-G find better hints than Bao-for-PostgreSQL? Remember that the key difference between AutoSteer and Bao-for-PostgreSQL is which and how hint-sets are selected. In contrast to Bao's 48 static hint-sets *chosen manually* by an expert, AutoSteer generates them *automatically*. Considering the 137 JOB queries, both approaches found the best known hint-set[6] in 99 cases. AutoSteer-G, however, discovered better hint-sets for 23 queries. In the remaining 15 cases, Bao-for-PostgreSQL found at least one hint-set, which performed better than all hints-sets generated by AutoSteer-G. However, for those queries where AutoSteer-G did not find the best hint-set but Bao-for-PostgreSQL did, we missed performance improvements of 2% on average. Bao-for-PostgreSQL decreases the overall JOB run time by 33.1%, whereas AutoSteer-G saves an additional 0.4%, which results in a relative improvement of 33.5%. However, further experiments showed that the explored hint-sets and their performance implications also depend on the PostgreSQL configuration. *In other words, AutoSteer-G matches Bao-for-PostgreSQL's performance improvements, even though its hint-sets are automatically explored and not pre-selected by human experts.*

What are the top hint-sets found by AutoSteer-G? The top hint-set AutoSteer-G's training mode found is turning off *Nested Loop-Joins*. That hint-set improves query performance the most for 29 JOB queries, reducing the run time by 30.7% on average. The second-best hint-set turns off *index scans*: in many cases, PostgreSQL overestimates the selectivity of complex predicates and, therefore, index lookups yield substantial overhead compared to a sequential scan. These two top hints are also part of Bao-for-PostgreSQL. However, AutoSteer-G also discovered brand-new hint-sets, e.g., disabling *Parallel Hashing* and *GatherMerge*, which improved selected queries by up to 38.5% and 91.1%, respectively.

Is the greedy hint-set exploration approach effective? In Section 3, we hypothesized that beneficial hint-sets are often composed of smaller beneficial hint-sets and argued for using a greedy hint-set exploration approach. In Figure 10, we experimentally evaluate this assumption and visually compare AutoSteer-G's hint-set exploration with Bao-for-PostgreSQL's 48 handcrafted hint-sets for the JOB queries 10b, 12b, and e2a: The y-axis shows the hint-set size. Some hint-sets (⬜) defined by Bao-for-PostgreSQL result in duplicated execution plans. Contrary, AutoSteer-G tracks already-seen query plans and does not execute duplicates. The colors encode the query plan's relative performance wrt. the best and the worst plans known from both approaches (on a logarithmic scale). Light colors indicate better plans, and vice versa. Compared to PostgreSQL's default plan (the bottom-most square), most hint-sets result in worse plans, but only a few lead to better plans. Squares with black edges were discovered by AutoSteer-G's greedy training mode. Here, AutoSteer finds the best hint-sets (★) for each query.

How does AutoSteer-G's greedy hint-set exploration compare to randomized approaches? The hint-set exploration approaches used in Bao-for-SCOPE [30, 47] first uniformly draw hint-sets and then use SCOPE's cost model to select the ten hint-sets generating the cheapest plans. As discussed in Section 1, cost models may not always be available or sufficiently accurate. Therefore, AutoSteer-G's greedy hint-set exploration strategy does not rely on cost models. To compare these two strategies on fairgrounds, we tested both variants (w/ and w/o using the DBMS cost model) under Setup 3. We use eight representative JOB queries {10a,...,17a}, execute all query plans (for this experiment, we include duplicates, as some changes might not be exposed in the QEP) seven times to get robust measurements and we do *not* limit the execution time.

Greedy vs. Randomized for a Single JOB Query (Figure 11): We first compare greedy to randomized exploration without using the PostreSQL cost model. We consider the *expected query performance improvements* $\mathbb{E}(k) = (\sum_{x \in \mathcal{X}_k} \max(x))/|\mathcal{X}_k|$ for randomly drawing k hint-sets. Here, $\mathcal{X}_k$ is the set of all combinations with k hint-sets and $\max(x)$ returns the relative improvement of the top hint-set in x, or 0, if there are no improvements. We show the expected query performance improvements for JOB query 10a in Figure 11, for which we executed 250 randomly selected hint-sets. Greedy quickly gains the expected improvements in iteration 1 and stops after the second iteration 2 after exploring eight hint-sets. In contrast, the randomized exploration has to search significantly more hint-sets to achieve similar improvements (e.g., 95%/99% of greedy's improvements after exploring 32/43 hint-sets). These findings indicate that our greedy approach reaches higher query performance improvements faster than the randomized approach.

Greedy vs. Randomized for Multiple JOB Queries (Table 5): Next, we compare the two approaches for JOB queries {10a, ..., 17a} (first w/o, then w/ using the cost model). As summarized in Table 5,

Table 5: Greedy vs. Randomized Hint-Set Exploration

Approach	Query Perf. Imp.	Time (min)
Greedy	48.68%	90.45
Randomized	50.22%	4597.10
Greedy w/ cost model	24.85%	1.58
Randomized w/ cost model	24.87%	10.77

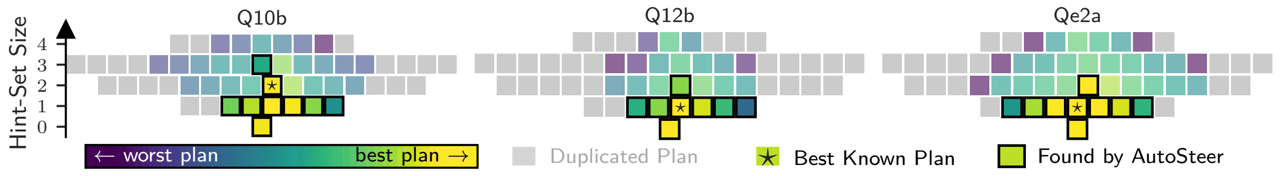


Figure 10: We compare AutoSteer-G's training mode with the 48 hint-sets defined in Bao-for-PostgreSQL [27] for three JOB queries. AutoSteer finds the best known hint-set for each of the three queries while it aggressively prunes the search space.

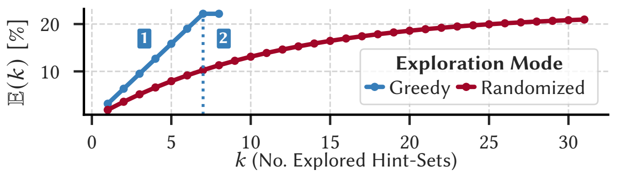


Figure 11: Expected performance improvements for greedy and randomized explorations for JOB Q10a and PostgreSQL.

greedy explores 86 hint-sets in about 90 minutes to reach a performance gain of 48.68%, while randomized explores 2000 hint-sets in about 4597 minutes to reach 50.22%. Thus, greedy reaches a similar level of performance improvement in about 2% of the time it takes for the randomized to do so. Finally, we compare variants of the two approaches that leverage the PostgreSQL cost model to limit their exploration to the ten cheapest QEPs. This significantly reduces the run times of both approaches but also degrades how much they can improve query performance to less than 25% since the PostgreSQL cost model overestimated the execution times of the most beneficial hint-sets. Overall, we observe that greedy strikes a good tradeoff between query performance improvement and exploration overhead while not requiring a reliable DBMS cost model.

RESULT SUMMARY. *AutoSteer-G's automated approach can find plans as good as those found based on Bao-for-PostgreSQL's expert-selected hint-sets, while also exploring the plan search space more efficiently. Furthermore, AutoSteer-G's greedy hint-set exploration strategy makes a better tradeoff between plan quality and exploration overhead than randomized alternatives.*

4.6 AutoSteer-G for SparkSQL

To further demonstrate our solution's general applicability, we also evaluated it with another widely used SQL engine, SparkSQL.

Can AutoSteer-G improve SparkSQL's query performance? Inspired by Intel's use of TPC benchmarks for its internal projects around SparkSQL, we experimented with different TPC-DS workloads and compared AutoSteer-G to SparkSQL's native optimizer for scale factors 1, 10, and 100. AutoSteer-G reduces the overall run time by up to 44.3% for SF1, 31.0% for SF10, and 22.0% for SF100.

Does the benchmark's scale factor impact AutoSteer-G's hint-set selection decisions? We interestingly observe that AutoSteer-G explored and selected a different collection of beneficial hint-sets at different scale factors. For smaller scale factors, the performance improvements primarily come from turning off expensive RRs such as *ConstantFolding*, which do not amortize for most short-running queries. SparkSQL primarily focuses on large-scale data processing. As performance is mainly dominated by query execution and not by its optimization, the RRs' efficiency was probably not a priority during development. With larger scale factors 10 and 100, performance improvements can be attributed to hint-sets improving the query plans on a structural level (e.g., ReorderJoins and CombineUnions).

RESULT SUMMARY. *AutoSteer-G can generate better query plans than SparkSQL's native query optimizer. Furthermore, due to its more adaptive approach, it can use different hint-sets at different scale factors – something that Bao would not be able to do due to its static approach to hint-set selection.*

4.7 AutoSteer-G vs. AutoSteer-C: Coding Effort

We implemented AutoSteer-G connectors for five well-known open-source database systems using Python3. We use the metric *lines of code* (LOC) to approximate the connectors' code complexity and exclude all comments, empty lines, import, and log statements. The connector for DuckDB has the fewest lines of code (34), followed by PrestoDB (53), PostgreSQL (49), MySQL (55), and SparkSQL's connector with 68 lines. For SparkSQL, we had to implement a post-processing step to remove random identifiers from the explained query plans. In contrast, implementing AutoSteer-C's custom integration for PrestoDB was significantly more complex, as we had to modify 1757 lines of code. The LOC metric indicates how simple the connectors are and prototyping a new connector for AutoSteer-G can be done in a few hours. All connectors are publicly available.[2]

5 LESSONS LEARNED AND FUTURE WORK

Applying AutoSteer in a Production Environment. We encountered several additional difficulties when applying AutoSteer to a large-scale production environment at Meta. First, PrestoDB's optimizer is cache-oblivious, meaning that the optimizer selects query plans without using information about the caches of a particular compute node. Cache hits or misses can significantly affect query performance. Thus, a change to a query plan that appears positive or negative may result from caching. Past works on learned query optimization dealt with this issue by simply assuming a warm or cold cache (e.g., [28, 42]), but in reality, the cache is rarely entirely warm or cold. An entirely warm or cold cache will often impact query performance more than many plan changes. Of course, the best solution to this problem would be to take the state of the cache into account as a feature (e.g., [46]), but this is easier said than done in large distributed environments: measuring the contents of Meta's PrestoDB deployment would take significantly longer than most queries. It is thus beneficial to examine many executions for a single query plan, preferably across a wide time range, to ensure different cache states are observed and accounted for statistically.

Second, many academic assumptions about query performance do not match the needs of some large organizations: (1) A few regressions are inevitable and acceptable with any optimizer change. Therefore, we use tail metrics – like P90/P95/P99 – to evaluate cluster performance and to accept or reject optimizer changes. (2) Changes in relative query performance are less important than changes in absolute query performance. For example, a 50ms query becoming a 200ms query looks like a 4x regression but is likely irrelevant in analytics. However, a 60s query becoming a 50s query, which "only" looks like a 15% improvement, is a desirable change. Thus, many past works using geometric means or relative latency metrics might be misleading. Metrics such as the geometric mean make an optimizer update that induces both previously described changes resemble a regression. However, it would actually be a significant upgrade. We suggest future evaluations of optimizers to include statistics about absolute changes in query latency.

Optimization Goals. In this work, we focus on minimizing query latencies. However, in industrial settings, there are different parameters that one would like to optimize for, including network transfers, I/O, and memory footprint, amongst others. For example, in Meta's PrestoDB deployments, memory is at a premium: increasing the concurrency of a particular query might improve its run time, but if the query's memory footprint increases substantially, other queries on the cluster might run out of resources, spilling to disk and causing general chaos. As a general rule of thumb, a 10% decrease in a query's memory footprint is as desirable as a 30% decrease in query latency (there are many exceptions to this rule, especially for queries with tight deadlines). Similarly, the CPU usage of a query is relevant to overall data center costs. Trading decreased latency for an overall increase in CPU time (e.g., again from parallelism) might be undesirable if the query was not time-critical.

Fortunately, AutoSteer can be easily extended to support arbitrary optimization functions. By changing the reward signal to whatever combination of measurable performance metrics is desired, AutoSteer can adapt to many different performance requirements. Unfortunately, real-world performance requirements often do not fit in single-query performance metrics. For example, a particular optimization might increase the memory footprint of one query by 60MB but decrease the footprint of two others by 25MB each. If these three queries run concurrently, this nets 10MB savings. However, such query-to-query tradeoffs are not expressible as a function of a single query's performance, so AutoSteer cannot yet handle them. We leave considerations for multi-query – and perhaps even entire workload – optimization to future work.

AutoSteer as a Tool for Human Experts. Query optimizers are highly complex software systems, as we observed firsthand in our collaboration with the PrestoDB team at Meta. Developed by over 130 software engineers, it has accrued almost 200 rewrite rules (RRs) [11]. While developers strive to make each rule applicable in general, this is impossible in practice; a rule that is helpful in one context may be harmful in another. For example, the *HashGenOptimizer* rule in PrestoDB enforces parallel generation of hash values for all joins, which improves the joining of large tables. However, the rule's overhead outweighs its performance gains for smaller tables. To address this issue, we crafted a heuristic that turned the HashGenOptimizer on or off based on the predicted input size, which resolved most of the regressions we observed in Meta's dashboard workload. It was AutoSteer that helped us improve PrestoDB's query optimizer by discovering this new heuristic, which is now being considered as a contribution to PrestoDB's upstream.

Furthermore, some rewrite rules can seem like "no-brainers" that ought to improve query performance wherever they are applied, but will surprisingly regress some queries. Since AutoSteer can automatically identify these kinds of surprising interactions like in the PrestoDB example above, it could serve as an invaluable tool for human experts in improving the design and implementation of their optimizers. On the other hand, investigating and debugging rule-based query optimizers for larger and more complex query plans that comprise tens to hundreds of relations would get increasingly more challenging. To simplify this process, one can use QO-Insight [12] – a visual tool that lets database experts explore AutoSteer's results interactively (i.e., by setting its interaction mode to *Debugging*) and supports a query- and rule-centric exploration mode. The former enables experts to analyze the potential improvements of benchmarks or individual queries, while the latter groups the performance results by hint-sets.

Integrating AutoSteer-G into other DBMSs. AutoSteer-G is easily applicable to other SQL databases. For MySQL and DuckDB, it took us less than an hour to implement the external connector. The main task is to figure out the database's syntax for toggling optimizer knobs. DuckDB has the session property `disabled_optimizers`, which is a string containing the list of disabled rules. MySQL exposes one session property per knob. For MySQL, AutoSteer found only around three effective rules because most of their changes were not exposed in the explained plan. For DuckDB, which exposes 14 knobs, we could improve the execution time of all JOB queries by 1.66% on average, but its tail is significantly enhanced, with a few queries improving by more than 10%.

To generalize, AutoSteer works best when the underlying DBMS (1) exposes a clean interface to modify the optimizer's configuration and (2) provides sufficiently detailed query plans to observe changes. Therefore, we argue that exposing more detailed optimizer statistics will help AutoSteer to detect better query plans. For example, database systems could have an option such as `EXPLAIN OPTIMIZATION <query>` returning a list or statistics describing the effective RRs. Such a feature would also help database developers see which RRs directly contribute to the final query plan.

6 CONCLUSIONS

In this paper, we introduced AutoSteer, a generic, learning-based query optimization framework that automatically steers traditional query optimizers of SQL databases. AutoSteer achieves this by extending Bao with automatically generated dynamic hint-sets, which can easily adapt to different query workloads and optimizers and lead to better plans than the manually selected static hint-sets in Bao. We have shown that our solution can be easily applied to several SQL databases and improve their query performance by up to 40% on well-known benchmarks. Furthermore, we tested AutoSteer on a real-world PrestoDB workload at Meta, where it achieved more than 20% reduction in 99% tail latency. Query optimization experts can also use AutoSteer as an interactive tool to generate insights that can be leveraged to improve existing RRs.

REFERENCES

[1] 2020. Bao for PostgreSQL. https://github.com/learnedsystems/BaoForPostgreSQL [Last Accessed: 2023/08/02].

[2] 2020. Solving Query Optimization in Presto. https://www.infoworld.com/article/3587781/solving-query-optimization-in-presto.html [Last Accessed: 2023/08/02].

[3] 2021. Applying Bao to Distributed Systems. https://rmarcus.info/blog/2021/06/17/bao-distributed.html [Last Accessed: 2023/08/02].

[4] 2021. Presto-on-Spark. https://prestodb.io/blog/2021/10/26/Scaling-with-Presto-on-Spark [Last Accessed: 2023/08/02].

[5] 2022. Bao Online Appendix. https://rm.cab/bao_appendix [Last Accessed: 2023/08/02].

[6] 2022. ML for Systems Papers. http://dsg.csail.mit.edu/mlforsystems/papers/ [Last accessed: 2023/08/02].

[7] 2022. MySQL Hints. https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_optimizer_switch [Last Accessed: 2023/08/02].

[8] 2022. PostgreSQL Hints. https://www.postgresql.org/docs/current/runtime-config-query.html [Last Accessed: 2023/08/02].

[9] 2022. PrestoDB Hints. https://prestodb.io/docs/current/optimizer/cost-based-optimizations.html [Last Accessed: 2023/08/02].

[10] 2022. SQLServer Hints. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query [Last Accessed: 2023/08/02].

[11] 2023. PrestoDB on GitHub. https://github.com/prestodb/presto [Last Accessed: 2023/08/02].

[12] Christoph Anneser, Mario Petruccelli, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, Ryan Marcus, and Alfons Kemper. 2023. QO-Insight: Inspecting Steered Query Optimizers. *Proc. VLDB Endow.* 16, 12 (2023), 3922 – 3925.

[13] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark.. In *SIGMOD Conference.* ACM, 1383–1394.

[14] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources.. In *SIGMOD Conference.* ACM, 221–230.

[15] Nicolas Bruno, Surajit Chaudhuri, and Ravishankar Ramamurthy. 2009. Power Hints for Query Optimization.. In *ICDE.* IEEE Computer Society, 469–480.

[16] Amol Deshpande and Joseph M. Hellerstein. 2002. Decoupled Query Optimization for Federated Database Systems.. In *ICDE.* IEEE Computer Society, 716–727.

[17] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[18] Peter J. Haas, Ihab F. Ilyas, Guy M. Lohman, and Volker Markl. 2009. Discovering and Exploiting Statistical Properties for Query Optimization in Relational Databases: A Survey. *Stat. Anal. Data Min.* 1, 4 (2009), 223–250.

[19] Benjamin Hilprecht and Carsten Binnig. 2022. One Model to Rule them All: Towards Zero-Shot Learning for Databases. In *CIDR.* www.cidrdb.org.

[20] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.

[21] Holger Kache, Wook-Shin Han, Volker Markl, Vijayshankar Raman, and Stephan Ewen. 2006. POP/FED: Progressive Query Optimization for Federated Queries in DB2. In *VLDB.* ACM, 1175–1178.

[22] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study.. In *SIGMOD Conference.* ACM, 1214–1227.

[23] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning.. In *CIDR.* www.cidrdb.org.

[24] Tim Kraska, Umar Farooq Minhas, Thomas Neumann, Olga Papaemmanouil, Jignesh M. Patel, Christopher Ré, and Michael Stonebraker. 2021. ML-In-Databases: Assessment and Prognosis. *IEEE Data Eng. Bull.* 44, 1 (2021), 3–10.

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[26] Zhenxiao Luo, Lu Niu, Venki Korukanti, Yutian Sun, Masha Basmanova, Yi He, Beinan Wang, Devesh Agrawal, Hao Luo, Chunxu Tang, Ashish Singh, Yao Li, Peng Du, Girish Baliga, and Maosong Fu. 2022. From Batch Processing to Real Time Analytics: Running Presto® at Scale. In *ICDE.* IEEE, 1598–1609.

[27] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical.. In *SIGMOD Conference.* ACM, 1275–1288.

[28] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *CoRR* abs/1904.03711 (2019).

[29] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration.. In *aiDM@SIGMOD.* ACM, 3:1–3:4.

[30] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc T. Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads.. In *SIGMOD Conference.* ACM, 2557–2569.

[31] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2020. Cost-Guided Cardinality Estimation: Focus Where it Matters.. In *ICDE Workshops.* IEEE, 154–157.

[32] Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032.

[33] Fatma Ozcan, Sena Nural, Pinar Koksal, Mehmet Altinel, and Asuman Dogac. 1995. A Region Based Query Optimizer Through Cascades Query Optimizer Framework. *IEEE Data Eng. Bull.* 18, 3 (1995), 30–40.

[34] Zhifei Pang, Sai Wu, Haichao Huang, Zhouzhenyan Hong, and Yuqing Xie. 2021. AQUA+: Query Optimization for Hybrid Database-MapReduce System. *Knowl. Inf. Syst.* 63, 4 (2021), 905–938.

[35] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything.. In *ICDE.* IEEE, 1802–1813.

[36] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: A Modular Query Optimizer Architecture for Big Data.. In *SIGMOD Conference.* ACM, 337–348.

[37] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.

[38] TPC-DS Benchmark 2022. TPC-DS Benchmark. https://www.tpc.org/tpcds [Last Accessed: 2022/11/27].

[39] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654.

[40] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. 2011. Query optimization for massively parallel data processing.. In *SoCC.* ACM, 12.

[41] Liqi Xu, Richard L. Cole, and Daniel Ting. 2019. Learning to Optimize Federated Queries.. In *aiDM@SIGMOD.* ACM, 2:1–2:7.

[42] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations.. In *SIGMOD Conference.* ACM, 931–944.

[43] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.

[44] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (2022), 3924–3936.

[45] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics.. In *CIDR.* www.cidrdb.org.

[46] Chi Zhang, Ryan C. Marcus, Anat Kleiman, and Olga Papaemmanouil. 2020. Buffer Pool Aware Query Scheduling via Deep Reinforcement Learning. In *AIDB@VLDB.*

[47] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc T. Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft.. In *SIGMOD Conference.* ACM, 2299–2311.

[48] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 15, 9 (2022), 1808–1821.

[49] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *VLDB J.* 21, 5 (2012), 611–636.

[50] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58.

[51] Rong Zhu, Ziniu Wu, Chengliang Chai, Andreas Pfadler, Bolin Ding, Guoliang Li, and Jingren Zhou. 2022. Learned Query Optimizer: At the Forefront of AI-Driven Databases.. In *EDBT.* OpenProceedings.org, 1–4.

PUBLICATION P3

Programming Fully Disaggregated Systems

P3.1 Synopsis

The shift towards disaggregated systems where multiple machines share their memory and compute resources over high-speed networks offers new potential for data processing. Furthermore, modern data center processors like Intel's 4th Generation Intel® Xeon® scalable processors [80] have on-die accelerators for streaming and encryption and adopt Compute Express Link™ (CXL™) [46] – a new industry standard for cache-coherent interconnects based on PCIe 5.0. CXL enables devices to share memory coherently and facilitates new data and compute placement options. Especially data-intensive applications, such as database systems, machine learning frameworks, high-performance computing, and streaming applications can greatly benefit from offloading tasks to specialized accelerators and sharing memory across different compute devices via CXL. However, several challenges make it difficult to exploit the full potential of these new technologies so that the development of basic algorithms and their optimization for disaggregated systems becomes a challenging endeavor even for experts [173]:

1. Memory devices differ in several ways, such as access bandwidth, latency, and persistence guarantees, which also depend on the accessing compute device and the hardware topology. These aspects must be kept in mind when developing and optimizing dataflow applications.
2. The availability and utilization of memory devices at runtime are often *not* known at development time but only at execution time, which requires applications to apply adaptive optimizations.

This publication proposes a new programming model that addresses the above-mentioned challenges and facilitates the sustainable development of data-intensive applications for the modern data center hardware landscape. It introduces a memory-centric view together with the concept of typed memory regions. The mapping of memory regions to memory devices is done by a runtime system, which adaptively co-optimizes data and compute placement during the application's execution.

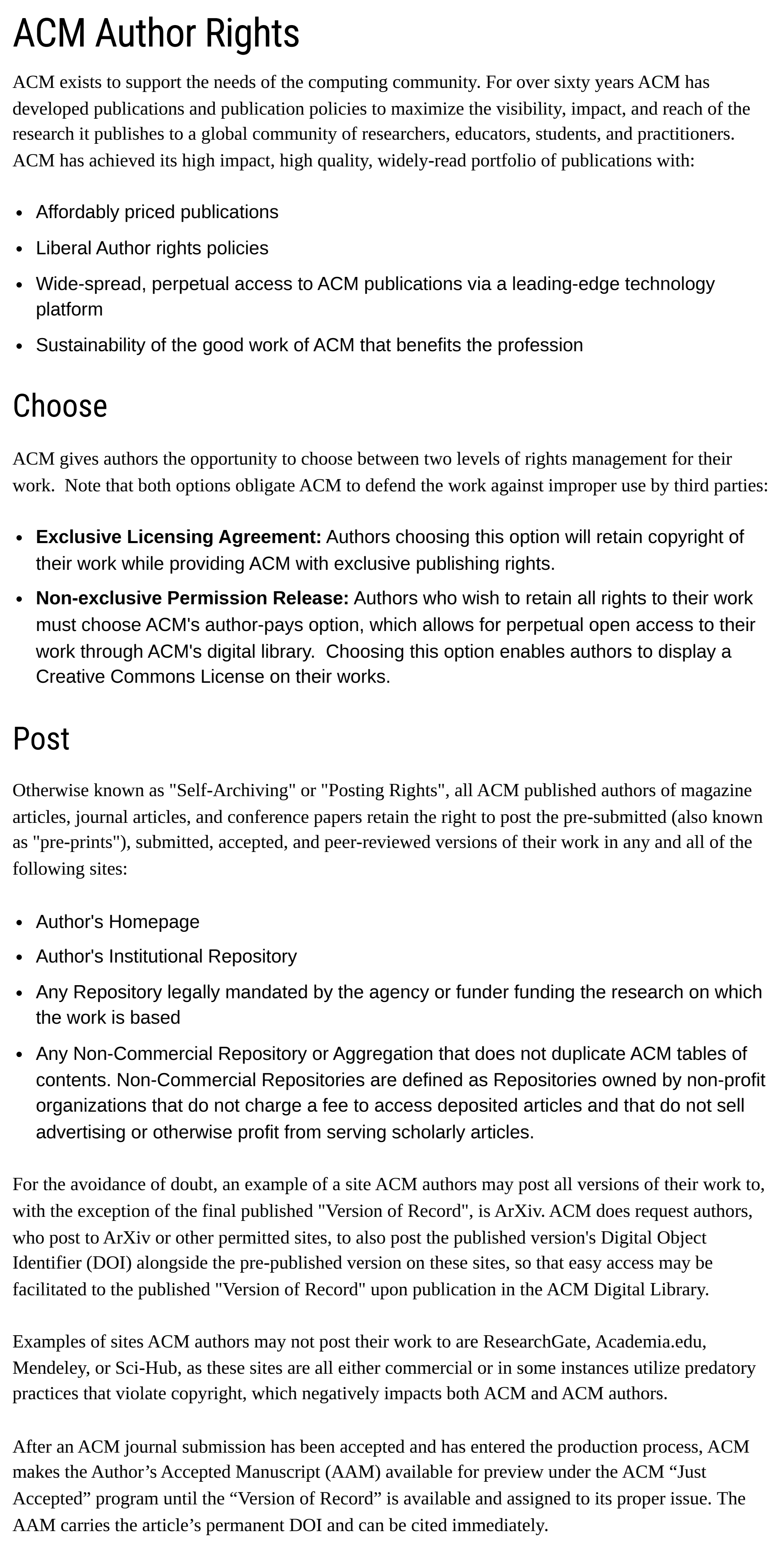
P3.2 Contributions and Publication Details

Author Contributions. Christoph Anneser contributed substantially to the content of the paper. He thoroughly reviewed the related work, which provided the foundation of the proposed programming model's blueprint. Additionally, he authored substantial parts of the publication.

Reference. Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. "Programming Fully Disaggregated Systems". In: *HotOS*. ACM, 2023, pp. 188–195.

DOI. https://doi.org/10.1145/3593856.3595889

https://authors.acm.org/author-resources/author-rights. Last accessed: 2024/06/10.



ACM Author Rights

ACM exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

Choose

ACM gives authors the opportunity to choose between two levels of rights management for their work. Note that both options obligate ACM to defend the work against improper use by third parties:

- **Exclusive Licensing Agreement:** Authors choosing this option will retain copyright of their work while providing ACM with exclusive publishing rights.
- **Non-exclusive Permission Release:** Authors who wish to retain all rights to their work must choose ACM's author-pays option, which allows for perpetual open access to their work through ACM's digital library. Choosing this option enables authors to display a Creative Commons License on their works.

Post

Otherwise known as "Self-Archiving" or "Posting Rights", all ACM published authors of magazine articles, journal articles, and conference papers retain the right to post the pre-submitted (also known as "pre-prints"), submitted, accepted, and peer-reviewed versions of their work in any and all of the following sites:

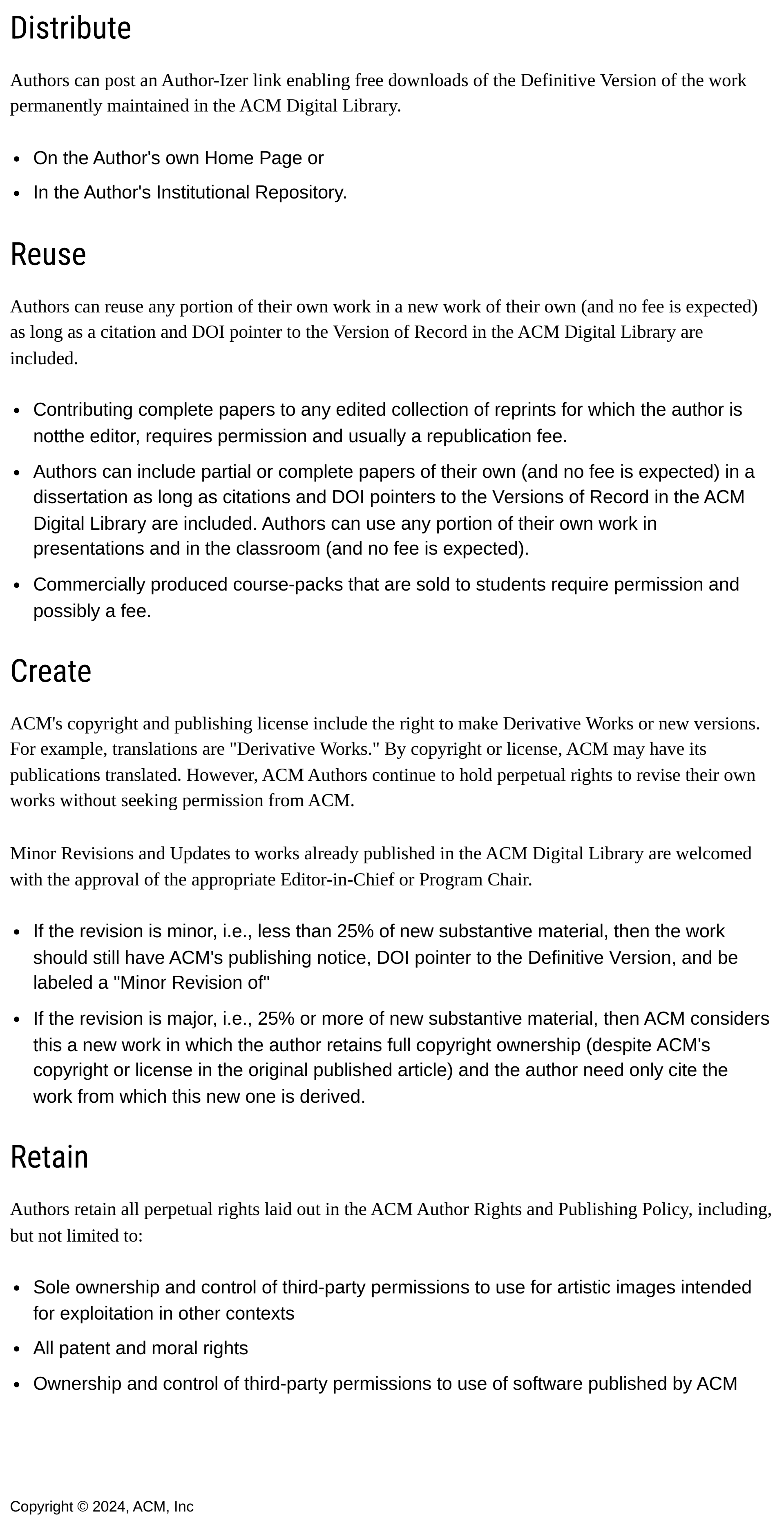
- Author's Homepage
- Author's Institutional Repository
- Any Repository legally mandated by the agency or funder funding the research on which the work is based
- Any Non-Commercial Repository or Aggregation that does not duplicate ACM tables of contents. Non-Commercial Repositories are defined as Repositories owned by non-profit organizations that do not charge a fee to access deposited articles and that do not sell advertising or otherwise profit from serving scholarly articles.

For the avoidance of doubt, an example of a site ACM authors may post all versions of their work to, with the exception of the final published "Version of Record", is ArXiv. ACM does request authors, who post to ArXiv or other permitted sites, to also post the published version's Digital Object Identifier (DOI) alongside the pre-published version on these sites, so that easy access may be facilitated to the published "Version of Record" upon publication in the ACM Digital Library.

Examples of sites ACM authors may not post their work to are ResearchGate, Academia.edu, Mendeley, or Sci-Hub, as these sites are all either commercial or in some instances utilize predatory practices that violate copyright, which negatively impacts both ACM and ACM authors.

After an ACM journal submission has been accepted and has entered the production process, ACM makes the Author’s Accepted Manuscript (AAM) available for preview under the ACM “Just Accepted” program until the “Version of Record” is available and assigned to its proper issue. The AAM carries the article’s permanent DOI and can be cited immediately.

https://authors.acm.org/author-resources/author-rights. Last accessed: 2024/06/10.



Distribute

Authors can post an Author-Izer link enabling free downloads of the Definitive Version of the work permanently maintained in the ACM Digital Library.

- On the Author's own Home Page or
- In the Author's Institutional Repository.

Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is notthe editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

Create

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.

Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"
- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

Retain

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
- Ownership and control of third-party permissions to use of software published by ACM

Programming Fully Disaggregated Systems

Christoph Anneser Lukas Vogel Ferdinand Gruber
Maximilian Bandle Jana Giceva
Technical University of Munich
firstname.lastname@in.tum.de

Abstract

With full resource disaggregation on the horizon, it is unclear what the most suitable *programming model* is that enables dataflow developers to fully harvest the potential that recent hardware developments offer. In our vision, we propose to raise the abstraction level to allow developers to primarily reason about their dataflow and the requirements that need to be met by the underlying system in a declarative fashion. Underneath, the system works with typed memory regions and uses the notion of ownership that allows for more flexible memory management across the different compute devices and the tasks mapped onto them. This requires a holistic approach that crosses multiple layers of the system stack, opening exciting systems research questions.

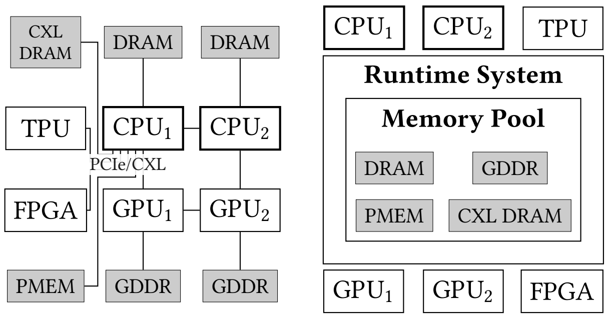
ACM Reference Format:
Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. 2023. Programming Fully Disaggregated Systems. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595889

1 Introduction

With the ever-increasing demand for data, where the datasphere volume is expected to reach 175ZB by 2025 [50], we have reached the point where moving data is the dominating cost factor in data centers [34, 45]. Cloud providers race to serve the different requirements of modern workloads better but with pressure to achieve it in a more sustainable fashion [51]. To improve efficiency, data centers have evolved to more loosely coupled software-defined racks, where they disaggregate resources over fast network interconnects [52].

However, until recently, coherent memory remained tightly coupled, and servers had to be equipped with large memory capacities to serve peak workloads reliably. This

HOTOS '23, June 22–24, 2023, Providence, RI, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595889



(a) Current Architectures **(b) Our Vision**

Figure 1: Moving from a compute-centric to a memory-centric architecture.

overprovisioning is a considerable cost (50% of Azure's servers [5] and 40% of Meta's rack costs come from memory [40]) for a resource that could not be properly pooled. The average memory utilization reported by many cloud vendors remains low, typically in the range of 50-65% [38, 56]. Therefore, data centers could reduce costs by pooling different types of memory [9, 11, 21, 57] and compute devices [6, 13, 17–19, 30, 33, 47] by connecting them with fast networks [14, 45].

However, data and compute placement within these pools significantly impacts the overall system performance. For example, non-uniform memory accesses (NUMA) can slow down algorithms by up to 3× [39]. Similarly, a naïve data placement in a heterogeneous storage landscape can reduce a database system's performance by up to 3× [59].

Moreover, today, optimal placement has become an issue even *within* single processors. For example, take the recently introduced Intel's 4th Generation Intel® Xeon® Scalable Processors – codenamed *Sapphire Rapids* [7]. They have built-in encryption, compression, streaming, and high-bandwidth memory accelerators. Its most promising feature, however, is the adoption of Compute Express Links™ (CXL™) – an industry standard for cache-coherent interconnects for processors, memory expansion, and accelerators based on PCIe 5.0, which has been adopted by companies like Intel, AMD, ARM, Samsung, and NVIDIA, amongst others [9]. CXL enables us to first *scale-up* nodes by extending their compute and memory pools with 'pluggable' compute devices and DRAM/PMem expansion cards before we have to rely on more expensive 'scale-outs' to other compute nodes that

Table 1: Memory device properties as seen from a CPU.

Name	Bw.	Lat.	Gran.	Attached	Sync	Persist.
Cache	++	++	1 B	CPU	✓	✗
HBM	++	+	64 B	CPU	✓	✗
DRAM	+	+	64 B	CPU	✓	✗
PMem	○	○	256 B	CPU	✓	✓
CXL-DRAM	○	○	64 B	PCIe	✓/✗	✓/✗
Disagg. Mem.	○	–	?	NIC	✗	✓/✗
SSD	–	–	4 KiB	PCIe	✗	✓
HDD	––	––	4 KiB	SATA	✗	✓

would require the implementation of more complex consistency protocols [38]. Furthermore, CXL-based compute devices can coherently access and cache host CPU memory, enabling new data and compute placement combinations but making optimal placement decisions much more complex, as Figure 1a shows.

We believe that in such a heterogeneous hardware landscape, existing programming models are not suitable anymore. Traditionally, a developer has to explicitly place data on a memory device and specify which accelerator performs the computation. In particular, this explicit data placement requires the developer to be aware of various memory types' different properties, as shown in Table 1. For example, to optimize applications and data placement, developers must consider access latencies, their granularities (bytes or logical blocks), and how devices are physically attached. Otherwise, they will be unable to fully unlock the potential of these exciting, emerging hardware platforms. Unsurprisingly, this topic is being discussed in several recent proposals on how to write programs for scale-out cloud systems [20, 29, 61] and how to do memory-tiering at warehouse scale [22, 40].

Recent work suggested that we should switch away from CPU- or process-centric architectures to overcome the complexity of disaggregated systems and thus allow developers to primarily focus on their application logic [22, 23, 28, 53, 54, 60]. For example, by lifting the abstraction level, Vogel et al. proposed a new framework enabling developers to get declarative control over data movement in heterogeneous, disaggregated environments [58], while HetCache co-optimizes data placement by taking different memory, compute devices, and queries into account [43].

In this paper, we ask '*what should be the appropriate programming model for implementing various dataflow frameworks in the era of full resource disaggregation.*'

2 Vision

This section presents our envisioned programming model and runtime system that would enable the writing of scalable code that leverages modern hardware with disaggregated compute and memory pools.

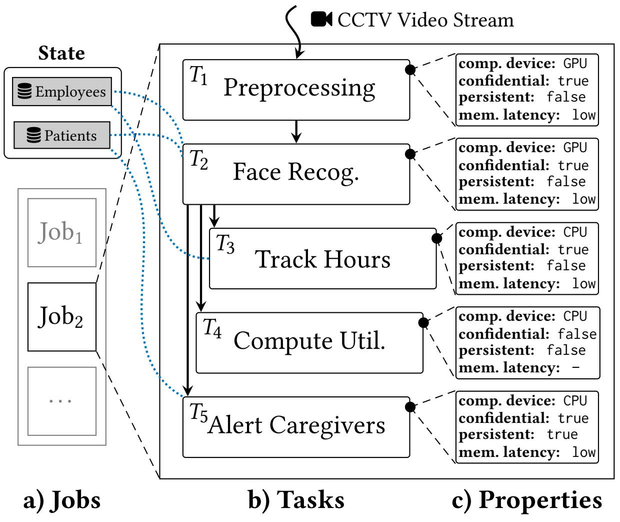


Figure 2: Example dataflow system of a hospital. Jobs consist of tasks that form a directed acyclic graph. Properties can be attached to tasks and dataflows.

2.1 Foundations

Data-intensive applications like database systems [42], machine learning frameworks [3, 4], or large-scale data analytics platforms [1, 2] can often be generalized to *dataflow systems.* To introduce the concepts of our approach, therefore, we rely on their well-known architecture, where applications launch *jobs* that consist of *tasks.* Tasks represent computational units, and connecting arrows between tasks represent the dataflow and its direction. Connected tasks form a directed acyclic graph.

Example. Figure 2 shows an example of such a job (2a) consisting of 5 tasks (2b): A hospital might have a CCTV camera recording entering and leaving persons (T_1) using GPU-accelerated face recognition connected to an employee and patient database (T_2). This information is then used to track the working hours of the employees (T_3), feed a public website displaying the utilization of the emergency ward (T_4), and alert caregivers if a confused patient exits the hospital and does not reappear after a grace period (T_5).

Declarative programming. As described in Section 1, recently introduced hardware platforms (such as Sapphire Rapids [7]) have built-in accelerators and support CXL1.1, allowing to *scale-up* one node with more accelerators and coherent memory expansion. From a programmer's perspective, implementing and optimizing applications, such as the hospital's dataflow, for modern hardware becomes increasingly complex and time-consuming. One viable option for developing high-performance dataflow applications is to introduce a new abstraction layer. This abstraction would hide the details of compute and memory devices during the application's *development* and defer the compute and memory placement decisions to *runtime*. Declarative programming concepts could allow developers to focus on the application logic (*what*) rather than *how* it is executed on a specific platform.

Common patterns. Today's dataflow applications share common patterns and often have similar requirements. For example, processing sensitive user data (i.e., T_2) requires them to implement security standards, including data encryption. Jobs and tasks could be either streamed or processed in batches. Machine learning-related tasks benefit from hardware acceleration. Implementing such requirements for each dataflow system individually and optimizing it to run on disaggregated systems is time-consuming and error-prone.

Properties for dataflow systems. Instead, a programming model should enable developers to attach common properties to their dataflow applications at different granularities. In Figure 2c), each task has some properties: While the video feed's confidentiality might depend on the country, the employee and patient database, and the tagged and cross-referenced persons are confidential. Furthermore, the video feed itself is not latency-sensitive, but since image recognition is computationally intensive, it requires low-latency memory (from the view of the GPU) to allow for real-time face recognition. The alerting task (T_5) has to store missing patients persistently, as a system crash would otherwise mean they might be forgotten. Furthermore, by attaching the property *confidentiality* to the tasks T_1–T_3 and T_5 in Figure 2, the application developer can indicate that the processed data is sensitive and must not be visible to other tasks or jobs. Another recurring pattern is the *materialization* of output data, as is the case for materialized views in database systems or the neural network's weights after training, making it another good candidate for a property being attached in dataflow systems.

Requesting properties. Current disaggregated systems introduce various memory devices, each having different properties regarding latency, bandwidth, persistency, and others (cf. Table 1). Deploying dataflow systems that serve thousands of jobs in parallel on such complex hardware landscapes with multiple physical memory devices makes efficient memory management more challenging, especially when tasks are deployed on different compute devices and the performance-critical inter-task communication is being implemented via message-passing over shared memory [41]. Therefore, the physical memory devices should be made transparent to applications that instead request memory based on the required properties. For example, the application could specify whether the allocated memory should be persistent and what latencies or bandwidths are acceptable.

Ownership Chunks of memory requested in such a way would then have a clear *owner* (i.e., a task, a job, or the whole application) allowing us to reason about the lifetime of chunks of memory and be aware of when we can re-assign it to new tasks. We could, thus, implement a reusable optimizer for various dataflow systems' data placement.

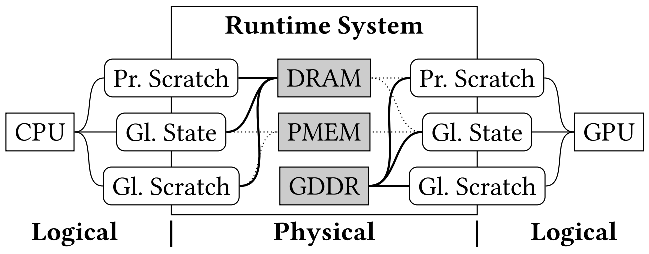


Figure 3: Mapping logical Memory Regions to physical memory depends on the compute device.

Summary. Given the challenges listed above, we need a programming model that enables application developers to utilize modern, disaggregated hardware platforms more efficiently. Such a model ideally enables the developer to attach commonly seen properties to tasks and facilitates managing disaggregated memory declaratively, which makes not only the application *development* more efficient but also the *applications* themselves by automatically co-optimizing data placement and the overall resource utilization.

2.2 Mapping to Disaggregated Systems

While the envisioned programming model abstracts from specific memory devices and instead lets the application specify what properties the requested memory must have, we need a runtime system that maps logical requests to the physical hardware in the background.

Memory devices. As shown in Table 1, various devices are already contributing to the pool of disaggregated memory, with more being added in the future. Each device has different properties concerning latency, bandwidth, coherency, and persistency. The mapping from a task's memory request and its declared properties must therefore be matched to the underlying hardware, which leads to three challenges:

(1) The 'optimal' memory device depends on the compute device executing the task *and* the type of memory accesses it performs (e.g., random vs. sequential, read- or write-intensive accesses, or access granularity). Figure 3 visualizes this problem: 'fast and local' scratch memory might preferably be DRAM when the task runs on a CPU. For tasks running on a GPU, however, GDDR provides better latency and bandwidth, although with less capacity.

(2) Tasks might share memory: The preceding task's output could become the succeeding task's input. If both tasks run on different compute devices, their shared memory must be addressable by both (e.g., via `CXL.mem`) or copied after the first task is done. Therefore, data placement depends not only on one task but also on the interaction of multiple tasks.

Table 2: Common Memory Regions.

Name	Properties	Purpose
Global State	{coherent, sync}	Syncing tasks
Global Scratch	{coherent, async}	Data exchange
Private Scratch	{noncoherent, sync}	Thread-local data

(3) Depending on how far memory is physically away, we want to expose different interfaces. In the case of *near* memory that provides low access latency, we would prefer synchronous `loads/stores` to reduce the task's execution time. If memory is 'far away', we should switch to an asynchronous interface that fetches memory in the background. For example, accessing CXL-attached memory will result in high latency comparable to accessing DRAM on a different NUMA socket. Asynchronous accesses improve the accelerator's utilization and overall throughput.

We propose three concepts to mitigate these problems:
(1) Memory regions. Since the properties of a device change depending on the task's point of view (i.e., by which compute device it is executed), we use the concept of *Memory Regions* to abstract from physical devices. A Memory Region is a *logical view* on a physical device: It guarantees some set of properties specified by a task (e.g., low latency, persistency) *relative to* the executing compute device. At runtime, the system maps the Memory Region to a physical device satisfying its properties. Memory Regions are thus *declared* and identified by their *properties*, not by their *location*, unlike traditional approaches. We group properties that are often used together and name the resulting Memory Region to express commonly used abstractions in programming – Table 2 describes three frequently used Memory Regions and Figure 4 shows how our dataflow system uses them: All threads of a task have their Private Scratch and hold a reference to a Global State and Global Scratch. Memory regions for dataflow systems for a *single* device have already been used in the past. Broom [25], for example, introduces memory regions and ownership to track lifetimes and, therefore, to remove the garbage collector. We build on this approach by generalizing memory regions to *multiple* devices.
(2) Memory ownership. To facilitate inter-task communication, we introduce the concept of *memory ownership*: Each chunk of allocated memory is either

- *exclusively* owned by a task. This applies if it is just task-local scratch space or handed over to the next task after completion. Here, consistency guarantees and memory ordering can be relaxed.
- or it *shares* the ownership with other tasks that may run concurrently. This puts additional requirements on the Memory Region, i.e., being cache-coherent or having strict memory ordering.

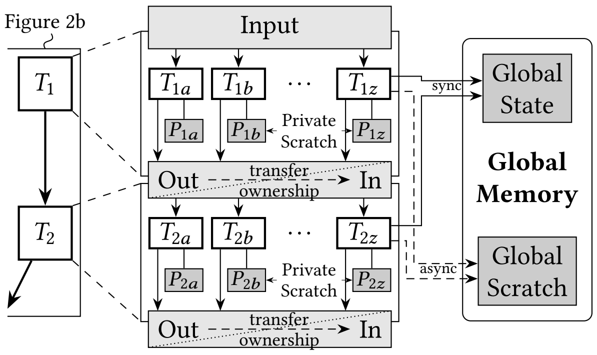


Figure 4: Tasks and Typed Memory Regions.

Note that memory being owned *exclusively* by a task does not mean it can only ever be owned by one thread of execution. As Figure 4 shows, ownership can be *transferred*, i.e., a reference to the memory chunk can be passed to the next task (the "`out`" becomes the "new `in`"), which is similar to `C++`'s move semantics. This explicit ownership model enables us to always allocate the most suitable memory per thread of execution. In contrast, in a traditional disaggregated system, users must choose placement, which increases complexity, especially as more kinds of memory become available.
(3) Access interfaces. It is beneficial to address different Memory Regions by different access modes to improve resource utilization. When accessing global memory, we might benefit from an asynchronous model where we can interleave computation with memory accesses. Memory Regions, thus, should expose different interfaces to access data.

2.3 Programming Model

After introducing the abstractions of Memory Regions, ownership, and interfaces, we now switch back to the application level and discuss integrating these concepts into the motivating dataflow example.
Runtime system. To implement the envisioned programming model, we need a runtime system that is responsible for (1) determining at runtime which physical memory device best fits each task's declared requirements, (2) allocating the Memory Regions that tasks have requested, (3) de-allocating Memory Regions after the last owning task finishes, (4) and resource-aware task scheduling.
Abstracting memory regions. As discussed in the previous section and Table 2, dataflow systems share common memory usage and access patterns and have similar requirements. The following Memory Regions should be pre-defined by the programming model:

- **Private Scratch** is memory local to each thread of the task. Since it is not shared, it may have relaxed coherence guarantees. As demonstrated in Figure 4, each task's thread has its own private scratch space ($P_{1a} \dots P_{1z}$), which is only

Table 3: How applications may use memory regions.

	Priv. Scratch	Glob. State	Glob. Scratch
DBMS	operator state (hashtables, ...)	synchronization (latches, ...)	(temp) indexes, caches
ML/AI	model training state	metadata, worker state	input data, cached transf. data
HPC	node-local working mem.	job metadata, node states	object/blob storage
Streaming	cache/buffer (send, recv.)	cluster/worker state	result/data cache

alive during its execution. It stores intermediate results not part of the task's output. Private scratch is visible to only one thread and not transferable.

- **Global State** is memory global to the application and shared between tasks to synchronize tasks and threads. It, thus, has to provide strict coherence guarantees and strong memory ordering but is expected to be slow as it has to be accessible from all compute devices.
- **Global Scratch** can pass data between tasks that are not connected. Passing data in such a way is helpful when two tasks do not depend on each other but may use an intermediate result from another task (such as a bloom filter) to speed up its processing. The Global Scratch exposes an asynchronous interface as threads should not block on `load/store` on slow memory.

Moving data. Data will be passed between tasks via Global Scratch memory or to the next task in the dataflow. For the second case, we need a concept of input and output as shown in Figure 4. The input consists of the data set the current task should operate on and is generated by the preceding task. The output is the data the task produces, i.e., the next task's input. Input and output can be modeled as Memory Regions, which the active task owns. Thanks to our concept of memory ownership, the output memory of the preceding task can directly become the input memory of the next task if it is addressable by the compute devices of both tasks. The runtime system allocates input and output memory so that handover is just a memory ownership transfer, and physical data movement is minimized.

2.4 Mapping Application Types

Different application types can be easily mapped to our proposed architecture. We illustrate four types in Table 3 and describe two in more detail.

Database systems internally represent queries as relational operator trees where the output of one operator becomes the input of the following operator, which nicely maps onto dataflow systems. Each operator must keep track of its state in private scratch (e.g., a group hash table for aggregation operators) and synchronize with other concurrently running operators via latches in the global state. Furthermore, some operators can re-use (transient) results of earlier operators stored in the global scratch space (e.g., a hash join might re-use a hash index created by an aggregation operator).

AI/ML applications must first transform and preprocess the input data (e.g., parsing, sampling, and feature extraction) before training a model on accelerators. This can also be modeled as a dataflow system, as demonstrated by Cachew [26]. Cachew stores the transformed data in a cache (global scratch) and uses a dispatcher accessing worker states (global state) to assign tasks running on accelerators (private scratch).

3 Discussion

Our proposed memory-centric programming model radically changes how applications and developers interact with memory in the disaggregated cloud. They should no longer have to deal with the complexity of handling different memory devices, which is further complicated by emerging technologies like CXL. Instead, memory should be requested declaratively based on desired properties like latency or bandwidth.

The way forward. Our programming model requires a runtime system (RTS) that should abstract away hardware-specific details of memory accesses and does the bookkeeping regarding ownership and the lifetime of regions. Furthermore, the RTS needs to make the deployment decisions on mapping tasks and memory onto the disaggregated resources. To make this come true, we need to address several challenges for which we begin the discussion in this section:

(1) Who oversees the management and utilization of the disaggregated resources?
(2) How do we make optimal deployment decisions?
(3) How to enforce deployment policies at runtime?
(4) Where should the RTS/control plane be placed?
(5) What support from the underlying system stack is needed on the critical data path?
(6) How can we make our concept of memory regions easy to use in general-purpose programming languages?
(7) How can we combine declarative and imperative principles in one programming model?
(8) What are the potential limitations of our approach?

Implementing the programming model and the runtime system is a non-trivial endeavor that calls for a holistic approach, crossing multiple layers of the systems stack. In the following paragraphs, we discuss how we can address these challenges and use prior work from the systems, compiler, and database communities to set the stage for our proposal.

Challenges 1-3: What is required from the RTS? Our RTS needs to manage memory resources –typed with different properties (cf. Figure 2)– of multiple machines or even cluster wide. Jobs and tasks request memory regions from the RTS that it then maps to physical memory (cf. Figure 3).

The allocation of memory regions goes beyond the capabilities of already known single-host memory management [46] and existing distributed managed runtimes [2]. With multiple, coherently accessible memory tiers, the RTS must manage memory regions based on pages or objects and their placement. Both approaches are actively researched by the systems community and have different implications on performance and scalability [48, 62]. To optimize the placement of memory regions, we can build on recent work that used pointer tagging to track the hotness of pages or objects and to implement remotable pointers that either point to objects in local or in remote memory (pointer swizzling) [37, 40, 48, 62].

Our RTS must also schedule and map tasks to different types of devices using cost models that consider topology and access paths [49] to optimize for concurrently running jobs. Therefore, it must know or predict the resource utilization of memory and compute devices. Scheduling also requires reusing results to avoid unnecessary copying [60] and lowering to different types of hardware. Thankfully, such cost models for optimization and lowering tasks to multiple devices are already well-known in the database and compiler communities [24, 35, 54]. Furthermore, new approaches using MLIR, such as LingoDB [31], have shown that it is feasible to provide the compiler with various statistics to make cost-based transformations and data and task placement decisions.

Challenges 4-5: What layer supports the RTS memory deployment? The RTS provides memory regions, but without some levels of abstraction, the complexity of handling disaggregated memory is just moved to the application. In our vision, the core responsibility of the operating system (OS) is mapping RTS-requested memory into the address space of our proposed tasks. The *processor-centric* design could lead to host congestion [10] and become a bottleneck in the future and the concept of memory ownership of today's OSes are not suitable anymore [53] because ownership is now globally managed by the RTS [23]. Thus, in the disaggregated cloud, OSes should be built *memory-centric*, like our jobs and tasks. Of course, this is a simplification of OS memory management, leaving out many aspects (e.g., the memory the OS requires for managing devices). We are not the first to propose such a shift in OS design and can rely on previous research [23, 53].

Challenges 6-7: How to get the developer on board and ease adoption? Until now, there is no consensus on handling the ownership and lifetime of memory objects and streams across different devices, and popular AI/ML frameworks handle them differently [8]. Furthermore, developers should not face the complexity of modern memory technology [61] and instead should request memory declaratively. This declarative approach is a paradigm shift for many programming languages (PL), where memory is managed manually or by a language runtime [46]. Consequently, the PL should either (1) allow programmers to provide different versions of code targeting different memory types or (2) provide a central compilation service that JIT compiles the programmer's declarative description of memory accesses. The latter –a mixture of declarative and imperative code– is actively researched [44, 55] and could be adapted for our approach.

Challenge 8: What are the potential limitations? Raising the abstraction level leads to new questions our approach does not yet solve: (1) How can we *debug*, *profile*, and *optimize* dataflow applications with multiple abstraction layers for performance when the runtime system hides performance-relevant details? Fortunately, the system community has already shown that – despite intricacies and difficulties – debugging [32] and profiling [16] across multiple abstraction layers is possible. (2) Legacy applications might not adopt a new programming model requiring significant source code modifications. A similar approach has been recently proposed by the Mojo programming language, which is a superset of Python and uses declarative programming to enable hardware acceleration with GPUs and FPGAs for AI and ML workloads. (3) How to mitigate faults and report them to the user? Network errors, corrupted memory, and planned and unplanned node faults such as kernel updates or power outages are common in data centers having thousands of interconnected compute and memory devices. If not handled properly, failures may lead to data loss and force applications to stop and restart. Therefore, our programming model and its runtime system must implement suitable mechanisms that guarantee fault tolerance *and* are compute- and storage-efficient. Several ideas have been recently discussed by the systems community, including replication-based approaches [12, 27, 53] and the striping of memory pages across multiple memory nodes [36]. The runtime system could also implement a combination of erasure-coding, one-sided remote memory accesses and compaction, and off-loadable parity calculations, as it is used by Carbink, a state-of-the-art approach for fault-tolerant far memory [62].

Conclusion. With our envisioned programming model, application developers can fully utilize emerging new hardware more easily without being concerned about the specifics of the underlying hardware or the complexity of memory coherency models. Building a distributed RTS is a complex task requiring support from the systems, compiler, and language community (cf. Legion [15]). However, the advantages of our proposal in terms of complexity reduction, resource utilization, and flexibility will make this effort worthwhile.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was funded by the German Research Foundation (DFG) within the SPP2037 under grant no. Ke 401/22-2.

References

[1] 2013. PrestoDB. https://prestodb.io/ Last Accessed: 2023/05/17.

[2] 2014. Apache Spark. https://spark.apache.org/ Last Accessed: 2023/05/17.

[3] 2015. Tensorflow. https://www.tensorflow.org/ Last Accessed: 2023/05/17.

[4] 2016. PyTorch. https://pytorch.org/ Last Accessed: 2023/05/17.

[5] 2020. CXL and GEN-Z iron out a coherent interconnect strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/ Last Accessed: 2023/05/17.

[6] 2020. Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/ Last Accessed: 2023/05/17.

[7] 2023. 4th Generation Intel® Xeon® Scalable Processors. https://ark.intel.com/content/www/us/en/ark/products/series/228622/4th-generation-intel-xeon-scalable-processors.html Last Accessed: 2023/05/17.

[8] 2023. Apache Arrow. https://github.com/apache/arrow/pull/34972 Last Accessed: 2023/05/17.

[9] 2023. Compute Express Links. https://www.computeexpresslink.org/download-the-specification Last Accessed: 2023/05/17.

[10] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David E. Culler, and Amin Vahdat. 2022. Understanding host interconnect congestion. In *HotNets*. ACM, 198–204.

[11] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *USENIX Annual Technical Conference*. USENIX Association, 775–787.

[12] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *EuroSys*. ACM, 14:1–14:16.

[13] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[14] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, István Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. 2020. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *SIGCOMM*. ACM, 782–797.

[15] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *SC*. IEEE/ACM, 66.

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

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95.

[18] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *FAST*. USENIX Association, 29–41.

[19] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *MICRO*. IEEE Computer Society, 7:1–7:13.

[20] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. 2021. New Directions in Cloud Programming. In *CIDR*. www.cidrdb.org.

[21] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*. USENIX Association, 401–414.

[22] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David E. Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *ASPLOS (3)*. ACM, 727–741.

[23] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan S. Milojicic. 2015. Beyond Processor-centric Operating Systems. In *HotOS*. USENIX Association.

[24] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. 1996. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *VLDB*. Morgan Kaufmann, 378–389.

[25] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *HotOS*. USENIX Association.

[26] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing as a Service. In *USENIX Annual Technical Conference*. USENIX Association, 689–706.

[27] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *NSDI*. USENIX Association, 649–667.

[28] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 417–433.

[29] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*. ACM, 1–12.

[31] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401.

[32] Timo Kersten and Thomas Neumann. 2020. On another level: how to debug compiling query engines. In *DBTest@SIGMOD*. ACM, 2:1–2:6.

[33] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *SOSP*. ACM, 756–771.

[34] Peter M. Kogge and John Shalf. 2013. Exascale Computing Trends: Adjusting to the "New Normal"' for Computer Architecture. *Comput. Sci. Eng.* 15, 6 (2013), 16–26.

[35] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020).

[36] Youngmoon Lee, Hassan Al Maruf, Mosharaf Chowdhury, and Kang G. Shin. 2019. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. *CoRR* abs/1910.09727 (2019).

[37] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. IEEE Computer Society, 185–196.

[38] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*.

[39] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR*. www.cidrdb.org.

[40] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *ASPLOS (3)*. ACM, 742–755.

[41] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455.

[42] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[43] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU-acceleration for Memory-Efficient Analytics. In *CIDR*. www.cidrdb.org.

[44] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (2018), 1002–1015.

[45] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Muhammad Mukarram Bin Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve D. Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM*. ACM, 66–85.

[46] Paula Pufek, H. Grgic, and Branko Mihaljevic. 2019. Analysis of Garbage Collection Algorithms and Memory Management in Java. In *MIPRO*. IEEE, 1677–1682.

[47] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*. IEEE Computer Society, 13–24.

[48] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *OSDI*. USENIX Association, 315–332.

[49] Enrico Russo, Maurizio Palesi, Salvatore Monteleone, Davide Patti, Giuseppe Ascia, and Vincenzo Catania. 2022. MEDEA: A Multi-objective Evolutionary Approach to DNN Hardware Mapping. In *DATE*. IEEE, 226–231.

[50] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. 2018. The digitization of the world from edge to core. *Framingham: International Data Corporation* 16 (2018).

[51] Ian Schneider. 2022. Building low-carbon computer systems: when does carbon diverge from cost? [Talk]. https://youtu.be/W7uTbxCxmPg Last Accessed: 2023/05/17.

[52] Boris M Shabanov and Oleg I Samovarov. 2019. Building the software-defined data center. *Programming and Computer Software* 45 (2019), 458–466.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2019. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *USENIX ATC*. USENIX Association.

[54] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. 2022. Towards a fully disaggregated and programmable data center. In *APSys*. ACM, 18–28.

[55] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endow.* 15, 5 (2022), 1119–1131.

[56] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *EuroSys*. ACM, 30:1–30:14.

[57] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *VLDB J.* 29, 6 (2020), 1223–1241.

[58] Lukas Vogel, Daniel Ritter, Danica Porobic, Pınar Tözün, Tianzheng Wang, and Alberto Lerner. 2023. Data Pipes: Declarative Control over Data Movement. In *CIDR*. www.cidrdb.org.

[59] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2662–2675.

[60] Stephanie Wang, Benjamin Hindman, and Ion Stoica. 2021. In reference to RPC: it's time to add distributed memory. In *HotOS*. ACM, 191–198.

[61] Yiying Zhang, Ardalan Amiri Sani, and Guoqing Harry Xu. 2021. User-defined cloud. In *HotOS*. ACM, 33–40.

[62] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *OSDI*. USENIX Association, 55–71.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *OSDI*. USENIX Association, 2016, pp. 265–283.

[2] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek R. Narasayya. "AutoAdmin: Self-Tuning Database Systems Technology". In: *IEEE Data Eng. Bull.* 29.3 (2006), pp. 7–15.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. "Automated Selection of Materialized Views and Indexes in SQL Databases". In: *VLDB*. Morgan Kaufmann, 2000, pp. 496–505.

[4] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. "Remote regions: a simple abstraction for remote memory". In: *USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 775–787.

[5] Adnan Alhomssi and Viktor Leis. "Contention and Space Management in B-Trees". In: *CIDR*. 2021, pp. 26–37.

[6] AMD. *4th Generation AMD EPYC® Processors*. 2023. URL: `https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html`. Last accessed: 2023/12/05.

[7] Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, and Alfons Kemper. "The Case for Hybrid Succinct Data Structures". In: *EDBT*. 2020, pp. 391–394.

[8] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. "Adaptive Hybrid Indexes". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 2022, pp. 1626–1639.

[9] Christoph Anneser, Mario Petruccelli, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, Ryan Marcus, and Alfons Kemper. "QO-Insight: Inspecting Steered Query Optimizers". In: *PVLDB* 16.12 (2023), pp. 3922–3925.

[10] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. "AutoSteer: Learned Query Optimization for Any SQL Database". In: *PVLDB* 16.12 (2023), pp. 3515–3527.

[11] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. "Programming Fully Disaggregated Systems". In: *HotOS*. ACM, 2023, pp. 188–195.

[12] *Applying Bao to Distributed Systems*. 2021. URL: `https://rmarcus.info/blog/2021/06/17/bao-distributed.html`. Last accessed: 2023/08/02.

[13] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *SIGMOD*. ACM, 2015, pp. 1383–1394.

[14] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. "Amazon Redshift Re-invented". In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2205–2217.

[15] *AutoSteer Prototype Implementation*. 2022. URL: `https://github.com/IntelLabs/Auto-Steer`. Last accessed: 2023/12/14.

[16] Ron Avnur and Joseph M. Hellerstein. "Eddies: Continuously Adaptive Query Processing". In: *SIGMOD*. ACM, 2000, pp. 261–272.

[17] John Aycock. "A Brief History of Just-In-Time". In: *ACM Comput. Surv.* 35.2 (2003), pp. 97–113.

[18] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, István Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. "Sirius: A Flat Datacenter Network with Nanosecond Optical Switching". In: *SIGCOMM*. ACM, 2020, pp. 782–797.

[19] *Bao for PostgreSQL*. 2020. URL: `https://github.com/learnedsystems/BaoForPostgreSQL`. Last accessed: 2023/08/02.

[20] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. "Cost-Model Oblivious Database Tuning with Reinforcement Learning". In: *DEXA (1)*. Springer, 2015, pp. 253–268.

[21] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing Locality and Independence with Logical Regions". In: *SC*. IEEE/ACM, 2012, p. 66.

[22] Rudolf Bayer. “The Universal B-Tree for Multidimensional Indexing: general Concepts”. In: *WWCA*. Vol. 1274. Lecture Notes in Computer Science. Springer, 1997, pp. 198–209.

[23] Rudolf Bayer and Edward M. McCreight. “Organization and Maintenance of Large Ordered Indices”. In: *Acta Informatica* 1 (1972), pp. 173–189.

[24] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. “Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources”. In: *SIGMOD*. ACM, 2018, pp. 221–230.

[25] Philip A. Bernstein, Michael L. Brodie, Stefano Ceri, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph M. Hellerstein, H. V. Jagadish, Michael Lesk, David Maier, Jeffrey F. Naughton, Hamid Pirahesh, Michael Stonebraker, and Jeffrey D. Ullman. “The Asilomar Report on Database Research”. In: *SIGMOD Rec.* 27.4 (1998), pp. 74–80.

[26] Timo Bingmann. *STX B+-tree*. 2007. URL: `http://panthema.net/2007/stx-btree/`. Last accessed: 2021/09/15.

[27] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (1970), pp. 422–426.

[28] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. “P4: Programming Protocol-Independent Packet Processors”. In: *Comput. Commun. Rev.* 44.3 (2014), pp. 87–95.

[29] Sebastian Breß. “Why it is time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS”. In: *PVLDB* 6.12 (2013), pp. 1398–1403.

[30] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. “GPU-Accelerated Database Systems: Survey and Open Challenges”. In: *Trans. Large Scale Data Knowl. Centered Syst.* 15 (2014), pp. 1–35.

[31] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. “Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems”. In: *SIGMOD*. ACM, 2022, pp. 617–630.

[32] *C++ Cuckoo Filter*. URL: `https://github.com/efficient/cuckoofilter`. Last accessed: 2021/03/01.

[33] *C++ HopscotchMap*. URL: `https://github.com/Tessil/hopscotch-map`. Last accessed: 2021/03/01.

[34] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. "POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database". In: *FAST*. USENIX Association, 2020, pp. 29–41.

[35] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook". In: *USENIX*. 2020, pp. 209–223.

[36] Jared Casper and Kunle Olukotun. "Hardware Acceleration of Database Operations". In: *FPGA*. ACM, 2014, pp. 151–160.

[37] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A Cloud-Scale Acceleration Architecture". In: *MICRO*. IEEE Computer Society, 2016, 7:1–7:13.

[38] Hokeun Cha, Xiangpeng Hao, Tianzheng Wang, Huanchen Zhang, Aditya Akella, and Xiangyao Yu. "Blink-hash: An Adaptive Hybrid Index for In-Memory Time-Series Databases". In: *PVLDB* 16.6 (2023), pp. 1235–1248.

[39] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing". In: *OOPSLA*. ACM, 2005, pp. 519–538.

[40] Surajit Chaudhuri and Vivek R. Narasayya. "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server". In: *VLDB*. Morgan Kaufmann, 1997, pp. 146–155.

[41] Surajit Chaudhuri and Vivek R. Narasayya. "AutoAdmin "What-if" Index Analysis Utility". In: *SIGMOD*. ACM Press, 1998, pp. 367–378.

[42] Surajit Chaudhuri and Vivek R. Narasayya. "Self-Tuning Database Systems: A Decade of Progress". In: *VLDB*. ACM, 2007, pp. 3–14.

[43] Surajit Chaudhuri and Gerhard Weikum. "Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System". In: *VLDB*. Morgan Kaufmann, 2000, pp. 1–10.

[44] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. "The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database". In: *PVLDB* 9.13 (2016), pp. 1401–1412.

[45] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. “HotRing: A Hotspot-Aware In-Memory Key-Value Store”. In: *FAST*. USENIX Association, 2020, pp. 239–252.

[46] *Compute Express Link*. URL: `https://www.computeexpresslink.org/download-the-specification`. Last accessed: 2023/12/02.

[47] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. “DiffLex: A High-Performance, Memory-Efficient and NUMA-Aware Learned Index using Differentiated Management”. In: *ICPP*. ACM, 2023, pp. 62–71.

[48] *CXL and GEN-Z Iron out a Coherent Interconnect Strategy*. 2020. URL: `https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/`. Last accessed: 2023/05/17.

[49] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. “Automatic SQL Tuning in Oracle 10g”. In: *VLDB*. Morgan Kaufmann, 2004, pp. 1098–1109.

[50] Sabyasachi Dash, Sushil Kumar Shakyawar, Mohit Sharma, and Sandeep Kaushik. “Big data in healthcare: management, analysis and future prospects”. In: *J. Big Data* 6 (2019), p. 54.

[51] *Data Processing Units*. 2020. URL: `https://www.nvidia.com/en-us/networking/products/data-processing-unit/`. Last accessed: 2023/05/17.

[52] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51.1 (2008), pp. 107–113.

[53] Amol Deshpande. “An Initial Study of Overheads of Eddies”. In: *SIGMOD Rec.* 33.1 (2004), pp. 44–49.

[54] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. “Adaptive Query Processing”. In: *Found. Trends Databases* 1.1 (2007), pp. 1–140.

[55] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. “AI Meets AI: Leveraging Query Executions to Improve Index Recommendations”. In: *SIGMOD*. ACM, 2019, pp. 1241–1258.

[56] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. “FaRM: Fast Remote Memory”. In: *NSDI*. USENIX Association, 2014, pp. 401–414.

[57] *Explain Statements in Amazon Redshift*. URL: `https://docs.aws.amazon.com/redshift/latest/dg/r_EXPLAIN.html`. Last accessed: 2023/11/30.

[58] *Explain Statements in DuckDB.* URL: `https://duckdb.org/docs/guides/meta/explain.html`. Last accessed: 2023/11/30.

[59] *Explain Statements in Hyper.* URL: `https://tableau.github.io/hyper-db/docs/sql/command/explain/`. Last accessed: 2023/11/30.

[60] *Explain Statements in MySQL.* URL: `https://dev.mysql.com/doc/refman/8.0/en/using-explain.html`. Last accessed: 2023/11/30.

[61] *Explain Statements in PostgreSQL.* URL: `https://www.postgresql.org/docs/current/using-explain.html`. Last accessed: 2023/11/30.

[62] *Explain Statements in PrestoDB.* URL: `https://prestodb.io/docs/current/sql/explain.html`. Last accessed: 2023/11/30.

[63] *Explain Statements in Snowflake.* URL: `https://docs.snowflake.com/en/sql-reference/sql/explain`. Last accessed: 2023/11/30.

[64] *Explain Statements in SparkSQL.* URL: `https://spark.apache.org/docs/latest/sql-ref-syntax-qry-explain.html`. Last accessed: 2023/11/30.

[65] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. "In-memory database acceleration on FPGAs: a survey". In: *VLDB J.* 29.1 (2020), pp. 33–59.

[66] Martin R. Frank, Edward Omiecinski, and Shamkant B. Navathe. "Adaptive and Automated Index Selection in RDBMS". In: *EDBT*. Springer, 1992, pp. 277–292.

[67] Florian Funke, Alfons Kemper, and Thomas Neumann. "Compacting Transactional Data in Hybrid OLTP&OLAP Databases". In: *PVLDB* 5.11 (2012). DOI: `10.14778/2350229.2350258`.

[68] David Gay and Alexander Aiken. "Memory Management with Explicit Regions". In: *PLDI*. ACM, 1998, pp. 313–323.

[69] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. "SALI: A Scalable Adaptive Learned Index Framework based on Probability Models". In: *CoRR* abs/2308.15012 (2023).

[70] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael Isard. "Broom: Sweeping Out Garbage Collection from Big Data Systems". In: *HotOS*. USENIX Association, 2015.

[71] Anastasios Gounaris, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. "Adaptive Query Processing: A Survey". In: *BNCOD*. Vol. 2405. Lecture Notes in Computer Science. Springer, 2002, pp. 11–25.

[72] Goetz Graefe. "The Cascades Framework for Query Optimization". In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29.

[73] Goetz Graefe and David J. DeWitt. "The EXODUS Optimizer Generator". In: *SIGMOD*. ACM Press, 1987, pp. 160–172.

[74] Goetz Graefe and William J. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search". In: *ICDE*. IEEE Computer Society, 1993, pp. 209–218.

[75] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A. Shah. "Adaptive Query Processing: Technology in Evolution". In: *IEEE Data Eng. Bull.* 23.2 (2000), pp. 7–18.

[76] John Hennessy and David Patterson. *ACM Turing Lecture*. 2018. URL: `https://www.acm.org/hennessy-patterson-turing-lecture`. Last accessed: 2023/12/14.

[77] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. "TCUDB: Accelerating Database with Tensor Processors". In: *SIGMOD*. ACM, 2022, pp. 1360–1374.

[78] Nina C. Hubig, Linnea Passing, Maximilian E. Schüle, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. "HyPerInsight: Data Exploration Deep Inside HyPer". In: *CIKM*. ACM, 2017, pp. 2467–2470.

[79] Clive Humby. *Data is the New Oil*. 2006. URL: `https://ana.blogs.com/maestros/2006/11/data_is_the_new.html`. Last accessed: 2023/12/01.

[80] Intel. *4th Generation Intel® Xeon® Scalable Processors*. 2023. URL: `https://ark.intel.com/content/www/us/en/ark/products/series/228622/4th-generation-intel-xeon-scalable-processors.html`. Last accessed: 2023/12/01.

[81] *ISO/IEC 9075-2*. Standard. June 2023.

[82] Guy Joseph Jacobson. "Succinct static data structures". PhD thesis. Carnegie Mellon University, 1988.

[83] Matthias Jarke and Jürgen Koch. "Query Optimization in Database Systems". In: *ACM Comput. Surv.* 16.2 (1984), pp. 111–152.

[84] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *ISCA*. ACM, 2017, pp. 1–12.

[85] Michael Jungmair, André Kohn, and Jana Giceva. “Designing an Open Framework for Query Optimization and Compilation”. In: *PVLDB* 15.11 (2022), pp. 2389–2401.

[86] Kaan Kara, Jana Giceva, and Gustavo Alonso. “FPGA-based Data Partitioning”. In: *SIGMOD Conference.* ACM, 2017, pp. 433–445.

[87] Alfons Kemper. *Datenbanksysteme - Eine Einführung, 10. Auflage.* De Gruyter Studium. de Gruyter Oldenbourg, 2015.

[88] 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.

[89] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. “LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism”. In: *SOSP.* ACM, 2021, pp. 756–771.

[90] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. “Learned Cardinality Estimation: An In-depth Study”. In: *SIGMOD.* ACM, 2022, pp. 1214–1227.

[91] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. “Learned Cardinalities: Estimating Correlated Joins with Deep Learning”. In: *CIDR.* www.cidrdb.org, 2019.

[92] Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins”. In: *EDBT.* OpenProceedings.org, 2020, pp. 347–358.

[93] Peter M. Kogge and John Shalf. “Exascale Computing Trends: Adjusting to the “New Normal” for Computer Architecture”. In: *Comput. Sci. Eng.* 15.6 (2013), pp. 16–26.

[94] Ron Kohavi, Neal J Rothleder, and Evangelos Simoudis. “Emerging Trends in Business Analytics”. In: *Communications of the ACM* 45.8 (2002), pp. 45–48.

[95] André Kohn, Viktor Leis, and Thomas Neumann. “Adaptive Execution of Compiled Queries”. In: *ICDE.* IEEE Computer Society, 2018, pp. 197–208.

[96] Artem Kroviakov, Petr Kurapov, Christoph Anneser, and Jana Giceva. “Heterogeneous Intra-Pipeline Device-Parallel Aggregations”. In: *DaMoN.* ACM, 2024, pp. 1–10.

[97] Philip Laird. “Dynamic Optimization”. In: *ML*. Morgan Kaufmann, 1992, pp. 263–272.

[98] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. “The Vertica Analytic Database: C-Store 7 Years Later”. In: *PVLDB* 5.12 (2012), pp. 1790–1801.

[99] Viktor Leis, Alfons Kemper, and Thomas Neumann. “The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases”. In: *ICDE*. Vol. 13. 2013, pp. 38–49.

[100] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. “Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark”. In: *VLDB J.* 27.5 (2018), pp. 643–668.

[101] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. “The ART of Practical Synchronization”. In: *DaMoN*. 2016, pp. 1–8.

[102] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. “Identifying Hot and Cold Data in Main-Memory Databases”. In: *ICDE*. IEEE. 2013, pp. 26–37.

[103] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. “Pond: CXL-Based Memory Pooling Systems for Cloud Platforms”. In: *ASPLOS*. 2023.

[104] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. “Adaptively Reordering Joins during Query Execution”. In: *ICDE*. IEEE Computer Society, 2007, pp. 26–35.

[105] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. “Algorithmic Improvements for Fast Concurrent Cuckoo Hashing”. In: *EuroSys*. 2014, pp. 1–14.

[106] Yongping Luo, Peiquan Jin, Zhaole Chu, Xiaoliang Wang, Yigui Yuan, Zhou Zhang, Yun Luo, Xufei Wu, and Peng Zou. “Morphtree: a polymorphic main-memory learned index for dynamic workloads”. In: *VLDB J.* (2023), pp. 1–20.

[107] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. “Query-based Workload Forecasting for Self-Driving Database Management Systems”. In: *SIGMOD*. ACM, 2018, pp. 631–645.

[108] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. “Active Learning for ML Enhanced Database Systems”. In: *SIGMOD*. ACM, 2020, pp. 175–191.

[109] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. "MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems". In: *SIGMOD*. ACM, 2021, pp. 1248–1261.

[110] Maria Malik and Houman Homayoun. "Big Data on Low Power Cores: Are Low Power Embedded Processors a good fit for the Big Data Workloads?" In: *2015 33rd IEEE International Conference on Computer Design (ICCD)*. 2015, pp. 379–382. DOI: `10.1109/ICCD.2015.7357128`.

[111] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. "Bao: Making Learned Query Optimization Practical". In: *SIGMOD*. ACM, 2021, pp. 1275–1288.

[112] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. "Neo: A Learned Query Optimizer". In: *CoRR* abs/1904.03711 (2019).

[113] Ryan Marcus and Olga Papaemmanouil. "Deep Reinforcement Learning for Join Order Enumeration". In: *aiDM@SIGMOD*. ACM, 2018, 3:1–3:4.

[114] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. "TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory". In: *ASPLOS (3)*. ACM, 2023, pp. 742–755.

[115] *Micron Memory Expansion Module*. 2023. URL: `https://investors.micron.com/news-releases/news-release-details/micron-launches-memory-expansion-module-portfolio-accelerate-cxl`. Last accessed: 2023/12/08.

[116] Ralf Mikut and Markus Reischl. "Data mining tools". In: *Wiley interdisciplinary reviews: data mining and knowledge discovery* 1.5 (2011), pp. 431–443.

[117] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[118] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc T. Friedman, and Alekh Jindal. "Steering Query Optimizers: A Practical Take on Big Data Workloads". In: *SIGMOD*. ACM, 2021, pp. 2557–2569.

[119] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. "Cost-Guided Cardinality Estimation: Focus Where it Matters". In: *ICDE Workshops*. IEEE, 2020, pp. 154–157.

[120] Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. “Flow-Loss: Learning Cardinality Estimates That Matter”. In: *PVLDB* 14.11 (2021), pp. 2019–2032.

[121] Thomas Neumann and Michael J Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *CIDR*. 2020.

[122] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet D. Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. “Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey”. In: *Artif. Intell. Rev.* 52.1 (2019), pp. 77–124.

[123] *NoisePage: Self-Driving Database Management System*. URL: `https://noisepage.com/`. Last accessed: 2023/12/12.

[124] *oneAPI Unified Memory Framework*. URL: `https://github.com/oneapi-src/unified-memory-framework`. Last accessed: 2023/12/14.

[125] *Open Street Map Cells*. URL: `http://www.opencellid.org`. Last accessed: 2022/01/05.

[126] HweeHwa Pang, Michael J. Carey, and Miron Livny. “Memory-Adaptive External Sorting”. In: *VLDB*. Morgan Kaufmann, 1993, pp. 618–629.

[127] HweeHwa Pang, Michael J. Carey, and Miron Livny. “Partially Preemptive Hash Joins”. In: *SIGMOD*. ACM Press, 1993, pp. 59–68.

[128] Philippos Papaphilippou and Wayne Luk. “Accelerating Database Systems Using FPGAs: A Survey”. In: *FPL*. IEEE Computer Society, 2018, pp. 125–130.

[129] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. “Automatic differentiation in PyTorch”. In: (2017).

[130] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. “Self-Driving Database Management Systems”. In: *CIDR*. www.cidrdb.org, 2017.

[131] Andy Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. “Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation”. In: *PVLDB* 14.12 (2021), pp. 3211–3221.

[132] Christina Pavlopoulou, Michael J. Carey, and Vassilis J. Tsotras. "Revisiting Runtime Dynamic Optimization for Join Queries in Big Data Management Systems". In: *SIGMOD Rec.* 52.1 (June 2023), pp. 104–113. ISSN: 0163-5808. DOI: 10.1145/3604437.3604460. URL: https://doi.org/10.1145/3604437.3604460.

[133] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. "Handling Task Dependencies Under Strided and Aliased References". In: *ICS.* ACM, 2010, pp. 263–274.

[134] Andrea Pietracaprina, Matteo Riondato, Eli Upfal, and Fabio Vandin. "Mining Top-K Frequent Itemsets Through Progressive Sampling". In: *Data Mining and Knowledge Discovery* 21.2 (2010), pp. 310–326.

[135] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Muhammad Mukarram Bin Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve D. Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. "Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking". In: *SIGCOMM.* ACM, 2022, pp. 66–85.

[136] *PrestoDB on GitHub.* 2023. URL: https://github.com/prestodb/presto. Last accessed: 2023/08/02.

[137] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. "One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA". In: *ADMS@VLDB.* 2021, pp. 17–26.

[138] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *ISCA.* IEEE Computer Society, 2014, pp. 13–24.

[139] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. "Using State Modules for Adaptive Query Processing". In: *ICDE.* IEEE Computer Society, 2003, pp. 353–364.

[140] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. "Automating Physical Database Design in a Parallel Database". In: *SIGMOD*. ACM, 2002, pp. 558–569.

[141] Alexander van Renen and Viktor Leis. "Cloud Analytics Benchmark". In: *Proc. VLDB Endow.* 16.6 (2023), pp. 1413–1425.

[142] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Building blocks for persistent memory". In: *VLDB J.* 29.6 (2020), pp. 1223–1241.

[143] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. "Locality-Sensitive Operators for Parallel Main-Memory Database Clusters". In: *ICDE*. IEEE Computer Society, 2014, pp. 592–603.

[144] Thomas A. Runkler. *Data Analytics - Models and Algorithms for Intelligent Data Analysis, Third Edition*. Springer, 2012. ISBN: 978-3-658-14074-8. DOI: 10.1007/978-3-658-14075-5.

[145] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. "The Digitization of the World From Edge to Core". In: *Framingham: International Data Corporation* 16 (2018).

[146] *Samsung CXL Memory Module*. 2023. URL: https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module. Last accessed: 2023/12/14.

[147] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. "B^2-Tree: Cache-Friendly String Indexing within B-Trees". In: *BTW*. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 39–58.

[148] Tobias Schmidt, Philipp Fent, and Thomas Neumann. "Efficiently Compiling Dynamic Code for Adaptive Query Processing". In: *ADMS@VLDB*. 2022, pp. 11–22.

[149] Maximilian E. Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python". In: *EDBT*. OpenProceedings.org, 2019, pp. 562–565.

[150] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. "Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL". In: *SSDBM*. ACM, 2020, 6:1–6:12.

[151] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. "In-Database Machine Learning with SQL on GPUs". In: *SSDBM*. ACM, 2021, pp. 25–36.

[152] Maximilian E. Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. “In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems”. In: *BTW*. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 247–266.

[153] Maximilian E. Schüle, Maximilian Springer, Alfons Kemper, and Thomas Neumann. “LLVM code optimisation for automatic differentiation: when forward and reverse mode lead in the same direction”. In: *DEEM@SIGMOD*. ACM, 2022, 5:1–5:4.

[154] Maximilian Emanuel Schüle. “Recursive SQL and GPU-Support for In-Database Machine Learning”. In: *BTW*. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, p. 931.

[155] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. “Access Path Selection in a Relational Database Management System”. In: *SIGMOD*. ACM, 1979, pp. 23–34.

[156] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. “Presto: SQL on Everything”. In: *ICDE*. IEEE, 2019, pp. 1802–1813.

[157] Boris M Shabanov and Oleg I Samovarov. “Building the Software-Defined Data Center”. In: *Programming and Computer Software* 45 (2019), pp. 458–466.

[158] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. “Optimizing ETL Processes in Data Warehouses”. In: *ICDE*. IEEE Computer Society, 2005, pp. 564–575.

[159] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. “Orca: A Modular Query Optimizer Architecture for Big Data”. In: *SIGMOD*. ACM, 2014, pp. 337–348.

[160] *Stack Benchmark*. 2022. URL: `https://rmarcus.info/stack.html`. Last accessed: 2022/11/27.

[161] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. “The Design and Implementation of INGRES”. In: *ACM Trans. Database Syst.* 1.3 (1976), pp. 189–222.

[162] *Storage Performance Development Kit (SPDK).* 2023. URL: https://spdk.io/. Last accessed: 2023/12/12.

[163] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. "Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation". In: *PVLDB* 15.1 (2021), pp. 85–97.

[164] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: the Next Generation". In: *EuroSys*. ACM, 2020, 30:1–30:14.

[165] Mads Tofte and Jean-Pierre Talpin. "Region-based Memory Management". In: *Inf. Comput.* 132.2 (1997), pp. 109–176.

[166] *TPC-DS Benchmark.* 2022. URL: https://www.tpc.org/tpcds. [Last accessed: 2022/11/27].

[167] *TPC-H Benchmark.* 2023. URL: https://www.tpc.org/tpch. [Last accessed: 2023/10/03].

[168] Immanuel Trummer and Christoph Koch. "Parallelizing Query Optimization on Shared-Nothing Architectures". In: *PVLDB* 9.9 (2016), pp. 660–671.

[169] Sandeep Tyagi. "Using Data Analytics for Greater Profits". In: *Journal of Business Strategy* 24.3 (2003), pp. 12–14.

[170] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. "DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes". In: *ICDE*. IEEE Computer Society, 2000, pp. 101–110.

[171] Jeffrey S Vitter. "Random Sampling with a Reservoir". In: *ACM Transactions on Mathematical Software (TOMS)* 11.1 (1985), pp. 37–57.

[172] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems". In: *PVLDB* 13.11 (2020), pp. 2662–2675.

[173] Lukas Vogel, Daniel Ritter, Danica Porobic, Pınar Tözün, Tianzheng Wang, and Alberto Lerner. "Data Pipes: Declarative Control over Data Movement". In: *CIDR*. www.cidrdb.org, 2023.

[174] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. "Are We Ready For Learned Cardinality Estimation?" In: *PVLDB* 14.9 (2021), pp. 1640–1654.

[175] Gerhard Weikum, Axel Mönkeberg, Christof Hasse, and Peter Zabback. "Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering". In: *VLDB*. Morgan Kaufmann, 2002, pp. 20–31.

[176] Leon Windheuser, Christoph Anneser, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. "Adaptive Compression for Databases". In: *EDBT*. OpenProceedings.org, 2024, pp. 143–149.

[177] Christian Winter, Andreas Kipf, Christoph Anneser, Eleni Tzirita Zacharatou, Thomas Neumann, and Alfons Kemper. "GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons". In: *EDBT*. OpenProceedings.org, 2021, pp. 169–180.

[178] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. "FASTgres: Making Learned Query Optimizer Hinting Effective". In: *PVLDB* 16.11 (2023), pp. 3310–3322.

[179] Jinsong Wu, Song Guo, Jie Li, and Deze Zeng. "Big Data Meet Green Challenges: Greening Big Data". In: *IEEE Syst. J.* 10.3 (2016), pp. 873–887.

[180] Juncheng Yang, Yao Yue, and K. V. Rashmi. "A large scale analysis of hundreds of in-memory cache clusters at Twitter". In: *OSDI*. USENIX Association, 2020, pp. 191–208.

[181] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. "Balsa: Learning a Query Optimizer Without Expert Demonstrations". In: *SIGMOD*. ACM, 2022, pp. 931–944.

[182] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. "NeuroCard: One Cardinality Estimator for All Tables". In: *PVLDB* 14.1 (2020), pp. 61–73.

[183] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. "Reinforcement Learning with Tree-LSTM for Join Order Selection". In: *ICDE*. IEEE, 2020, pp. 1297–1308.

[184] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *HotCloud*. USENIX Association, 2010.

[185] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. "In-Memory Big Data Management and Processing: A Survey". In: *TKDE* 27.7 (2015), pp. 1920–1948.

[186] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. "Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes". In: *SIGMOD*. ACM. 2016, pp. 1567–1581.

[187] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. "SuRF: Practical Range Query Filtering with Fast Succinct Tries". In: *SIGMOD*. 2018, pp. 323–336.

[188] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc T. Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. "Deploying a Steered Query Optimizer in Production at Microsoft". In: *SIGMOD*. ACM, 2022, pp. 2299–2311.

[189] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. "SCOPE: parallel databases meet MapReduce". In: *VLDB J.* 21.5 (2012), pp. 611–636.

[190] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. "Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory". In: *SIGMOD*. 2021, pp. 2195–2207.

[191] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. "Two is Better Than One: The Case for 2-Tree for Skewed Data Sets". In: *CIDR*. www.cidrdb.org, 2023.

[192] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. "A Learned Query Rewrite System using Monte Carlo Tree Search". In: *PVLDB* 15.1 (2021), pp. 46–58.

[193] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. "Lero: A Learning-to-Rank Query Optimizer". In: *PVLDB* 16.6 (2023), pp. 1466–1479.

[194] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. "DB2 Design Advisor: Integrated Automatic Physical Database Design". In: *VLDB*. Morgan Kaufmann, 2004, pp. 1087–1097.