Smart Access Strategies for Data-Centric Processing

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Maximilian Berens

Dortmund

2025

Tag der mündlichen Prüfung:	01. September 2025
Dekan:	Prof. Dr. Jens Teubner
Gutachter:	Prof. Dr. Jens Teubner
	Prof. Dr. Kai-Uwe Sattler

Abstract

Data movement in analytical database systems is a critical bottleneck, driving energy consumption and infrastructure costs. In the context of storage access, this thesis contributes techniques to mitigate these costs through Cooperative Refinement, the symbiotic interplay between indexing and data-centric processing.

First, we address the intersection of these two fields: We by analyze probabilistic data structures, such as Bloom filters and binary sketches, as candidates for Processing-in-NAND (PiN) due to their high error tolerance. To expand the applicability of current PiN architectures, we propose a scheme for emulating inequality comparisons inside NAND. To maximize the potential of fine-granular index information, we present early work on Gravity Store, a data-centric in-storage materialization engine for declarative analytics.

The second major contribution is Team-based indexing, a generalization of bitmap indexing for selective, high-dimensional range queries. By forming "Teams" of moderately-sized attribute subsets, this strategy improves runtime efficiency and reduces storage overhead compared to traditional indexing. We address the central challenges of efficient index intersection and Team composition. Finally, we introduce TeamBench, a benchmark generator specifically designed to evaluate these index intersection performances at scale.

Acknowledgements

In the pursuit of my work and while writing this thesis, I had the pleasure of getting to know many people who helped and enabled me greatly, making my time in Dortmund interesting, enlightening, and fun.

First and foremost, I'd like to express my deep gratitude to my supervisor, Jens Teubner. Jens not only invited me to do a PhD in the first place, but also always supported my goals and independence. He also always had an open ear for my plights and ideas, and I knew I could always rely on his rich experience and insightful questions when I felt stuck the most.

During my time, I shared countless discussions with my colleagues Jan Mühlig and Roland Kühn, and I enjoyed our exchanges about our many common professional and non-professional interests at our frequent weeknight dinners. Moreover, I'd like to thank all my other colleagues and alumni from the DBIS group, such as, Henning Funke, Florian Grießkamp, Michael Kußmann, Stefan Noll, with whom I shared project group during my masters program, Thomas Lindemann, who supervised my master thesis, Sebastian Breß, who initially hired me as a HiWi for the group, and more recently Jannik Wolff and Sebastian Hilker.

Over the years, I was able to nurture my broad interests in a multitude of exciting fields and collaborated with several people, both for the purpose of research and teaching. First, I'd like to thank Joachim Biskup and Marcel Preuß, with whom I collaborated in the field of random database generation, offering interesting combinatorial and probabilistic puzzles to solve. Further, I want to thank the colleagues from the IPS chair and Statistics department, most notably Lukas Schulte, Daniel Boiar, Nikolai West, Roman Möhle, and Marléne Baumeister, for several years of amusing teaching experience of the Industrial Data Science courses. I would also like to mention our "floor-neighbors," Erich Schubert, Erik Thordsen, Alexander Lochmann, and others. I shared many intriguing discussions with Erich on the challenges and perils of teaching, ML, and AI, to whom I am also grateful for making time to head my dissertation committee.

Moreover, I want to thank Yun-Chih Chen and Jian-Jia Chen, who significantly broadened my understanding of the intricacies of modern storage hardware and processing-in-memory techniques, which became an integral part of my later research. At this point, I would also like to thank Nils Hölscher, Mario Günzel, Kuan-Hsun Chen, Christian Hakert, Kay Heider, and all the others of the LS12 chair. Our groups grew close ties over the years, and we shared multiple amazing

and insightful opportunities to visit Taiwan and our colleagues there, which also yielded insightful workshops and collaborations. I also thank Jian-Jia Chen for being part of my committee and inviting me to the group retreat in 2025.

As the second reviewer of my dissertation and the fourth member of the committee, I thank Kai-Uwe Sattler for taking the time to assess this document and visiting Dortmund for my disputation. Further, I thank Peter Buchholz for being my PhD mentor.

Lastly, I want to thank all my family and friends. I'm grateful that my friends provided me with welcome opportunities to escape stress whenever I really needed it. My father Bernd, Sigrid, and my brother Sebastian, but also Uta and Jörg, kept prodding me, which ensured I never felt lost. Even though my mother Heike never even got the chance to learn about my time as a university student, I hope she would have been proud that I finally reached the end of this journey.

Parts of the research included in this thesis have received funding from the Bundesministerium für Forschung, Technologie und Raumfahrt (BMFTR, formerly BMBF) project "Industrial Data Science" (grant no. 01IS17063A) and the Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 (project C5, grant no. 124020371).

Contents

1 Introduction **1**
1.1 A Shared Pursuit . . . 2
1.1.1 Open Challenges . . . 3
1.2 Contributions & Outline . . . 5
1.2.1 Miscellaneous Contributions . . . 6
1.2.2 Declaration of Author Contributions . . . 6

2 The End of Bandwidth Doubling **7**
2.1 Understanding NAND Flash Performance . . . 8
2.1.1 Errors in NAND Flash . . . 10
2.1.2 Bandwidth Scaling & Dark NAND . . . 12
2.2 Near-Data Processing is Not (Close) Enough . . . 13
2.2.1 Design Limitations of PiN . . . 13
2.2.2 Specialized- and Analog-based PiN . . . 14
2.2.3 Off-The-Shelf PiN . . . 14
2.2.4 Simplicity Preferred . . . 16
2.3 Processing in Unreliable Memory . . . 16
2.4 Error-Adaptive Databases . . . 17
2.4.1 Adaptive vs. Worst-Case . . . 18

3 Cooperative Refinement **21**
3.1 Bloom Filter Evaluation . . . 23
3.1.1 Integrating Bit-Flips . . . 24
3.1.2 False Negatives Guarantees . . . 26
3.1.3 Processing-in-NAND Implementation . . . 27
3.2 Inequality Emulation . . . 28
3.2.1 From-Below and From-Above Emulation . . . 29
3.2.2 Executing the Program . . . 30
3.2.3 Improving Efficiency . . . 31
3.3 Binary Sketch Evaluation . . . 32
3.3.1 Integrating Bit-Flips . . . 32
3.3.2 Processing-in-NAND Implementation . . . 34
3.4 Gravity Store . . . 35

3.4.1 Layout Description Layer 36
3.4.2 Materialization Graph & Engine 40
3.5 Discussion 43
3.5.1 Error-Analysis Scope 43
3.5.2 Further PiN Applications 44
3.5.3 No Singular Processing Paradigm 45
3.5.4 Transparent Interfaces & Database Kernels 46

4 Team-Based Indexing 47
4.1 Methodology 48
4.1.1 Evaluation Strategy 49
4.1.2 Grid-based Index 49
4.1.3 Application Conditions 51
4.1.4 Other Related Work 52
4.2 Index Creation 53
4.2.1 Storage Layout 53
4.2.2 Meta Data & Query Mapping 54
4.3 Index Precision & the Blessing of High Dimensionality 54
4.3.1 Sparsity of High-Dimensional Spaces 55
4.3.2 Sparsity Without Exponential Cost 55
4.3.3 Diminishing Returns 56

5 Efficient Index Intersection 57
5.1 Logical Optimization 57
5.1.1 Leaf Grouping & Good ISE Count 58
5.1.2 Taking the Complement 59
5.1.3 Intersection Order 60
5.2 Physical Implementation 60
5.3 Experimental Setup 61
5.4 Performance Factors 62
5.4.1 Volume & Overhead 63
5.4.2 Evaluation Strategies 63
5.4.3 Imbalance 65
5.5 Scaling Behavior 68
5.5.1 Dimensional Scaling & Phases of Index Intersection 68
5.5.2 High-Dimensional Scaling 70
5.5.3 The Impact of Team-Composition 71
5.5.4 Varying Table Size 71

6 Team Composition 73
6.1 Attribute Coverage & Ad-Hoc Queries 74
6.1.1 Definitions 75
6.1.2 The Probability of l-efficient Access 75
6.2 Exploiting Combined Selectivity 78

6.2.1 Query–Data Dependencies 78
6.2.2 Workload-Aware, Optimal Volume 79
6.2.3 Allowing Attribute Overlap 81
6.2.4 Overhead vs. Volume 81
6.2.5 Storage Costs vs. Runtime 82
6.2.6 Discussion 84

7 TeamBench 85

7.1 Defining the Generator 86
7.1.1 Input Parameters 86
7.1.2 The Sampling Procedure 87
7.1.3 Computing Distribution P 89
7.2 Choosing Parameters 90
7.2.1 Estimated, Uncompressed Volume 90
7.2.2 Sufficient Sample Count 90
7.2.3 Custom Distributions 91
7.3 Discussion 91
7.3.1 On Inter-Grid Dependence 92
7.3.2 Compression & Order of ID Generation 92

8 Conclusions 95

8.1 Towards Data-Centric Database Processing 96
8.2 Beyond Team-based Indexing 97

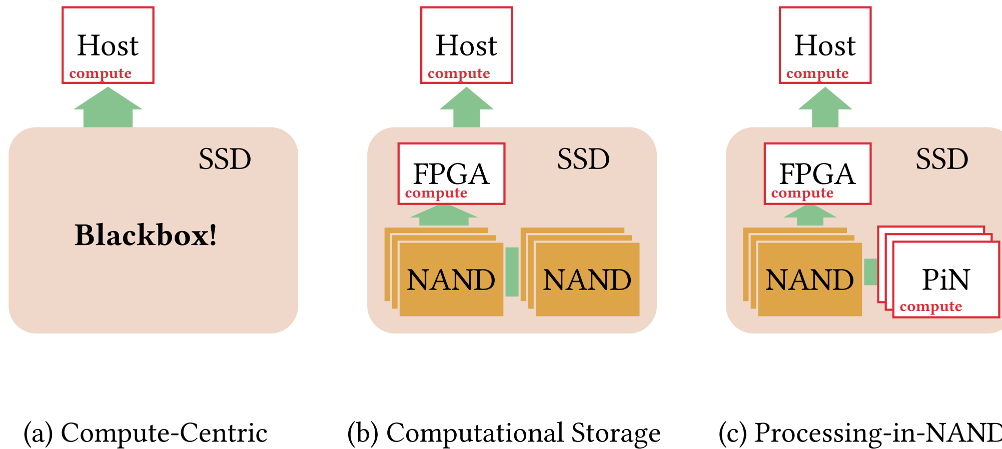
A Abstract: Uniform Database Generation 99

1 Introduction

The cost of many of today's analytical workloads is dominated not by computation, but by the data movement [1, 2]. The problem is especially acute in scan-heavy or data-intensive workloads, where a large, storage-resident volume of data is touched, yet only a small fraction is truly relevant. In these compute-centric system architectures, where powerful CPUs must be constantly fed data, this Data Movement Tax manifests at every level of the memory hierarchy—from a processor cache miss, down to a NAND flash memory access within a Solid-State Drives (SSDs). The recent availability of PCIe 4.0 and 5.0 NVMe SSDs provides databases a raw, aggregate read *bandwidth* that rivals DRAM, albeit at significant cost. For example, Micron's 6550 ION SSD [3] can deliver the full $3.5\,\mathrm{GB/s}$ *per PCIe 5.0 lane*. With modern server CPUs offering 96–128 lanes, this theoretically amounts to a total bandwidth of up to 448 GB/s. However, the SSD also draws about $5\,\mathrm{W}$ per lane when active for a combined power demand of $5 \cdot 128 = 640\,\mathrm{W}$. For comparison, even a very power hungry CPU like the AMD EPYC 9965 has "only" a Thermal Design Power (TDP)[1] of $500\,\mathrm{W}$ [4]. Moreover, fully utilizing these devices—especially if access is not entirely sequential—can occupy multiple cores per drive, solely for handling the necessary I/O requests [5, 6]. Excessive data movement impacts not only raw performance but also system complexity, infrastructure, and energy costs, and by extension, the ecological footprint. Even more crucially, it takes its toll even deep within modern microchips, where it reduces power efficiency and ultimately constrains their performance scaling [1].

Traditionally, such inefficiencies motivate the use of index structures, which aim to selectively narrow the access path by skipping irrelevant portions of the data. Although indexing can be very effective, it is not universally applicable and suffers from high storage overhead. Moreover, its potential to avoid data movement

[1] The maximum (temporary) power draw can be higher, but is usually not explicitly specified for CPUs.



(a) Compute-Centric (b) Computational Storage (c) Processing-in-NAND

Figure 1.1: In data-centric architectures (middle and right), computation is performed prior to traversing high-cost system buses, thereby addressing the Movement Tax.

is often diminished by large access granularity. For example, processing even a single bit from an SSD still requires fetching a full (4–16 KiB) page, which results in *read-amplification*.

In contrast to a traditionally compute-centric architecture (Fig. 1.1a), Data-Centric Processing promises to reduce the Movement Tax by taking another, much more radical approach: moving computation toward the data. Having the ability to aggregate or filter data closer to its source subsequently results in less data transfer. However, the full access path involves many stages and therefore requires consideration of opportunities for data reduction not merely at a single point but across all levels of the memory hierarchy: Computational Storage Devices (CSDs) [7–9] (Fig. 1.1b), and various forms of Processing-in-Memory—most notably using DRAM [10–13], and also NAND flash memory [14–17] (Fig. 1.1c)—as examples of DCP, each offering a distinct set of capabilities. A truly data-centric database system must orchestrate these techniques in concert, but doing so demands a significant departure from traditional system design [18]. It requires a holistic hardware-software co-design approach and a willingness to embrace specialized components, workload decomposition, and resource-constrained computation.

1.1 A Shared Pursuit

Both Data-Centric Processing (DCP) and indexing have a common goal of reducing the Data Movement Tax, but each pursues it in a constrained and incomplete way.

By pushing down compute, DCP comes with costs and limitations, to which we will collectively refer to as the Proximity Tax: The closer one gets, the more one sacrifices in terms of computational flexibility, generality, and resource richness. Near-data accelerators, such as FPGAs, often lack general-purpose programmabil-

ity, offer limited memory, and require complex setup. In the most extreme case, i.e., processing directly within memory chips, the movement cost is minimal, but execution becomes purely local and therefore is applicable only to a narrow class of problems. For example, operations (e.g. joins) partially conflict the data-centric idea due to their inherent need of initial data movement. Moreover, limited resource availability restricts the overall load that a near- or in-memory architecture can handle reliably. Having the ability to skip work for irrelevant data is therefore instrumental to minimize the hardware burden. This is especially crucial when considering that microchips with data-centric compute capabilities need to be resource-constrained *by design*: their ubiquitous deployment throughout the memory hierarchy is only economically feasible if they are cheap to design, manufacture, and run.

Indexing, on the other hand, does not focus on relocating computation, but still has its own costs. For example, access to a leaf in a B^+-tree [19] still requires moving it all the way up to the CPU, only to (potentially) result in yet another I/O request. Due to the large access granularity of storage-class memory and the limits of traditional block device interfaces, the potential of indexing remains largely untapped. Moreover, some types of indices, such as bitmap indices [20, 21], can require significant data transfers on their own. These costs can be alleviated by pushing the index evaluation down into the storage device.

In this sense, the two concepts can support each other to achieve what each alone cannot. We identify two complementary directions of combining indexing and DCP: First, *Data-Centric Index Evaluation*, where index structures are evaluated using data-centric mechanisms, and second, *Index-Driven DCP*, where index information is used within a data-centric architecture.

We will refer to this symbiotic interaction as Cooperative Refinement, a conceptual framing in which indexing and DCP work together to transform raw data into query results. In this thesis, we will use it as a guiding structure to relate and contextualize our contributions that each aims at addressing the Movement Tax.

1.1.1 Open Challenges

Combining indexing with Data-Centric Processing introduces a variety of new challenges and opportunities, which we highlight below and revisit throughout the thesis. Beyond their intersection, the continued development of each approach in isolation also presents untapped potential to address the Data Movement Tax.

Cooperative Refinement The potential of Cooperative Refinement remains largely untapped in both directions, primarily because various forms of DCP are not yet fully established and hardware implementations are scarce: UPMEM [10] presents the first commercially available Processing-in-DRAM platform. At the time of writing, existing CSD designs are either prototypical [22] or highly specialized, e.g., ScaleFlux's CSD 3000 [23] implements transparent compression capabilities.

Research on Processing-in-NAND is largely based on simulations [14] or on the modification of existing, commercially available NAND flash chips [15].

Although PiN has gained traction in the architecture and storage systems domains [14, 15, 17, 24, 16], it has seen limited integration into database management systems. One reason may be the significant constraints inherent to NAND flash-based compute: limited resources and programmability, a restricted instruction set, economic viability challenges, and the necessity to address intrinsic bit errors. These hurdles can only be overcome through deliberate hardware-software co-design. In this work, we identify evaluation of *probabilistic index structures* as a particularly compelling candidate for PiN, as they can both benefit greatly and work within the limits of the architecture.

Similarly, leveraging detailed index information within a CSD poses challenges with respect to interfaces and the orchestration of data movement within the device. Even though skipping whole pages within the traditional block device perspective is straightforward, it suffers from read-amplification and "host-centric" communication overhead for managing I/O. This overhead is still paid even if the device is somehow able to drop irrelevant parts of a page. Furthermore, additional data movement and operations are required for materializing more complex elements, such as tuples in a column store. Transferring pre-selected, pre-assembled (and potentially batched) elements out of the device instead has the potential to improve efficiency significantly. However, to enable this capability, the logical structure of the stored data must first be made internally accessible within the CSD through dynamic mapping of data structures to physical addresses. Such a mapping presents a first step towards declarative and domain-specific storage interfaces.

Extending Indexing In analytical queries on large-scale scientific data sets, such as the LHCb particle accelerator at CERN [25], indexing is often considered infeasible because it suffers from the curse of high dimensionality and substantial storage costs. As a result, full-table scans are deemed the only appropriate solution, despite the grueling Data Movement Tax at petabyte-scale workloads. Without indexing, the use of DCP is also limited by the need to assemble mostly irrelevant, often high-dimensional tuples—a task that already involves substantial data movement and operations before any meaningful compute can occur, particularly in resource-constrained environments. It is therefore beneficial to adopt new strategies for tuple-accurate indexing that improve overall storage space consumption and offer dimensional scalability in order to push the limits of feasibility and thus enable Cooperative Refinement at the very large scale. The Team-based indexing strategy we suggest in this thesis introduces new challenges with respect to index selection and intersection.

1.2 Contributions & Outline

The chapters of this thesis can be divided into two parts.

Cooperative Refinement The first two chapters focus on how indexing and DCP can cooperate to refine data into query results, with a particular focus on NAND flash SSDs. In Chapter 2, we motivate the database integration of Processing-in-NAND as an answer to the design limits of NAND flash memory. We specifically discuss the importance of NAND's intrinsic susceptibility to bit-flips and the absence of error correction. Chapter 3 sharpens our vision of data-centric databases by suggesting the evaluation of probabilistic data structures as a compelling indexing application for PiN due to their simplistic evaluation and inherent error tolerance. The content of these parts was published in

[26] Maximilian Berens, Yun-Chih Chen, Jian-Jia Chen, and Jens Teubner: "Beyond Bandwidth Doubling: Embrace Bit-Flips and Unlock Processing-in-NAND," in *2025 IEEE 41st International Conference on Data Engineering (ICDE), pp. 4649–4661*

Moreover, we introduce an emulation scheme for inequality operations, leveraging instructions from existing PiN designs, to broaden the applicability of PiN. In Sec. 3.4, we describe the challenge of utilizing index information and early materialization within CSDs. We present our early ideas on *Gravity Store*—a physical storage description for dynamically generating device-internal requests from declarative input.

Extending Indexing The following four chapters are dedicated to our work on *Team-based indexing*—a generalization of the bitmap indexing strategy. In Chapter 4, we suggest creating index structures over *moderately*-sized subsets of attributes, so called *Teams*. This offers significant reductions in storage costs and access volume during evaluation. In Chapter 5, we formulate logical execution plan optimizations for index intersection and evaluate them with our prototype implementation. Moreover, we discuss domain-agnostic methods for choosing Teams from the set of attributes and present out (Team-dependent) performance evaluation in Chapter 6. The Team-based indexing approach was published in

[27] Maximilian Berens and Jens Teubner: "Index Intersection for High-Dimensional Range Queries," in *Proc. VLDB Endow. Vol 19 Issue 4, pp. 767-779*

Thereafter, Chapter 7 describes an efficient, highly configurable generator for benchmark data, tailored specifically for (Team-based) index intersection. Lastly, we provide a summary of the thesis and concluding remarks in Chapter 8.

1.2.1 Miscellaneous Contributions

During the writing of this thesis and previously, the author contributed to publications that are not part of this thesis. This work includes the following:

[28] Michael Kußmann, Maximilian Berens, Ulrich Eitschberger, Ayse Kilic, Thomas Lindemann, Frank Meier, Ramon Niet, Margarete Schellenberg, Holger Stevens, Julian Wishahi, Bernhard Spaan, and Jens Teubner: "DeLorean: A Storage Layer to Analyze Physical Data at Scale," in *BTW 2017: 413-422*

[29] Maximilian Berens and Joachim Biskup: "On Sampling Representatives of Relational Schemas with a Functional Dependency," in *FoIKS 2022: 1-19*

[30] Maximilian Berens, Joachim Biskup, and Marcel Preuß: "Uniform Probabilistic Generation of Relation Instances Satisfying a Functional Dependency," in *Inf. Syst. 103: 101848 (2022)*

An abstract on [29, 30] can be found in Appendix A.

1.2.2 Declaration of Author Contributions

According to §10 (2) of the doctoral regulations of the computer science department at TU Dortmund University from August 29, 2011, the author should indicate their own contributions to the results of collaborations that are used in this thesis.

In article [26], I contributed conceptualization, formal proofs and writeup, and Yun-Chih Chen, Jian-Jia Chen, and Jens Teubner contributed to the conceptualization and writeup. In article [27], I contributed conceptualization, experiments, formal proofs and writeup, and Jens Teubner contributed to the conceptualization and writeup. I am the principle author of all unpublished content presented in chapters of this thesis.

Beyond this thesis, I assisted with experiments in [28]. I also co-authored articles [29] and [30], where I contributed to conceptualization, experiments, formal proofs and writeup.

2
The End of Bandwidth Doubling

The deeper we descend in the memory hierarchy, the higher the total cost of data movement. Access to SSDs, the bedrock of today's large-scale data centers, is therefore particularly expensive. However, the distance to the CPU is not the only problem: SSDs also contend with significant internal power, heat and reliability issues [31, 32]. Soon, SSDs, and therefore databases as well, may face the limits of Dennard scaling, meaning no more performance gains can be achieved without hitting physical limitations. Just like microprocessors before, NAND flash memory—the underlying technology of SSDs—has Dark Silicon, too, preventing performance from improving at the same pace as capacity [31]. Much of the power (and thus heat) within a NAND chip results from transferring data at a high rate [14], which is yet another symptom of a compute-centric style of processing. Although recent generations of SSDs offer unprecedented *bandwidth*, the access *latency* of NAND-flash memory has actually been deteriorating for some time now [33]. This is not always apparent when looking at product descriptions, because manufacturers are employing various techniques to compensate for the compromises that come with the quest for ever-increasing storage density. However, as an SSD fills up its capacity or when its memory cells age with usage, NAND's performance deterioration becomes visible [31].

Processing-in-NAND promises to reduce the need for data movement, but it is no panacea. Power and transistors are finite resources in every microchip design. Adding more logic circuitry to NAND memory, such as for computation or more read-out parallelism, inevitably reduces capacity and increases design cost [34]. Moreover, complex (and *flexible*) functionality places a considerable demand on the chip's power budget, is harder to manufacture, and thus often infeasible.

Although targeting the peripheral area of the actual memory array, several recent PiN concepts [17, 16] still represent significant departures from state-of-the-art NAND flash design. In contrast to these highly *specialized* approaches, *Off-the-*

Shelf (OTS-) PiN presents a different take by *repurposing* existing circuits within the memory chip to perform basic calculations [14, 15]. This reduces data transfer volume at a more affordable $/GB while also preserving the chip's primary storage functionality. However, OTS-PiN typically only supports primitive operations and may sacrifice some storage capacity to improve reliability. Additionally, it generally lacks hardware error correction, requiring applications to incorporate built-in resilience against bit-flip errors.

Reliability, i.e., the occurrence of random bit-flips, poses a central challenge for PiN and Processing-in-Memory (PiM) in general. Although NAND flash memory is especially susceptible [35, 32, 36], these errors also increasingly plague DRAM [37], and High-Bandwidth Memory (HBM) [38]. Due to their proximity to memory cells, PiM architectures usually lack costly error correction, necessitating alternative methods to manage errors. However, even beyond the DCP paradigm, maintaining the illusion of error-free memory hinders performance growth [37]. This inspires a need for systems that are not just error-aware but tolerant.

Outline In this chapter, we first provide background on NAND flash memory and its performance characteristics. We then motivate the use of Processing-in-NAND as a means to cope with the inevitable demise of NAND's performance scaling and discuss its inherent challenges, most notably bit-errors. Lastly, we broaden our discussion of memory errors to highlight the broader need to address them at the database system level.

2.1 Understanding NAND Flash Performance

To understand the limitations of NAND flash design and why its performance scaling is in jeopardy, we will first describe two primary factors that impact bandwidth: NAND parallelism and access latency. We then discuss why the measures currently employed by manufacturers offer no sustainable path forward and ultimately require us to rethink how to access storage.

Flash Parallelism To achieve sufficient bandwidth despite high latency, SSDs employ various forms of parallelism. A single SSD has multiple NAND-flash *chips* (usually 4–8), each partitioned into so called *planes* (usually 4–6).[1] Each plane contains a set of page-sized SRAM registers (*page buffers* in Fig. 2.1) and is capable of sequentially serving requests targeting its (several hundred) flash *blocks*, each block containing (thousands of) pages. For a write, an entire block must be erased first before individual pages can be programmed, a so-called *program-erase* (P/E) cycle. Reads are performed at page granularity of usually 4 KiB to 16 KiB.

[1]We omit some levels of the hierarchy, e.g., package & die, for simplicity.

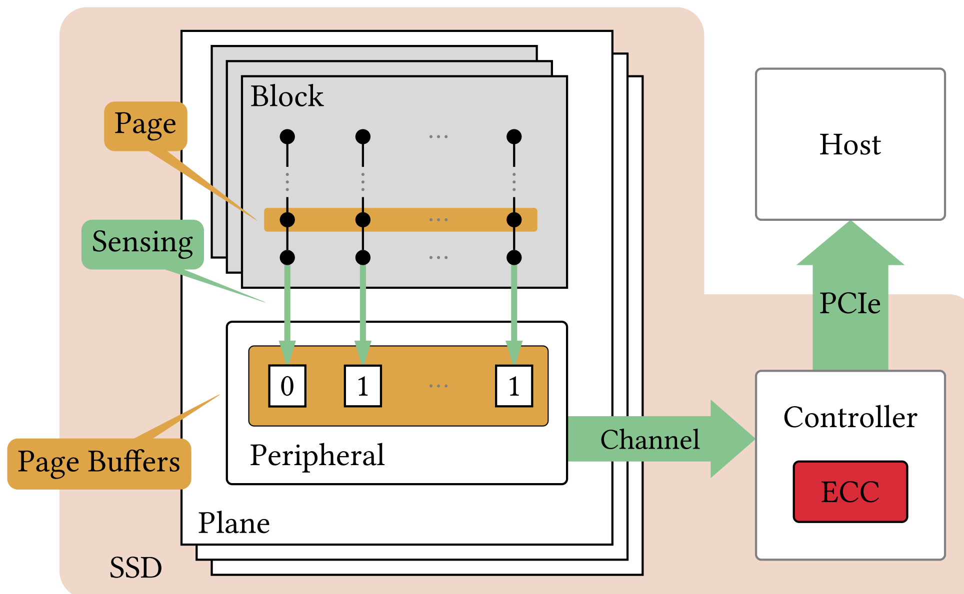


Figure 2.1: Simplified structure of a NAND flash memory chip, showing the path of a read request out of a *plane*, an independent read unit.

Read Latency When a read request is issued, a command is first sent from the controller to the plane's command queue. To execute the read, the plane first *senses* the data through individual "bit-lines" into the page buffers by applying a certain voltage: when a cell is initially programmed, electrons are deposited to represent a specific charge level. During sensing, the cell may or may not have enough charge to conduct the applied voltage, representing a single bit in Single-Level Cell (SLC) NAND. After the page is fully sensed, it is transferred over a bus (shared by multiple planes) to the device DRAM (*channel transfer* in Fig. 2.1) and then checked for bit-flips. If the page passes error correction, it will be sent to the host via PCIe, otherwise the read process may be restarted with different parameters to hopefully observe fewer bit-flips. A breakdown of the entire latency can be found in Fig. 2.2 (next page).

Writing NAND flash Memory Writing to NAND flash is inherently a *block*-level operation, meaning that an entire block must first be erased before any *page* within can be programmed. For denser technologies like TLC (Triple-Level Cell), this programming is an iterative process that carefully adjusts voltage levels across multiple pulses to precisely store more bits per cell. This meticulous approach ensures higher capacity but introduces complexities that reduce performance and increase error susceptibility.

Capacity Over Latency Memory technology scaling was the driver of both performance and capacity for a long time. But decreasing the feature size of memory cells comes with significant hurdles, namely high manufacturing cost and

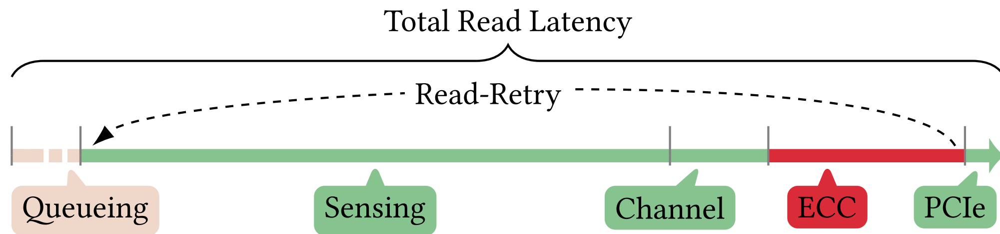


Figure 2.2: Breakdown of a typical page read latency. Segment length indicates the relative duration of each component: command send and queuing, sensing, channel transfer, error correction and external transfer. Latency may increase further due to read-retry.

degrading performance and reliability [39]. To avoid this issue, manufacturers have instead focused[2] on 3D stacking [39] and increasing the number of layers [40].

Another way to scale up capacity is to store more than one bit per cell. Because both initial programming charge and sensing voltage can be varied between multiple levels, probing for more than one state becomes possible: Triple-Level Cell (TLC), for example, utilizes the voltage stored in a cell to represent three bits, resulting in dividing the voltage range into $2^3 = 8$ intervals, each associated with a different state. Although this, too, increases capacity, access latency becomes longer. Specifically, TLC requires up to seven read operations to decode all data bits from a single cell, resulting in a $2.3\times$ latency compared to SLC for the same amount of data (1 read for 1 bit versus 7 reads for 3 bits).

Latency Mitigation To mitigate the long latency, some NVMe SSDs can initially write data in "SLC mode" by storing only one bit per cell. Once this initial capacity threshold (about 20% to 30% of the advertised capacity) is exceeded, the SSD switches to "TLC mode". This can degrade random read throughput by more than 30% [41]. Subsequently, latencies deteriorate with increased capacity usage.

2.1.1 Errors in NAND Flash

Another important aspect of latency is bit-flips, and the necessity of error correction. There are various situations for errors to occur in NAND flash: Bits may flip during both write and read processes, but most errors[3] today are *retention errors* caused by electrons leaking from a cell over time [35, 42].

NAND is known to degrade on write, causing error rates to differ temporally and spatially: Repeated program-and-erase (P/E) cycles on the same block "age" the underlying material, causing it to leak electrons faster and be generally more susceptible to disturbances [35]. Further, due to lithographic imperfections, the

[2] As of 2021, the feature size *increased* from 14 nm to 15 nm to 38 nm [39].

[3] Depending on how multi-bit cells encode the different voltage levels, retention errors can result in both types of bit-flips ($0 \rightarrow 1$ and $1 \rightarrow 0$).

material quality across 3D-NAND layers can vary, a phenomenon known as *process variation* [32]. Error rates can also differ across pages stored in the same set of cells, a so called *word-line* [35]. For additional information on NAND flash errors see for example [32, 35].

Error guarantees for today's memory have to be very strict in order to ensure viability for a wide range of applications. For enterprise SSDs, the JEDEC's JEP122H standard defines 10^{-16} as the highest, acceptable Uncorrectable Bit Error Rate (UBER), i.e., after ECC [43]. That means 1 unrecoverable bit error is expected per 1.25 PB read. The Raw Bit Error Rate (RBER), however, can be significantly larger—in the order of 1 in a 100 bit—after a few days of retention time [32, 42]. As a result, powerful error-correction codes (ECC) are necessary to uphold the illusion of error-less data stored in (actually very error-prone) memory.

Bit-Flips vs. Latency Low-Density Parity-Check (LDPC), the de-facto standard for ECC in SSDs, is applied iteratively: If a step is not able to correct enough errors, the read-process is *restarted* (see Fig. 2.2), applying different voltage levels during sensing and a more heavy-weight error correction step. This can have detrimental impact on the latency. For example, an initial 85 µs access latency can grow to more than 1 ms due to ECC [32]. As memory cells become smaller, fewer charges are used to store data bits, making them more susceptible to errors due to charge leakage. Thus, these so called Fail-Slow symptoms in SSDs are expected to become more severe with further technology scaling [31, 44].

Error Mitigation SSDs employ various strategies to avoid errors. Ultimately, error rate, which directly impacts latency, is dependent on many factors. For example, material quality, endured P/E cycles, retention time (i.e., the time since a page was written) and *temperature*: Heat causes cells to leak electrons faster and disturbs signal transmission in electrical circuits. To illustrate, a page can require up to 6 read-retries when retained for 2 days at 85 °C, which is equivalent to 168 days of retention time at 40 °C [42]. This makes both passive (cooling) and active heat-management (performance throttling) a necessity in modern SSDs.

Another measure relevant for Processing-in-NAND is *data scrambling* [35]. To avoid data-pattern-specific program disturbance errors, a page is deterministically turned into a pseudo random stream of equally many zeros and ones before being actually written. Because data is de-scrambled only after it was already retrieved from the NAND array, the ability to perform PiN operations may be hindered. Some approaches alleviate the problem of program disturbance errors by reducing the error rate with an enhanced programming step at the cost of latency [15]. Moreover, some of the data structures discussed in Chapter 3 are designed to achieve uniformity and independence between bits, which allows us to skip this measure. When errors do rise beyond a certain rate, data is *scrubbed* by writing a fresh version of the entire block to a different location [31]. Eventually, blocks deemed too unreliable by the controller are decommissioned, reducing (internal)

capacity. This is invisible to the user, because SSDs ship hidden blocks specifically allocated for this purpose. Notably, manufacturers usually opt for performance degradation over offering less storage capacity to the host [31]. To use up P/E each blocks cycles uniformly and prolong the lifetime of the NAND flash chips is the concern of complex wear-leveling strategies employed by the SSD controller.

2.1.2 Bandwidth Scaling & Dark NAND

Thermal constraints have been widely studied in processor design and are commonly cited as a driving force behind the end of Dennard scaling [45]. This has been recognized as the *Dark Silicon* problem [45], in which silicons are not able to be utilized due to high temperature. Temperature plays a critical role in maintaining data integrity, but it also heavily influences the hardware design of NAND flash chips. Just like microprocessors before, NAND flash meets the end of Dennard scaling. With growing transistor density, the power draw also increases, converting into heat. Generally, ensuring stable functionality and longevity becomes increasingly more difficult once we approach the limits of the physical laws. The aforementioned halt to NAND technology scaling is one consequence, shifting the focus more on increasing capacity.

Scaling Parallelism Given stagnant access latencies, bandwidth scaling within a single NAND flash chip may be gained by scaling up the plane count. Although planes in modern SSDs are only capable of very rudimentary logic functionality, they are (mostly) independent units and can individually serve requests, determining the level of parallelism within a chip. But manufacturers currently still primarily focus on improving the number of *pages per plane*, rather than plane count, as a way to reduce the cost per gigabyte. This decision to invest chip area in memory cells rather than in peripheral logic is influenced by economical rationales.

Even though the latency-component of channel transfers is comparatively small (see Fig. 2.2), achieving it requires a significant investment of *instantaneous* power for every single transfer. Previous designs [34], that focus on parallelism over capacity, draw significant amounts of power, require strong cooling to compensate and are very expensive.

Dark NAND Increasing the power draw beyond a certain degree eventually leads to "*Dark NAND*", where not all parts of the NAND chip can be used at any given point in time in order to avoid surpassing the thermal budget. Explicit cooling and thermal throttling are required to maintain stable operations. Like multi-core scaling for CPUs [45], multi-plane scaling is reaching its limit.

Note that our discussion so far focused on the relative costs of the steps of *reading* data, instead of program & erase, i.e., write, operations. Although writes do have higher latency and cost more power, they are usually executed as background operations in times of less device load. Further, as we will argue below, the

high, relative power requirements of bus transfers during reads are what motivates processing-in-memory as a potential avenue for future performance scaling.

Scaling Flash-Chips Increasing the number of NAND-flash chips per SSD as a means to improve bandwidth comes with considerable disadvantages. Adding more chips is cost-inefficient, i.e., doubling bandwidth also (at least) doubles cost. Further, contention of shared internal resources, such as power supply, DRAM and controller, increases. Additional chips also represent additional points of failure, and a single chip failing usually warrants decommissioning the entire device.

2.2 Near-Data Processing is Not (Close) Enough

One of the largest factors of power consumption (although not latency) during a NAND-flash read is *channel transfers*, i.e., moving the data out of the plane's page buffers. Thus, (data-centric) methods that reduce data transfer volumes only after it has already left the chip are not enough. Instead, only computation directly within the chip can address the high power demand of channel transfers and thus, the thermal limitations of NAND flash design [14].

PiN, a subset of Processing-in-Memory (PiM), promises better *power efficiency* by empowering individual planes with compute capabilities or, even more extreme, using the memory array itself for computations. However, PiN's functionality is inherently *limited* because complex logic circuits compete with memory for chip area and generate heat, which can compromise memory reliability. This constrains technical and economic feasibility. As a result, PiN designs typically face two options: either *specialize* for a specific application or support only a limited set of *primitive* operations.

2.2.1 Design Limitations of Processing-in-NAND

In domains where a workload can be reduced to a very narrow set of elementary operations, such as matrix multiplication in machine learning, developing dedicated hardware is economically feasible, even if designs are complex and expensive. Although relational queries, too, can be composed of only a small set of elementary operations, databases do not enjoy this luxury. Instead, real world Database Management Systems (DBMS) are expected to support a wide range of data types and extensions that break with the purity of the relational model. But also for very basic types, even seemingly simple functionality, such as "WHERE A > 42", can quickly become a major challenge for Processing-in-NAND: Inequality operations are usually implemented with arithmetic subtraction to allow for an efficient comparison with 0, e.g., as $0 > A - 42$. However, hardware circuits for subtraction (or addition) require passing a carry bit "between bit-lines". To compute inequality directly within NAND flash memory would therefore necessitate connecting adjacent bit-lines—a feature that would directly go against the

economically successful hardware design of NAND flash memory. Another class of data-reducing operations, (grouped) aggregations, require even more complexity, e.g., for holding intermediates. Notably, instructions that would require comparing two values (or bits) *within* the same page are subsequently also infeasible.

Fortunately, the initial *sensing* step is not very power-hungry (albeit relatively slow) and the peripheral area of the circuit—directly outside of the actual memory fabric—offers more opportunities for modifications. For example, ICE [16] suggests considerable extensions to implement similarity search, which requires dedicated, domain-specific error correction in this area. But significantly changing the peripheral circuitry still incurs significant costs for the design and testing of hardware. In this context, it is helpful to think of a plane's page buffers (see Fig. 2.1) not as a single page-sized register, but rather as a collection of independent 1-bit registers (or *latches*) much like the individual registers found in a CPU. This perspective emphasizes the fine-grained nature of the peripheral logic and highlights the potential (and challenge) of repurposing it for compute tasks. These challenges vary significantly depending on how radically the peripheral logic is modified, which motivates a classification of PiN approaches based on their degree of specialization and reliance on analog computation.

2.2.2 Specialized- and Analog-based Processing-in-NAND

Some designs, such as ICE (mentioned above), specifically redesign the circuitry to target a narrow range of applications—a pattern we refer to as *specialized PiN*. Another example is conditional page reads based on string matching [17]. Such specialization reduces flexibility and departs substantially from conventional hardware designs and manufacturing processes. While theoretically feasible, large divergences of existing designs are bound to complicate the manufacturing process and thus significantly increase the cost of their development. They are therefore less appealing to manufacturers who generally prioritize more versatile, general-purpose solutions with larger markets. Similarly, *analog-based PiN* [24] uses the memory array fabrics to compute in the analog domain, but requires expensive, high-resolution analog-to-physical converters.

2.2.3 Off-The-Shelf Processing-in-NAND

Instead of creating an entirely new design from scratch, repurposing functions in existing circuit designs offers a different approach. The majority of space in a flash chip is dedicated to memory cells. But the plane's peripheral circuit, i.e., the area close to the actual memory, already contains a rich amount of logic responsible for accessing or verifying the memory. Off-The-Shelf (OTS) PiN makes use of these circuits to conduct computations, thus limiting modifications to the chip and lowering the adoption barrier.

In the following, we discuss Search-in-Memory (SiM) [14] and FlashCosmos [15], two prominent examples of OTS PiN that form a basis for further discussions.

Search-in-Memory Search-in-Memory (SiM) repurposes existing circuits in conventional TLC NAND flash chips to support two new functions: a SIMD-style SEARCH command for fine-grained matching and the GATHER command. SEARCH treats a data page as an array of 8-byte words and matches an 8-byte input key against each word stored in the target page, using a (masked) bit-wise AND. It evaluates to True, iff all (relevant) bits match. A SEARCH on a 4 KiB page therefore results in a 512-bit bitmap. The GATHER command pulls a subset of 64-byte chunks, indicated by a bitmap operand, from a page. The two commands always start with a *page open* command, which senses the target page into the page buffers.

Unlike typical page reads, data from the page is not sent to the SSD controller. Instead, a SEARCH command sends a query word (the "key") to the page buffer for comparison and only transmits the result back. Multiple SiM commands can be run on the page while it is cached in the registers. Once all relevant commands have been completed, a *page close* command will release the data from the register. When the SSD controller executes SiM commands, it can reduce the clock rate of the I/O bus to reduce the power consumption for the channel transfer. This does not affect the latency of the operation because the amount of transferred data is considerably reduced. SiM's efficiency relies heavily on making use of word-aligned arrays.

To implement SEARCH, which emits only one bit per word, SiM exploits the Failed Bit Counting (FBC) [46], which is usually used by SSDs during writes to verify the integrity of data programmed into flash memory. A program operation requires multiple rounds, with each round generating a pass/fail signal from each page buffer to indicate whether the cells have reached the target accuracy level. These signals are then aggregated to determine if the programmed data meets reliability requirements. This same circuitry can be adapted to perform a (word-wise) POPCNT operation,[4] though the result must be quantized to avoid high circuit overhead from precise counting.

For SiM, one measure to cope with bit-errors is to use the memory in "SLC mode", sacrificing storage density. Storing only a single bit means that only two states need to be distinguished, which makes both programming, reading and retaining data much more robust. Modern SSDs usually support switching between TLC and SLC in order to offer these benefits and avoid the performance toll, while capacity is not in great demand.

FlashCosmos FlashCosmos [15] leverages the sensing mechanism of NAND flash memory to perform bit-wise operations: logical NOT on a single page, AND across pages within the same block, and OR across pages in different blocks. By combining these basic operations, FlashCosmos can implement more complex logic functions, such as NOR, while reducing data transfer overhead by sending only a single *result* page instead of multiple *operand* pages.

FlashCosmos requires careful placement of operand pages to align with the

[4]The POPCNT instruction counts the number of bits set to 1.

logic operations being performed. For example, if a bit-wise OR operation involves pages within the same block, page migration is necessary to move one operand to a different block. Moreover, unlike SiM, which sends queries directly to the page buffer for matching, FlashCosmos requires operands to be pre-programmed into memory. This makes FlashCosmos less suitable for workloads with long computation sequences that produce multiple intermediate results, as writing intermediates to NAND memory is slow. Despite these limitations, SiM and FlashCosmos adhere to existing memory sensing mechanisms and can complement each other.

Similar to SiM, FlashCosmos relies on SLC to reduce errors in bit operations.

2.2.4 Simplicity Preferred

Generally, the layout of data structures must adhere to the SIMD-style processing of PiN and architecture specific limitations. For instance, FlashCosmos can only combined bits logically *across different pages*, resulting in a page-sized result. With Search-in-Memory, an 8 byte query argument is compared individually with all 512 many 8 byte words on a 4 KiB page. While only rudimentary, SiM performs a "bit-wise aggregation" to check if all bits in a word did match or not, producing a 512 bit result for the entire page. Many types of indices, such as B^+-trees and zone maps, are based on (integer or float) inequality operations, which are not easily available in OTS-PiN (see Sec. 2.2.1).

With this processing style, data should be carefully aligned, and page-sized arrays of a single data type are preferred. Workloads with complex arithmetic should not be offloaded to PiN and are better handled by upper-layer compute units. PiN is most effective for tasks with simple primitives. For example, it can be used to probe external hash table buckets containing fixed-length hash fingerprints [14] or to perform an exhaustive exact search across a large array of embeddings, which are commonly used to represent images or objects [47].

2.3 Processing in Unreliable Memory

Beyond NAND flash memory, bit-flips pose a significant challenge for any Processing-in-*Memory* architecture, because logic and memory are tightly integrated, using manufacturing techniques like 3D wafer bonding [48] and Peripheral-under-Cell (PuC) [49]. Bypassing existing ECC mechanisms this way requires new approaches to error management.

Hardware Error Correction Many data-center-grade DRAM memory arrays are tightly integrated with ECC on the same chip to present a seemingly error-free interface. However, as memory cell becomes smaller, data reliability continues to degrade, making on-die ECC increasingly costly and, at times, ineffective [37]. For example, DRAM refresh overhead has increased $9.4\times$ when scaling from 128 Mbit

to 16 Gbit chips [37]. Similarly, DDR5 ECC-DIMMs with on-die ECC for single-bit error correction now incur a 32.8% increase in area and parity overhead [50].

Despite these efforts to create an error-free memory interface, errors are still inevitable as memory reliability can deteriorate beyond ECC's correction capabilities. In some cases, ECC can even introduce additional errors [37]. High-Bandwidth Memory (HBM), which builds on DRAM technology, inherits these reliability challenges while introducing additional error sources due to its multi-layered architecture [38]. As memory scaling continues to degrade reliability, applications that lack built-in resilience against silent bit errors and rely solely on hardware-based error correction face significant scalability challenges. Further, they are more vulnerable to side-channel attacks like Rowhammer [37], which can induce bit-flips even with ECC in place.

Hardware ECC Conflicts with PiM (On-chip) hardware ECC would require significant power and chip area and therefore directly compete with resources needed for PiM functionality. As a result, UPMEM [10], a PiM implementation for DRAM, omits hardware ECC entirely and relies on the assumption that the underlying DRAM is error-free. High-Bandwidth Memory (HBM) can also be used for PiM [51] and therefore also faces the issue of bit-flips. Being based on DRAM, HBM both inherits it's error characteristics and also introduces additional error sources [38] due to the layering architecture.

Specifically for NAND flash, the challenge is even greater due to its inherently higher error susceptibility, compared to DRAM. Powerful ECC mechanisms, like LDPC, are essential to ensure reliability but come with significant costs: increased storage overhead for parity bits, increased read latency, and substantial demands on chip area and power. Additionally, these ECC circuits require significant testing and design costs. Consequently, integrating comprehensive ECC capabilities into PiN may be impractical [17, 52].

2.4 Error-Adaptive Databases

Beyond Processing-in-Memory, random memory errors are becoming an increasingly larger issue [37]. Many software systems recognize that storage is not flawless but still assume that bit-flips are rare [53]. For example, file systems such as ZFS and databases such as RocksDB [54] detect errors and simply abort operations, expecting that a reread will resolve the issue. However, as memory reliability worsens with advanced technology, errors become more frequent and operation aborts become a more common occurrence, causing severe service disruptions in the process.

Ultimately, solely relying on hardware to maintain an error-free memory interface is both costly and increasingly impractical. One way to address these challenges is to expose information about error characteristics to software, allowing it to make informed decisions and lessen the burden (and reliance) on

hardware-based ECC [37]. To take it one step further, memory hardware may be bought with different error guarantees [37].

Varying Impact Depending on the application, a single bit-flip may have varying impact. For example, a flip in the less significant bits of a floating-point representation usually has much less severe consequences than a flip in the exponent. Similarly, flips in pointers are more detrimental than in a column value and a crash is usually preferable over silent data corruption. Further, the impact of a single bit-flip is particularly severe in compact or highly compressed data structures [55], where every bit carries—by design—a high amount of information. As we will discuss in Chapter 3, approximate or probabilistic data structures are more suitable for coping with bit-flips, because they can be designed and configured to account for additional sources of inaccuracy.

Error-Tolerant Operations The relevance of errors in database applications has been a topic for some time now [56]. For example, Koldiz et al. [55] suggest to use arithmetic coding to enable robust computations and on-the-fly error detection for in-memory databases. In [57], they suggest methods to improve the reliability of B^+-trees. While previous works mostly focus on DRAM and are not necessarily compatible with the PiN functionality considered in this work, we believe that many of their approaches still offer important pointers to enable databases on increasingly less-reliable hardware.

2.4.1 Adaptive vs. Worst-Case

As discussed in Sec. 2.1.1, error rates in different memory regions of a NAND flash chip can vary significantly. Given the error differences across layers of 3D-memory [32], the general approach is to apply the strongest ECC across the entire memory, targeting the least reliable regions. However, this worst-case approach is costly and unscalable, as reliability discrepancies between layers are likely to increase with continued technology scaling. On the other hand, since information on current and historical error rates are often already available to devices for wear-leveling and data scrubbing [58] and can also be predicted [32], data structures able to adapt to the error can unlock many potential optimization opportunities.

With knowledge of error rates across memory regions, software can make informed decisions about data placement and data structure configuration [37]. For example, Tian et al. [32] propose partitioning NAND blocks by error levels, enabling SSD controllers to place "hot" pages in regions requiring less error correction, thereby improving latency for frequently accessed data. Similarly, applications can exploit this knowledge to tailor the protection levels of memory regions to specific reliability requirements, reducing the overhead of overprotection and improving efficiency [59]. Aligning an application's error tolerance levels with the device's varying error rates unlocks further optimization potential.

(How To) Embrace Bit-Flips If applications can deal with random bit-flips on their own, reading pages without the usual error correction and its associated overhead is preferable. The NVMe 1.4 standard introduces *Read Recovery Levels* [60], an optimization that aligns with this approach. This feature was initially intended to allow software to control the recovery effort during reads to avoid long decoding latencies for aged data pages and lets software RAID take over the recovery. This can be taken one step further for bypassing recovery entirely, if the accessed data that is known to be *resilient*.

Domain-Specific Error-Correction Applications may employ their own domain-specific error-correction mechanisms, which can be more efficient than general-purpose approaches [57]. This approach gives applications more control over performance and storage overhead trade-offs, allowing these decisions to be made before *writing* data, rather than being fixed at design time by the manufacturer.

Tailored Maintenance Error-tolerance levels specified by applications can flow back to the controller to optimize maintenance and data scrubbing, potentially increasing the longevity of NAND memory. For example, scrubbing a page may be delayed if its current error level is still acceptable for a specific application, or the controller may decide to skip scrubbing altogether, allowing the application to recreate the data later. Similarly, applications may continue using blocks that would otherwise have been decommissioned based on general worst-case criteria.

3 Cooperative Refinement

Both DCP and indexing can significantly contribute to reducing the Movement Tax, though neither is a silver bullet and each has clear limitations. However, when combined, many of their individual shortcomings can be mitigated, enabling capabilities that are otherwise infeasible. The notion of *Cooperative Refinement* proposes indexing as a workload to execute near or within storage memory, directly leveraging the resulting insights at their point of greatest impact. Despite its significant potential, realizing this paradigm increases system complexity considerably and introduces new challenges.

Data-Centric Index Evaluation The conceptual simplicity of indexing has led to its widespread adoption since the 1960s, beginning with IBM's ISAM [61]. However, index structures can also be large and their evaluation may involve considerable data movement on its own, especially for complex, high-dimensional objects. For example, image search—even after an initial pruning step and with an efficient storage layout—can still require "walking the last-mile" and exhaustively compute similarity measures for a set of candidates [47]. But even holding a large number of traditional index structures, such as B^+-trees [19], in precious DRAM can be a costly endeavor—especially in disaggregated cloud environments. To this end, evaluating index structures close to their place of storage has many advantages with respect to DRAM utilization, access latency, CPU overhead, and overall data movement cost.

However, not only do indices directly benefit from being processed near data, some also conveniently address several challenges of Processing-in-NAND. Probabilistic data structures, which are used to index complex objects, are particularly suitable: their approximate nature allows them to also account for bit-flips—one of the biggest issues of NAND flash memory. Unlike many traditional database indices, where a single bit-flip can corrupt the entire structure, these data structures

are able to degrade gracefully: query efficiency decreases proportionally to the bit error rate rather than failing arbitrarily. Moreover, workloads such as nearest neighbor search, already rely on approximate *success metrics.* When accounting for one type of uncertainty, it is often possible to incorporate another. This opens up the opportunity to *embrace a certain error rate* instead of fixing all errors explicitly (and indiscriminately) on every single access.

Some data structures, such as Bloom filters, also have the beneficial property that they rely only on relatively simple bit-wise equality tests for their evaluation. Such functionality is already implemented, for instance, via the masked equality instruction in Search-in-Memory (SiM) by Chen et al. [14]. A larger, more powerful class of index data structures relies on measures like the Hamming distance or Jaccard similarity, but already requires capabilities beyond current designs, e.g., the support of instructions like POPCNT. Although this type of bit-wise *aggregation* already requires much more complex circuitry than a basic zero check, it remains far simpler than general-purpose database aggregation, which involves intermediates and full arithmetic logic. Yet, commercially available Processing-in-NAND (PiN) prototypes capable of these basic operations are unavailable for testing. Existing designs are mostly conceptual, and their efficiency in specific applications is not well established. As a result, there is a need to consider the usefulness of specific low-level operations from a database perspective and identify feasible extensions of existing PiN concepts that would better support index evaluation.

Index-Driven Data-Centric Processing Similar to indexing, DCP has been around for some time now as well. For example, Active (Hard-Disk) Disks [62] are an early instance of Computational Storage Devices, were discussed in 1998; and ideas on processing directly with memory go as far back as 1969 [63]. Yet, historically improving bus connectivity, significantly increased system complexity, the simplicity of a “scale-out” approach, and the inherently limited resources on such devices, hindered their adoption so far. Moreover, modern SSDs are still primarily accessed via the traditional “block device” interface, where read requests operate on fixed-size pages addressed through Logical Block Addresses (LBAs). This “black-box” approach to storage access incurs non-negligible CPU overhead per request [6, 5], prompting some systems to adopt kernel-bypassing techniques like the SPDK library [64]. Despite such efforts, all communication remains explicitly orchestrated by the host—an inherently *host-centric* design.

In data-intensive analytical workloads, database systems often determine early during query planning which tables and columns will be accessed. This enables “sequential” access patterns, where *logically adjacent* pages are fetched in bulk to amortize overhead, both on the host and within the SSD. However, this bulk access strategy is often very inefficient when most of the data is actually irrelevant. Yet, even if precise knowledge about relevant bytes is available beforehand, NAND’s large access granularity still forces reading entire pages. Especially column stores, the de-facto standard for analytical systems, are susceptible to this issue. Their

high page cardinality makes it more likely that at least one value qualifies for the query, triggering the retrieval of the entire page. Consequently, they must suffer the full Movement Tax and fetch pages for *all query-relevant columns*—even for just a single tuple.

These observations highlight an untapped potential: when information on the relevance of data from an index or an earlier processing stage is available, it should be possible to exploit this knowledge *directly within* the storage device itself. To realize this potential, however, both the capabilities of the device and the interface must evolve. Using index information to issue ever more targeted page requests may reduce waste, but it does not eliminate the core problem of host-centric communication.

Instead, what is needed is a computational storage design that can fundamentally change this communication model and autonomously act on relevance information. This is not only beneficial to improve access efficiency, but also to augment the data-centric processing model itself: skipping irrelevant data early enables a more intelligent use of the device's limited computational resources, avoiding unnecessary computational effort and (device-)internal bandwidth utilization. Crucially, exploiting this relevance information is not merely about supporting subsequent operations; it already constitutes a fundamental form of computation.

Outline In the remainder of this chapter, we sharpen our vision of Cooperative Refinement by first focusing on specific approaches to implement *data-centric index evaluation*. We begin by showing the Bloom filter's inherent resilience to bit flips and then discuss it as a possible application that can be implemented using the (masked) equality test functionality implemented with SiM's SEARCH [14]. Next, we extend the capability of the instruction by proposing an emulation scheme to execute inequality operations—a core primitive in database-index evaluation. In Sec. 3.3, we discuss the capabilities of a visionary PiN system that offers POPCNT-like functionality and would therefore greatly increases the scope of feasible index applications. Our error analysis for Binary Sketches, data structure that rely on a such POPCNT instruction, shows their general feasibility for PiN. In Sec. 3.4, we introduce Gravity Store—our initial concept for materializing indexed data and utilizing index information directly within storage devices—our contribution to *index-driven Data-Centric Processing*. In the last section, we conclude the topics of this chapter by outlining potential directions for future research.

3.1 Bloom Filter Evaluation

Three primary aspects of PiN are its suitability for bulk operations, limited instruction set, and susceptibility to memory errors. Fortunately, the Bloom filter, a popular data structure for probabilistic set-membership queries, presents itself as a prime candidate. It consists of a fixed-length bit-vector and a set of independent, uniform hash-functions, each mapping to positions in this vector. Starting with an

all-zero vector, elements are inserted by setting the bits determined by the hash functions to one; previously set bits remain unchanged. After inserting all elements, the final filter, or multiple in batch, is persisted to storage. To test for membership, we hash the query element with the same hash functions, building a *query mask*. Within the NAND flash chip, we then test whether the corresponding bits were set in the filter using an instruction for logical AND, such as SiM's SEARCH. As a bonus particularly relevant to PiN, Bloom filters have—by design—a roughly uniform distribution of zero and one bits, which coincides with the goal of data scrambling, a technique used in SSDs to avoid data-dependent disturbance errors (see Sec. 2.1.1). The result may be a *false positive*, indicating membership despite the element not actually being inserted before.

They have many applications, such as in key-value databases for look-ups [65] or (rudimentary) range-query support [66], in distributed joins [67], in-memory hash-joins [68], and recently Bloom filter-aware plan optimization [69]. Bloom filters are also used beyond classic database systems, for instance in genomics [70, 71] or network applications [72]. See [73] for an overview of techniques.

3.1.1 Integrating Bit-Flips

Given the inherent susceptibility of NAND to random bit-flips, data structures operating directly within the memory, prior to error correction (ECC), require robustness. As we will demonstrate in the following, Bloom filters—only ever testing a small subset of bits—fulfill this property, yielding reasonable results even at error rates far higher than those tolerated by industry standards ($\approx 10^{-16}$ [43]). For the analysis, we assume that bit flips occur independently across the page, regardless of position or surrounding state.

Usually, Bloom filters are designed to not surpass a specified false-positive rate (FPR). Given a fixed number of bits for the filter, this leads to a maximum number of elements that can be inserted before this rate would be surpassed, i.e., its *capacity*. For target FPR t_{FP}, size of the filter m and a number of hash-functions k, the maximum capacity[1] of a Bloom filter is

$$n = -\frac{(m-1)}{k} \ln\left(1 - {t_{\text{FP}}}^{1/k}\right) - 0.5. \tag{3.1}$$

Let $p_{0\to 1}$ denote the probability of a flip from 0 to 1, and $p_{1\to 0}$ the probability of a flip from 1 to 0. Moreover, $p_{1\to 1} = 1 - p_{1\to 0}$ represents the probability that a 1 is retained and $p_{[0]} := 1 - p_{[1]}$ the probability of an initial 0. Then, the probability of *observing* a 1 after bits may have flipped is

$${p_{[1]}}^{\text{final}} = p_{[0]} p_{0\to 1} + p_{[1]} p_{1\to 1}.$$

The chance of a false positive is then $\left({p_{[1]}}^{\text{final}}\right)^k$.

[1] This slightly modified version sacrifices "1 element and 0.5 bits" for a more accurate estimate of the FPR (or, after rearranging, capacity) [74].

For fixed bit-flip probabilities and a target FPR t_{FP}, this limits the number of hash-functions k that we are allowed to employ from below. Note that the lowest inherent FPR (and thus maximum capacity) of a Bloom filter is achieved if $p_{[1]} = p_{[0]} = 0.5$, which we will substitute in the following. We rearrange for k and choose the smallest complying number of hash functions:

$$\begin{aligned}
\left(p_{[1]}{}^{\text{final}}\right)^k &\leq t_{\text{FP}} \\
k &\geq \frac{\ln(t_{\text{FP}})}{\ln((1-p_{1\to 0})p_{[1]} + p_{0\to 1}(1-p_{[1]}))} \\
k &\geq \frac{\ln(t_{\text{FP}})}{\ln(0.5 \cdot ((1-p_{1\to 0}) + p_{0\to 1}))} \\
\Rightarrow k_{\text{opt}} &:= \left\lceil \frac{\ln(t_{\text{FP}})}{\ln(0.5) + \ln(1-p_{1\to 0} + p_{0\to 1})} \right\rceil
\end{aligned} \tag{3.2}$$

Plugging k_{opt} into the Eq. (3.1) yields the *FPR-limited capacity*.[2] Observe that $0 \to 1$ flips increase the FPR and $1 \to 0$ flips decrease it. As a result, both types cancel each other out: If $p_{1\to 0} = p_{0\to 1}$, Eq. (3.2) coincides with the usual formula without errors and bit-flips have no impact at all on the FPR of the filter. In other words, for every decision that becomes false-positive due to bit-flips, we also count a false positive that turns into a true-negative decision. Without bit-flips, the optimal capacity is usually associated with a uniform distribution of ones and zeros in the filter, so a bias in either direction impacts capacity (as determined by k_{opt}) positively or negatively.

Impact of Bit-Flips on Filter Capacity The upper limit to k_{opt} and the capacity formula gives us the tools to assess the robustness of Bloom filters. We illustrate the impact of bit-flips on the bits per element under false-positive restrictions in Fig. 3.1 (next page). For growing $0 \to 1$ errors, the storage cost per element increases eventually, or—equivalently—the capacity decreases. But using a larger k is not necessary until $p_{0\to 1}$ is well above 0.01, a value significantly higher than JEDEC's 10^{-16}. For example, for $t_{\text{FP}} = 0.1$ and $p_{1\to 0} = 0$, the very first jump (from $k = 3$ to $k = 4$) occurs at $p_{0\to 1} \approx 0.125$, raising storage cost from 4.86 bits to 5.03 bits per element. Before, $0 \to 1$ errors do not have any impact on the choice of k and therefore the (FPR-limited) capacity, offering the same performance as a Bloom filter in regular "error-less" memory. Further, $1 \to 0$ flips delay the increase in bits per element. As indicated by the vertical bars, these error ranges can be reached by existing memory chips.

Note that the step-wise increase (y axis) is due to rounding k, the number of hash functions, to the next larger integer.

[2] Note that, due to hash collisions, the actual number of relevant bit may be smaller than k, the number of hash functions, especially for very small m (relative to k).

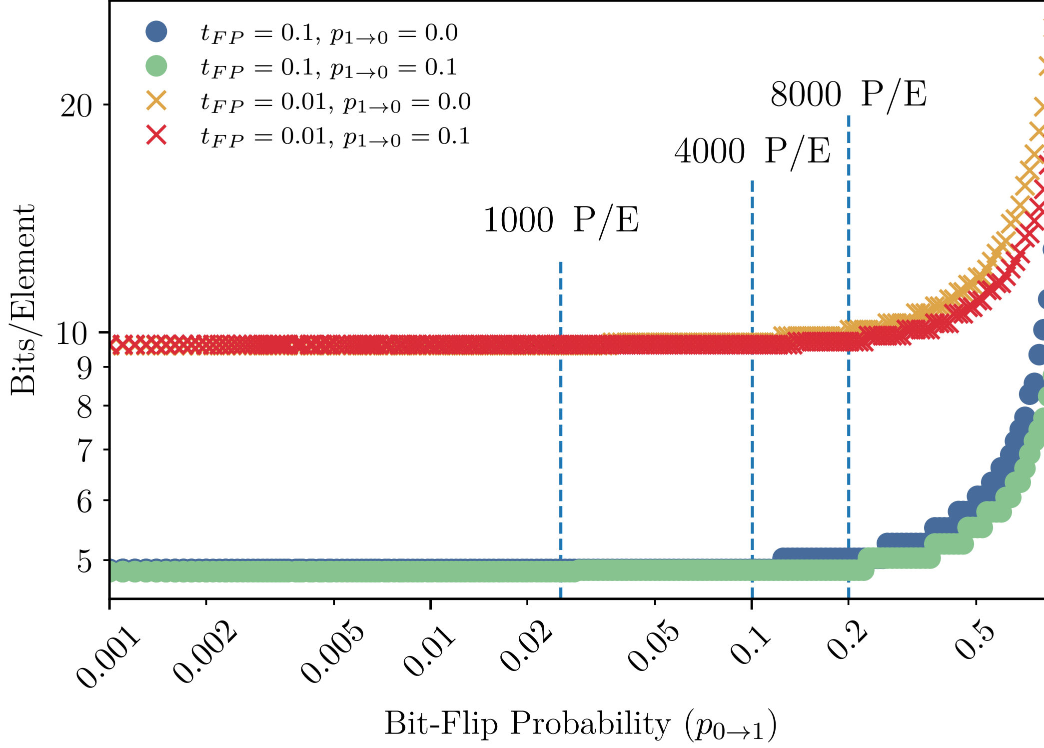


Figure 3.1: FPR-restricted (t_{FP}) bits-per-element for a Bloom filter, given bit-flip probabilities $p_{0\to1}$ and $p_{1\to0}$. For reference, the dashed vertical bars indicate the RBER of an older flash chip from Micron for different P/E cycles, after retaining data for 64 days (P/E values reproduced from [32]).

3.1.2 False Negatives Guarantees

Although $1 \to 0$ bit-flips are (surprisingly) beneficial for the capacity of the filter, a non-zero false-negative rate impacts the quality of the actual result, because actual members of the set can now be "overlooked". The false-negative rate is given by $\text{FNR} = 1 - (1 - p_{1\to0})^k$, which is the complement of the probability that none of the 1-bits flip. For applications that can tolerate a certain false-negative rate of t_{FN}, we can rearrange for k and obtain an *upper limit* to the number of hash-functions:

$$
\begin{aligned}
t_{\text{FN}} &= 1 - (1 - p_{1\to0})^k \\
\Rightarrow k_{\max} &= \left\lfloor \frac{\ln(1 - t_{\text{FN}})}{\ln(1 - p_{1\to0})} \right\rfloor
\end{aligned}
$$

If $k_{\max} < k_{\text{opt}}$, the filter can not be build optimally. This is because the optimal capacity can only be achieved with an even chance of encountering a 0 or 1, but the corresponding number of hash functions would violate our restrictions on either error rate. To compensate, we therefore add fewer elements in order to require fewer hash-functions for the same FPR. We call this a *FNR-limited capacity*.

Although the introduction of false negatives is not acceptable for some applications, others can tolerate it. In fact, across a wide range of applications in distributed systems surveyed in [75], various Bloom filter variants allow for false negatives. It is often considered an acceptable trade-off to enable item deletion [76] or improve the false-positive ratio [77].

Exploiting Error-Asymmetry If $p_{1\to0} = p_{0\to1}$, the capacity is not FPR-limited, but may be FNR-limited. In case our memory experiences one-sided bit-flips, we are able to exploit that $1 \to 0$ flips *increase both capacity and FNR*. This allows, for example, to strengthen correctness over capacity by inverting the bit-interpretation of the bitmap of a Bloom-filter, when $p_{1\to0} \geq p_{0\to1}$.

SLC NAND presents a hardware design that offers a high asymmetry. This is because the largest source of bit-flips today is electron leakage [35]: if there are only two states to differentiate, cells can only drop once, from the higher to the lower. In contrast, any substantial increase of the cell charge that would also cause a change in state is significantly less likely [35]. While being more expensive, SLC is also overall less error prone, allows faster read and write and offers more durability than, for example, TLC NAND.

3.1.3 Processing-in-NAND Implementation

We envision pages in a PiN-capable device to store multiple Bloom filters, or parts thereof. For the example configurations considered by Chen et al. [14], a 4 KiB page consist of 512 64-bit words. A page then contains, for instance, the first 64 bits of a Bloom filter, which may be longer. For evaluation, a Search instruction is issued with a query element representing the (partial) result of the hash function evaluation. Matching the query with each filter prefix results in a positive response, if all relevant bits are set in both query and filter. Compared to a regular page access, which requires transmitting the whole 4 KiB page, only the 512-bit result must be transmitted over the NAND flash chip's channel. Results from multiple pages are combined with a logical AND, potentially using an accelerator within the storage device. Each bit in the final result then indicates whether the input set at that bit's position contained the query, or not. This setup is particularly suited for applications such as the large-scale analytics file format Apache Parquet [78]. Another application domain may be Key-Value stores, such as RocksDB, where separate indices are used for many independent sets of elements or to give lightweight support for range queries [66].

Error-Adaptive Parametrization As stated in Sec. 3.1.1, the optimal capacity depends on the bit-flip rate and limits the number of hash functions. To illustrate, even the pages within the same 3D NAND block and written at approximately the same time can have vastly different error levels that vary by an order of magnitude [79]. Crucially, error rates are usually not uniformly distributed, with

pages exhibiting extreme rates potentially being outliers [79]. As a result, pages with an error above a certain threshold will drop in effective capacity (see Fig. 3.1). An application may therefore under-utilize Bloom filters stored across this block if it allocates capacity based on *global worst-case error rates.* While this is less of an issue for highly asymmetric error environments, overestimating $1 \to 0$ errors leads to severe restrictions.

For example, assuming $p_{0\to1} > p_{1\to0}$ and $t_{\text{FP}} = t_{\text{FN}} = 0.01$, classifying all pages in a block to have, say, a worst-case ratio of $p_{1\to0} = 0.0034$, forces k to be no greater than $\left\lfloor \frac{\ln(1-0.01)}{\ln(1-0.0034)} \right\rfloor = 2$. With $k = 2$, the bit-per-element ratio in the worst case would be $\frac{m-1}{n+0.5} = 18.98$. If the actual probability is $p_{1\to0} = 0.002$ or lower instead, k may be as large as 5, costing only 9.85 bits per element, a factor-of-two difference. Relying on worst-case assumptions to characterize NAND flash errors can be misleading and should therefore be avoided.

3.2 Inequality Emulation

As discussed in Sec. 2.2.1, native inequality instructions are non-trivial to implement for PiN. However, databases rely heavily on this functionality, e.g., during a B$^+$-tree traversal. It may therefore be beneficial to consider ways to emulate inequality operations with the tools already considered feasible, such as the aforementioned SIMD-style masked-equality instruction offered by Search-in-Memory (see Sec. 2.2.3). SiM's Search instruction goes beyond simple word-wise comparison operations and allows to restrict the scope of the comparison using a bit mask as an additional argument besides the comparison constant. This permits expressing logical *less-than* operations by synthesizing a series of Search instructions and combining their results logically. This approach is reminiscent of how UPMEM emulates floating-point operations with integer arithmetic [10].

Note that, in the following, we will temporarily ignore bit errors and return to it in the next section.

Example. Intuitively, the emulation works by iteratively testing for available power-of-two "frequencies" that make up the binary representation of an integer. For example, in the 8-bit binary representation of the query $q = 64 = 2^6$, only the 6th bit is set. Any value $x < q$ will therefore have both the 6th and 7th bits unset. We can implement this with a single Search instruction using a mask value of 192 ($\texttt{1100 0000}_2$ in binary) and testing for equality with 0. We denote this (abstract) operation as MEQ$(x, 0, 192)$.

For non-powers-of-two queries, multiple tests are required, each testing for a different power-of-two component. Or in other words, each emitted MEQ selects a disjoint slice of the comparison interval, and the slice sizes form a strictly descending sequence of powers-of-two. For example, for $q = 2^7 + 2^3 + 2^2 = 140$, we can use disjunction of three tests:

Algorithm 1 From-below synthesis of the predicate $x < q$ using MEQ

Input: $q \in \{1, \ldots, 2^b - 1\}$
Output: List E_{FB} of MEQ clauses such that $\bigvee_{e \in E_{FB}} e(x) \iff x < q$

1: lower $\leftarrow 0$; rest $\leftarrow q$; $E_{FB} \leftarrow \varnothing$
2: **while** rest > 0 **do**
3: $p \leftarrow \lfloor \log_2(\text{rest}) \rfloor$
4: mask $\leftarrow (2^b - 1) \wedge \neg(2^p - 1)$
5: $E_{FB} \leftarrow E_{FB} \cup \{\text{MEQ}(x, \text{lower}, \text{mask})\}$
6: lower $\leftarrow$ lower $+ 2^p$
7: rest $\leftarrow q -$ lower
8: **return** E_{FB}

$$\text{LT}(x, 140) = \text{MEQ}(x, 0, 128) \vee \text{MEQ}(x, 128, 248) \vee \text{MEQ}(x, 136, 252),$$

which tests the intervals $[0, 128)$, $[128, 136)$, and $[136, 140)$, respectively.

3.2.1 From-Below and From-Above Emulation

The synthesis algorithm for this strategy is described in Alg. 1. It proceeds by iteratively emitting operations for increasingly smaller parts of the initial query. In each iteration, the bit mask is extended by the next lower power of two present in the query. Overall, the emulation emits a number of MEQ operations that depends on the number of 1-bits in the q's binary representation.

There are two ways to approach the emulation, depending on which power-of-two components we test for. In the examples above, we tested "from-below" for the lower components in q, but we can also test for components of values larger than q and negate the final result of the disjunction. To obtain "from-above"—the dual of *from-below*—we initialize *rest* not with q but its arithmetic (not bit-wise) complement $2^b - q$ and update it via rest $\leftarrow$ rest $- 2^p$ instead. Moreover, we keep increasing *lower*, now named "upper," but initialize it with q instead of 0. Lastly, we always negate the result of the conjunction after the evaluation.

Maximum Program Length For each query q we can pick the strategy that requires the fewest operations, so we have at most

$$\min\{\textsc{popcnt}(q),\ \textsc{popcnt}(2^b - q)\} \leq \lceil b/2 \rceil$$

operations. This bound follows from the observation that *from-above* tests the arithmetic complement $2^b - q$ instead of q itself. We have that either q or its bit-wise complement $\bar{q}$ never have more than half of their bits set. Since $2^b - q = \bar{q} + 1$, either q or its arithmetic complement has at most $\lceil b/2 \rceil$ bits set.

Example. For $q = 13$ we have

$$\begin{aligned} \textsc{popcnt}(q) &= 3 & (q &= 0000\,\mathbf{1101_2}), \\ \textsc{popcnt}(\bar{q}) &= 5 & (\bar{q} &= \mathbf{1111}\,00\mathbf{1}0_2), \\ \textsc{popcnt}(2^8 - q) &= 6 & (2^8 - q &= \mathbf{1111}\,00\mathbf{11}_2). \end{aligned}$$

Hence $\min\{3, 6\} = 3 \leq \lceil 8/2 \rceil$, and the *from-below* strategy is optimal.

3.2.2 Executing the Program

A storage device that implements MEQ operations via the SIMD-style SEARCH instruction allows us to test all words in a NAND page at once. First, we execute the full sequence of SEARCH instructions, and then combine the results via logical OR to obtain a bitmap with all matching words. Notably, an "interactive" inspection of intermediate results would incur additional latency and overhead. Thus, a "one-shot" execution of the program, where it runs to completion without interruption, will be necessary to make the entire approach worthwhile.

Volume & Energy Savings Returning results instead of the full page leads to a considerable reduction in transfer volume. For example, with a word size of 64-bit and a page size of $4\,\text{KiB}$, a SEARCH instruction results in a 64-bit result bitmap—a reduction of $64\times$. Even in the worst-case, where the benefits may be offset by the need to execute up to $\lceil b/2 \rceil = 32$ operations, we still safe 50% of the volume. This reduction in volume can then be used to transfer data at a slower rate, requiring less power in the process [14]. The energy-saving potential is significantly greater, however. Modern memories (not just NAND) incur energy cost for transmitting 1-bits in the data stream, instead of a fixed amount for every transmitted word.[3] When transferring results of SEARCH instructions, we therefore pay the transfer not for all tested words, but only for those that match. Because we test for disjoint intervals, our less-than emulation therefore has energy costs that are proportional to the predicate selectivity. Note, however, that the overall latency cost of the emulation will usually be larger than a regular page transfer.

Two's Complement & `DECIMALS` The emulation so far assumes unsigned integer values. Representing signed integers instead requires some adjustments: With Two's complement for signed integers, the MSB is 1 for negative values and values are no longer comparable in the same way (as unsigned integers), because negative values are now "larger" than the positive. The issue can be avoided, if we flip the MSB of two's-complement numbers, such that a 0 indicates a negative number.

Comparisons for floating point values can be enabled by using fixed-precision `DECIMALS`: multiplying a floating point value (before storage) with a power of

[3] Signals transmitted over an electrical bus require differing amounts of energy. This allows to optimize the transmission towards specific patterns with features like *Data Bus Inversion* [80].

10 with specified exponent (which dictates the desired precision) and dropping remaining decimal digits yields an integer that can be evaluated with our emulation.

3.2.3 Improving Efficiency

There are several ways to improve efficiency, either by ensuring fewer operations are required, or by evaluating multiple words in a single MEQ operation.

Reducing the Operation Count Depending on the query q, costs may differ with our emulation. One way to reduce the number of operations is by increasing the tested query constant to a nearby value that has fewer 1 bits, thereby sacrificing accuracy.

However, another way to ensure that the operation count remains low is to give up parts of the integer domain. For example, we can *remap* the integers of an interval $[0, n)$ to a subset of the interval $S \subseteq [0, 2^b)$, where b is the word size, in an order preserving way. Ideally, S—and therefore a remapped q—would then only contain values that can be tested with few MEQ operations, i.e., that have few 1-streaks.

We denote the set that only contains integers with r many 1-streaks as S_r. One such mapping, where every word in the image has exactly two streaks of 1s, is

$$[0, 1, 2, 3, 4] \rightarrow [5, 9, 10, 11, 13] = [\texttt{0101}_2, \texttt{1001}_2, \texttt{1010}_2, \texttt{1011}_2, \texttt{1101}_2].$$

In general, the number of binary words with exactly r 1-streaks is given by $\binom{b+1}{2 \cdot r}$ [81] and thus we have

$$|S_r| = \sum_{s=0}^{r} \binom{b+1}{2 \cdot s}$$

with $r \in [0, \frac{b}{2}]$ integers to map to. So for example, for $b = 32$ bit, we can differentiate between $\sum_{s=0}^{4} \binom{32+1}{2 \cdot s} = 15033173$ different keys using at most 4 MEQ operations.

We leave deriving a generic (and efficient) procedure to establish the mapping for future work.

Sub-Word Evaluation So far, we silently assumed that the word size coincides with the size of the values we compare, i.e., every word is matched with exactly one MEQ operation. However, this may be wasteful for architectures that have word sizes larger than the values we store, e.g., 64-bit words for 16-bit values. This results in worse storage space utilization and degrades the overall efficiency.

One way to address this is to “pack” multiple values and the corresponding masks into one word each. However, when a 64-bit word packs four 16-bit keys, a single MEQ operation fires only if all four sub-words meet the masked condition. With *unsorted* sub-words that is rarely useful, because one failing lane voids the whole word.

If the sub-words are sorted, as in a *dense* key array of a B$^+$-tree inner node[4], we regain some structure. The effect is limited, however, because MEQ operations test for disjoint intervals and the sub-words may contain values from more than one integer interval. To regain the ability to pin-point the largest-qualifying value for a LT-operation, we therefore need to execute the synthesized sequence for each sub-word separately.

As a result, correct execution when storing multiple values in a single word does not decrease the required number of operations, when compared to an execution across 4 pages without word-packing. However, we may still save on storage capacity, and only have to pay the latency cost for *sensing* data from the memory cells once, not 4 times.

3.3 Binary Sketch Evaluation

Emulation schemes, like the one previously introduced in Sec. 3.2, come with additional complexity and usually require post processing with another compute unit, such as a CSD's co-processor. Moreover, restrictions still remain in what can be feasibly expressed.

A large class of data structures becomes available, if the underlying PiN hardware would be capable of POPCNT-style bit-wise aggregation. For example, combined with a bit-wise XOR operation, a POPCNT gives the Hamming distance between query bit vector $\mathbf{q}$ and bit vector $\mathbf{x}$:

$$\textsc{Hamming}(\mathbf{q}, \mathbf{x}) = \textsc{popcnt}(\mathbf{q} \veebar \mathbf{x}),$$

which is a common measure used to compare binary sketches [83, 84]. Binary sketches are a basic building block for nearest neighbor search and allow identifying all elements that have a Hamming distance below a certain threshold. Just like Bloom filters above, they share the objective of achieving independence between bits, as well as *balanced* bits, i.e., a uniform distribution of 0 and 1 bits, which is beneficial for storage in NAND memory. The inherently approximate character of the search, as well as a potential of the two types of bit-flips to negate each other's impact on the distance calculation, makes binary sketch evaluation another potential application for PiN. See [85] for a survey on the data structure.

3.3.1 Integrating Bit-Flips

Similar to our prior analysis of Bloom filters in Sec. 3.1.1, we will now analyze how bit-flip probabilities affect the effectiveness and result quality of finding the set

$$R := \{s \mid d(s, q) \leq r\}.$$

[4]This mirrors Cache-Sensitive B$^+$-trees [82], where only the first child pointer is stored and the rest are inferred arithmetically from the relative slot.

This set contains all neighbors s that have Hamming-distance d of at most r to query sketch q.

When a bit flips within a sketch, its Hamming distance to a query is altered by exactly 1. Specifically, the distance decreases if the flipped bit now matches the corresponding query bit, and increases if it no longer matches. Therefore, for the distance to become smaller than threshold r, decreases in distance must be more numerous than increases. This implies that sketches with a high actual distance are less likely to become neighbors due to bit-flips.

However, as $p_{\updownarrow}$, the overall bit-flip probability, increases, two effects emerge: more and more false positives appear as (mostly not-too-distant) non-neighbors are "swapped-in," while some actual neighbors may be lost as false negatives.

To illustrate this effect, we derive the probability of selecting a candidate with a *true* distance Hamming distance (before bit-flips) within threshold r, despite our selection being based on an *observed* distance (after bit-flips). Let S be the length of the binary sketch and $H_0, H \in [0, 1, \dots, S]$ denote the random variables for the actual and the *observed* Hamming distance, respectively. Under the assumption of independent, uniformly distributed bits of a random sketch from the database, $H_0 \sim \text{binom}(0.5, S)$. Let $N, M \sim \text{binom}(p_{\updownarrow}, S)$ be independent random variables for the bit-flips that decrease or increase the distance, respectively. We can then express the observed distance H as the sum of these variables

$$H = h_0 - N + M.$$

The probability of observing a specific distance of h, given an actual distance of h_0, is then:

$$\Pr(H = h \mid H_0 = h_0) = \sum_{n=0}^{h_0} \Pr(N = n) \cdot \Pr(M = m) \cdot I(0 \leq m \leq S - h_0)$$

where $m = n + h - h_0$ and the indicator function I drops invalid sum terms for given n. The probability distribution of H_0 after filtering is then:

$$\begin{aligned}\Pr(H_0 = h_0 \mid H \leq r) &= \frac{\sum_{h=0}^{r} \Pr(H_0 = h_0, H = h)}{\Pr(H \leq r)} \\ &= \frac{\sum_{h=0}^{r} \Pr(H = h \mid H_0 = h_0) \cdot \Pr(H_0 = h_0)}{\sum_{h=0}^{r} \Pr(H = h)}\end{aligned}$$

with $\Pr(H = h) = \sum_{\hat{h}_0=0}^{S} \Pr(H = h \mid H_0 = \hat{h}_0) \cdot \Pr(H_0 = \hat{h}_0)$. $\Pr(H_0 = h_0 \mid H \leq 51)$, the probability distribution of having a true distance of h_0 after filtering out all sketches with observed distance $h > 51$, is shown in Fig. 3.2 (next page). We chose the threshold $r = 51$, which corresponds to a query selectivity of ≈ 0.001.

The figure illustrates how $p_{\updownarrow}$ affects the probability distribution of *actual* Hamming distances within the result set R, if we filter according to the distances we

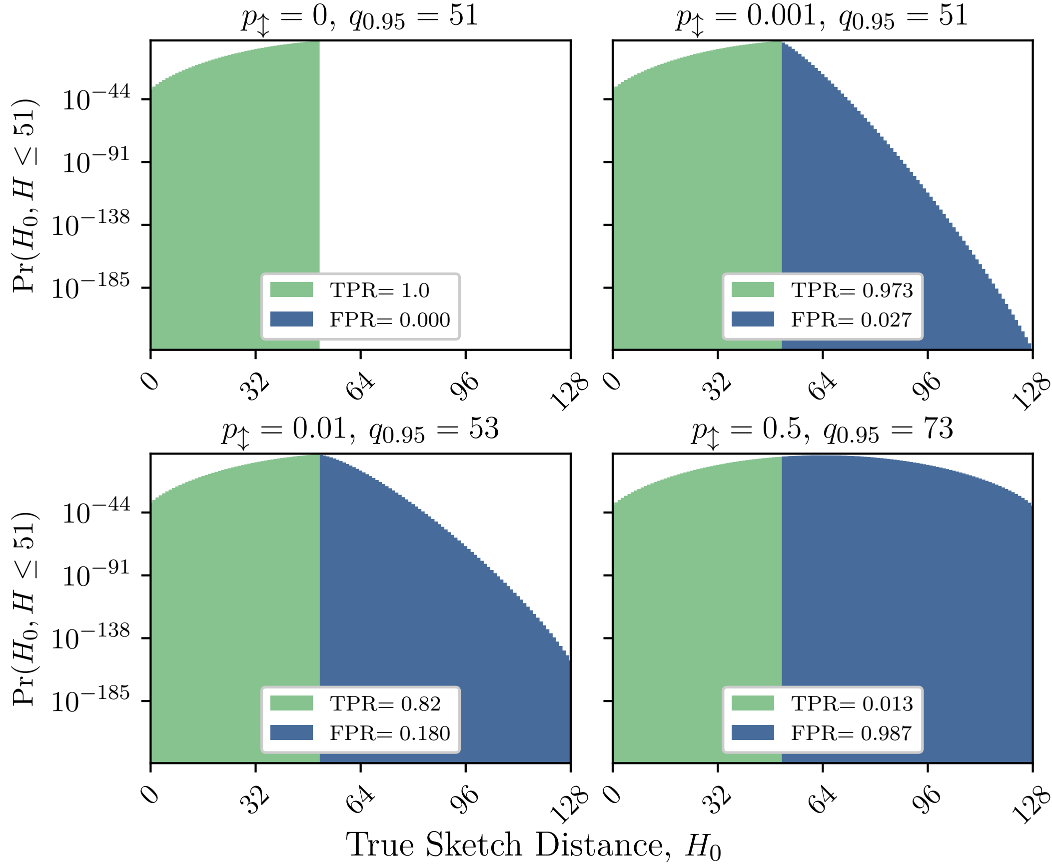


Figure 3.2: Distribution $\Pr(H_0 \mid H \leq 51)$ for sketch size $S = 128$, threshold $r = 51$ and different bit-flip probabilities $p_{\updownarrow}$. The result quality degrades gracefully for increasing bit-flip rates and high rates are required to have meaningful impact.

observe through the PiN instruction. As the error rate increases, true positives are gradually lost in exchange for false positives. However, the 95%-percentile above each sub-figure shows that—even with an error of $p_{\updownarrow} = 0.01$—the distance of 95% of all sketches is still barely larger than our threshold r and the true-positive rate is still at 0.82. With very errors, however, the result will eventually be a random sample of the dataset, with distances following the initial binomial distribution.

3.3.2 Processing-in-NAND Implementation

Following a similar approach as for Bloom filters, a page may contain a sequence of fixed-length sketches. But, as mentioned before, binary sketches are more challenging to realize via PiN. Another issue is the word size, which ideally coincides with the sketch size. Smaller words would require adding up multiple POPCNTs before thresholding can occur. Splitting the operation as well is possible but introduces another source of errors. Moreover, these modifications specialize the circuit further towards a single application, reducing general applicability.

Hardware Implementation The Failed Bit Counting (FBC) offers only rudimentary aggregation capabilities [46], i.e., to determine if the POPCNT is 0 or not. Although this is sufficient to enable SiM's SEARCH, full support of the operation would require significant changes with respect to both the hardware design of the peripheral circuitry. Moreover, the result of a SIMD-style POPCNT operation is no longer a bitmap, but a sequence of (potentially binary-encoded) numbers that represent each popcnt. One way to potentially avoid this issue may be to also add a thresholding operation, that compares the result of the POPCNT with a (ideally dynamic) constant, i.e., $\text{POPCNT}(x) < c$, and returns only the result. We acknowledge that such a hardware design requires further investigation that goes beyond the scope of this thesis.

Error-Adaptive Parametrization A single distance alteration due to a bit-flip has less relative impact, if the sketch is larger. However, considering more bits also increases the probability of encountering an error. To compensate for larger error, false positives can be filtered out in a later step, when the actual objects are accessed and refined further. This allows to choose a less strict threshold at the cost of additional post processing and larger result size.

3.4 Gravity Store

In previous sections, we focused on the specific challenges posed directly by a PiN-capable architecture. However, *complementary* near-data processing is equally important to avoid the long latency of direct host control. For example, for our emulation in Sec. 3.2, issuing SEARCH instructions as separate I/O requests and transmitting their results one by one to the host just to perform a simple bit-wise operation is unlikely to be feasible in practice. Performing the (cheap) operation directly near the chip would require a much shorter communication path.

A similar problem arises for (deceptively simple) selective access to one or more pages: Retrieving, for instance, a single tuple from a five-column dataset in a column store still pulls five (full) pages—even when the tuple's exact offset is known to the host. Even if we gain the option of cache line-granular access, for example with SiM's GATHER, we must still handle I/O operations and result transfers individually—with all the associated overhead. Issuing only a single request and receiving the fully assembled *tuple*—rather than separate, abstract *pages* (or parts thereof)—would be significantly more efficient, especially if we can specify the request for *all relevant* tuples at the same time.

To accomplish such a feature, storage devices need to do more than passively serve pages. We therefore envision CSDs capable of autonomous materialization, that is, *independently planning* data access, selectively materializing *relevant* data, and subsequently preparing *compact, meaningful data batches.* To support this autonomy in practice, the way requests are formulated needs to change fundamentally: *imperative* page requests, e.g., "fetch pages 10, 15, 73" must give way to

declarative, expressive requests, e.g., "retrieve column `A`". An autonomous device would then consult on-device metadata to generate its own internal I/O requests, fetch the necessary pages, and materialize the final, dense result (for further transfer or near-data processing). A richer request, e.g., "retrieve column `A` using bitmap `B`," would return only the matching, pre-filtered values of column `A`.

Although materialization usually does not require long sequences of complex operations, requirements on storage layouts, data types and access patterns vary significantly and are domain-specific. A framework underpinning such a device therefore requires sufficient flexibility, but still has to work within the resource constraints of near-data processing. In the following we present our current implementation of *Gravity Store*—an execution model for in-device materialization, focused on pulling together data that *matters*.

Central Components Unconstrained flexibility introduces significant complexity and overhead. To maintain predictable and efficient data access, Gravity Store adopts a structured execution model that deliberately limits programmability. It has the following core components:

1. **Layout Description Layer (LDL)**. A formal methods for mapping logical structures to their position in pages and deriving access paths. A key component is a formal *schema* notation that serves as the foundation for layout calculations and dependency inference.

2. **Materialization Graph**. Based on the LDL, Gravity Store translates a declarative request into a directed graph of tasks. These tasks can be, for example, iterators for spawning (device-internal) I/O requests and materialization operations, e.g., gather and shuffle.

3. **Materialization Engine**. A lightweight execution engine responsible for executing the graph, optimized for resource-constrained operation.

For the remainder of this section, we discuss each component and its central challenges in turn, with particular focus on the LDL.

3.4.1 Layout Description Layer

Storage layouts in analytical databases require considerable flexibility due to the wide range of data types and workloads they are expected to support. This flexibility is even more critical for data-centric architectures, where metadata, such as indices, is becoming a first-class citizen for the processing engine. In these formats, and in analytical workloads in general, a desirable property is often the decomposition of heterogeneous types into homogeneous arrays, which has several primary advantages: The ability to select only part of the information despite the large hardware-imposed access granularity, an vectorization-friendly memory layout and improved compressibility. This, however, comes at the occasional need (and cost)

for materializing or assembling tuples with values from different physical locations. As a result, depending on the workload, we require the ability to represent both heterogeneous struct-style types, as well as homogeneous array types. Moreover, there exists a plethora of indices and operations, each referring to data at different granularities, e.g., per-tuple, cache-line, page, or "row-group".[5] In addition, device-internal reasoning, e.g., component utilization or wear-leveling, may instead refer to storage constructs such as NAND-flash blocks, "super pages"[6] or planes.

Overall, we require a unified representation that maps logical data structures to both physical and logical addresses, and supports reasoning about access paths and request dependencies. In Gravity Store, all data structures are mapped to a continuous, logical memory address range. This "file-like" perspective can then be superimposed on, for instance, the Flash Translation Layer (FTL) of the SSD, to derive actual, physical addresses.

A central part of the LDL is the *storage schema*, a type-based formalism for representing the storage layout within the device. A *schema* is a *tree* and built from the following types:

- *Physical* types:
  - **Basic**: A primitive type of *static* or *dynamic* size
  - **Meta**: OFFSET, SIZE, and CARDINALITY
- *Logical* types:
  - **Struct**: heterogeneous product type, i.e., the fields have arbitrary type
  - **Array**: homogeneous type of static or dynamic *cardinality* (and size)

The instance of a type in the schema may be either physical or logical and can represent multiple values. Values of physical types occupy physical storage space, whereas logical types are containers with no inherent *physical* size; their value's (total) physical extent is derived from the sum of sizes of their nested children. Meta types are special variants of basic types, their values have static size (such as 4 bytes) and refer to other values in the schema that have dynamic size and/or cardinality. OFFSET and SIZE have bytes as their unit; CARDINALITY counts values in the schema. "*Dynamic*" in our context means that properties, e.g., size and cardinality, are not statically known at plan time and must be inferred at runtime. We may refer to specific type instances also as (schema) *elements*.

By convention, the schema itself is a struct, and the leaves of the schema are required to be physical types, which need not be statically sized. Thus, the simplest schema is $\langle T \rangle$, that is, a single physical type T wrapped in a struct.

[5] A *row-group*, as used by Apache Parquet [78], is a set of tuples of configurable physical size. The key takeaway in our context is that a row-group can have essentially arbitrary tuple cardinality and physical size.

[6] *Super pages* are an optimization used within SSDs: a 16 KiB super page consisting of 4 4 KiB flash pages is associated with 4 different NAND flash planes, which allows to read or write it in parallel with a single so-called *multi-plane* instruction.

Example Walk-Through A more complex schema is

$$\langle\ \textsc{Offset}(\text{MyArray}),\\ [\,\textsc{Cardinality}(\text{MyNestedArray})\,]_{\mathbf{i}},\\ \text{MyArray: } \Big[\ \text{MyNestedArray: } [\ \text{T}\ ]_{\mathbf{j}}\ \Big]_{\mathbf{i}}\ \rangle$$

The schema's first field is an $\textsc{Offset}(\text{MyArray})$ and stores the starting address of the third field, a nested array *labeled* MyArray. In terms of notation, $[\text{MyNestedArray}]_{\mathbf{i}}$ refers to the $\mathbf{i}$-th value of MyNestedArray, $\text{MyNestedArray}_{\mathbf{j}}$ denotes the $\mathbf{j}$-th value inside a MyNestedArray, and $[\text{MyNestedArray}_{\mathbf{j}}]_{\mathbf{i}}$ is a (physical) value of type T.

The second field in the schema struct is an array iterated by the same variable $\mathbf{i}$ as array MyArray. Using the same variable implies an association among the children of both arrays. In this example, a $[\textsc{Cardinality}(\text{MyNestedArray})]_{\mathbf{i}}$ value defines the domain of sub-variable $\mathbf{j}_{\mathbf{i}}$, which denotes the $\mathbf{i}$-th instance of variable $\mathbf{j}$. Type T is basic with statically known size $\text{size}(T)$.

Ignoring access granularity for now, we can request data via *label* by first determining its dependencies in a "backwards" scan of the schema. A label refers to exactly one (physical or logical) value in the data underlying the schema. If there are no dependencies, we infer both offset and size of the requested value and issue a read command based solely on schema information.

For example, the value of $\textsc{Offset}(\text{MyArray})$ is trivially positioned at offset 0 of the address range implied by the schema, since it is the very first physical value in the schema. Assuming the domain of variable $\mathbf{i} \in [0, I)$ is *statically* known, we can also infer the *physical* size of the cardinality array $[\textsc{Cardinality}(\text{MyNestedArray})]_{\mathbf{i}}$, which is $I \cdot \text{size}(\textsc{Cardinality})$. Its offset, $\text{off}([\textsc{Cardinality}(\text{MyNestedArray})]_{\mathbf{i}}) = \text{size}(\textsc{Offset})$, is derived from the size of its *predecessor* field in the struct.

At runtime, reading entries of the element $[\textsc{Cardinality}(\text{MyNestedArray})]_{\mathbf{i}}$ also gives the ability to fulfill the dependencies of the remaining elements and formulate requests for them. For example, the value $[\textsc{Cardinality}(\text{MyNestedArray})]_{15}$ gives us the cardinality of $\text{MyNestedArray}_{15}$. To generate a request for this entry, we also need the respective offset, which depends on the accumulated cardinality of all entries $\mathbf{i} < 15$.

Contiguity, Alignment & Label Constraints When an *array* element is one of several fields inside a struct, and this parent struct itself is repeated, the values of that element need not refer to a *contiguous* range of bytes. In

$$\langle\ \big[\,\langle\,\text{T},\ \text{X} : [\,\text{S}\,]_{\mathbf{j}}\,\rangle\big]_{\mathbf{i}}\,\rangle$$

the bytes belonging to X are interleaved with the T-fields, yielding the pattern $[\text{T}]_{\mathbf{0}}, [\text{X}]_0, [\text{T}]_1, [\text{X}]_1 \ldots$, and so on.

If either $[\text{T}]_{\mathbf{i}}$ or $[\text{X}]_{\mathbf{i}}$ has dynamic size, their offsets depend on the runtime size of preceding elements. Consequently, their alignment may change as well. Until the schema encounters the next field whose offset is "hard-coded" at schema-design time, compile-time reasoning about absolute offsets must be suspended;

only runtime access to the corresponding metadata can reestablish exact offsets beyond that point. To this end, we suggest *label constraints* to encode alignment guarantees by defining restrictions, e.g., "off($[\mathrm{X}]_{\mathbf{i}}$) mod $4096 =$ size(T)," to enforce physical page alignment. This can also be interpreted relative to an unknown but aligned parent offset, for example, to encode alignment within a page.

Note that we explicitly require encoding any padding in the schema using dedicated a type (not shown for brevity). Padding elements may be omitted for clarity from any interface facing the user (or host application).

Restrictions, Slots and Logical Size To avoid inconsistencies and inefficiencies, schema design must follow certain restrictions. In particular, the size of variable domains must be either statically known or derivable from metadata also specified within the schema. Moreover, we need to avoid cycles to ensure resolvable dependencies and therefore enforce a strict *metadata-first* policy; metadata values therefore only refer to elements that are appear *later* in the schema.

Beyond the mentioned logical constraints, a schema may still be *inefficient*. For instance, an offset may be stored "before" the element it refers to, but still end up in the same physical page. Assuming an otherwise correct schema, an access may then simply result in a single I/O to the page and subsequent pointer traversal within the fetched page. This may or may not be acceptable within a Computational Storage Device, depending on its execution model or latency constraints. So far, we have not specified restrictions in this respect and leave them to future work, as these depend on device-specific scheduling and execution capabilities.

To reason about such constraints, we introduce the notion of *slots*, which denote a *logical position*. The slot of a schema element is obtained by a pre-order enumeration of *all* elements in the schema. Most notably, an array type can occupy multiple slots, and these slots may also be non-contiguous. Moreover, we define the *logical size* recursively as the number of direct descendants, plus the logical size of each of those descendant.

Let $\mathcal{S}$ denote the label that always refers to the schema itself. By definition slot($\mathcal{S}$) $= 0$. For the small example above and $\mathbf{i} \in [0, 2)$ and $\mathbf{j} \in [0, 3)$, the elements in their slot order are

$$\mathcal{S}, [\mathrm{T}]_0, [\mathrm{X}_0]_0, [\mathrm{X}_1]_0, [\mathrm{X}_2]_0, [\mathrm{T}]_1, [\mathrm{X}_0]_1, [\mathrm{X}_1]_1, [\mathrm{X}_2]_1.$$

The logical size of the schema is card($\mathcal{S}$) $= 1 + I \cdot (1 + J) = 1 + 2 \cdot (1 + 3) = 9$.

Different elements always have different slots, but may share a physical position. For example, $\mathcal{S}$ and $[\mathrm{T}]_0$ share offset 0.

Labels always correspond to exactly one slot and, if referring to an array, may be sub-scripted to refer to its children. Conversely, referring to the label without the subscript implies *all* associated values. For example X refers to slots $2, 3, 4, 5, 6, 7$, $[\mathrm{X}]_0$ to $2, 3, 4$, and so on. Furthermore, we define *predecessor* and *successor* in terms of slots, which can differ from *parent* and *child*.

Another useful notion for reasoning about the structure of the schema is the *tuple notation*, which encapsulates nesting and ignores physical types: the length of

the tuple is determined by nesting depth, and at each (struct) position we enumerate the children on this level. In case of an array, we use the corresponding variable instead. For the small example above, its two leaves (and also implicitly the full tree structure) can be represented as $(\mathbf{i}, 0)$ and $(\mathbf{i}, \mathbf{j})$, respectively.

Annotations & Superimposing Layers As mentioned before, the same data may be considered from different perspectives, such as “physical pages” or “row-groups”. We intend layout analysis to be able to consider the schema in different “layers” for cross-referencing meta information, where each schema is used for a different purpose. Depending on the type of layer, this may be done using simple modulo-based arithmetic, more complex layers may be implied by mapping tables, e.g., the FTL. Another feature are range *annotations*, e.g., for caching header pages semi-permanently in a small device-internal DRAM buffer for fast access.

3.4.2 Materialization Graph & Engine

As already hinted at with the (nested) calculation of “card($\mathcal{S}$)” for the example above, we use the schema notation to derive the necessary information to infer (at query plan time) and fetch (at runtime) dependencies. Given a specific logical schema, we map requests, defined in terms of labels, by first identifying all mentioned elements, and then backtrack the dependencies. This allows us to build the *materialization graph* that guides execution, where vertices represent operations and edges dependencies.

The most essential node is the `IOGenerator`, and its most basic variant issues exactly one I/O for a given of offset and size. More complex generators are capable of emitting more than one request and are defined by a code-generated math expression, derived from the schema. This allows to flexibly account for strides and variable access patterns. In our current python-based prototype implementation, we use the Sympy package [86] to resolve symbolic math expressions derived from the schema. These complex generators may require runtime arguments of well defined format (as per the schema), e.g., a contiguous array of $\textsc{Offset}(\text{MyArray})$. Note that most generators may be simple enough to be hard-coded, e.g., if the requests are sequential and have constant strides or if the generator can directly issue requests from the input without computation.

Other nodes will represent materialization operations (“compute”), e.g., for gathering relevant values from a page or scattering values into a result buffer. Determining the exact scope of this functionality is subject to future work.

Executing the Graph As mentioned before, edges represent dependencies. If the preceding node is a generator, then this requires awaiting an asynchronous I/O and represents a central (and implicit) aspect of plan execution.

The graph also exhibits *loop-carried* edges, because most generators run repeatedly, e.g., the offset of element $i+1$ depends on the size of element i. These

feedback edges add a second dependency dimension (iteration order), but they do not hinder static schema analysis as long as the feedback always points *forward*, i.e., from iteration i to $i + 1$. Conceptually, we can "unroll" the loop, yielding a finite, acyclic execution pipeline *at runtime*.

However, these cycles may complicate resource management. Without explicit limitations such contention can deadlock the pipeline even though the logical dependency graph is cycle-free. For instance, different iterations of the graph may try to unsuccessfully acquire a subset of multiple necessary buffers from a limited pool, which can lead to deadlocks. This necessitates safe locking protocols for resource allocation or "windowing" techniques to restrict the number of simultaneously active loop iterations.

Although we do not consider explicit filtering or aggregation functionality in the scope of Gravity Store, size and latency for any (partial) result are still potentially difficult to predict. We hope to address this issue via schema reasoning.

Pull or Push? Another important topic is the execution model of the engine, which impacts aspects, such as operator implementations, workload balancing and also the host-facing interface.

This is usually discussed under the "push vs. pull-based" execution model [87–89] in the database context. In the context of Gravity Store, many of the relevant aspects of the discussion are less critical, since operators are designed to have low computational complexity and execution is frequently interrupted by asynchronous I/O requests. For instance, one of the main motivations for push-based execution in in-memory databases is the exploitation of code and data locality [88]. In contrast, pipeline continuations within Gravity Store can potentially run on an entirely different compute unit and may be therefore be more accurately described as *event-based*.[7]

An open issue in this context is how a result data stream is organized at runtime. In principle, the declarative nature of the interface allows to request the content of the entire storage device, which can exceed buffer capacity on the host side. Especially in the case where the host may consume data slower than the device produces it, back-pressure needs to be applied to throttle the data stream. In the context of push-based in-memory systems, this is often solved by a *credit*-based communication [91] along the pipeline: each credit represents buffer space and can be passed downstream by an upstream operator (or the host); if no credit is available, producers stop their operation. Note that this system can also be used to implement operations, such as `LIMIT`. Implementing this approach in our context may be done via coherent memory exchange between device and host for the credit-information via CXL [92]. Ultimately, device-local hardware constraints should be the main driver of execution, not the (very loosely) specified host request.

[7]Similar to the discussion in this thesis, the *gravitational force in general relativity* is considered neither a pushing nor a pulling force, but rather the movement of matter along curved space-time [90]. Nevertheless, popular consensus seems to hold that gravity is a *pulling* force.

Thus it may be necessary to complement a credit-based mechanism with additional device-internal resource schedulers, e.g., in case thermal throttling is needed.

Declarative Interface Declarative access over tree-shaped, nested layouts is extensively discussed in the field of comprehension-based query languages, such as monadic list comprehensions [93], XQuery [94, 95], or LINQ [96]. These languages are very expressive, but can nevertheless inspire the interface of Gravity Store.

A key aspect of the interface should be the ability to *nest* parts of the query using the schema's array variables. Without nesting, a request would always result in a "flattened," type-homogeneous representation and not allow to "assemble tuples" from, e.g., a columnar format. To illustrate this, consider this variation of the earlier example:

$$\mathcal{S} : \big\langle\, \big[\, \langle\, \mathrm{T},\ \mathrm{X} : [\,\mathrm{S}\,]_{\mathbf{j}}\, \rangle \big]_{\mathbf{i}},\ \mathrm{Y} : \big[\, [\,\mathrm{S}\,]_{\mathbf{j}}\, \big]_{\mathbf{i}}\, \big\rangle.$$

A request may be formulated as

$$\mathcal{S} : \{([\mathrm{X}_{\mathbf{j}}]_{\mathbf{i}}, \mathrm{Y}_{\mathbf{i},\mathbf{j}}) \mid \forall \mathbf{i}, \mathbf{j}\}$$

and would result in

$$\Rightarrow ([\mathrm{X}_{\mathbf{0}}]_{\mathbf{0}}, [\mathrm{X}_{\mathbf{1}}]_{\mathbf{0}}, \ldots, [\mathrm{X}_{\mathbf{0}}]_{\mathbf{1}}, \ldots, \mathrm{Y}_{0,0}, \mathrm{Y}_{0,1}, \ldots, \mathrm{Y}_{1,0}, \mathrm{Y}_{1,1}, \ldots).$$

where X and Y are delivered separately, i.e., "flat". With nesting, we may formulate our request as

$$\mathcal{S} : \{\ \{([\mathrm{X}_{\mathbf{j}}]_{\mathbf{i}}, \mathrm{Y}_{\mathbf{i},\mathbf{j}}) \mid \forall \mathbf{j}\} \mid \forall \mathbf{i}\ \}$$

which may yield

$$\Rightarrow (([\mathrm{X}_{\mathbf{0}}]_{\mathbf{0}}, \mathrm{Y}_{0,0}), ([\mathrm{X}_{\mathbf{1}}]_{\mathbf{0}}, \mathrm{Y}_{0,1}), \ldots, ([\mathrm{X}_{\mathbf{0}}]_{\mathbf{1}}, \mathrm{Y}_{1,0}), \ldots).$$

instead.

Overall, the required feature set of the interface (language) has yet to be determined and depends on which capabilities are feasibly implemented within a Computational Storage Device.

3.5 Discussion

In this chapter, we first sharpened our notion of Cooperative Refinement by discussing its two forms: *data-centric index evaluation* and *index-driven Data-Centric Processing*. We delved into the feasibility of Bloom filters as an application for PiN-based indexing leveraging masked-equality instructions—a technique already introduced conceptually in the PiN literature. We also examined potential extensions to this approach by emulating inequality conditions via masked equality, and by analyzing binary sketches as an alternative index structure relying on POPCNT. With Gravity Store, we took first steps toward enabling the use of index information within a Computational Storage Device, and toward in-storage materialization and declarative, domain-adaptive storage interfaces for databases. Each of these discussions present an early, conceptual contribution, and their success will rely on several complementary innovations, such as innovative PiN functionality (POPCNT) or the availability of testable hardware prototypes for PiN.

The remainder of this discussion outlines several directions for expanding on these topics.

3.5.1 Error-Analysis Scope

The analyses we have considered for Bloom filters and binary sketches only account for a single evaluation of the respective data structure, which limits their scope. For example, even a single NAND flash page typically contains multiple data strutures and workloads access potentially many pages for one query, increasing the overall probability that the entire operation contains errors. Moreover, as discussed in Sec. 3.1.3 as well, bit-flip ratios vary across the memory fabric. This variability necessitates accounting for fluctuating error rates in predictions [32] and through regular checkups. Additionally, data still requires scrubbing by re-writing the block to reset its error state. A more comprehensive treatment should therefore also incorporate economical aspects, such as life-time extensions or energy savings from reduced write-rates. Lastly, the actual impact of the power saving potential needs to be established, ideally via hardware prototype or, alternatively, through simulations.

Other Data Structures Beyond binary sketches, there are various other data structures that rely on metrics that are similar in complexity to the Hamming distance. An example is the Jaccard similarity

$$\textsc{Jaccard}(\mathbf{q}, \mathbf{x}) = \frac{\textsc{popcnt}(\mathbf{q} \wedge \mathbf{x})}{\textsc{popcnt}(\mathbf{q} \vee \mathbf{x})},$$

which may be used to evaluate MinHashes [97]—a data structure to efficiently compare the overlap between two sets (if represented by bit-vectors). A PiN-based execution may produce both nominator and denominator, whereas the division

would be executed in a near-storage compute unit. Extending the analysis to these types of data structures may therefore yield additional applications for PiN.

We also suggest MinHashes as a means to detect expensive posting list intersections in Chapter 5. However, a PiN-based evaluation may not be suitable, given that their application in the context of Team-based indexing would require *pair-wise* comparison within a larger set of MinHashes. Instead, PiN-style operations are more suited to workloads that involve point-like accesses, i.e., a comparison where one query object is compared with a large set of other objects.

3.5.2 Further PiN Applications

In addition to the techniques discussed throughout this chapter, several other indexing strategies show potential for PiN acceleration. These approaches explore different encoding, masking, and summarization techniques that align with PiN's processing style.

Other Match-based Indices PiN can support *partial* index searches by applying masking during query evaluation. This idea extends beyond the *Bloom filter* discussed in Sec. 3.1. Another example is querying a relational database table encoded as compact keys. These keys are created by concatenating truncated or compressed attribute values from a tuple. Such an encoding is employed in MyRocks [98], a NoSQL store layered on top of a relational system. Storing these encoded keys in OTS-PiN-capable storage allows SiM's SEARCH operation. The instruction's mask argument facilitates skipping irrelevant attributes and exhaustive fuzzy searches. In addition, quantization—representing value ranges with a single code—enables rudimentary range-query support via logical AND operations.

An alternative indexing strategy uses a short, fixed-size bitmap per attribute and table chunk. Each bit signals whether the chunk contains values from a specific domain interval, akin to the range filters in [99]. For example, the domain of a floating-point variable may be split into 64 intervals using 63 quantiles, q_i. Each chunk sets the i-th bit in its bitmap if it contains values in $[q_i, q_{i+1})$. To evaluate a query, we set all corresponding bits in a query mask and perform a bit-wise AND. A non-zero result indicates that the chunk may contain qualifying values. Although this approach may yield false positives for values close to the predicate constants, it captures the active domain of the chunk in greater detail than a min-max Block-Range Indexes (BRIN) [100] and allows for more expressive conjunctions across multiple attributes.

Bitmap Index Evaluation As NAND capacity continues to grow, workloads can afford to spend more space on indices to accelerate queries, trading improved capacity for runtime performance. Heavier-weight structures, such as (binned) *bitmap indices*, often consume too much space to be held in DRAM, especially so if the datasets are high-dimensional. As a result, their high access cost makes

them unattractive in current von Neumann-style systems. By evaluating indices in-place, PiN avoids transferring large index structures to the host, reducing both data movement and relieving pressure on the host's page cache [14]. While bitmap indices consume more space than lightweight formats like Apache Parquet [78], they offer significantly greater discriminative power. Moreover, their ability to efficiently combine arbitrary attributes with simple logic instructions ensures flexibility for ad-hoc queries, where the set of relevant attributes may change from query to query. FlashCosmos enables page-wise logic operations and therefore directly implements the evaluation of bitmap indices [15].

However, an open issue is the potential susceptibility to bit-flips: FlashCosmos proposes improved programming and the use of SLC NAND to reduce error rates. Still, it remains unclear whether such measures are sufficient for bitmaps, especially given the growing unreliability of dense NAND cells. Another closely related issue is compression: storing bitmaps without encoding exacerbates their already large footprint. Since a single bit-flip can cause an entire tuple to be missed, increasing information density through compression further amplifies this vulnerability.

Beyond error-handling, FlashCosmos' concept generally aligns with our Team-based indexing strategy that we will introduce in Chapter 4.

3.5.3 No Singular Processing Paradigm

As discussed earlier, simply equipping NAND with compute capabilities is not sufficient. Neither PiN nor Computational Storage (CS) alone can fully address the performance and flexibility needs of modern database workloads. PiN is highly effective for simple, early-stage data reduction tasks by operating directly within the memory fabric and minimizing internal data movement.

However, PiN's scope is inherently constrained by what can be efficiently expressed. Computational Storage, by contrast, provides more flexible logic near the memory [8, 7], thereby enabling, for instance, in-storage aggregation, join pre-processing, or the orchestration of PiN output across channels. The two approaches are therefore complementary: PiN reduces volume, while CS adds adaptability and elementary materialization functionality, forming a symbiotic relationship.

This hybrid model enables a streaming[8] architecture, in which raw data from multiple storage sources is incrementally refined as it moves upward through the stack. By the time it reaches the CPU, it has been significantly filtered, condensed, and aligned with the needs of the query. This vision closely mirrors the data-flow models discussed in cloud-native environments [18], but now grounded in storage-level processing.

Heterogeneous PiN Generally, specialization provides higher efficiency. In this sense, instead of just one type, a database system may want to make use of various

[8]The early works on Active Disks [62] also suggested a streaming model for the access to Computational Storage.

PiN-capable memories, each tailored to a different function. For example, one type of memory may be used for bitmap indices [15], another for Bloom filters (see Sec. 3.1), and yet another for binary sketches (see Sec. 3.3).

3.5.4 Transparent Interfaces & Database Kernels

When integrating PiN, a key consideration is how its functionality can be made available to (host-)applications.

Compute Express Link (CXL) [92] promises a more efficient alternative to the block device interface by moving data transfer obligations to a dedicated controller. It enables the exchange of memory between devices, the host, and other platforms in a unified, (optionally) coherent way. Beyond simply exchanging memory, it is possible to back memory ranges exposed via CXL not just with actual physical memory: accessing such a *logical* range may transparently trigger a computation and dynamically generate the necessary data. This concept, dubbed *Database Kernels* [7], enables applications to expose, for example, the same physical data in both columnar and row-based representations via two different memory ranges. For more dynamic operations, the host can expose operands of the computation before the access using a different, dedicated address range, which is coherently available to the device via CXL.

Such an interface may also be used to expose data in PiN-capable memory to the host and/or an accelerator. For example, one memory range may present a page containing a set of search keys in their raw form, e.g., for initially writing the data into storage. Another (smaller) range could then contain the result of a PiN operation executed on the very same page. Taking this approach one step further, CXL may be used to implement the data exchange for Gravity Store. For example, one memory range may expose materialized data structures, and another may hold associated meta-data. Overall, we consider CXL—and more specifically Database Kernels—a complementary technique to the concepts we have discussed so far.

4 Team-Based Indexing

Executing analytical range queries at petabyte scale poses considerable challenges due to the sheer volume and high dimensionality typical of scientific datasets. Early data reduction is therefore pivotal in minimizing resource usage and ensuring feasibility. This is particularly critical for selective queries, which are often intentionally designed to produce result cardinalities small enough for subsequent processing without excessive overhead.

However, traditional index structures are inherently ill-suited due to the "curse of dimensionality," which severely degrades their efficiency. Compounding this, their storage footprints can become similar or even larger in size to the data being indexed, competing for the already limited storage capacity. Consequently, multi-dimensional index structures (MDIS) are frequently considered infeasible, leaving exhaustive scans as the only perceived option for large-scale data analysis. For particularly massive datasets, these scans may even be performed only on materialized subsets to manage performance [101]—a compromise that can unfortunately affect the correctness of query results. The absence of effective indexing further complicates predicate fine-tuning, as obtaining accurate high-dimensional cardinality estimates becomes an exceedingly difficult task, thereby hindering robust query optimization and workload management. In addition, applications at this scale often produce their extensive datasets over a long time in an append-only fashion and have to support a wide range of workloads. This generally rules out strategies that rely on extensive pre-processing or sorting of the whole dataset. Use-cases that match these characteristics include, for example, physics analyses at the particle accelerator at CERN and other scientific projects, which continuously generate experimental or observational data.

With the absence of indexing, the gains that can be had from a data-centric paradigm by pushing operations closer to the data are also limited. Although its potential is great, the relevance of tuples can typically only be determined after

combining information from multiple (usually separately stored-) columns. Subsequently, even highly selective queries still entail significant initial data movement, e.g., from the NAND flash chip to the CSD's accelerator. With the unavailability of (index) information on what to skip, CSDs lack the ability to focus their efforts on what is relevant and are therefore more easily limited by their constrained computational resources. To minimize the Movement Tax in applications where its impact is severely felt, enabling Cooperative Refinement through effective use of indexing may be a promising strategy.

Outline In this chapter, we introduce the Team-based indexing strategy, which extends the concept of bitmap indices. We describe the *grid-index*—a simple yet sufficient data structure that serves as an easy-to-understand foundation for the following discussions and also inspires the design of our benchmark data generator, introduced in Chapter 7. Finally, we outline the conditions under which Team-based indexing is applicable.

4.1 Methodology

Secondary indexing for multi-dimensional range query evaluation in the context of databases is usually considered at two extremes: either all dimensions are indexed together by one MDIS or each attribute has its own single-dimensional index structure (SDIS). However, either variant has considerable caveats: Multi-dimensional indices, e.g., the X-tree [102] or R^*-tree [103], generally suffer greatly from the curse of dimensionality, resulting in infeasible storage and execution costs. On the other hand, creating indices for all dimensions individually, e.g., bitmap indices [21], also comes with a large storage footprint and access to large parts of the index on every query. Moreover, predicates usually offer only moderate selectivity, so loading and intersecting many bitmaps can become expensive.

Instead of indexing all dimensions at once or every dimension separately, we consider a middle ground: multiple indices over *moderately-sized subsets* of attributes, which we dub *Teams* in the following. This strategy enables a trade-off between (individual) index structure complexity, storage overhead and index intersection performance. Similar to bitmap indices, moderately-sized Teams can still cope with the curse of dimensionality, but also harness benefits from multi-dimensional indexing. Bitmap indices therefore present a (one-dimensional) edge case of Team-based indexing.

However, we are now presented with new design challenges that warrant careful consideration. For one, there are now multiple ways to choose Teams, i.e., the "Team composition," as well as suitable (and potentially differing) index structures for each. Moreover, since modern SSDs offer significantly faster (random) access to index data compared to HDDs, achieving efficient index intersection requires both careful implementation and logical optimization.

4.1.1 Evaluation Strategy

Team-based index evaluation generally follows four steps:

1. **map** the query to relevant Teams and respective inverted lists
2. **optimize** the intersection logically
3. **retrieve & combine** relevant lists
4. (refine candidates)

For example, for the special case of bitmap indices, the query is first mapped to relevant (potentially binned [104]) bitmaps of relevant attributes and these individual index results are subsequently combined. At the other extreme, using a single Team requires substantial effort to map the query to "leaf nodes" that contain matching tuples, while the subsequent intersection phase becomes trivial.

Logical optimization (of the intersection) becomes important for non-trivial index intersections, e.g., if there is more than one relevant Team. We will discuss it further in Sec. 5.1.

In general, index evaluation may lead to *false-positive* results, i.e., tuple IDs that are near the query region but lie outside it. This is usually the case if the query rectangle[1] does not perfectly align with how the index structure partitions the data space. For example, a (binned) bitmap index has to access all bitmaps that overlap even partially with the query. Similarly, an X-tree's leaf may contain irrelevant data, if the corresponding bounding box overlaps. False-positive results need to be pruned later in a step usually referred to as *candidate refinement* and therefore inflate the overall index selectivity—the index looses *precision*.[2] Note that we do not discuss candidate refinement, which is highly application specific, further in this thesis.

4.1.2 Grid-based Index

Team-based indexing is generally compatible with a wide range of index structures. However, in this work, we will focus on a specific type of index: a simple, Cartesian grid-based partition of each Team-space into a fixed number of disjoint, multi-dimensional *bins*. To partition a d-dimensional Team space, we define bin borders along each dimension/attribute by $b - 1$ equidistant statistical quantiles. For example, with $b = 10$, the first bin of one dimension spans the interval $(-\infty, q_{0.1}]$ and the last bin covers $(q_{0.9}, \infty)$, where $q_{0.1}$ and $q_{0.9}$ denote the 0.1 and 0.9 quantile values, respectively. Using quantiles implies that the marginals[3] along each grid

[1] We consider only (hyper-)*rectangular* queries, i.e., conjunctions of independent range predicates.

[2] Precision $= \frac{|\{\text{actual result}\}|}{|\{\text{index result}\}|}$.

[3] The term "marginal" refers to the discrete 1-dimensional distribution resulting from summing up bin cardinalities over all dimensions except one, rather than over an arbitrary, proper subset of dimensions.

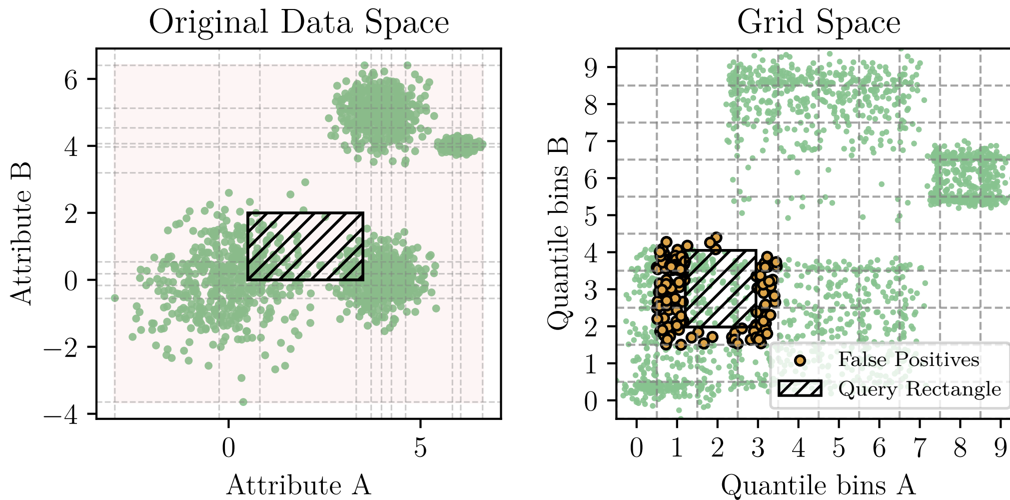


Figure 4.1: A 2-dimensional Team data space and the corresponding Cartesian-grid space with $b = 10$. False-positive results are elements located outside the query rectangle but still inside query-relevant bins.

dimension have an approximately uniform distribution, i.e., cardinalities of all bins in an interval sum up to $1/b$-th of the data. The way this transforms the data representation is illustrated in Fig. 4.1: dense regions are, to a degree, spread out and some bins may remain empty. Overall, there are exactly b^d bins and each bin has a relative cardinality ranging between 0 to $1/b$. Note that the relative position of the values within each bin is not stored, which makes it impossible for the index to distinguish true-positive and false-positive results.

We will call a subset of attributes a *Team*, the associated SDIS/MDIS *a Team index* and the result it produces the *Team result*. A specific set of Teams forms a *Team composition*. A *leaf* refers to the posting list associated with a specific, *non-empty* bin in each of the Team grids, akin to a leaf in a R*-tree or a bitmap in FastBit. A *list* may refer to either a leaf or an intermediate result.

Alternative Bin Borders In practice, marginals do not have to be exactly uniform and the bins can be chosen independently and with some flexibility. For example, in our running example (see Sec. 5.3 below), we use a separate bin for non-domain values (`NULL` and similar) in some attributes, which slightly skews the marginal distribution of the grid. The use of such out-of-support- or "sentinel" values is not uncommon in scientific applications and can yield spurious quantiles, resulting in non-uniform marginal distributions, even without explicit attention. We found the chance of *precisely* evaluating predicates involving these special values (e.g., `A == NULL`) to be an efficient application of domain knowledge.

Choosing grids with (roughly) uniform marginals is the most robust choice in the absence of domain knowledge. However, if information on frequently queried

regions of space is available, e.g., from previous queries, bin borders may be adapted accordingly. For example, using more bins in frequently queried regions of an attribute domain can reduce false-positive rates. Note that the number of bins per attribute is ultimately limited, which implies a trade-off: when specializing bin borders, predicates formulated over other regions become more expensive or potentially non-selective.

Why Not Another MDIS? We chose to focus on this approach for various reasons. First, the cost of storing metadata and mapping a query to relevant leaves is decoupled from the database cardinality. We make use of the metadata, such as leaf counts and cardinalities, for the purpose of optimization and can cheaply obtain it during at query planning time. Second, defining queries on grid coordinates instead of attribute domains gives a convenient and abstract way of generating predicates/queries for benchmarking. Moreover, the parameter b, i.e., the number of bins per dimension, gives a method of flexibly and uniformly configuring the precision of the index, which, similar to other MDIS, can return false-positive results. Lastly, and perhaps most importantly, high-dimensional queries do not require high precision per Team index. See Sec. 4.3 for a discussion.

4.1.3 Application Conditions

Team-based indexing has specific conditions that need to be fulfilled in order to reveal its full potential. We pay particular attention to the following properties:

1. high query dimensionality
2. sufficiently large table cardinality
3. sufficiently selective predicates (**SSP**s)
4. sufficient attribute coverage
5. highly selective queries

High query dimensionality is central to the usage of Team-based indexing. Otherwise, a single Team with a different, potentially more suitable MDIS, is likely more efficient on its own. Moreover, we require the table to be large enough or the (fixed) costs associated with using our grid-index are not efficiently amortized (see Sec. 5.5). Predicates are considered *sufficiently* selective, if they allow pruning at least one bin. For example, for $b = 10$, a predicate with selectivity $s > 0.9$ can not prune *any* bins in order to avoid overlooking results, i.e., false-negatives. Further, most attributes in query-relevant Teams should have SSPs, i.e., *coverage* of the considered Teams, otherwise the access and intersection performance per Team degrades. We define attribute coverage more formally in Sec. 6.1 and discuss the cost of irrelevant attributes in Sec. 5.4.3. Lastly, the overall query needs to be

highly selective in order to make indexing in general worthwhile. This criterion is generally hard to assess, since the same selectivity may have different performance implications, depending on the application. For example, determining upper bounds for cardinality estimation may still be worth-while for low selectivities. In this work, queries we classify as highly selective typically have selectivities of $s \ll 0.01$.

4.1.4 Other Related Work

The concept of *vertically* splitting a higher dimensional space into lower-dimensional projections and to consider them separately finds application in various works. The most relevant is "tree striping" [105], where a Team-based indexing strategy is suggested as a new MDIS. A key difference to our work is that they consider the intersection of *clustered* indices as an external merge-sort problem. Modern DRAM capacity and lightweight compression algorithms allow implementing the intersection of *secondary* indices in-memory, even for very large problems. Beyond tree striping, the overarching concept also finds application in other fields, e.g., product quantization [106] for approximate nearest neighbor search, where the larger data space is split into a set of independent, lower-dimensional subspaces.

Other MDIS The simple, Cartesian grid-based index considered in this work differs in various aspects from many other MDIS. Due to the relatively small number of leaves and use of (axis-aligned) quantiles, mapping a query rectangle to relevant leaves is cheap and does not require a hierarchical structure used by tree-like indices, such as X-tree [102] or R*-tree [103]. Just like bitmap indices, our index structure may be considered "leaf-only." Similar to the hyperoctree [107], the space partitioning of the grid index is *strict*, i.e., there is no overlap between bins and leaves. Moreover, leaves in the grid index do not have equal cardinality and can become larger than the leaves of traditional indices that usually aim at a small, fixed leaf size of typically 4 KiB to facilitate efficient storage access. Due to the smaller leaf size, these indices can offer higher precision and be overall more efficient for low-dimensional queries.

Another, conceptually related MDIS is the Z-order index [108], which encodes multi-dimensional values via bit-interleaving. It *implicitly* defines a regular (hierarchical) grid whose resolution (and thus the number of bins) depends explicitly on the chosen bit-precision. Similar to the grid spanned by dimension-independent intervals, bins defined by Z-order indexing are fixed by the encoding and their occupancy depends on the underlying data distribution, meaning many bins may remain empty. Note that a Z-order index can be applied on a quantile-based, quantized representation. In this sense, the Z-order index can be used "on top" of our Cartesian grid index to find relevant leaves.

A direct end-to-end runtime comparison between many established MDIS techniques is not straightforward and requires considerable programming effort. Most implementations, such as FastBit [21], are single-threaded, purely in-memory or make use of synchronous I/O. Our prototype makes use of asynchronous I/O,

lightweight list compression and task-based multi-core parallelism. See Sec. 5.2 for more details.

Compression & Search Engines An important aspect of the performance of search engines and bitmap indices alike is a representation that reduces the storage footprint of "posting lists" but still allows for efficient processing. Subsequently, many insights from these research directions can be leveraged within our framework as well. Of particular interest are compression and encoding schemes for unsigned integer sets, such as WAH [21], PFOR [109] and Roaring bitmaps [110]. These schemes were developed to support fast set operation algorithms, such as (k-way) galloping intersection [111–113]. Another relevant topic is the cost-based optimization of intersection order [113].

4.2 Index Creation

The entire index can be built in linear time of the table cardinality N. First, we specify bins using estimated per-attribute quantiles [114]. Second, we determine the maximum Team size and subsequently the actual Team composition (see Chapter 6). In a second pass over the dataset, we create the posting lists by assigning every tuple to exactly one *bin per Team* using the respective (quantile-based) grid definitions. Lastly, separately for each Team, the assignments are persisted in one of two ways: *inverted* as explicit posting lists/leaves or *flat* as a VA-style linear file [115]. We focus our attention on the inverted representation.

4.2.1 Storage Layout

Posting lists are always stored at the beginning of a page with padding at the end of a partially filled page. In all our experiments, we use serialized 32-bit Roaring bitmaps [110] when beneficial, and uncompressed storage otherwise (e.g., for very short leaves). Roaring bitmaps partition the 32-bit integer domain into 2^{16} blocks based on the upper 16 bits of each integer. For each block, a "container" stores the lower 16 bits of the values. Containers can be stored either as arrays (for sparse data), bitmaps (for dense data), or runs (for sequences). This design enables efficient memory usage and fast operations. Many implementations provide SIMD-accelerated operations such as union, intersection, and difference.

We also tested other compressions and found that the need for explicit decompression can defeat any gains from higher compression ratios [116], resulting in worse runtime performance. If storage cost is of larger concern, a combination of delta- and Varint-encoding with ZStd general purpose compression [117] offered the best compression rates for basically all tested configurations. For example, for a composition with $b = 10$ and $d = 4$ on the 16-dim. LHCb dataset, the index with 32-bit ids was ≈ 19.6 GB in size without compression, ≈ 10.75 GB with Roaring and ≈ 2.98 GB with aforementioned Varint-ZStd combination.

Bitmap representations are generally preferred for very high-cardinality leaves (with respect to ID domain size), whereas simple lists are more efficient for low-cardinality leaves—a property Roaring explicitly exploits. We refer to [116] for a comparative study of various compression methods for the two representations.

VA-file-style Layout When storing the bin assignments as a flat file, each tuple value is explicitly represented by the corresponding (1-dimensional) bin's ID, requiring $\log_2 b$ bits per Team dimension. The storage order coincides with the data order, allowing it to be scanned as a proxy to the actual data to produce the index result, i.e., a series of IDs. This style of index is called a *VA-file* [115] and has the beneficial property of a constant scan cost—regardless of query selectivity. As a result, such an approach may be preferable over "inverted" access in cases where predicates offer only very low selectivity, even when combined. We found that the partial access of an inverted representation can significantly outperform sequential access pattern when using modern SSDs (see the experiment in Sec. 5.5). Moreover, compared to (sorted) posting lists, VA-files are less susceptible to lightweight compression, especially so if stored in a row-wise or PAX-style format for efficient predicate combination within a Team. Compression is an important aspect, because even simple approaches can yield significant gains in both storage costs and evaluation performance.

4.2.2 Meta Data & Query Mapping

During index creation, we keep track of bin cardinalities, storage offsets of the associated leaves, as well as compressed sizes for each Team. Further, we store the compression codec in use, i.e., Roaring and *uncompressed* for the shown experiments, for each leaf. Each d-dimensional matrix requires b^d unsigned integers. For the considered configurations of b and d, metadata overhead was negligible and fixed with respect to table cardinality. Using this metadata and the grid definitions, queries can be mapped to relevant bins at negligible cost for the considered configurations. Fig. 4.1 illustrates this process.

4.3 Index Precision & the Blessing of High Dimensionality

As mentioned before, evaluating indices can lead to additional false-positive results, which inflates the overall index selectivity. For individual or very few dimensions, this error can be large—especially for low b, where a predicate has to have a selectivity of at least $1 - 1/b$ in order to have any potential effect at all.

4.3.1 Sparsity of High-Dimensional Spaces

However, index *precision* is not just a matter of the grid's resolution along individual attributes: for (rectangular) queries with *fixed* selectivity, combining several predicates offers substantial pruning potential, even if individual selectivity (or b) is low. The reason for this is the *sparsity of high dimensional spaces.* Intuitively, irrelevant tuples differ sufficiently in some dimension (of the grid) *eventually.* With growing dimensionality, the chance for this variation increases exponentially, even if the number of possible values/bins is limited.

In the context of our Cartesian-grid index, this has several implications. First, the volume of a query rectangle (in unit-cube space $[0, 1]^D$) continuously shrinks with every additional SSP (Item 3), allowing us to discard most of the data space safely, since bins are non-overlapping. By using quantiles as bin borders, we ensure that, for about $1/b$ of all values, there are at least $b - 1/b$ others that can be clearly distinguished. This ability to distinguish values is what defines the *overall precision* of the index, i.e., how many ids it returns, compared to how many are actually relevant. In practice, we found that $b \approx 10$ is sufficient to offer a sufficiently high chance of a predicate contributing to pruning. For example, predicates with selectivity $s \leq 0.8$ will always be able to prune bins. Another benefit of high query dimensionality is that unused or a few poorly selective predicates are more easily compensated for—the query is still highly selective. In this sense, high (query) dimensionality *blesses* us with high levels of index precision, as well as some robustness for low-selectivity attributes.

To illustrate the benefits of high-dimensional sparsity, we have partitioned all tuples of the LHCb dataset into a 10^{16} dimensional grid. The resulting number of leaves, i.e., non-empty bins, is about $0.33 \cdot N$, primarily due to repeated values. For a more uniform dataset, the number of leaves would be closer to $1 \cdot N$, i.e., basically every tuple would be in its own bin. But even with 3 ids per bin (on average), identifying relevant portions of the grid quickly becomes the most expensive task.

4.3.2 Sparsity Without Exponential Cost

The value b should be selected with Team-dimensionality in mind: A grid-based Team index eventually suffers from the *curse of dimensionality* due to the exponential increase in complexity, just like other MDIS. Thus, *moderately-sized* Teams exploit some of the combined selectivity for more efficient access, while *intersection* allows us to tap into the remaining selectivity for an overall high index precision. For example, intersecting 4 Teams with 4 dimensions (instead of one 16-dimensional Team) still allows differentiation between all b^{16} disjoint regions of the data space, even if each Team only separately considers a 4-dimensional projection of the table. At the same time, combining 4 predicates translates to higher Team index selectivity (than 1-dimensional selectivity), which—in turn—enables us to consider fewer ids during the intersection step.

4.3.3 Diminishing Returns

Given the sparsity of high-dimensional space, choosing a large b only gives diminishing returns for precision. Further, utilizing increasingly more predicates in a Team decreases the benefit of higher Team *selectivity* for a cheaper intersection. However, this, in turn, is offset by needing overall *fewer* Teams. This suggests a sweet spot in Team dimensionality (or, equivalently, Team count [105]). Conversely, if query dimensionality is low, the precision of individual predicates matters more and, eventually, a single Team with a precise MDIS becomes the superior option. Overall, Team-based indexing allows to pay less attention to individual index quality and instead shift the focus towards efficient index intersection.

One may argue that higher compression rates for 1-dimensional Teams reduce the number of *bytes* involved and therefore compensates for low predicate selectivity. We found that the need to represent every tuple ID *at least once per Team* (implicitly or explicitly) generally outweighs the effect of compression (see Sec. 6.2.5). In this sense, multi-dimensional Teams offer an overall smaller storage footprint and also translate *combined* predicate selectivity into reduced index intersection cost more efficiently.

5 Efficient Index Intersection

Given the multi-dimensional character of a Team-based approach, the number of leaves involved in an intersection can be significantly larger than for bitmap indices. This necessitates a careful approach to index intersection—a central challenge for Team-based indexing. Logical runtime optimizations in particular can greatly contribute to more efficient processing. The described measures are agnostic of the actual MDIS in use and we make no assumptions on whether a Team index has overlapping leaves or strictly partitions the data space.

Outline We describe *intersection push-down* and *leaf grouping* as two key tuning knobs for logical optimization. Our prototype implementation, workload, and experimental setup, reused in Chapter 6 for evaluating Team composition, are presented in detail. As a foundation for subsequent discussions, we systematically categorize relevant performance factors and evaluate strategies for configuring the proposed logical optimizations.

5.1 Logical Optimization

Index intersection can be implemented by first collecting (unifying) all component parts of each Team index result, followed by an intersection of the results—either through pairwise reduction or by applying a single operator to multiple sets simultaneously [112, 113].

This "union-first" approach—and index intersection in general—can be formalized as a set-algebraic expression to represent an *execution plan*. For example, for three Team indices, A, B, and C, where each denotes *a set of leaves*, we can write:

$$R = \Big(\bigcup_i A_i\Big) \;\cap\; \Big(\bigcup_j B_j\Big) \;\cap\; \Big(\bigcup_k C_k\Big), \tag{5.1}$$

where i, j and k enumerate the query-relevant leaves of every Team, respectively.

However, a union-first approach has significant drawbacks: To facilitate efficient intersection as the second step, forming the union requires first fetching all IDs and then either to sort them or insert them into a suitable data structure, such as a Roaring bitmap [110]. Such a Team union represents a forced synchronization point—before any intersection can start—and produces potentially large intermediate results, even if the final index result ends up being empty. For our grid index, a union of leaves always produces the largest possible size, whereas MDIS with overlapping leaves requires (implicitly or explicitly) removing irrelevant/duplicate IDs. In addition, such a plan offers low inherent parallelism, if the operator implementations do not inherently utilize the available processing threads.

There are various ways to improve upon a union-first approach, avoid costly unions, increase potential parallelism and prune irrelevant IDs early. One way is to re-write the equation and *push intersections* into a common union operator. In its most extreme form, we obtain an "intersection-first" approach:

$$R = \bigcup_{i,j,k} \Big(A_i \ \cap \ B_j \ \cap \ C_k \Big) \tag{5.2}$$

where only actual result IDs survive an intersection chain and require unification. Although Eq. (5.2) minimizes the number of IDs we have to unify and increases the inherently available parallelism, it also forces us to consider every combination of lists separately. In this example, the big union indicates that there are $|A| \cdot |B| \cdot |C|$-many intersection terms, which quickly results in an infeasible number of intermediate results. For more than two Teams, it also results in redundant work, e.g., $A_i \cap B_j$ is (independently) evaluated k times. Thus, we opt for a middle ground, where we distribute the terms of C over combinations of (i, j) pairs and therefore "expand" only the terms of A and B:

$$R = \bigcup_{i,j} \underbrace{\Big[\bigcup_{k} \Big(\Big(A_i \ \cap \ B_j \Big) \ \cap \ C_k \Big) \Big]}_{\text{Independent Sub-Expression}}, \tag{5.3}$$

In this example, there are $|A| \cdot |B|$ many "Independent Sub-Expressions" (ISEs), where each evaluates $1 + |C|$ operations and requires two intermediate results: one for $A_i \cap B_j$, and another for the union over k. Note that longer ISEs may terminate early, if the result set becomes empty.

5.1.1 Leaf Grouping & Good ISE Count

To control the number of terms per Team, we employ partial unions over subsets of leaves ("leaf groups"), which allows us to reduce, e.g., $|A_i|$ many leaves to a freely configurable number of terms. Since this directly controls the number of ISEs, it provides another important lever to balance parallelism, memory consumption

and overhead of the execution. Detailed information on the number of terms and the involved leaf cardinalities are readily available from the metadata of our simple grid-based index approach, before any leaf data has to be accessed (see Sec. 4.2.2).

In practice, we found that expanding the smallest Team (by *accessed data volume*) first consistently yielded good results. In cases where the participating lists in the smallest Team are too few, we also expand a second Team to increase the ISE count for utilizing the available CPU threads more efficiently. We usually paired the expansion of a second Team with leaf grouping in order to ensure that the overall number of ISEs was never much larger than 2-3 times the number of CPU threads.

Note that we can apply grouping also on subsequent intersections and unions that span many terms in order to avoid synchronization points when using single-threaded implementations (such as Roaring).

5.1.2 Taking the Complement

In cases where individual Team results require accessing more than 50% of all IDs of a Team, it may be worthwhile to operate on the *complement* of the relevant leaves instead, i.e., on the IDs of bins that lie strictly outside of the query region. In place of set intersection, we perform *set subtraction* and remove irrelevant IDs from an intermediate result. This optimization gives us the theoretical benefit of never requiring access to more than about 50% of any Team and can be straightforwardly integrated into Eq. (5.3). This use-case rarely applies when utilizing two or more dimensions per Team. Our implementation does support the optimization and it finds notable application in the edge case of one-dimensional Teams, where every predicate with selectivity $s > 0.5$ will always be considered as a complement.

To show how this optimization impacts the index expression, we extend our earlier example, such that B and D are now being considered as complements. We expand A and B:

$$
\begin{aligned}
&\Big(\bigcup_i A_i\Big) \cap \Big(\bigcup_k C_k\Big) \setminus \Big(\bigcup_j B_j\Big) \setminus \Big(\bigcup_l D_l\Big) \\
\Rightarrow &\Big(\bigcap_j \bigcup_i A_i \setminus B_j\Big) \setminus \Big(\bigcup_{k \in C} C_k\Big) \setminus \Big(\bigcup_l D_l\Big) && \text{(expand } A, B\text{)} \\
\Rightarrow &\bigcap_j \Big\{ \bigcup_i \Big[\underbrace{\Big(\bigcup_k (A_i \setminus B_j) \cap C_k\Big) \setminus \Big(\bigcup_l D_l\Big)}_{\text{ISE}} \Big] \Big\} && \text{(push } C, D\text{)}
\end{aligned}
$$

Observe that the subtraction of B becomes an *outer intersection* over what previously only was an outer union. This can be disadvantageous, because the results obtained from ISEs are no longer independent parts of the final result and require further processing, before they can be considered part of the final result. It may therefore be preferable to expand only non-complement Teams, if possible.

When combining this optimization with leaf grouping and additional grouping (for additional parallelism) at later stages, the final equation becomes:

$$R = \bigcap_{s\in S}\left\{\bigcap_{j\in J_s}\left\{\bigcup_{r\in R}\left\{\bigcup_{i\in I_r}\left[\underbrace{\left(\bigcup_{t\in T}\left(\bigcup_{k\in K_t}((A_i \setminus B_j)\cap C_k)\right)\right)\setminus\bigcup_{l} D_l}_{\text{ISE}}\right]\right\}\right\}\right\}.$$

where I_r, J_s, and K_t represent the ID groups of A,B and C, respectively. Grouping D as well usually did not offer any benefit in our experience.

5.1.3 Intersection Order

The order of intersecting more than two lists, i.e., 3 or more Teams, can impact performance. In our implementation, the order of (pairwise) ISE operations is determined globally, based on the total combined result sizes of the Teams. However, the ideal order within a specific ISE may differ. As a first step, reordering set operations individually based on the respective list cardinalities can already give performance improvements. However, while the low cardinality of one list is a useful and cheap indicator, the ideal processing order also depends on data dependencies and favorable tuple ordering. Thus, an accurate assessment requires additional information. For example, fixed-size, prebuilt MinHashes [97] can yield an estimation of the Jaccard index $J = |A_i \cap B_j|/|A_i \cup B_j|$ [118] and subsequently the expected volume of an intersection. This approach incurs a constant-factor increase in both storage footprint and per-leaf runtime cost. Note that this optimization should be considered in conjunction with the selection of *physical* k-way operator implementations.

5.2 Physical Implementation

Implementing the intersection of storage-resident leaf data requires employing various techniques in order to make use of modern hardware, most notably fast SSD storage (or networks), and multi-core CPUs. To this end, our C++ prototype implementation is based on liburing [119] (with kernel polling/`IOPOLL`) for asynchronous I/O, Roaring bitmaps [110] for efficient set operations and the Taskflow library [120] for fine-grained parallelism and work-stealing-based load-balancing. To perform the intersection, we dynamically build a directed task graph of set operations over leaves initially retrieved from storage, with each operation being executed as soon as all resources, e.g., a pair (A_i, B_j), become available for processing. In our current implementation, ISEs are executed sequentially in fixed, global order and the operation is considered finished when all result tuples are collected into one final Roaring bitmap (see “big union” over i, j in Eq. (5.3)). Roaring offers

cheap short-circuit operations for set intersection, e.g., if all values are positioned in entirely different regions of the tuple ID space or one of the sets is empty. In case leaf grouping was applied, we assigned leaves greedily to the currently smallest group to achieve roughly balanced union sizes.

Notably, in some cases, our prototype was sensitive to fine-tuning of logical optimization parameters due to inefficient work balancing in situations involving many small leaves. Manual fine-tuning, for example of the group count, was able to achieve better efficiency and more consistent performance.

I/O I/O requests are issued for the smallest Team first, then for the second smallest Team, etc. We also implemented I/O-request merging for accesses to adjacent regions in the file, with mixed results. While this enables amortizing some of the I/O overhead, it also increased overall latency of the requests, which can decrease performance. As a result, we have not applied request merging in the experiments shown in this work. Further adapting the spawning of I/O requests to account for leaf group assignments and the order of necessity within the plan may offer additional performance benefits.

Streamed Execution Depending on the application, we may start using results from finished ISEs and skip forming any “big union”. This enables the seamless integration of subsequent operations, e.g., candidate refinement or a simple COUNT aggregation, into the workflow and avoids the potentially unnecessary synchronization step. Further, our approach lends itself to horizontal workload partitioning: Using shared, fast network-attached SSDs for index storage, multiple machines may each consider separate subsets of ISEs by partitioning leaves of the largest Team result and sharing other Teams’ leaves, for instance.

Physical Optimization Further note that we do not consider *physical* plan optimization in this work, i.e., the selection of the most suitable implementation of, e.g., a pairwise (or multi-way) intersection [111, 112], aside from what the implementation of Roaring already offers. As applications in search engines show [112, 113], considerable gains are possible by adapting the intersection operator.

5.3 Experimental Setup

For the remainder of this chapter and in Chapter 6, we will use the same workload and server setup.

Workload We make extensive use of a large real-world dataset representing particle decays from the LHCb experiment at CERN. The table has $D = 16$ double precision float attributes describing properties of multiple particles participating in the decay. It has $N = 1.222 \cdot 10^9$ tuples, which amounts to $\approx 153\,\text{GB}$, or $\approx 19\,\text{GB}$

in Snappy-compressed [121] Parquet format [78]. The large compression factor stems from repeated values and beneficial tuple order. Our running example will be a highly selective real-world query with 14 predicates, ranging in selectivity between ≈ 0.1 and ≈ 0.92. For $b \leq 10$, the query has 12 SSPs due to the lack of per-attribute precision. The index results for $b \in \{5, 10, 16\}$ yield 38109, 3728 and 1312 tuple ids, which corresponds to a selectivity of $3.12 \cdot 10^{-5}$, $3.05 \cdot 10^{-6}$, and $1.07 \cdot 10^{-6}$ respectively. Note that this illustrates that the grid index can indeed achieve significant selectivity of $3.12 \cdot 10^{-5}$—$1.07 \cdot 10^{-6}$ for this query.

We also use a "data sweep" from the Sloan Digital Sky Survey (SDSS) representing stellar objects identified as stars. We use $D_{\text{table}} = 85$ single precision attributes and $N = 0.434 \cdot 10^9$ of the **SDSS dataset**, which amounts to ≈ 147 GB. The dataset was overall incompressible (Parquet with Snappy). The attributes are made up of 17 underlying physics properties, such as brightness, flux and noise statistics, which are then separately tracked for 5 separate optical *bands*, each representing a different wave-length and one attribute in the table. We will use this "band"-grouping later to investigate the impact of domain-specific Team compositions (see Paragraph "The Impact of Team-Composition" in Sec. 5.5.3).

Workload Randomization We also created a uniform version of this dataset without data dependencies but equivalent marginal cardinalities, i.e., *individual* predicates evaluated on similarly-built indices on both datasets have exactly the same selectivity. The dataset was almost incompressible (≈ 151 GB parquet file). We also make use of Team compositions drawn uniformly at random in the following way: first, a list of attributes is shuffled and then split according to an integer partition of its length, e.g., 16 elements were split into three lists of size 6, 5, and 5, respectively. Further, we created random queries by translating a query (rectangle) uniformly at random in grid-space. Note that, despite equal predicate selectivities, overall query selectivity still varies slightly due to inter-attribute dependencies.

Hardware and Software Platform All our experiments were run on a AMD EPYC 9124 processor with 16 cores/32 threads, 384 GB DDR5 RAM and a Linux 6.8.0-55-generic kernel. We always use all available threads. For storage, we used a Kioxia CM7-R with up to 14 GB/s PCIe 5.0 sequential read bandwidth.

5.4 Performance Factors

Having introduced methods to influence index intersection efficiency, the focus shifts to understanding how workload-specific properties govern runtime performance. For example, the performance of index intersection is not necessarily only determined by the overall number of operations [113]. In fact, intersection push-down increases the number of operations and union-first has the minimal number of operations. We differentiate three categories of runtime factors: *Volume*, *Overhead* and *Imbalance*.

5.4.1 Volume & Overhead

We define *Volume* as the sum of the compressed sizes of all *relevant* leaves across all Teams. Its influence on performance is straightforward: less work has to be done if fewer IDs are involved or if these IDs are highly compressible. Volume is reduced with higher predicate selectivity, but also with higher Team dimensionality by allowing to combine more predicates and therefore to reduce the involved ID volume drastically. However, the latter comes with a cost of additional *Overhead* due to a larger number of *relevant leaves* (or bins), as well as reduced compressibility. There are various forms of Overhead, e.g., the need for more tasks to run set operations in our execution plan, more and smaller I/O requests and increased read amplification due to padding in the storage layout. Query mapping costs, i.e., "tree traversal", may also be considered a form of Overhead that increases with dimensionality. Eventually, at high Team dimensionality, Overhead is no longer amortized and essentially *paid per id*. Overhead costs are therefore the primary reason why we consider *moderately*-dim. Teams. With our Cartesian grid index, we consider Overhead to be a cost linear in the total number of *relevant leaves*. Occasionally, we will use the *bin* count and the union *cardinality* of query-relevant leaves for back-of-the-envelope calculations of Overhead and Volume, respectively.

As we will confirm later, Volume *decreases* exponentially with Team dimensionality, whereas Overhead *increases* exponentially. Thus, 1-dim. Teams have the lowest Overhead, but usually also the highest Volume. In contrast, D-dim. Teams consider the lowest Volume but also the highest Overhead. This implies an optimal configuration, where Volume reduction and Overhead increase balance out, but *Imbalance* factors can shift performance towards better or worse.

5.4.2 Evaluation Strategies

To first illustrate different execution strategies solely with respect to Volume and Overhead, we measured intersection runtime on a uniform dataset with $D = 12$ and one composition with 3 Teams, i.e., $d = 4$. To control Overhead, we varied $b \in \{4, 8, 16\}$, such that each Team has b^d bins. Queries were created randomly with equally selective predicates over all D dimensions with $s \in \{0.25, 0.5\}$, such that we access $(s \cdot b)^d \cdot 3$ many bins in every query. For uniform data, the overall selectivity of the query is given by s^D. We created 20 random queries for each pair (s, b) and executed them with different execution strategies, namely *union-first* (with and without grouping), as well as various forms of partial expansions. We expanded only one Team, exactly two Teams or between one to two Teams in an adaptive manner. Except for *expand-first*, we also varied the number of leaf groups. When applying grouping to a Team A, we reduced the number of leaves to $\sqrt{(|A|)}$. The runtime (log-scale) is shown in Fig. 5.1 (next page), with a small horizontal jitter to each group of measurements for visual clarity.

Performance generally degrades with larger Volume and increasing Overhead, i.e., larger s and number of leaves b. The strategy with the best and most con-

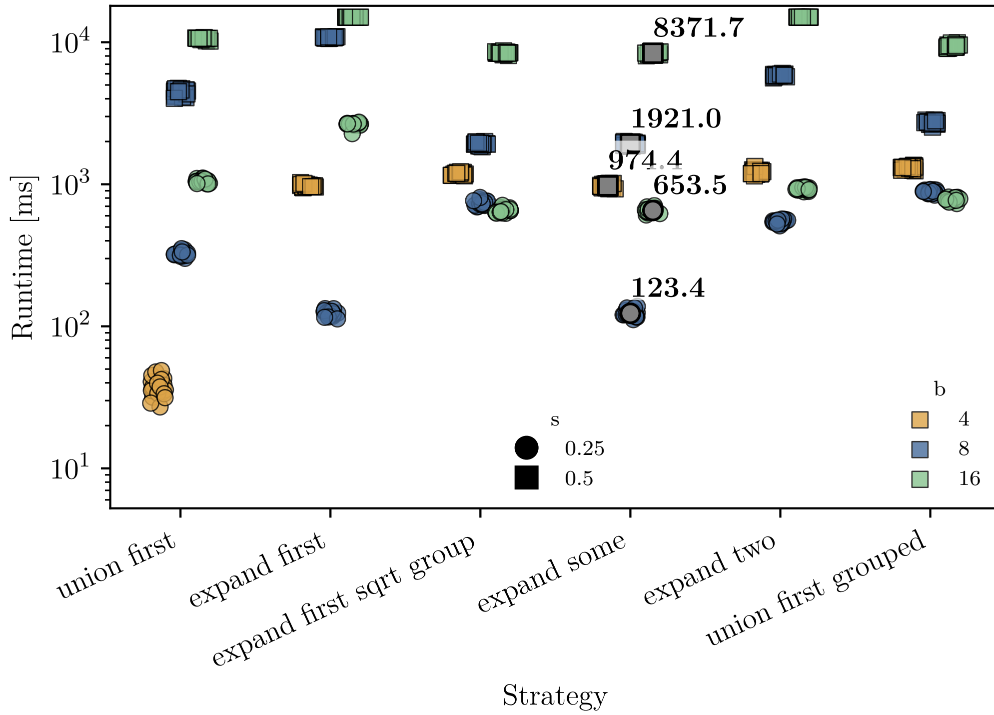


Figure 5.1: Runtime for different execution strategies over two sets of random queries and a uniform dataset. The adaptive *expand-some* strategy dominates.

sistent performance was an adaptive *expand-some*, which always expanded the first/smallest-Volume Team and potentially the second, if the first had too few lists (< 16). Grouping for expand-some was only applied (on either Team) when the number of ISEs grew beyond 128. We found that the union-first strategy can be improved with grouping for additional parallelism, but is still dominated by other strategies. Further, grouping is necessary for queries/indices with many leaves (cmp. expand-first), but can also deteriorate performance, especially for configurations with few lists ("expand first sqrt group" for $s = 0.25$ and $b = 8$). The execution strategy for $s = 0.25, b = 4$ is always trivial, since there is only a single list per Team and therefore no union required. We show the corresponding measurement in the union-first group. Overall, we observe that partial expansion is universally helpful, but may require additional grouping for queries that access many leaves. We will use the adaptive expansion strategy in the later experiments.

I/O Overhead To assess the impact of storage access Overhead, we ran most of our measurements on already DRAM-resident index data as well. For example, in Fig. 5.1, DRAM-based access improved intersection runtime by $1.16 - 1.78\times$. The impact grew with larger b and smaller s, where leaves were smaller and more numerous, indicating that the costs are mostly associated with Overhead.

5.4.3 Imbalance

There are various Imbalance factors that impact Overhead, Volume and the efficiency of individual operations. The absence of data dependencies and differences in predicate selectivities across Teams presents a worst-case scenario for indexing for various reasons. First, given the lack of any favorable order in the data, compression efficiency of individual leaves deteriorates significantly. This results in higher Volume and storage costs, but set operations are also less efficient: When intersecting two lists from a uniform distribution, no clustering or other forms of exploitable structure imply that intersection essentially degrades to an exhaustive, pairwise comparison of IDs. Further, leaves are about equal in cardinality, which also degrades operation efficiency. If integer set cardinalities differ significantly, we can make use of specialized implementations for intersection, such as galloping [113]. Union operations similarly benefit from inter-attribute and inter-leaf dependencies within a grid. Further, differences in Team sizes and the selectivity of individual predicates can further impact the performance of index intersection as a whole.

Imbalanced Data Dependencies To illustrate the impact of data-induced Imbalances, we ran the same benchmark as in Fig. 5.1 on the real LHCb dataset as well. The results are shown in Fig. 5.2 (next page). Using real-world data yielded consistent performance improvements. The runtimes for the best configurations of $b \in \{8, 16\}$ and $s = 0.25$ were about $2.5\times$ faster ($1.85\times$ faster for $s = 0.5$). Further, the overall performance differences between the strategies markedly shrank, with even the worst performing execution strategies observing significantly lower runtime. Also notable is the still low but slightly larger (vertical) runtime variance across random queries, compared to the uniform dataset. Data dependencies cause bin cardinalities to fluctuate more and our adaptive strategy still dominates. Beneficial order in the data gives overall better compressibility, resulting in more efficient intersections. Note that, from a probabilistic perspective, the *conditional* bin cardinality distribution can shift greatly, depending on *where* predicates restrict the distribution's support—even if individual selectivities remain the same.

To further investigate the influence of data distributions, we varied our highly-selective real-world query (see Sec. 5.3) solely based on its *location* in grid space, both on the LHCb dataset, and on the uniform variant. The results are shown in Fig. 5.3 (next page). Selectivity varied more for the real dataset, but the total involved ID volume ($\approx 541 \cdot 10^6$) was about equal across both datasets and all queries. But with compression, the actual Volume was $\approx 20\%$ smaller and the Volume *difference* between the most selective and least selective Team was larger, an indication of why non-uniform queries performed better. As indicated by the low (vertical) runtime variance, performance of the query on the real dataset was less impacted by differences in overall selectivity, compared to uniform data. Note that the impact of query locality is likely more pronounced for bigger query rectangles due to the larger number of involved leaves.

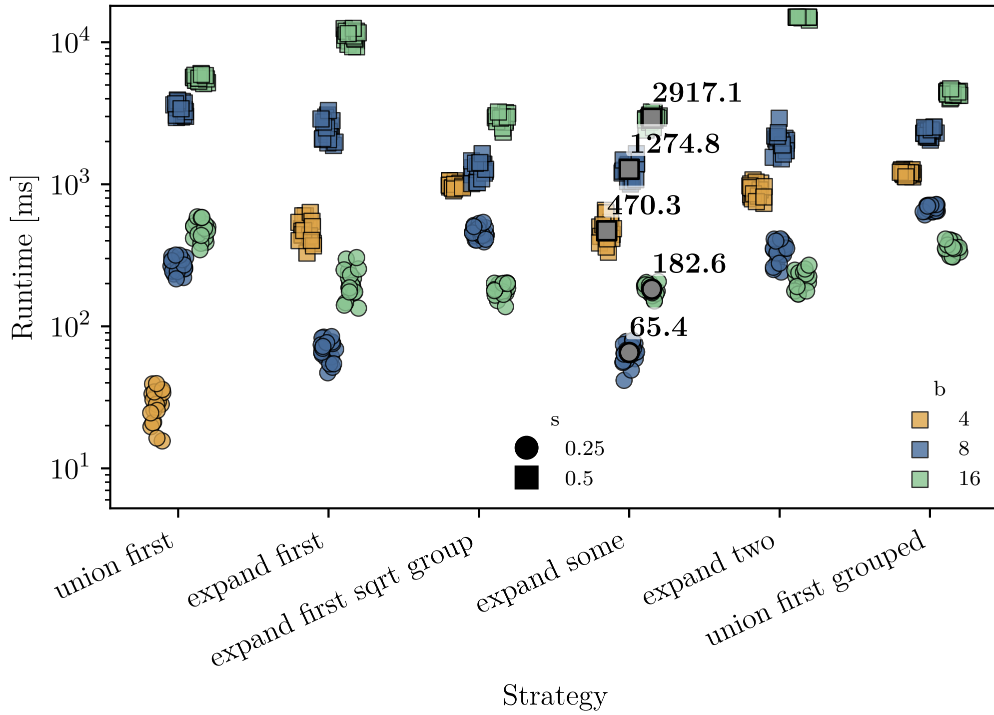


Figure 5.2: Runtime for different execution strategies over two sets of random queries and the LHCb dataset. Compared to Fig. 5.1, runtime on real data is better throughout.

Imbalanced Team Sizes Team compositions over the same set of attributes but differing Team sizes can have different performance. Understanding how this difference may generally affect performance in a more isolated setting will be useful for our discussion on Team composition in Chapter 6. To quantify it, we ran 20 random queries with $b = 10$ and uniformly selective predicates with $s = 0.4$ on two Team compositions, one with 2×4-dimensional Teams ("balanced") and one with Teams of size 6 and 2 ("imbalanced"). We used both the real and the uniform version of the LHCb dataset.

Overall, Volume and Overhead differed significantly. The balanced Team required $2 \cdot 0.4^4 \cdot N \approx 0.0512 \cdot N$ IDs, whereas the imbalanced Team $(0.4^6 + 0.4^2) \cdot N \approx (0.004096 + 0.16) \cdot N = 0.164096 \cdot N$ IDs—$3.1\times$ more. Further, the number of relevant bins was $2 \cdot 4^4 = 512$ and $4^6 + 4^2 = 4112$, respectively. As a result, for uniform data, the imbalanced composition required approximately $1.5\times$ as much time for the intersection (0.919 vs. 0.583 seconds). When using real data instead, the actual Volume difference in byte was lower at about $2.5\times$ as much due to better compression and the runtime advantage of the balanced composition shrank to $1.23\times$ (0.475 vs. 0.385 seconds). We attribute this to the higher efficiency of intersecting two lists of dissimilar size on real data, which required $4.4\times$ more

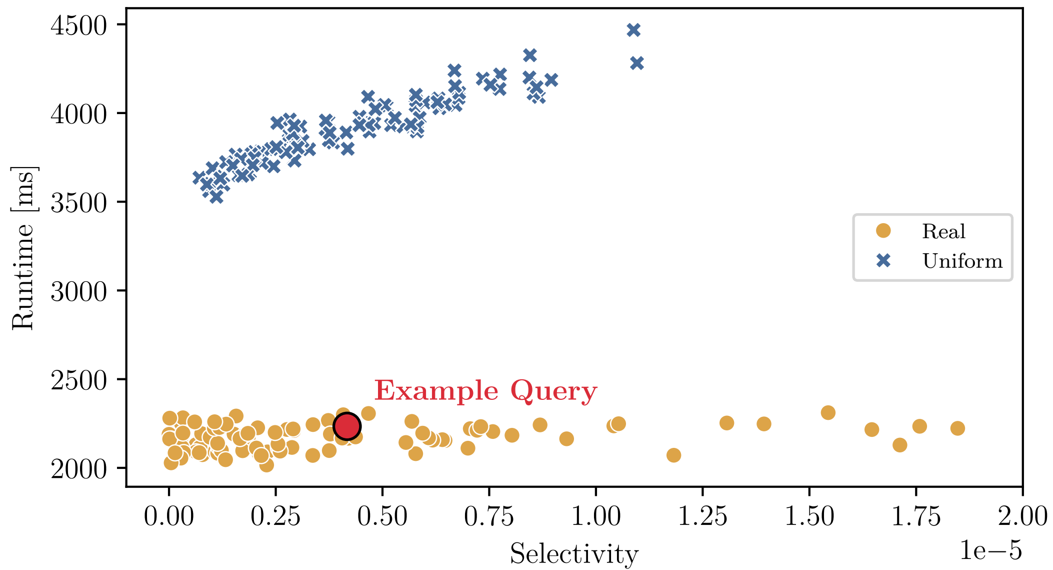


Figure 5.3: Runtime vs. selectivity for 100 randomly shifted queries and data with and without data dependencies. Using real data improves performance.

CPU time for the imbalanced case. For uniform data, this difference was $5.6\times$ instead. In addition, reducing 4096 leaves to only 16 via grouping was also cheaper, requiring only $1.58\times$ the cost compared to $2.45\times$ in the uniform case. Overall, this shows that Imbalance between Teams may lead to additional Overhead and access Volume, but actual differences in runtime may be smaller.

Imbalanced Predicates Attributes and their associated predicates usually differ in their selectivity, with some offering none at all. In the latter case, access to a Team that also includes unused attributes is less efficient. By comparing access to a smaller Team without these unused attributes—while keeping overall Team selectivity constant—we can quantify the resulting inefficiency. We considered two compositions: The first with a 2-dim. Team and the second with a 6-dim. Team, both accessed with similarly (low) Team selectivity 0.04. We paired each with another 4-dim. Team of fixed selectivity $s \in \{0.3, 0.7\}$ and generated 20 random queries per s. This represents a situation where indexing 4 additional attributes does not offer a selectivity benefit and instead increases *Overhead* by partitioning the data into more bins than necessary.

For real data, the runtime advantage for using the 2-dimensional Team differed greatly, depending on how selective the second Team was. If the second Team offers more selectivity than the first, i.e., $0.3^4 \approx 0.0081$, intersecting with the 2-dim. Team was $12\times$ faster than intersecting with the 6-dim. Team (0.259 vs. 3.145 seconds). On the other hand, if the second Team selects $0.7^4 \approx 0.24$, an intersection with the 2-dim. Team was only about $1.5\times$ faster (5.28 vs. 3.36 seconds), because the runtime is dominated by processing the Volume of the larger Team, not the

difference in Overhead. Note that efficiency loss from unused attributes is not unique to the grid-based MDIS used here; in other (tree-based) MDIS, irrelevant attributes similarly reduce the number of prunable subtrees.

5.5 Scaling Behavior

By limiting the maximum Team dimensionality d, and keeping it constant, the cost of Team-based indexing scales linearly with both the table cardinality and the query dimensionality, i.e., every additional Team we intersect with adds a constant factor to overall runtime and storage requirement.

Note that the maximum value for d depends on the underlying index structure in use. We chose the grid-index in order to decouple the cost of meta-data storage, which is solely dependent on Team size d (and b), from the table size, making it negligible for large tables. Thus, the storage space requirement for indexing a table with N tuples and D attributes is capped at most $N \cdot D/d$ IDs—one ID per tuple per Team (without compression).

Generally, we can imagine the intersection process as a chain of ISEs (see Eq. (5.3)), which represent a gradual reduction of intermediate result sets. Across multiple intersections, an ISE can evaluate to an empty set, terminating this string of computations (early). However, if our intermediate results of our ISEs are small but non-empty, we still have to consider leaves of later Teams entirely. This roughly separates index intersection into two phases.

At the beginning of the computation, many ISEs remain active, and our intermediate results are relatively large (with respect to the initial leaf size). Here, an intersection between two sets are more balanced, since both sets can be comparable in their cardinality and overall compute costs are expensive (relative to the involved data volume). In this *early reduction* phase, multi-core parallelism and efficient scheduling can have a large impact on performance.

After several intersections, however, our intermediate results become smaller or even empty, reducing both the number of active ISEs (and also exploitable parallelism) and the cost of individual set operations. In the second phase, runtime is mostly dominated by the involved volume of later Teams, since we still need to fetch (and decompress) every ID of these later Teams. Note that implementations, such as Roaring [110], also offer optimizations, such as galloping [113], improving performance in case one set of the intersection has low cardinality.

5.5.1 Dimensional Scaling & Phases of Index Intersection

To quantify runtime for the first *early reduction* phase, which also represents index intersection for lower-dimensional queries, we created indices with $b = 10$ over increasingly more dimensions $D \in \{1, 2, 4, \ldots, 16\}$, using attributes from the LHCb dataset. Fig. 5.4 (log-log) shows the runtime of 20 random queries, each with constant selectivity $s = 0.4$ per predicate/dimension. We start out with a

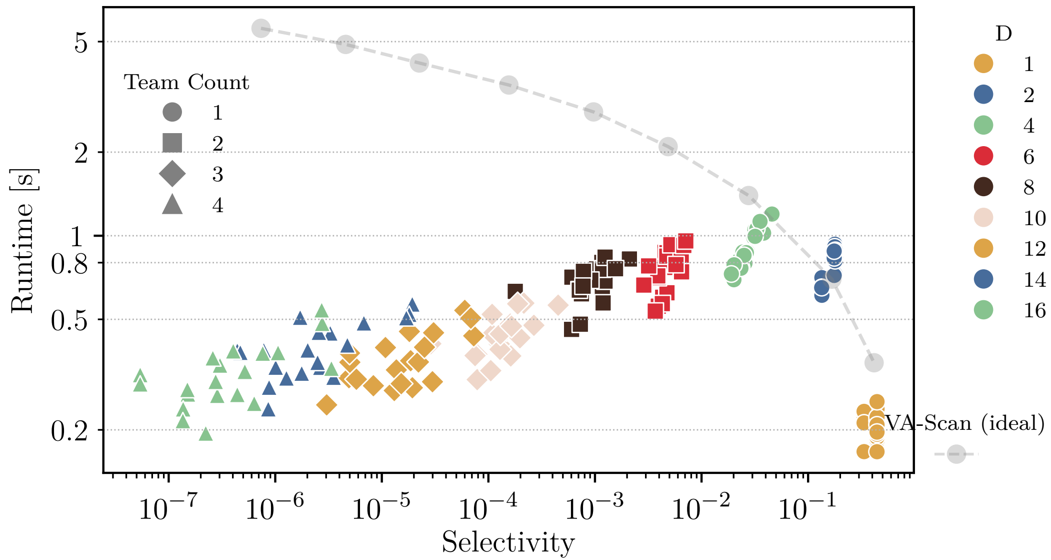


Figure 5.4: Intersection runtime for increasing dimensionality D and predicate count (log-log, $s = 0.4$). Performance improves with selectivity. VA-files can not exploit selectivity.

single attribute (far-right measurements) and gradually add new ones (in table order) for each measurement, decreasing overall query selectivity and increasing overall Overhead and Volume in the process. Note that the dimensionality (and Team count) grows from right to left. For $D > 2$, compositions were built balanced, using Team sizes of $\{2, 4\}$. We used the *expand-some* optimization strategy (see Sec. 5.4.2) in all our measurements.

Performance improves with higher query selectivity, both across dimensionality groups and between queries (same symbol & color). While this decrease in runtime may be surprising due to the increase in Overhead and Volume, we attribute this drop to the improved ability of the system to utilize available parallelism and lower cost of materializing the (now-smaller) index result. Adding more Teams offers us the ability to expand more Teams (see Sec. 5.1), and more intersections enable a cheaper (final) union of intermediate results. I.e., a union over many small leaves is costly, and pruning via intersection reduces intermediate result size – and thus runtime – considerably. Moreover, gradual addition of Teams offers new opportunities to start the intersection order: Freshly added Teams with high selectivity “jump the line” and intersect early, reducing runtime in the process. This behavior requires a sufficiently low per-predicate selectivity, s, or the fetch costs diminish any benefits from early pruning (see below). Notably, in this phase, compute dominates even the cost of I/O, as also indicated in later experiments (cmp. Fig. 6.3 and Fig. 6.4).

An interesting outlier is the low runtime for $D = 1$ (lower right corner), which only requires retrieving and unifying four highly compressible leaves. Note that we generally recommend using only a single Team with a suitable single- or multi-

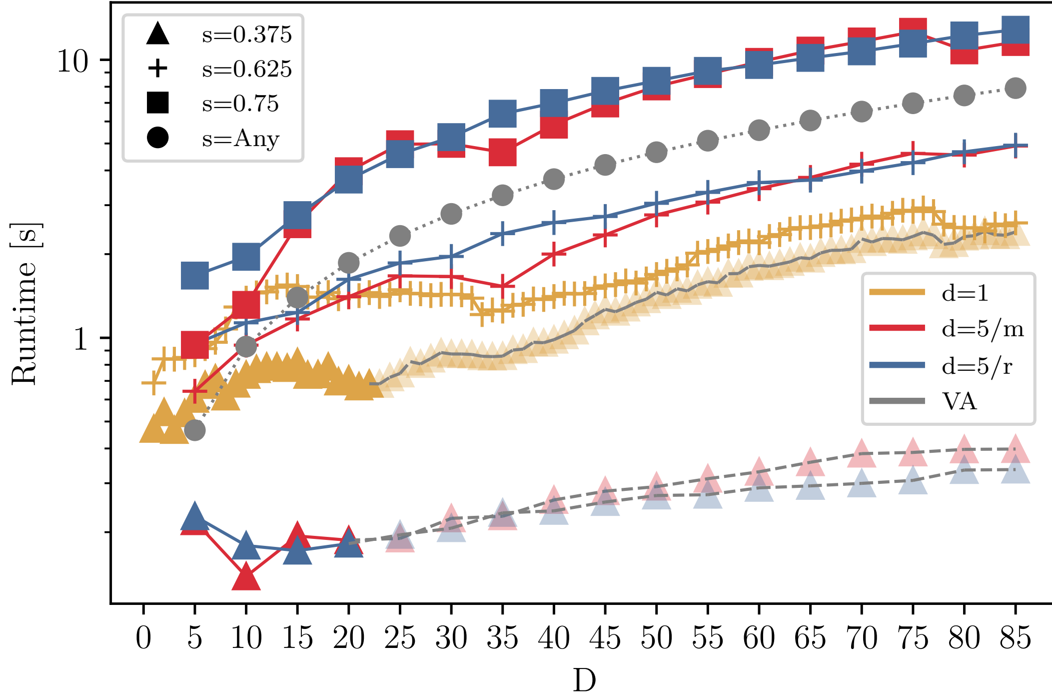


Figure 5.5: Intersection runtime (log y-axis) for increasing D. Runtime scales linearly and Team compositions become similar. VA-files and Teams with $d = 1$ are faster for large s.

dimensional index [19, 103] for lower-dimensional queries ($D \in [1, 2, 4]$ in Fig. 5.4), and we have not fine-tuned our implementation or optimizer for such cases.

For reference, we calculated the I/O time of scanning VA-files (built over the same attributes and without the cost of intersecting multiple VA-files), which represents their ideal, SSD-limited runtime. Due to the inverted access, Team-based indexing outperforms VA-files for low selectivity/high dimensionality (even if accounting only for I/O), since the Volume is significantly lower and reduces not only overall I/O cost but also the associated compute cost. However, we expect the overall runtime to generally increase linearly with (larger) D, as we will demonstrate in our next experiment.

5.5.2 High-Dimensional Scaling

To show how the runtime evolves for high-dimensional queries, we performed a similar experiment as in Fig. 5.4 for the SDSS dataset, which features up to 85 dimensions. We varied selectivity $s \in \{0.375, 0.625, 0.75\}$ and created indices for $b = 8$ in three different flavors: a manually-chosen composition (indicated as *m*; see below) and a random composition (indicated as *r*) with $d = 5$ each, and a composition with $d = 1$ represents bitmap indices. The manual composition formed Teams of 5 wavelength bands (see the description of the SDSS dataset in

Sec. 5.3). The three indices required 13, 17, and 27 GB of storage for *m*, *r*, and $d = 1$, respectively. We show runtime averages over 5 random queries (log y-axis; 5 repetitions) versus query dimensionality (linear scale) in Fig. 5.5.

Starting with a single Team, we observe that adding more Teams generally grows runtime by a constant factor (we also show VA-file runtime for reference). Similar to Fig. 5.4, we can also observe the initial dip in runtime for small $s = 0.375$.

As indicated by the gray line continuations for $s = 0.375$, all 5 random queries eventually evaluate to the *empty set*, and execution could be terminated early at this point. Subsequent runtime increases solely come from fetching and decompressing leaves of later Teams.

Baseline Comparison For reference, we have again calculated the ideal time of evaluating VA-files. Due to the inverted access, Team-based indexing with inverted index outperforms VA-files for selective queries (around $0.625 < s < 0.75$) due to their inability to skip any data. Similar results can be seen for $d = 1$, albeit for lower s. For $s = 0.375$, Teams with $d = 5$ members can reduce overall Volume by up to $80\times$ (for $D = 85$) and subsequently offer a speedup of $6\times$ to $7\times$. Here, 1-D Teams can not exploit any *combined* (per-Team) selectivity and the evaluation is bound by Volume alone, since Overhead is just 3 leaves per Team.

5.5.3 The Impact of Team-Composition

To show the impact of Team composition for large D, we compare a *hand-crafted*- and a randomly drawn Team composition (*r*) for the SDSS dataset. In the *manual* composition (*m*), wavelength measurements can have very strong correlation (up to Pearson coefficient of 1) compared to *r*, where strong correlations are unlikely. To illustrate, the *smallest* average pairwise correlation within any Team of *m* is 0.27, whereas *random*'s averages are *at most* 0.28. Higher combined selectivity made it more likely for correlated Teams to jump the intersection chain. Accordingly, runtime is lower early (where pruning potential matters most), but can still drop runtime later on. For $D = 35$, the new Team offered significant pruning at significantly lower Overhead cost: *r* processed 42851 leaves in 6.3s, whereas *m* requires 4.6s for only 30868. However, *r* does occasionally outperform *m* for $s \geq 0.625$ and most D at $s > 0.325$. Moreover, in the fetch-dominated phase—where queries are truly high-dimensional—runtime differences due to Team composition eventually become negligible.

5.5.4 Varying Table Size

To confirm that Overhead is amortized with larger tables, we measured storage costs and runtime per tuple for our running example, a fixed 4×4 Team composition and $b = 10$. Indices were build over increasingly more tuples from the LHCb dataset to retain the beneficial order and subsequently compression efficiency. The runtime

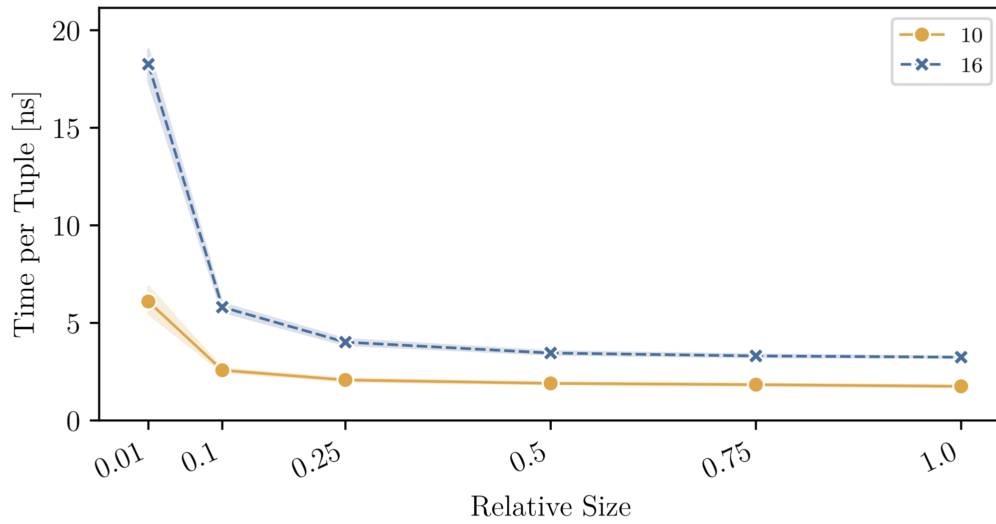


Figure 5.6: Runtime spend per database tuple (y-axis) for increasingly larger tables and running example query. Indexing more tuples amortizes Overhead costs.

per tuple is shown in Fig. 5.6: The constant Overhead costs are quickly amortized and only noticeable for small tables.

6
Team Composition

Storage costs and intersection performance are impacted by how attributes are grouped, i.e., how Teams are composed. The runtime efficiency of a specific composition depends on the application and one composition may favor one query but not another. Further, Team indices usually have to be created within a certain storage capacity budget and practical constraints on Team size and precision. In the following, we discuss various criteria for assessing compositions and reducing a potentially very large design space.

Size Restrictions Although larger Teams allow combining more predicates or attributes for greater selectivity and therefore to access less Volume, it also comes with an escalating Overhead—regardless of the MDIS in use. Therefore, Team size is effectively limited by table cardinality and index precision, resulting in a limit on d, and thus Overhead.

Generally, for a fixed Team composition and index precision b, we expect larger tables to be more efficiently indexed. With fixed grid-index parameters, Overhead costs, e.g., for formulating I/O requests or execution plan traversal, are essentially constant. An additional aspect is compression efficiency, which improves with leaf cardinality. However, these benefits only really matter for moderate Team sizes. Meaningfully increasing the average cardinality ($\frac{N}{b^d}$) of bins in high-dim. grids requires exponentially more data. Although strong data dependencies can help with concentrating the "mass" in only a subset of the high-dimensional grid's bins, Team sizes are still ultimately limited and even extreme table cardinalities require eventually yielding to the curse of dimensionality. Given the sparsity of high dimensional space, new tuples are more likely to be placed into an entirely new, previously unoccupied bin first—a property that *helped* us relax the requirement on index precision before. For our simple, grid-based index and the LHCb dataset, we found 4–6 dimensions (with $b \approx 10$) to be the limit.

The *order* of attributes within a Team may potentially allow to amortize some I/O Overhead by exploiting storage access to adjacent (logical) addresses with *merged* requests more frequently, but this effect was not noticeable during our investigations. In the following we ignore the order of attributes within a Team.

Balanced Team Sizes With a limit for the maximum Team size d, we still have to decide on how to distribute D many attributes over $\lceil D/d \rceil$ or more Teams. As indicated in Sec. 5.4.3 (*Imbalanced Team sizes*), choosing compositions with imbalanced Team sizes can be beneficial (due to reduced Overhead) for queries that exactly cover a (proper) subset of Teams. However, under the assumption that all indexed attributes feature SSPs, balanced Teams of maximum size d can minimize storage costs and overall Volume for a larger workload. Even with further domain-agnostic assumptions, a large design space still remains: When we consider only true set partitions of the D attributes, i.e., no attribute appears in more than 1 Team, the number of possible composition grows quickly with D and can counted using the Stirling numbers of the second kind [122]. For example, for 16 attributes and 4 Teams, we already have $S2(16, 4) = 171 \cdot 10^6$ possible compositions. We will discuss how to further reduce this design space after motivating additional criteria to assess compositions.

Outline For the remainder of this chapter, we first approach Team composition from a *domain-agnostic* perspective, where every attribute is considered equally important, providing a probabilistic model for assessing the efficiency in the presence of unused attributes. Thereafter, we consider *combined selectivity* as a first (incomplete) quality metric to systematically assess compositions. Finally, we provide a runtime (and storage consumption) evaluation for Team compositions and also correlate performance with the measure of combined selectivity.

6.1 Attribute Coverage & Ad-Hoc Queries

As demonstrated in Sec. 5.4.3, attributes without SSP (see Sec. 4.1.3) incur an indirect cost. Even when using one-dimensional Teams, e.g., as in bitmap indices, the flexibility to ignore all irrelevant attributes entirely is paid for with increased storage cost and fewer opportunities for faster access. Thus, our initial goal in Team composition is to maximize *coverage*: a query should access most or all attributes in all relevant Teams with SSPs (or skip Teams entirely). In real-world *ad-hoc* queries, users may define predicates over arbitrary attributes. The goal of exploiting combined selectivity for lower access Volume implies that advantages over (one-dimensional) Teams are only possible if more than one attribute is accessed at once in a Team. In the worst case, every query-relevant attribute is located in a separate Team. Single-dimensional bitmap indices then have an advantage due to better compression ratios and less Overhead.

To more generally assess the coverage criterion for ad-hoc queries, we formally compute the probability that any query (chosen uniformly at random) allows for *efficient access*. This analysis represents a *workload agnostic* approach to Team composition, where every possible query is equally likely and every predicate equally relevant and valuable.

6.1.1 Definitions

Let the set partition $T = T_1, \ldots, T_g$ of attribute set A denote a Team composition into g many disjoint subsets/Teams $T_i \subset A$ with $\bigcup_{i=1}^{g} T_i = A$ and $\forall i, j : T_i \cap T_j = \emptyset$. Note that a different order of enumeration of the Teams does not result in a different Team composition.

The subset $Q \subset A$ of the indexed attributes A, chosen uniformly at random, represents a query with predicates defined on the attributes of Q. For any Q and fixed composition T, we represent the access of Q on T as the sequence of intersections $T_1 \cap Q, T_2 \cap Q, \ldots, T_g \cap Q$, which will be called the *cover* of T for Q. Note that a cover is also a set partition, in this case of Q. The sequence of cardinalities of a covering will be denoted $C = (c_1, \ldots, c_g)$ with $c_i := |T_i \cap Q|$. Lastly, the access for an arbitrary combination of query Q and composition T is considered l-efficient, if *at least* l many attributes of all (query-relevant) Teams are accessed at a time. More formally, access is l-efficient, if

$$\forall c_i \in C, l \in \{1, \ldots, d\} : c_i \geq l \lor c_i = 0.$$

Example. Let $A = \{A, B, \ldots, L\}$ be an attribute set, $D := |A|$, and

$$T = \{\{A, B, C\}, \{D, E, F\}, \{G, H, I\}, \{J, K, L\}\}$$

be a Team composition. Evaluating a query $Q = \{A,\ D, E,\ G, H, I\}$ then requires access to attributes $\{A\}$ from the first Team, attributes $\{D, E\}$ of the second and $\{G, H, I\}$ of the third Team—Team $\{J, K, L\}$ can be ignored. The access is therefore 1-efficient, because predicates are defined on at least 1 attribute per query-relevant Team.

6.1.2 The Probability of l-efficient Access

To simplify the notation, we set the size of every Team to a constant value, i.e., $T_i = d$, and have $D = \sum_{i=0}^{g} T_i = d \cdot g$. The probability that a random query Q with q many different attributes allows an l-efficient access is given by the number of all l-efficient covers, divided by the number of all possible covers. Since queries are considered distinct of they differ by at least one attribute, queries necessarily also produce unique covers. Further, given that T is fixed, drawing any subset Q from A also determines the corresponding cover. The number of covers for all possible queries with $|Q| = q$ is then counted by the binomial coefficient $\binom{D}{q}$.

But only a subset of all possible covers represents an l-efficient access. To count only *relevant* covers, we observe that C represents a *weak integer compositions*[1] of query size q, such that $\sum_{i=1}^{g} c_i = q$ with $c_i \in \{0, 1, 2, \ldots, q\}$. This means that a sequence $(3, 0, 1)$ is distinct from $(0, 1, 3)$. Moreover, multiple covers may share the same sequence of sizes C: for any Team T_i, there are $\binom{d}{c_i}$ ways to select attributes differently. Therefore, there are $\prod_{i=1}^{g} \binom{d}{c_i}$ distinct covers for any composition C.

Our definition of l-efficient accesses can be enforced by restricting the set of all possible weak integer compositions to those that only contain terms from the set $\{0\} \cup \{l, l+1, \ldots, d-1, d\}$. Let $\mathcal{E}_l(q)$ denote this set of l-restricted weak compositions of integer q with g many terms. Notably, $\mathcal{E}_{l=1}(q)$ is the unrestricted set. Summing over all integer compositions $C \in \mathcal{E}_l(q)$ gives $\sum_{C \in \mathcal{E}_l(q)} \prod_{i=1}^{g} \binom{d}{c_i}$, i.e., the number of all covers (and queries) that grant l-efficient access. Hence, the probability $\Pr(g, d, q, l)$ that a uniformly chosen query with exactly q attributes and composition T allows an l-efficient access is:

$$\Pr(g, d, q, l) := \frac{\sum_{C \in \mathcal{E}_l(q)} \prod_{i=1}^{g} \binom{d}{c_i}}{\binom{D}{q}}. \tag{6.1}$$

To confirm our counting method of l-efficient covers, we may sum over *all possible* integer compositions C instead, which gives

$$\sum_{c_1 + \cdots + c_g = q} \prod_{i=1}^{g} \binom{d}{c_i} = \binom{D}{q}.$$

This identity is known as the *generalized* Vandermonde convolution [122]. Note that all formulas directly apply to Team compositions with arbitrary Team sizes $|T_i|$ as well, if we replace $\binom{d}{c_i}$ with $\binom{|T_i|}{c_i}$.

A slightly more expressive perspective can be obtained by accumulating all queries that use q *or more*, obtained by summing up both nominator and denominator:

$$\Pr_{\geq q}(g, d, q, l) := \frac{\sum_{p=q}^{D} \sum_{C \in \mathcal{E}_l(p)} \prod_{i=1}^{g} \binom{d}{c_i}}{\sum_{p=q}^{D} \binom{D}{p}}. \tag{6.2}$$

Interpretation A plot of the cumulative probability $\Pr_{\geq q}(g, d, q, l)$ for $g = 10$, $d = 6$ and $l \in \{2, 3, 4\}$ is shown in Fig. 6.1. Also highlighted are the $q_{0.9}$ quantiles—each rounded up—that indicate the number of query attributes at which at least 90% of all queries have l-efficient (or better) access. For example, there is a (larger

[1] A *weak* composition of integer q is an *ordered* sequence of integer parts that sum up to q and may can contain 0 parts. Not to be confused with *Team compositions*.

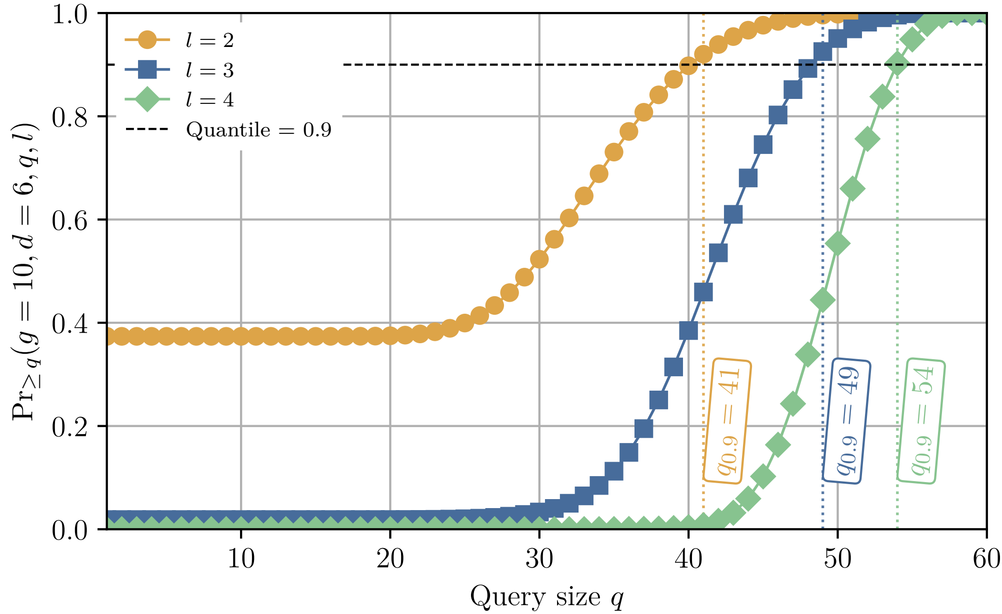


Figure 6.1: The cumulative probability $\Pr_{\geq q}(g = 10, d = 6, q, l)$, that a query over at least q-many different attributes accesses either none or at least l different attributes across *all* 10 of the 6-sized Teams. The distribution indicates the query dimensionality at which access with multiple attributes per Team becomes likely.

than) 0.9 probability that a random queries that use at least 49 of 60 attributes is 3-efficient. For $l = 2$, the probability drops towards ≈ 0.374—or in other words, more than a third of all possible queries are always 2-efficient. For configurations with larger g, i.e., more Teams, the distributions shift to the right and plateau lower, i.e., the changes for efficient access drop. Larger d, i.e., Team size, has the opposite effect. Notably, for $l > 2$ and low q, the probabilities are not 0, but the chances for l-efficient access become vanishingly low.

Computing the numerators of the distributions requires computing a sum over a set of integer compositions. A naïve implementation can become very costly for larger attribute counts, if it requires full enumeration of restricted compositions [123], which is some subset of a set of size 2^{q-1}. Using generating functions instead offers an equivalent but more efficient way of computing the distribution.[2]

Overall, queries that leave out *some* attributes are still very likely to provide *some* benefit in combined selectivity for every Team. On the other hand, access quickly deteriorates because accesses with only a single attribute become more

[2]The above described proof was developed in an earlier submission version of the Team index paper, but the idea to use *generating functions* came from a discussion with ChatGPT [124]. The generating function suggested by the tool and as used in the plotting code is $F(x)^g = \sum_{q=0}^{gd} \left[\sum_{C \in \mathcal{E}_l(p)} \prod_{i=1}^{g} \binom{d}{c_i} \right] x^q$, such that the *coefficient* of x^q gives the numerator of Eq. (6.1).

frequent, diminishing the benefit of using Teams. This analysis is also *pessimistic*, because efficiency is determined solely by the worst Team access—regardless how efficient all other accesses may be. However, even if sufficiently high coverage is given, compositions may still differ in how attributes are grouped. Making choices on how to actually group attributes requires domain-specific knowledge about which attributes are going to be used together and their *combined* selectivity.

6.2 Exploiting Combined Selectivity

Ensuring sufficient attribute coverage presents a first step, but there are various other factors to the quality of a Team composition. For example, combining two low-selectivity predicates in one Team and two high-selectivity predicates in another is likely to have less efficient performance than mixing both—although our efficiency notion in Sec. 6.1 considered both to be equal. This is similar to how imbalanced Team sizes have different performance than balanced ones (see Sec. 5.4.3).

As safe assumption can make is that combining multiple attributes in one Team allows us to assume the Volume can be reduced to at least the selectivity of the most selective predicate. And additional predicates usually improve this selectivity further. Improving upon this assessment quickly becomes difficult, however.

6.2.1 Query–Data Dependencies

Aside from simple selectivity, performance depends on *where* predicates restrict the attribute domain, since multi-dimensional data dependencies are "conditioned" differently. Altering the position (not the size) of the query in a Team—or just a single predicate—results in a different intersection result and potentially also a different selectivity, similar to how shifting the entire query rectangle does (see Sec. 5.4.3).

Multi-Dimensional Dependencies This dependency can be approached by empirically examining filter predicate placement relative to the data distribution. For instance, a query might target the sparse tail of a skewed domain or a slice from a dense region, implying a correlation between the predicate constant and data space. However, simple (linear) correlation statistics are insufficient, as they fail to account for query patterns. Furthermore, dependencies can be non-linear or conditional, particularly within (and across) multi-dimensional Teams. Ultimately, query-data dependencies can also stem from workload patterns or domain-specific constraints.

If information on such a dependency between query and data space is available, we can leverage it for Team composition. For example, we might prefer combining attributes that frequently yield high selectivity directly, rather than achieving this selectivity through intersection later. While potentially beneficial, obtaining information on combined selectivities is generally difficult. An estimation

requires knowledge of the high-dimensional joint probability distribution in order to assess (all relevant subsets of) predicate combinations. This notoriously hard problem of cardinality estimation is similarly encountered in query plan optimization [125]. Overall, this makes Team composition a hard problem, but we found that even simple selectivity statistics can help us to at least discard compositions with unfavorable attribute combinations.

Assuming Independence In contrast to high-dimensional distributions that are costly to estimate, selectivity statistics for a query workload can be efficiently obtained from quantile information. Although a product of individual selectivities (usually incorrectly) assumes independence, grouping attributes with good selectivities and bad selectivities into their own separate groups strongly suggests an increased Volume across Teams. For example, a composition of two Teams and 4 attributes may consider a Volume of roughly $0.1 \cdot 0.1 + 0.9 \cdot 0.9 = 0.82$ IDs, whereas another for the same query may be closer to $0.1 \cdot 0.9 + 0.9 \cdot 0.1 = 0.18$ IDs. Intuitively, combining attributes enables compensating for badly selective attributes with more selective ones. Although performance generally improves with reduced Volume (see Sec. 5.4), it does not always result in lower runtime, with one reason being that low Volume may come at the cost of high Overhead. Regardless, we will still use these statistics to take first steps towards developing more suitable cost models for Team composition.

6.2.2 Workload-Aware, Optimal Volume

If a query workload with a set of per-attribute selectivities is available, we can use it to systematically find a Team composition that minimizes the total access Volume over an entire workload. We assess the Volume of a particular composition using $\sum_{q \in W} \sum_{T \in C} \prod_{a \in T} s_q(a)$, where W is a workload and $s_q(a)$ the selectivity of query q for attribute a. Note that we also restrict the maximum Team size to d, which reflects a limit on Overhead. Further, we require that composition C is a set *partition* of the D attributes into Teams denoted as T.

The term $\prod_{a \in T} s_q(a)$ is *super-modular* and (non-strict) *monotone decreasing* in T: adding additional attributes/predicates cannot increase Volume, and the marginal decrease diminishes with increasing Team size. Exact minimization of a supermodular function is NP-hard, since it is equivalent to maximization of a submodular function [126]. However, there are various ways to simplify the problem and enable heuristics. For example, restricting the domain of $s_q(a)$ to a discrete domain, e.g., multiples of $1/b$, allows for cheaper cost comparisons.

For configurations with low Team size and limited table dimensionality, e.g., $d \in \{2, \dots, 8\}$ and $D \leq 16$, an exhaustive search remains feasible, especially when combined with branch-and-bound pruning techniques. Note that we insist on a proper set partition and require all attributes to appear in at least one Team. Another way would be to soften the requirement on Team count (or alternatively *size*), which turns it into a *weakly-α super-modular* problem [127]. Allowing overlap

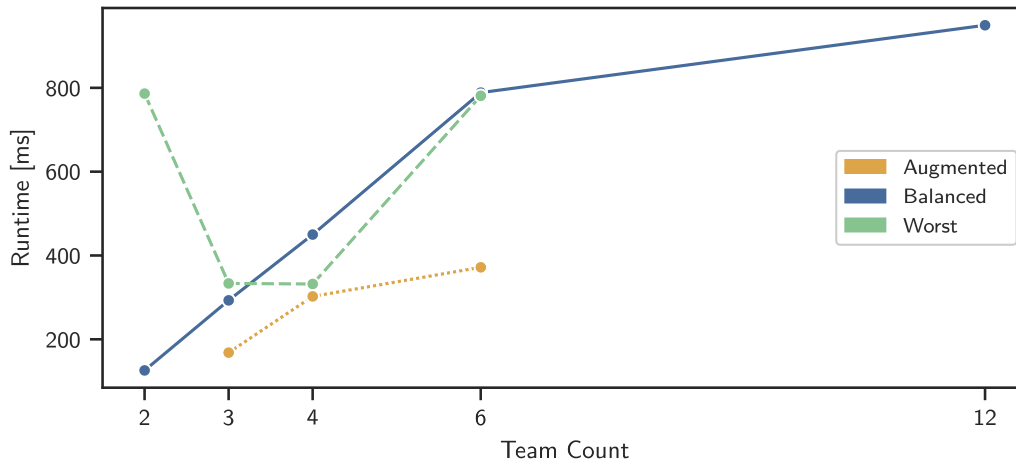


Figure 6.2: Intersection runtime for Team compositions with lowest and highest Volume. Lowest-Volume wins for larger Teams. Augmentation can improve performance.

between Teams increases the solution space further and turns the problem into a set-cover problem, but could open up additional heuristics.

Volume-Optimality vs. Runtime To show the actual impact of choosing Volume-optimal (as modeled above) Team compositions, we determined the ideal and worst compositions for our running example query using exhaustive search with strict partition requirement and Team count restriction. We measured runtime (Y-axis), along with the number of relevant leaves and the corresponding I/O volume (in bytes; both not shown). We used $b = 10$ for every composition over the $D = 12$ SSP attributes. The results are shown in Fig. 6.2. Using larger Teams reduces the total number of Teams, i.e., high dimensional (up to $d = 6$) compositions are on the left.

The Volume difference between best and worst composition ranged from $1.25\times$ (low d) to $10.52\times$ (high d). Overall, performance improves significantly by using fewer and larger Teams, but optimal access Volume does not imply optimal runtime performance. If Volume differences are not too large, with the "worst" composition for $d = 3$ even performing slightly faster, despite a 40% higher Volume. In these cases, factors other than Volume, such as a beneficial order within ISEs for the particular Team compositions, were more influential. Runtime differed significantly in the high dimensional case of $d = 6$, however, where the Volume difference was more substantial—about $10.52\times$. Generally, for fixed b, d and number of Teams, a reduction in relevant Volume obtained solely by swapping attributes between Teams, correlates with a lower Overhead as well. Or in other words, if the sum of query rectangle sizes (in grid space) becomes smaller, each rectangle usually also selects fewer bins. In this experiment, this is particularly noticeable for $d = 6$,

where the worst had about $3.5\times$ as much Overhead as the best. We will confirm the general trend in a later experiment.

Overall, a holistic cost model for Team composition, beyond Volume estimation and simple constraints like Team size, should not only penalize high-Overhead compositions, but also account for queries that can mitigate Overhead through high per-Team selectivities.

6.2.3 Allowing Attribute Overlap

Another interesting application of selectivity information is the re-use of highly-selective attributes across multiple Teams. This can improve overall Team selectivity, but at the cost of storage and larger Overhead at runtime. To show the effect of this domain-specific optimization, we augmented the ideal compositions by adding the most selective attribute to each of the other Teams as well, increasing their size. The results are also shown in Fig. 6.2. Although Overhead was larger, the reduced Volume from the additional selectivity significantly improved intersection runtime compared to compositions without overlap. Note that this optimization has diminishing returns for Teams that are already high dimensional ($d \geq 6$) and more likely to result in performance degradation.

6.2.4 Overhead vs. Volume

The reduction in access Volume with increasing Team dimensionality is not specific to our example query. Generally, as Volume decreases, Overhead, i.e., the number of relevant leaves, increases. To demonstrate this, we measured 16-dimensional random queries and 10 random Team compositions for various levels of precision and dimensionalities on our LHCb dataset. Overhead versus accessed Volume is shown in Fig. 6.3 (next page) on a log-log scale. We evaluated $b \in \{5, 10, 16\}$ bins per attribute, balanced Team compositions with $16/d$-many Teams and Team sizes $d \in \{1, 2, 4, 6, 8\}$. For $d = 6$, we used one 6-dimensional Team and two 5-dimensional Teams, otherwise Team sizes are equal. We omitted configurations with $b^d > 10^6$ due to prohibitively long runtimes. As workload, we generated five queries in three different *flavors*: Two with equal predicate selectivities of $16 \times s$ for $s \in \{0.2, 0.5\}$ ("balanced"/"balanced selective") and one with varying selectivities denoted as "diverse" ($4 \times 0.2, 4 \times 0.4, 4 \times 0.6, 4 \times 0.8$). A different color indicates a different index configurations, i.e., pairs (b, d), and symbols represent the query flavor. Points with identical color, hue and symbol represent different random indices. As indicated by the "constellations," points with the same color *group* (green, brown, blue) and symbol are directly comparable.

Generally, the trend from the upper-left (low d) to lower-right corner (high d) shows the exponential trade-off between Overhead and Volume. Further, larger b increases Overhead and slightly decreases Volume. Balanced, highly selective queries ($\times$) require both less Overhead and access less Volume than less selective (•) or imbalanced (▪) query flavors. Within a specific configuration of b and d,

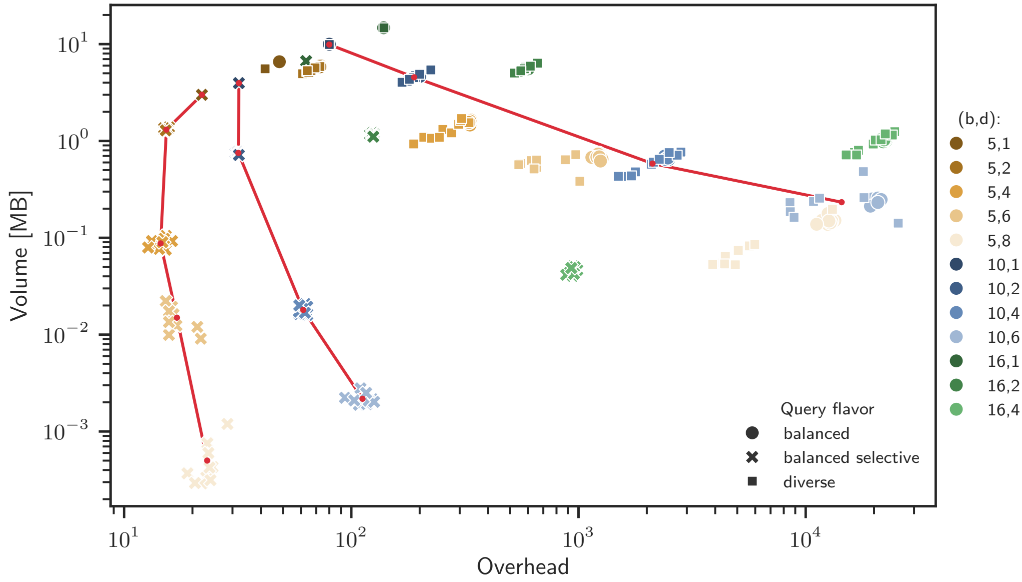


Figure 6.3: Number of accessed leaves (Overhead) vs. accessed data Volume for different configurations (d and b) and 10 balanced random Team compositions. Access Volume decreases as Team dimensionality—and thus Overhead—increases.

i.e., identical color, accessing less data also implies fewer relevant leaves. The variance within each configuration is low, with the exception of the "diverse" query flavor with $b = 10, d = 6$, where the average bin cardinality is only $N/b^d \approx 1221$. Both Overhead and Volume can vary significantly due to differences in effective predicate selectivity, favoring some compositions and resulting in a larger spread.

6.2.5 Storage Costs vs. Runtime

Storage cost and runtime are the two core metrics for evaluating index performance. For the Team indices considered in this work, runtime is almost entirely determined by the cost of intersection. Fig. 6.4 shows the impact of Team compositions on these metrics for the same experiment as in Fig. 6.3. As before, measurements with the same symbol and color group (b) are directly comparable.

One-dimensional Teams require more storage space with the notable exception of $b = 5$, where each attribute is represented with only five bitmaps. However, this advantage vanishes quickly for increased levels of precision. Otherwise, storage costs decrease as d increases (from right to left), due to a decrease in Team count. For configurations with many bins, i.e., $b = 5, d = 8$ and $b = 10, d = 6$, decreasing compressibility and (for this D) diminishing gains from fewer Teams again lead to a slight increase. Absolute (horizontal) variation across random Team compositions is low if bin count is not too large.

In terms of runtime, results vary depending on query flavor and index configuration. First, we observe that the *balanced* query generally performs worse than the

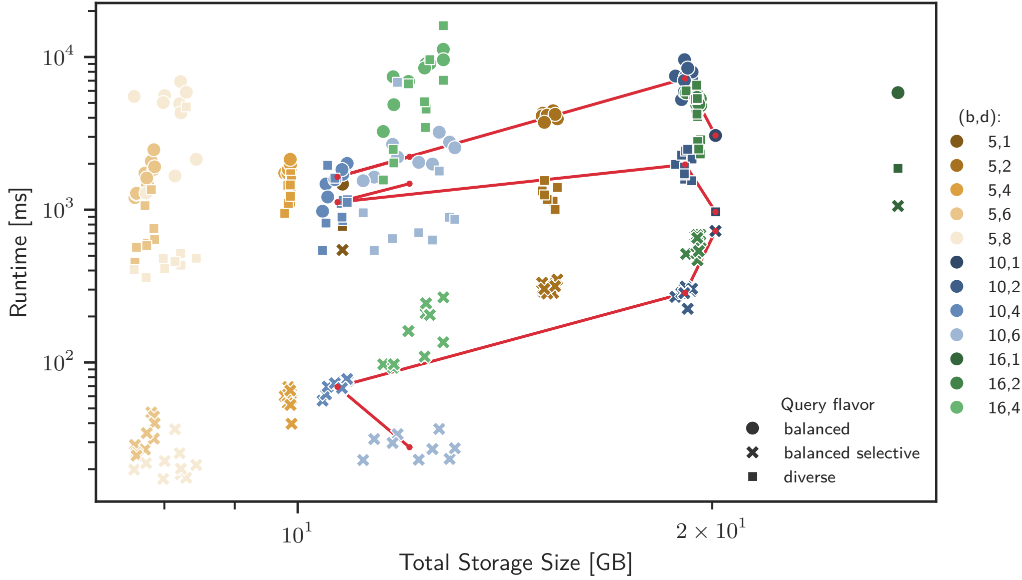


Figure 6.4: Storage costs vs. intersection runtime. One-dimensional Teams incur higher storage costs. For highly selectivity queries, higher dimensional Teams offer better runtime.

diverse query (black square), especially for low-dimensional Teams. Further, the *selective-balanced* query flavor was fastest by a wide margin, because of both a very small Volume and Overhead. We attribute the difference across flavors primarily to their respective result cardinalities, which were between $2.5 - 10 \times$ smaller (depending on b) for the *diverse* queries and 0 (or very close) for the $\times$-flavor. Smaller results lead to faster execution, especially for low-dimensional Teams, where ISEs involve intersections across more Teams and a larger result implies that fewer ISEs can terminate early due to an empty result. Two-dimensional Teams generally offer no performance benefits and are dominated by other configurations.

The most interesting combination was $b = 10$ for the diverse queries, where 1-dimensional Teams were outperformed by some compositions of $d = 4, 6$, whereas others performed significantly worse and had large variance. Upon closer investigation, we found that the composition-dependent differences were caused by multiple factors. First, our implementation lacks fine-grained adaptivity to query-specific variation, e.g., per-ISE intersection orders. Given the larger Team dimensionality and larger variety of predicate selectivities, differences in data dependencies were more likely to be noticeable across queries of this flavor. This is also reflected in the larger variance of Volume- and Overhead shown in Fig. 6.3. In addition, Overhead for these configurations was relatively high (w.r.t. N) and therefore more influential on less-ideal configurations, causing larger fluctuations.

6.2.6 Discussion

In our experiments, we observed that the particular choice of Team composition had less effect on overall performance and storage costs than the general configuration, i.e., the number of bins per dimension (b) and the maximum Team dimensionality (d). Therefore, it is possible to significantly reduce Volume with larger Teams, but this scaling has limits. Although Overhead also increases, its costs are independent of the table cardinality. For queries with high selectivity predicates, e.g., $s = 0.2$, Team-based yields significant reductions in both storage footprint and runtime. Moreover, Teams must have sufficient size ($d > 2$) to more efficiently compensate for non-selective attributes.

Given our current prototype implementation, the tested table sizes and various open optimizations, it was not always possible to fully translate reductions in Volume into an equal reduction in runtime. Our physical implementation was not sufficiently fine-tuned to mitigate the impact of Overhead costs, which was particularly noticeable for high-dimensional configurations. Moreover, indexing larger tables offers a simple way to more effectively amortize Overhead with larger leaves and thus improve performance and unlock (a few) additional parameter configurations, most notably with respect to d. We plan to investigate this further using the TeamBench benchmark generator (see Chapter 7).

Another direction is logical optimization, e.g., individually optimizing the ISE operation order. But also the selection of the two tuning knobs introduced in this work, leaf grouping and expansion, can be specialized further for edge cases—for instance, when one Team selects very few leaves but another selects many. More holistic formal objectives for Team composition can offer additional gains by adapting to workload knowledge.

7 TeamBench

The performance of index intersection is determined by various factors, such as data distribution, query placement and selectivity, Team composition, and number of leaves. In Sec. 5.4, we broadly categorized these factors into Volume, Overhead and Imbalance. Improving existing implementations and developing more sophisticated optimization strategies requires understanding these factors further. Unfortunately, a systematic assessment purely on real-world data is complicated by the inability to isolate the impact of these influences and study them separately. Micro benchmarks aimed at specific components are also limited in what they can reveal about the performance of the whole process. For example, a benchmark dedicated to the purpose of accessing the performance of new task scheduling strategies may not be be able to account for runtime variation of operators in actual workloads.

Another issue is the large flexibility in Team composition and potentially very large table cardinality necessary to assess the behavior of the design under realistic load. Explicitly adjusting Team compositions and binning to produce exactly the desired test environment is a costly and difficult endeavor. Moreover, doing so repeatedly may not be feasible for problems at the very large scale. As a result, it would be of great benefit to be able to directly generate *indices* that exactly meet specifications and allow to isolate aspects of interest.

We introduce our initial findings on *TeamBench*—a data generator for index intersection. TeamBench is highly configurable and efficient, i.e., it directly generates leaves for Team indices and only IDs that actually take part in the index intersection. Moreover, it is trivially parallelizable and avoids repeated sampling draws, i.e., has a fixed runtime that is fully determined by the input parameters.

Outline After introducing the core algorithm, we provide additional metrics to aid in choosing input parameters. Thereafter, we discuss complementary methods for creating custom probability distributions that formally resemble bin-cardinality

distributions of real-world grid index structures (as described in Sec. 4.1.2). We close with a discussion on possible extensions.

7.1 Defining the Generator

The design of TeamBench is inspired by the Grid Index introduced in Sec. 4.1.2, but generalizes beyond its specific design decisions. Our primary goal is to directly generate leaves, which involves much less data than the underlying dataset and also avoids an explicit index creation step. We also aim to control the *Volume*,[1]i.e., the "cost" of the intersection, and also to incorporate the structure and volume contributed by each relevant Team. Finally, want to have some measure of control on how selective a query executed on this data is going to be. The final result is a set of leaves for each Team.

7.1.1 Input Parameters

There are three parameters to control the benchmark generator for an intersection of 2 or more Teams. We explain every parameter in turn below:

- ▷ $\{(p_t, Q_t)\}_t$ A set of Team-specific *bin cardinality distributions and queries*
- ▷ N The *number of unique IDs* involved in the intersection
- ▷ S_{rel} A *selectivity* parameter to control the actual result size

The distribution p_t represents the relative size of the leaves of team t. We treat it as a flat vector, but the respective Team index may still be multi-dimensional. It's length corresponds to the number of all bins in the Team. p_t may be taken from an existing Team index or have an explicitly specified form (see Sec. 7.2.3 below). Q_t is a set of bin indices of *all query-relevant leaves* for Team $t \in T = [1, n]$, i.e., $\forall i \in Q_t : p_t(i) > 0$. Note that Q_t must be a proper, non-empty subset of all possible leaves, i.e., p_t's entire support. Otherwise the Team would not contribute selectivity (or select nothing). The components of the *conditional distribution* are given as

$$p_{t|Q}(i) := \frac{p_t(i)}{\sum_{j \in Q} p_t(j)}.$$

By using a set of p_t and Q_t as input parameters for TeamBench, we can precisely control the intra-grid Imbalance for each Team, i.e., the relative sizes of the leaves in each Team. This also directly specifies the respective Team selectivities and the overall Overhead.

The integer N determines the number of *unique* IDs (and samples) we use to generate the benchmark data set. We create two types of IDs: *result IDs*, which survive the intersection, and *drag IDs*, which take part in the intersection but are pruned eventually.

[1]Our tool is not able to offer precise control over the actual *Volume* in byte, as we have defined the term in Sec. 5.4.1, because this would require to account for compression. TeamBench can directly control the number of IDs involved in the intersection, however.

The $S_{\text{rel}} \in [0, 1]$ parameter controls how many IDs survive the intersection and implies the actual result selectivity relative to N, defined as $S := S_{\text{rel}} \cdot \min_t(s_t)$, where $s_t := \sum_{i \in Q} p_t(i)$ is Team t's selectivity. Selecting $S_{\text{rel}} = 0$ gives an empty intersection, whereas $S_{\text{rel}} = 1$ produces the largest possible result size ($N \cdot \min_t(s_t)$). Note that specifying S_{rel} instead of S ensures the result selectivity conforms with the most selective Team. As we will see below, the total, uncompressed intersection Volume is impacted by all three parameters.

7.1.2 The Sampling Procedure

The basic procedure is straightforward and generally works "in reverse" to the actual intersection. We repeat the following exactly N times:

Initially, we flip a biased coin to determine which kind of value we will generate, with probability S for a result ID and $1 - S$ for a drag ID. A result needs to be present in all Teams, so we assign it to exactly one bin in every Team, chosen according to the respective conditional distribution $p_{t|Q}$.

For exactly two Teams, assigning a drag ID proceeds similarly. Since the ID is going to partake in the intersection but gets dropped eventually, there has to exist (at least) one Team where its not in any bin selected by the respective Q_t. Thus, we choose to assign it to $U \subsetneq T$, a *proper* random subset of all Teams. In case of two Teams, we may choose either Team according to a distribution π, where $\pi(t) := \frac{s_t - S}{\sum_{u \in T} s_u - S}$ is the normalized weight of Team t after accounting for the mass that we assigned for result IDs. Note, in case of $S_{\text{rel}} = 1$, the subtraction of S results in (at least) one Team being unable to take on any drag IDs.

To see why this procedure ensures that we get exactly the desired number of survivors, consider the *intersection first* approach described on Eq. (5.2). The expression highlights that we (either explicitly or implicitly) exhaustively considered the intersection of every possible combination of leaves across all relevant Teams. Thus, assigning the ID to (exactly) one *relevant* bin per Team ensures it will be included in the final result. Conversely, leaving one or more Teams out during the assignment guarantees the value will be pruned. Notably, this process assigns IDs *independently* for each Team. This design choice is deliberate and we discuss it further in Sec. 7.3.1 below.

In case we have more than two Teams, the procedure becomes more complex, since we now first have to choose *how many* Teams we will be assigning drag IDs to. Thus, we first draw $k = |U|$ from random variable $K \sim P$ with $P = (P_1, \ldots, P_{n-1})$, before choosing any (proper) subset. However, P has to be chosen with care as to not distort the drag volume distribution π. We discuss the design of P further in Sec. 7.1.3.

There are various algorithms for choosing subsets of size k according to an (arbitrary) distribution π, we choose Gumbel Top-k [128]. Another suitable choice is the algorithm by Efraimidis and Spirakis [129].

The complete TeamBench sampling algorithm is shown in Alg. 2 (next page).

Algorithm 2 TeamBench Sampling Algorithm

Input: $\{(Q_t, p_t)\}_{t \in T}$ (queries, Team distributions)

Input: $S_{\text{rel}} \in [0,1]$ (relative result size)

Input: N (total sample count)

▷ **Preprocessing:**

1: **for** $t = 1$ **to** n **do**
2:     **for** $i \in Q_t$ **do**
3:         $p_{t|Q}(i) \leftarrow \frac{p_t(i)}{\sum_{j \in Q} p_t(j)}$ ▷ conditional distribution
4:     $s_t \leftarrow \sum_{i \in Q} p_t(i)$ ▷ Team selectivity
5: $S \leftarrow S_{\text{rel}} \cdot \min_t s_t$ ▷ selectivity w.r.t. N
6: **for** $t = 1$ **to** n **do**
7:     $\pi_t \leftarrow \frac{s_t - S}{\sum_u (s_u - S)}$ ▷ drag volume weights
8: $P \leftarrow$ ComputeDistributionP(π) ▷ $P = (P_1, \ldots, P_{n-1})$

▷ **Start sampling:**

9: **for** $x \in [0, N)$ **do**
10:     Draw $c \leftarrow$ RandomFloat($[0,1]$) ▷ Flip a coin
11:     **if** $c < S$ **then** ▷ **result** ID
12:         **for** $t = 1$ **to** n **do**
13:             $\text{bin}_t \leftarrow$ RandomInteger($[0, |Q_t|),\ p_{t|Q}$)
14:     **else** ▷ **drag** ID
15:         $k \leftarrow$ RandomInteger($[1, n-1],\ P$)
16:         $U \leftarrow$ RandomSubset($T,\ k,\ \pi$) ▷ without replacement
17:         **for** $t = 1$ **to** n **do**
18:             **if** $t \in U$ **then**
19:                 $\text{bin}_t \leftarrow$ RandomInteger($[0, |Q_t|),\ p_{t|Q}$)
20:             **else**
21:                 $\text{bin}_t \leftarrow$ None
22:     **assign** x to $(\text{bin}_1, \ldots, \text{bin}_n)$

23: **function** ComputeDistributionP(π)
24:     $I \leftarrow [\Pr(t \mid k)]_{t,k}$ ▷ via sampling using π
25:     $P \leftarrow$ SolveRLSOP(I, π) ▷ solve $I \cdot P = \pi$
26:     **return** P

7.1.3 Computing Distribution P

Before assigning a drag ID to a subset of Teams, we need to determine the number of Teams, $k \in [1, n-1]$. To see why we this can not be done naïvely, imagine we first pick the number k uniformly at random, and then subsets according to π. Because we now assign IDs to exactly k Teams, Teams that overall contribute less are then forced to take on a value, even though their "capacity" would be overtaxed. Thus, the probability of choosing a specific Team should depend on how many Teams we want to assign drag IDs to. Intuitively, picking a larger k should be less likely, if there are Teams that have only a small contribution to the overall drag-mass. On the other hand, if all Teams contribute the same, choosing k may be done in a uniform way.

We therefore need to derive the distribution P from a formulation that incorporates this requirement. To achieve this, we express that we need to adapt the probability for choosing a specific t depending on the value k as a matrix $I \in \mathbb{R}^{x \times (n-1)}$, where each column in I represents a distribution for choosing a specific t given some k, i.e., $I_{t,k} = \Pr(t \mid k)$. Given such a matrix, we can enforce our goal of assigning drag volume exactly in the proportion defined by π by restricting the row corresponding to t to add up to π_t, if we pick t with probability P_t. We require $I \cdot P = \pi$, where both distribution P and matrix I are unknown.

As illustrated in the case of $n = 2$, we observe that the first column of I, which corresponds to $k = 1$, is always exactly π. However, the other columns differ to account for the fact that we may choose more than one Team. An interesting insight is that we can estimate the remaining $\Pr(t \mid k)$'s *empirically* by using the above mentioned algorithms to draw weighted subsets of size k. Specifically, for all $k > 1$, we repeatedly draw subsets of size k using probability π and then record which grids got sampled. Normalizing the columns afterwards yields an approximation of the respective (column) distributions.[2]

After finding (an approximation of) I, the expression $I \cdot P = \pi$ becomes a system of linear equations and solving it provides the desired distribution P for sampling k. Since I is not a square matrix, we cannot derive P by inverting I. Instead, we formulate it as an (overdetermined) least-squares optimization problem with constraints to enforce the probability distribution property:

$$
\begin{aligned}
\min_{P} \quad & \|I \cdot P - \pi\|_2^2 \\
\text{s.t.} \quad & \sum_k P(k) = 1, \quad P(k) \geq 0 \text{ for all } k
\end{aligned}
$$

This convex optimization problem can be solved using software libraries, such as SciPy's LSQ_LINEAR [130], to yield distribution P.

[2]The original idea for the simulation of I was developed in a dialogue with OpenAI's ChatGPT [124]. The discussion also led to a possible closed form solution for computing I, which we omit for brevity.

7.2 Choosing Parameters

The parameter choices have an intertwined impact on the performance of the intersection. For example, the (uncompressed) Volume, which directly influences the actual runtime cost of the intersection, is not chosen explicitly but implied by the other parameters. We therefore give estimations to aid in determining a suitable N. Thereafter, we discuss how distributions p_t may be deliberately designed.

7.2.1 Estimated, Uncompressed Volume

The *total Volume* in terms of ID count across all Teams is not determined by the parameter N alone, since it also depends on the respective Team selectivities. We can estimate it by making use of the independence assumptions and the P distribution we use for sampling k. In expectation, we assign every drag ID to $\mathbb{E}[K] = \sum_{k=1}^{n-1} k \cdot P(k)$ many Teams and there are $N \cdot (1 - S)$ unique drag IDs in total. Result IDs are assigned to all n Teams, thus there are $n \cdot N \cdot S$ in total. The total Volume involved in the intersection then comes to

$$\text{(uncompressed) Volume} = N \cdot (\mathbb{E}[K] \cdot (1 - S) + n \cdot S)\,.$$

7.2.2 Sufficient Sample Count

Although *drag Volume* may give a suitable impression of the overall memory consumption of the operation, it does not account for the probability distribution specified for each Team. To obtain a dataset that fills even small, i.e., less probable bins, we need to select an N that is sufficiently large. Thus, we may require that every bin/leaf receives at least C many IDs in total.

Let $p_{\text{min},u} := \min_{t,i}(p_{t|Q}(i))$ be the smallest non-zero bin probability across all grids and let u be the corresponding Team grid. The chance that u's bin receives a *result ID* is given by $S \cdot p_{\text{min},u}$. For a *drag ID*, we also need to account for the subset selection, which is independent of the bin selection. For any i, we have $(1 - S) \cdot p_{t|Q}(i) \cdot \pi(t)$ for the chance of a drag ID. Thus, we need to draw at least

$$\begin{aligned} C &\leq N \cdot (S \cdot p_{\text{min},u} + (1 - S) \cdot p_{\text{min},u} \cdot \pi(u)) \\ N &\geq \frac{C}{p_{\text{min},u} \cdot (S + (1 - S) \cdot \pi(u))} \end{aligned}$$

samples in order to have C or more IDs in all leafs.

In practice, filling up even the smallest leaf may not be necessary and we may choose a larger target probability than $p_{\text{min},u}$ instead. For example, the 0.1 quantile across all probability values may already result in sufficiently high cardinality across *most* bins.

7.2.3 Custom Distributions

Instead of relying on empirical grid distributions derived from Team indices built over real data, we can also configure a parameterized grid distribution. This allows us to, for example, deliberately generate Teams with bin cardinality distributions that have specific skew for studying the impact of "heavy hitter" ISEs involving expensive intersections.

Although TeamBench is agnostic to the design decisions and constraints that went into the grid index used in our Team-based approach, we may still want to reproduce them to get more realistic behavior. For example, the primary property of a Team's grid cardinality distribution are the uniform marginal distributions, which forced all bins to be at most $1/b$ in size; the distribution essentially only captures the dependency structure of the Team attributes, not the actual (one-dimensional) attribute distributions. Moreover, real-world bin cardinality distributions usually exhibit some form of *smoothness*, i.e., cardinalities tend to not vary abruptly across adjacent bins.

One way to enforce these criteria on a multi-dimensional *prior* distribution, such as a Gaussian or mixture thereof, is to formulate an optimization problem. Starting with a chosen prior distribution w_t, one can minimize the Kulback-Leibler divergence $\mathrm{KL}(x_t) = \sum_i x_t(i) \cdot \log(x_t(i)/w_t(i))$ with respect to additional uniform-margin and probability constraints. The divergence is a well-known measure for comparing distributions and its *convexity* allows us to solve the problem with a Python library such as CVXPY [131]. Geometrically, this transformation tends to "squash" the prior distribution to enforce the constraints, pushing mass from higher cardinality bins to nearby bins, leading to a smoother, more "rectangular" version. This parallels the ID distribution across bins that results from using quantiles to define Team grids, as shown in Fig. 4.1.

Although this gives a faithful way to construct a "realistic" cardinality distribution for Team indices, there may be a more straightforward way. After conditioning, the distribution no longer conforms with (all) the marginal constraints and only the maximum probability of $1/b$ is still satisfied. Specifying the conditional distribution directly and scaling it down may therefore offer a more immediate approach.

7.3 Discussion

This chapter presented TeamBench, our sampling procedure for benchmarking index intersection. Although its input parameters allow for flexible configurations, certain simplifications were made that may affect result quality and subsequently observed performance characteristics. Moreover, there are various aspects of TeamBench that allow for modifications and generalizations and may provide directions for future work.

7.3.1 On Inter-Grid Dependence

As previously discussed, we assign (both drag and result) IDs to Teams independently, that is, bin selection for one Team does not impact bin selection for other Teams. Consequently, each bin assignment is independent. In contrast, real-world data usually has data dependencies: intersections of certain combinations of leaves across Teams yield larger results than others or may exhibit different runtime despite similar cardinalities. In the current approach, intersection mass is spread independently across all leaf combinations and assignments solely depend on the (relative) size of the leaves.

This has implications on the performance of index intersection. The absence of any exploitable pattern, that is, clustered sequences of IDs within leaves, leads to comparable set operation runtimes primarily determined by size alone. That means that intersections across different but similarly sized leaves are also likely to have similar runtime. This can be a desirable trait for benchmarking, since it allows for a focused analysis of other aspects of the algorithm, such as, scheduling or Overhead costs.

Moreover, deliberately controlling these dependencies makes the whole sampling approach significantly more involved and costly: not only do we have to somehow specify the multi-dimensional relationships, we also have to take care to not introduce bias into the target distributions, and, for instance, increase the number of values assigned to rare bins.

7.3.2 Compression & Order of ID Generation

As indicated before, TeamBench does not explicitly control compressibility (and therefore actual Volume in byte) and any leaves produced will likely to exhibit poor compression rates. Deliberately introducing a specific order can lead to unwanted behavior, however. In Alg. 2, we flipped a coin to determine the ID type, but we may also generate each in a fixed sized batch. Although this would ensure drag and result IDs are created in precise proportions, they are now potentially produced in a specific and overly unrealistic order. For example, creating all result IDs first may yield the continuous range $[0, S \cdot N)$ as index result. Due to compression this may lead to unrealistic behavior and index intersection runtime, for example Roaring is able to represent the result as a simple range, which impacts the size of the representation and the cost of set operations and full matches would only ever be found for low ID values. To avoid this generation artifact, we opted for the (cheap) coin toss approach instead.

Similar to our discussion in Sec. 7.3.1, we may still want to change the way we emit IDs to obtain a more "organic" order. For example, natural data tends to form streaks of consecutive or close IDs within the same bin, which leads to better compressibility; or certain bins tend to have lower ID values. With our current procedure, this behavior is unlikely to emerge on its own. However, relabeling the IDs after the generation may be computationally expensive, since every change in

labels has to be coordinated across all Teams. An alternative may be correlating subsequent sampling steps by increasing the chance to hit the same bin multiple times in a row. But similar to the introduction of inter-grid dependencies, distortion of the target probability distributions must be carefully avoided.

8 Conclusions

With ever-growing data volumes, the cost of data movement continues to rise, particularly in read-intensive, storage-bound analytics workloads. Indexing and Data-Centric Processing each present techniques and architectures to remedy this cost, but they also entail limitations and challenges. To achieve a truly holistic solution, a multi-faceted approach is therefore essential.

Cooperative Refinement As a first step, this thesis explored Cooperative Refinement, the combined application of indexing and Data-Centric Processing that is not merely complementary but symbiotic: Indexing can benefit from processing both near and in memory to decrease its own non-negligible cost. Simultaneously, it is highly suitable to adapt to the limitations and hazards of data-centric architectures. Moreover, utilizing index information directly within a storage device provides a fast path for efficient in-device materialization, which is a critical capability for making the most of the constrained resources of Computational Storage Devices. However, Cooperative Refinement also comes with its own set of unique challenges, requiring deliberate hardware-software co-design, thereby bridging the gap between the inherent resource restrictions of hardware and the functional requirements of software, specifically index structures.

The initial focus in Chapter 2 was on the design limitations of NAND flash memory, that any Processing-in-NAND architecture inherits: a limited power budget, a small instruction set and susceptibility to bit-flips. To position index structure evaluation as a prime candidate for PiN, we analyzed the inherent error tolerance of Bloom filters and binary sketches in Chapter 3. In an effort to extend the applicability of existing PiN concepts, we also introduced a strategy that uses masked-equality operations to emulate inequality—a primitive that is not supported in current PiN architectures but central for evaluating indices, such as the ubiquitous B^+-tree. The final topic of the chapter was Gravity Store, our initial

vision for an in-storage materialization framework. Its components are designed to offer a flexible basis for declarative, low-overhead interfaces and efficient, selective access, thereby addressing the inefficiency of today's host-centric communication.

Extending Indexing As our second step towards a holistic solution for addressing the Data Movement Tax, we presented Team-based indexing, a generalization of the bitmap indexing strategy for high-dimensional range queries at the very large scale. Indexing Teams, i.e., moderately-sized subsets of attributes, rather than all dimensions individually or all at once, allows us to exploit combined selectivity for faster index evaluation without suffering from the curse of dimensionality. It also enables to shift the focus toward an efficient intersection of individual Team results and away from the (individual) quality of index data structures. Moreover, it allows for considerable storage space savings, which is another crucial performance metric of index data structures.

After introducing the strategy in Chapter 4, we suggested optimization techniques for index intersection in Chapter 5, which is not dominated by (initial) storage access costs but the execution of in-memory set operations. Team-based indexing also requires addressing the problem of Team composition, which we investigated in Chapter 6. We provided an analysis for assessing ad-hoc queries, given that Teams may contain irrelevant attributes, and an early cost-model to guide the process of finding suitable compositions. The chapter also included our composition-dependent runtime evaluation of our prototype implementation. In Chapter 7, we introduced TeamBench, a highly configurable sampling procedure for generating benchmark data. TeamBench was specifically designed to test the performance of index intersection and therefore complements the preceding contributions on Team-based indexing.

8.1 Towards Data-Centric Database Processing

While indices are well-established in databases and beyond, data-centric architectures introduce significant complexity and are scarcely implemented in hardware. This not only hinders progress, it also shows that the research field is still in its infancy, despite its age [63, 62]. Part of the reason is the conflicting need to reconcile strict physical and economical hardware design restrictions with sufficiently flexible functionality. This is particularly acute for NAND flash memory: its limited lifespan and key role as the most affordable and widespread mass storage medium tighten these constraints even further, making high storage density ($/GB) the dominant design priority.

In contrast to many machine learning applications, databases are especially ill-suited for having *a* dedicated chip, due to the high degree of flexibility they require: table schemas, storage layouts, and workloads are extremely heterogeneous, which leads to entirely different requirements. Given this conflict of interests, our primary goal was to maximize the utility of a given hardware design for PiN-based index

evaluation. However, since our work is conceptual and addresses only isolated aspects, the practical feasibility of PiN-based index evaluation requires further investigation. Identifying a chip design with sufficiently broad applicability (and thus market potential) and a low enough design cost to justify prototypical hardware implementation would be an ideal next step. As mentioned in Sec. 3.5.1, our analyses should be extended to holistically incorporate whole-system considerations, economic factors and tighter integration with specific database applications.

This applies less to our initial concept of Gravity Store, which may already demonstrate its efficacy when used as a code generation engine for host-side materialization. An early prototypical implementation may therefore not require access to a sufficiently flexible CSD device for testing. Beyond the need for a suitable testing platform, Gravity Store itself may be used to identify conceptual limitations and refine self-imposed design restrictions regarding storage layouts, access patterns, and structural requirements for CSD compute capabilities. To this end, given the limited resources, it is not yet established what degree of in-device materialization is acceptable, and when full transfer to the host is warranted. This decision may ultimately need to be dynamic, for example based on workload characteristics or runtime resource pressure, rather than static.

Another aspect not covered in this thesis is the materialization of data for the express purpose of additional near-data processing, such as (grouped) aggregation or joins. Various existing CSD designs propose FPGAs as co-processors, which impose their own restrictions on the structure and modality of their inputs. Ideally, data flow and materialization efforts within the device would enable seamless integration with such components, thereby unlocking the full potential of data-centric storage design.

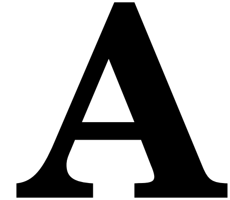
8.2 Beyond Team-based Indexing

Although Team-based indexing can lead to significant reductions in the accessed Volume, Overhead costs, such as the cost of generating I/Os or of scheduling set operations, still account for a considerable portion of the runtime of our prototype. Extending the evaluation to larger and more diverse datasets may help to improve both implementation and runtime optimization strategies—an endeavor that may be systematically aided by our TeamBench benchmark data generator.

Our discussion on Volume-based Team composition assessment suggested that other aspects should also be taken into consideration. For example, Team composition is ideally based on domain-specific prior knowledge of possible workloads. This also allows more targeted success evaluation of composition strategies, where performance for certain types of queries is weighted according to their importance.

Lastly, our investigations focused on index evaluation as an isolated problem, where producing a list of qualifying tuple IDs is the final result. However, a more holistic perspective would also take into account the distributed system, the underlying storage layout, and domain-specific workload characteristics. Without

this information, the runtime evaluation of an implementation for Team-based index intersection is difficult to put into perspective. Moreover, the usage of index evaluation for secondary tasks, such as cardinality estimation, may also be a promising direction for further exploration.



Abstract: Uniform Database Generation

Benchmark-oriented research and fuzz testing often require synthetic database instances that satisfy specific semantic constraints, such as tuple counts, attribute cardinalities, or functional dependencies. However, naïve generation procedures introduce bias because equivalence classes (of database instances) can vary significantly in size, causing larger classes to be disproportionately over-sampled. Instances within these classes differ only in literal values and can therefore exhibit the same runtime performance and error behavior.

We address this by introducing an unbiased generator that samples equivalence-class *representatives* of concrete databases sharing identical functional-dependency patterns, active domain sizes, and cardinalities. The generator focuses on schemas where attribute `A` functionally determines attribute `B`, and where `A` together with a third attribute `C` forms a primary key.

The key insight is that equivalence classes can be represented by *integer partitions*, which encode the active domain size and multiplicities of active values of attribute `A`. The algorithm first picks an integer partition uniformly at random and imposes suitable restrictions on part size and count. Distinct `C` values are then selected per occurrence to ensure a sufficiently large number of unique tuples. Values for `B` can be chosen independently, after the active domain of A is determined.

In [29], we prove the special case of normalized schemas and detail an efficient implementation design. The generator and its implementation are generalized to non-normalized schemas in [30].

Generative AI Declaration

In compliance with §11 (2) of the doctoral regulations and my own ethical responsibility for transparency, I hereby declare the ways I made use of generative AI for authoring this thesis. Generative AI was primarily used to provide editorial support in the writing process, for drafting figures, and for literature research. It was used to help with grammar, typographical errors, and the restructuring of original sentences and sections with the goal of improving the correctness and clarity of the representation. Moreover, it was used to confirm the correctness of the formal aspects of the thesis developed in prior works. Two notable contribution details that were inspired by a discussion with ChatGPT were made explicit via footnotes in Sec. 7.1.3 and Sec. 6.1.2. Any text or program code generated or modified by a generative AI tool and used in this thesis was fully reviewed and modified as needed. All content presented in the final version of this thesis is my own, and I claim full responsibility.

Bibliography

[1] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. "A Modern Primer on Processing in Memory." In: *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann.* Ed. by Mohamed M. Sabry Aly and Anupam Chattopadhyay. Singapore: Springer Nature Singapore, 2023, pp. 171–243. DOI: 10.1007/978-981-16-7487-7_7.

[2] Peter Kogge and John Shalf. "Exascale computing trends: Adjusting to the" new normal"'for computer architecture." In: *Computing in Science & Engineering* 15.6 (2013), pp. 16–26.

[3] Micron. *Micron 6550 ION NVMe SSD Product Page.* Accessed: 2025-06-11. URL: https://www.micron.com/products/storage/ssd/data-center-ssd/6550-ion.

[4] AMD. *AMD Epyc 9965 Processor Product Page.* Accessed: 2025-06-11. URL: https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html.

[5] Gabriel Haas and Viktor Leis. "What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines." In: *Proc. VLDB Endow.* 16.9 (2023), pp. 2090–2102.

[6] Gabriel Haas, Michael Haubenschild, and Viktor Leis. "Exploiting Directly-Attached NVMe Arrays in DBMS." In: *CIDR*. Vol. 20. 2020, p. 2.

[7] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. "Database Kernels: Seamless Integration of Database Systems and Fast Storage via CXL." In: *Conference on Innovative Data Systems Research (CIDR), Chaminade, HI, USA, January 14-17.* 2024.

[8] Alberto Lerner and Philippe Bonnet. "Not your Grandpa's SSD: The Era of Co-Designed Storage Devices." In: *International Conference on Management of Data (ICDE).* Virtual Event, China, 2021, pp. 2852–2858.

[9] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. "Biscuit: A Framework for Near-Data Processing of Big Data Workloads." In: *43rd ACM/IEEE Annual*

International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016. IEEE Computer Society, 2016, pp. 153–165. DOI: 10.1109/ISCA.2016.23.

[10] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. *Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture*. 2022. arXiv: 2105.03814 [cs.AR].

[11] Onur Mutlu, Ataberk Olgun, Geraldo F Oliveira, and Ismail E Yuksel. "Memory-Centric Computing: Recent Advances in Processing-in-DRAM." In: *2024 IEEE International Electron Devices Meeting (IEDM)*. IEEE. 2024, pp. 1–4.

[12] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology." In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 2017, pp. 273–287.

[13] İsmail Emir Yüksel, Yahya Can Tuğrul, Ataberk Olgun, F Nisa Bostancı, A Giray Yağlıkçı, Geraldo F Oliveira, Haocong Luo, Juan Gómez-Luna, Mohammad Sadrosadati, and Onur Mutlu. "Functionally-complete boolean logic in real dram chips: Experimental characterization and analysis." In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2024, pp. 280–296.

[14] Yun-Chih Chen, Yuan-Hao Chang, and Tei-Wei Kuo. "Search-in-Memory: Reliable, Versatile, and Efficient Data Matching in SSD's NAND Flash Memory Chip for Data Indexing Acceleration." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43.11 (2024), pp. 3864–3875.

[15] Jisung Park, Roknoddin Azizi, Geraldo F. Oliveira, Mohammad Sadrosadati, Rakesh Nadig, David Novo, Juan Gómez-Luna, Myungsuk Kim, and Onur Mutlu. "Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory." In: *55th International Symposium on Microarchitecture (MICRO)*. 2022, pp. 937–955.

[16] Han-Wen Hu, Wei-Chen Wang, Yuan-Hao Chang, Yung-Chun Lee, Bo-Rong Lin, Huai-Mu Wang, Yen-Po Lin, Yu-Ming Huang, Chong-Ying Lee, Tzu-Hsiang Su, Chih-Chang Hsieh, Chia-Ming Hu, Yi-Ting Lai, Chung Kuang Chen, Han-Sung Chen, Hsiang-Pang Li, Tei-Wei Kuo, Meng-Fan Chang, Keh-Chung Wang, Chun-Hsiung Hung, and Chih-Yuan Lu. "ICE: An Intelligent Cognition Engine with 3D NAND-based In-Memory Computing for Vector Similarity Search Acceleration." In: *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 2022, pp. 763–783. DOI: 10.1109/MICRO56248.2022.00058.

[17] Myoungjun Chun, Jaeyong Lee, Sanggu Lee, Myungsuk Kim, and Jihong Kim. “PiF: in-flash acceleration for data-intensive applications.” In: *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems* (2022).

[18] Alberto Lerner and Gustavo Alonso. “Data Flow Architectures for Data Processing on Modern Hardware.” In: *40th International Conference on Data Engineering (ICDE)*. 2024, pp. 5511–5522.

[19] Rudolf Bayer and Edward McCreight. “Organization and maintenance of large ordered indices.” In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 1970, pp. 107–141.

[20] Chee Yong Chan and Yannis E. Ioannidis. “Bitmap Index Design and Evaluation.” In: *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. Ed. by Laura M. Haas and Ashutosh Tiwary. ACM Press, 1998, pp. 355–366. DOI: `10.1145/276304.276336`.

[21] Kesheng Wu, Sean Ahern, E Wes Bethel, Jacqueline Chen, Hank Childs, Estelle Cormier-Michel, Cameron Geddes, Junmin Gu, Hans Hagen, Bernd Hamann, et al. “FastBit: interactively searching massive data.” In: *Journal of Physics: Conference Series*. Vol. 180. 1. IOP Publishing. 2009, p. 012053.

[22] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. “Cosmos+ OpenSSD: Rapid prototype for flash storage systems.” In: *ACM Transactions on Storage (TOS)* 16.3 (2020), pp. 1–35.

[23] ScaleFlux. *CSD 3000 Computational Storage Drive*. Accessed: 2025-06-09. URL: `https://scaleflux.com/products/csd-3000/`.

[24] Han-Wen Hu, Wei-Chen Wang, Chung Kuang Chen, Yung-Chun Lee, et al. “A 512Gb In-Memory-Computing 3D-NAND Flash Supporting Similar-Vector-Matching Operations on Edge-AI Devices.” In: *IEEE International Solid- State Circuits Conference (ISSCC)* 65 (2022), pp. 138–140.

[25] A Augusto Alves Jr, LM Andrade Filho, AF Barbosa, I Bediaga, G Cernicchiaro, G Guerrer, HP Lima Jr, AA Machado, J Magnin, F Marujo, et al. “The LHCb detector at the LHC.” In: *Journal of instrumentation* 3.08 (2008), S08005.

[26] Maximilian Berens, Yun-Chih Chen, Jian-Jia Chen, and Jens Teubner. “Beyond Bandwidth Doubling: Embrace Bit-Flips and Unlock Processing-in-NAND.” In: *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 2025, pp. 4649–4661. DOI: `10.1109/ICDE65448.2025.00373`.

[27] Maximilian Berens and Jens Teubner. “Index Intersection for High-Dimensional Range Queries.” In: *Proc. VLDB Endow.* 19.4 (Dec. 2025), pp. 767–779. DOI: `10.14778/3785297.3785315`.

[28] Michael Kußmann, Maximilian Berens, Ulrich Eitschberger, Ayse Kilic, Thomas Lindemann, Frank Meier, Ramon Niet, Margarete Schellenberg, Holger Stevens, Julian Wishahi, Bernhard Spaan, and Jens Teubner. "DeLorean: A Storage Layer to Analyze Physical Data at Scale." In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*. Ed. by Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland. Vol. P-265. LNI. GI, 2017, pp. 413–422. URL: `https://dl.gi.de/handle/20.500.12116/643`.

[29] Maximilian Berens and Joachim Biskup. "On Sampling Representatives of Relational Schemas with a Functional Dependency." In: *Foundations of Information and Knowledge Systems - 12th International Symposium, FoIKS 2022, Helsinki, Finland, June 20-23, 2022, Proceedings*. Ed. by Ivan Varzinczak. Vol. 13388. Lecture Notes in Computer Science. Springer, 2022, pp. 1–19. DOI: `10.1007/978-3-031-11321-5`.

[30] Maximilian Berens, Joachim Biskup, and Marcel Preuß. "Uniform probabilistic generation of relation instances satisfying a functional dependency." In: *Inf. Syst.* 103 (2022), p. 18. DOI: `10.1016/J.IS.2021.101848`.

[31] Ziyang Jiao, Xiangqun Zhang, Hojin Shin, Jongmoo Choi, and Bryan S. Kim. "The Design and Implementation of a Capacity-Variant Storage System." In: *USENIX Conference on File and Storage Technologies, FAST, Santa Clara, CA, USA, February 27-29*. 2024, pp. 159–176.

[32] Huan Tian, Jiewen Tang, Jun Li, Zhibing Sha, Fan Yang, Zhigang Cai, and Jianwei Liao. "Modeling Retention Errors of 3D NAND Flash for Optimizing Data Placement." In: *ACM Trans. Des. Autom. Electron. Syst.* 29.4 (June 2024).

[33] Chun-Yi Liu, Jagadish B. Kotra, Myoungsoo Jung, Mahmut T. Kandemir, and Chita R. Das. "SOML Read: Rethinking the Read Operation Granularity of 3D NAND SSDs." In: *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) , April 13-17, 2019*, pp. 955–969.

[34] Tatsuo Shiozawa, Hirotsugu Kajihara, Tatsuro Endo, and Kazuhiro Hiwada. "Emerging Usage and Evaluation of Low Latency FLASH." In: *International Memory Workshop (IMW)*. 2020, pp. 1–4.

[35] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives." In: *Proc. IEEE* 105.9 (2017), pp. 1666–1704.

[36] Min Ye, Qiao Li, Yina Lv, Jie Zhang, Tianyu Ren, Daniel Wen, Tei-Wei Kuo, and Chun Jason Xue. "Achieving Near-Zero Read Retry for 3D NAND Flash Memory." In: *International Conference on Architectural Support for*

Programming Languages and Operating Systems, Volume 2. La Jolla, CA, USA: ACM, 2024, pp. 55–70.

[37] Minesh Patel, Taha Shahroodi, Aditya Manglik, Abdullah Giray Yaglikçi, Ataberk Olgun, Haocong Luo, and Onur Mutlu. "Rethinking the Producer-Consumer Relationship in Modern DRAM-Based Systems." In: *IEEE Access* 12 (2024), pp. 196207–196239. DOI: `10.1109/ACCESS.2024.3514377`.

[38] Ronglong Wu, Shuyue Zhou, Jiahao Lu, Zhirong Shen, Zikang Xu, Jiwu Shu, Kunlin Yang, Feilong Lin, and Yiming Zhang. "Removing Obstacles before Breaking Through the Memory Wall: A Close Look at HBM Errors in the Field." In: *USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, July 10-12*. 2024, pp. 851–867.

[39] Seung Soo Kim, Soo Kyeom Yong, Whayoung Kim, Sukin Kang, Hyeon Woo Park, Kyung Jean Yoon, Dong Sun Sheen, Seho Lee, and Cheol Seong Hwang. "Review of semiconductor flash memory devices for material and process issues." In: *Advanced Materials* 35.43 (2023), p. 2200659.

[40] *NAND Flash density race*. Accessed: 2025-06-25. 2024. URL: `https://www.trendforce.com/news/2024/06/27/news-outpacing-samsung-or-the-end-of-race-kioxia-eyes-1000-layer-nand-by-2027/`.

[41] The Overclocking Page. *The Impact of SLC Cache in Performance of an NVMe SSD: Benchmarks and Results*. Accessed: 2025-06-09. Aug. 2024. URL: `https://theoverclockingpage.com/2024/08/23/the-impact-of-slc-cache-in-performance-of-an-nvme-ssd-benchmarks-and-results/?lang=en`.

[42] Min Ye, Qiao Li, Yina Lv, Jie Zhang, Tianyu Ren, Daniel Wen, Tei-Wei Kuo, and Chun Jason Xue. "Achieving Near-Zero Read Retry for 3D NAND Flash Memory." In: *International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS), La Jolla, CA, USA, 27 April - 1 May 2024*, pp. 55–70.

[43] JEDEC Solid State Technology Association. *Failure Mechanisms and Models for Semiconductor Devices*. Tech. rep. JEP122H. Sept. 2016.

[44] Cristian Zambelli, Rino Micheloni, Luca Crippa, Lorenzo Zuolo, and Piero Olivo. "Impact of the NAND flash power supply on solid state drives reliability and performance." In: *IEEE Transactions on Device and Materials Reliability* 18.2 (2018), pp. 247–255.

[45] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling." In: *IEEE Micro* 32.3 (2012), pp. 122–134.

[46] Nayoung Choi and Jaeha Kim. "Modeling and Simulation of NAND Flash Memory Sensing Systems with Cell-to-Cell Vth Variations." In: *International Conference On Computer Aided Design (ICCAD)*. 2020.

[47] Fan Yang, Ajinkya Kale, Yury Bubnov, Leon Stein, Qiaosong Wang, M. Hadi Kiapour, and Robinson Piramuthu. "Visual Search at eBay." In: *SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pp. 2101–2110.

[48] Zhiheng Yue, Huizheng Wang, Jiahao Fang, Jinyi Deng, Guangyang Lu, Fengbin Tu, Ruiqi Guo, Yuxuan Li, Yubin Qin, Yang Wang, Chao Li, Huiming Han, Shaojun Wei, Yang Hu, and Shouyi Yin. "Exploiting Similarity Opportunities of Emerging Vision AI Models on Hybrid Bonding Architecture." In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA).* 2024, pp. 396–409.

[49] Jae-Woo Park, Doogon Kim, Sunghwa Ok, Jaebeom Park, Taeheui Kwon, Hyunsoo Lee, Sungmook Lim, Sun-Young Jung, Hyeongjin Choi, Taikyu Kang, Gwan Park, Chul-Woo Yang, Jeong-Gil Choi, Gwihan Ko, Jaehyeon Shin, Ingon Yang, Junghoon Nam, Hyeokchan Sohn, Seok-In Hong, Yohan Jeong, Sung-Wook Choi, Changwoon Choi, Hyun-Soo Shin, Junyoun Lim, Dongkyu Youn, Sanghyuk Nam, Juyeab Lee, Myungkyu Ahn, Hoseok Lee, Seungpil Lee, Jongmin Park, Kichang Gwon, Woopyo Jeong, Jungdal Choi, Jinkook Kim, and Kyo-Won Jin. "30.1 A 176-Stacked 512Gb 3b/Cell 3D-NAND Flash with 10.8Gb/mm2 Density with a Peripheral Circuit Under Cell Array Architecture." In: *IEEE International Solid-State Circuits Conference (ISSCC).* Vol. 64. 2021, pp. 422–423. DOI: 10.1109/ISSCC42613.2021.9365809.

[50] Dongwhee Kim, Jaeyoon Lee, Wonyeong Jung, Michael B. Sullivan, and Jungrae Kim. "Unity ECC: Unified Memory Protection Against Bit and Chip Errors." In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, November 12-17.* 2023, 48:1–48:16.

[51] Jin Hyun Kim, Yuhwan Ro, Jinin So, Sukhan Lee, Shin-haeng Kang, YeonGon Cho, Hyeonsu Kim, Byeongho Kim, Kyungsoo Kim, Sangsoo Park, Jin-Seong Kim, Sanghoon Cha, Won-Jo Lee, Jin Jung, Jong-Geon Lee, Jieun Lee, JoonHo Song, Seungwon Lee, Jeonghyeon Cho, Jaehoon Yu, and Kyomin Sohn. "Samsung PIM/PNM for Transfmer Based AI : Energy Efficiency on PIM/PNM Cluster." In: *2023 IEEE Hot Chips 35 Symposium (HCS).* 2023, pp. 1–31.

[52] Hüsrev Cilasun, Salonik Resch, Zamshed I. Chowdhury, Masoud Zabihi, Yang Lv, Brandon Zink, Jianping Wang, Sachin S. Sapatnekar, and Ulya R. Karpuzcu. "On Error Correction for Nonvolatile Processing-In-Memory." In: *International Symposium on Computer Architecture (ISCA), Buenos Aires, Argentina, June 29 - July 3.* 2024, pp. 678–692.

[53] Shehbaz Jaffer, Stathis Maneas, Andy A. Hwang, and Bianca Schroeder. "The Reliability of Modern File Systems in the face of SSD Errors." In: *ACM Trans. Storage* 16.1 (2020), 2:1–2:28.

[54] Inc. Facebook. *RocksDB: A Persistent Key-Value Store for Flash and RAM Storage.* Accessed: 2025-03-31. 2013. URL: https://github.com/facebook/rocksdb.

[55] Till Kolditz, Dirk Habich, Wolfgang Lehner, Matthias Werner, and Stefan T.J. de Bruijn. "AHEAD: Adaptable Data Hardening for On-the-Fly Hardware Error Detection during Database Query Processing." In: *International Conference on Management of Data.* Houston, TX, USA, 2018, pp. 1619–1634.

[56] Matthias Böhm, Wolfgang Lehner, and Christof Fetzer. "Resiliency-Aware Data Management." In: *Proc. VLDB Endow.* 4.12 (2011), pp. 1462–1465.

[57] Till Kolditz, Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. "Online bit flip detection for in-memory B-trees on unreliable hardware." In: *International Workshop on Data Management on New Hardware (DaMoN).* Snowbird, Utah, USA, 2014.

[58] Qiao Li, Min Ye, Yufei Cui, Liang Shi, Xiaoqiang Li, Tei-Wei Kuo, and Chun Jason Xue. "Shaving Retries with Sentinels for Fast Read over High-Density 3D Flash." In: *Annual International Symposium on Microarchitecture (MICRO).* 2020, pp. 483–495.

[59] Yun-Chih Chen, Chun-Feng Wu, Yuan-Hao Chang, and Tei-Wei Kuo. "Zone-Life: How to Utilize Data Lifetime Semantics to Make SSDs Smarter." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.8 (2023), pp. 2488–2499.

[60] NVM Express, Inc. *NVM Express 1.4 Specification.* Tech. rep. 1.4. Accessed: 2025-06-25. NVM Express, Inc., 2019. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[61] IBM Corporation. *System/360 Operating System Indexed Sequential Access Method (ISAM) Program Logic Manual.* IBM Corporation. Poughkeepsie, NY, USA, 1966.

[62] Anurag Acharya, Mustafa Uysal, and Joel H. Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998.* Ed. by Dileep Bhandarkar and Anant Agarwal. ACM Press, 1998, pp. 81–91. DOI: 10.1145/291069.291026.

[63] William H. Kautz. "Cellular Logic-in-Memory Arrays." In: *IEEE Trans. Computers* 18.8 (1969), pp. 719–727. DOI: 10.1109/T-C.1969.222754.

[64] SPDK Developers. *Storage Performance Development Kit (SPDK).* Accessed: 2025-06-19. 2025. URL: https://spdk.io/.

[65] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "WiscKey: Separating Keys from Values in SSD-Conscious Storage." In: *ACM Trans. Storage* 13.1 (2017), 5:1–5:28. DOI: 10.1145/3033273.

[66] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. "Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 2071–2086.

[67] Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. "Optimizing Distributed Joins with Bloom Filters." In: *Distributed Computing and Internet Technology, 5th International Conference, ICDCIT 2008, New Delhi, India, December 10-12, 2008. Proceedings.* Ed. by Manish Parashar and Sanjeev K. Aggarwal. Vol. 5375. Lecture Notes in Computer Science. Springer, 2008, pp. 145–156. DOI: 10.1007/978-3-540-89737-8_15.

[68] Maximilian Bandle, Jana Giceva, and Thomas Neumann. "To Partition, or Not to Partition, That is the Join Question in a Real System." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 168–180. DOI: 10.1145/3448016.3452831.

[69] Tim Zeyl, Qi Cheng, Reza Pournaghi, Jason Lam, Weicheng Wang, Calvin Wong, Chong Chen, and Per-Åke Larson. "Including Bloom Filters in Bottom-up Optimization." In: *Proceedings of the 2025 International Conference on Management of Data Companion (SIGMOD '25 Companion).* Berlin, Germany. 2025. DOI: 10.48550/ARXIV.2505.02994. arXiv: 2505.02994.

[70] Gaurav Gupta, Minghao Yan, Benjamin Coleman, Bryce Kille, RA Leo Elworth, Tharun Medini, Todd Treangen, and Anshumali Shrivastava. "Fast processing and querying of 170tb of genomics data via a repeated and merged bloom filter (rambo)." In: *Proceedings of the 2021 International Conference on Management of Data.* 2021, pp. 2226–2234.

[71] Nathaniel McVicar, Chih-Ching Lin, and Scott Hauck. "K-Mer Counting Using Bloom Filters with an FPGA-Attached HMC." In: *25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017.* IEEE Computer Society, 2017, pp. 203–210. DOI: 10.1109/FCCM.2017.23.

[72] Rafael P. Laufer, Pedro B. Velloso, and Otto Carlos Muniz Bandeira Duarte. "A Generalized Bloom Filter to Secure Distributed Network Applications." In: *Comput. Networks* 55.8 (2011), pp. 1804–1819.

[73] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. "Optimizing Bloom Filter: Challenges, Solutions, and Comparisons." In: *IEEE Commun. Surv. Tutorials* 21.2 (2019), pp. 1912–1949.

[74] Ashish Goel and Pankaj Gupta. "Small subset queries and bloom filters using ternary associative memories, with applications." In: *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, New York, USA, 14-18 June.* 2010, pp. 143–154.

[75] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. "Theory and Practice of Bloom Filters for Distributed Systems." In: *IEEE Communications Surveys & Tutorials* 14.1 (2012), pp. 131–155.

[76] Deke Guo, Yunhao Liu, Xiangyang Li, and Panlong Yang. "False Negative Problem of Counting Bloom Filter." In: *IEEE Transactions on Knowledge and Data Engineering* 22.5 (2010), pp. 651–664. DOI: 10.1109/TKDE.2009.209.

[77] Benoit Donnet, Bruno Baynat, and Timur Friedman. "Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives." In: *Conference on Emerging Network Experiment and Technology, CoNEXT, Lisboa, Portugal, Dec. 4-7.* 2006, p. 13.

[78] Apache Software Foundation. *Apache Parquet Format Specification.* Version 2.9.0, Accessed: 2025-05-31. 2013. URL: https://github.com/apache/parquet-format.

[79] Qiao Li, Liang Shi, Yufei Cui, and Chun Jason Xue. "Exploiting Asymmetric Errors for LDPC Decoding Optimization on 3D NAND Flash Memory." In: *IEEE Trans. Computers* 69.4 (2020), pp. 475–488.

[80] Open NAND Flash Interface Working Group. *ONFI 5.0 Specification.* Accessed: 2025-06-13. 2021. URL: https://onfi.org/specs.html.

[81] OEIS: The On-Line Encyclopedia of Integer Sequences. *Sequence A034839.* Accessed: 2025-06-21. URL: https://oeis.org/A034839.

[82] Jun Rao and Kenneth A Ross. "Making B+-trees cache conscious in main memory." In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data.* 2000, pp. 475–486.

[83] Erik Thordsen and Erich Schubert. "An Alternating Optimization Scheme for Binary Sketches for Cosine Similarity Search." In: *Similarity Search and Applications - 16th International Conference, SISAP 2023, A Coruña, Spain, October 9-11.* Vol. 14289. Lecture Notes in Computer Science. Springer, 2023, pp. 41–55.

[84] Yair Weiss, Antonio Torralba, and Robert Fergus. “Spectral Hashing.” In: *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*. Ed. by Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou. Curran Associates, Inc., 2008, pp. 1753–1760.

[85] DA Rachkovskij. “Index structures for fast similarity search for binary vectors.” In: *Cybernetics and Systems Analysis* 53 (2017), pp. 799–820.

[86] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondrej Certík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason Keith Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Stepán Roucka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony M. Scopatz. “SymPy: symbolic computing in Python.” In: *PeerJ Comput. Sci.* 3 (2017), e103. DOI: 10.7717/PEERJ-CS.103.

[87] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. “How to Architect a Query Compiler, Revisited.” In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 307–322. DOI: 10.1145/3183713.3196893.

[88] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware.” In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550. DOI: 10.14778/2002938.2002940.

[89] Goetz Graefe. “Volcano - An Extensible and Parallel Query Evaluation System.” In: *IEEE Trans. Knowl. Data Eng.* 6.1 (1994), pp. 120–135. DOI: 10.1109/69.273032.

[90] Albert Einstein. “Die Grundlage der allgemeinen Relativitätstheorie.” In: *Annalen der Physik* 49.7 (1916), pp. 769–822.

[91] Mark A Cusack, John Adamson, Mark Brinicombe, Neil A. Carson, Thomas Kejser, Jim Peterson, Arvind Vasudev, Kurt Westerfeld, and Robert Wipfel. “Yellowbrick: An Elastic Data Warehouse on Kubernetes.” In: *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024. URL: https://www.cidrdb.org/cidr2024/papers/p2-cusack.pdf.

[92] Compute Express Link Consortium, Inc. *CXL® Specification*. Accessed: 2025-06-18. 2025. URL: https://computeexpresslink.org/.

[93] Torsten Grust. “Monad comprehensions: a versatile representation for queries.” In: *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. Springer, 2004, pp. 288–311.

[94] Torsten Grust, Sherif Sakr, and Jens Teubner. "XQuery on SQL Hosts." In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer. Morgan Kaufmann, 2004, pp. 252–263. DOI: 10.1016/B978-012088469-8.50025-5.

[95] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. "Pathfinder: XQuery - The Relational Way." In: *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. Ed. by Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi. ACM, 2005, pp. 1322–1325. URL: http://www.vldb.org/conf/2005/papers/p1322-boncz.pdf.

[96] Erik Meijer, Brian Beckman, and Gavin M. Bierman. "LINQ: reconciling object, relations and XML in the .NET framework." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. Ed. by Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis. ACM, 2006, p. 706. DOI: 10.1145/1142473.1142552.

[97] Andrei Z Broder. "Identifying and filtering near-duplicate documents." In: *Annual symposium on combinatorial pattern matching*. Springer. 2000, pp. 1–10.

[98] Facebook MyRocks. *MyRocks Record Format*. GitHub wiki; Accessed 2025-06-18. 2019. URL: https://github.com/facebook/mysql-5.6/wiki/MyRocks-record-format.

[99] Kapil Vaidya, Tim Kraska, Subarna Chatterjee, Eric R. Knorr, Michael Mitzenmacher, and Stratos Idreos. "SNARF: A Learning-Enhanced Range Filter." In: *Proc. VLDB Endow.* 15.8 (2022), pp. 1632–1644.

[100] PostgreSQL. *64.5 BRIN Indexes*. Accessed 2025-06-20. Nov. 2014. URL: https://www.postgresql.org/docs/17/brin.html.

[101] R Aaij, J Albrecht, F Alessio, S Amato, E Aslanides, I Belyaev, M Van Beuzekom, E Bonaccorsi, R Bonnefoy, L Brarda, et al. "The LHCb trigger and its performance in 2011." In: *Journal of Instrumentation* 8.04 (2013), P04022.

[102] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. "The X-tree : An Index Structure for High-Dimensional Data." In: *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. Ed. by T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda. Morgan Kaufmann, 1996, pp. 28–39. URL: http://www.vldb.org/conf/1996/P028.PDF.

[103] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles.” In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990.* Ed. by Hector Garcia-Molina and H. V. Jagadish. ACM Press, 1990, pp. 322–331. DOI: 10.1145/93597.98741.

[104] Doron Rotem, Kurt Stockinger, and Kesheng Wu. “Efficient binning for bitmap indices on high-cardinality attributes.” In: (2004).

[105] Stefan Berchtold, Christian Böhm, Daniel A. Keim, Hans-Peter Kriegel, and Xiaowei Xu. “Optimal Multidimensional Query Processing Using Tree Striping.” In: *Data Warehousing and Knowledge Discovery, Second International Conference, DaWaK 2000, London, UK, September 4-6, 2000, Proceedings.* Ed. by Yahiko Kambayashi, Mukesh K. Mohania, and A Min Tjoa. Vol. 1874. Lecture Notes in Computer Science. Springer, 2000, pp. 244–257. DOI: 10.1007/3-540-44466-1_24.

[106] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. “Product Quantization for Nearest Neighbor Search.” In: *IEEE Trans. Pattern Anal. Mach. Intell.* 33.1 (2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57.

[107] Mann-May Yau and Sargur N. Srihari. “A Hierarchical Data Structure for Multidimensional Digital Images.” In: *Commun. ACM* 26.7 (1983), pp. 504–515. DOI: 10.1145/358150.358158.

[108] Hermann Tropf and Helmut Herzog. “Multimensional Range Search in Dynamically Balanced Trees.” In: *Angew. Inform.* 23.2 (1981), pp. 71–77.

[109] Daniel Lemire and Leonid Boytsov. “Decoding billions of integers per second through vectorization.” In: *Softw. Pract. Exp.* 45.1 (2015), pp. 1–29. DOI: 10.1002/SPE.2203.

[110] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. “Better bitmap performance with Roaring bitmaps.” In: *Softw. Pract. Exp.* 46.5 (2016), pp. 709–719. DOI: 10.1002/SPE.2325.

[111] Ricardo Baeza-Yates and Alejandro Salinger. “Fast Intersection Algorithms for Sorted Sequences.” In: *Algorithms and Applications, Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday.* Ed. by Tapio Elomaa, Heikki Mannila, and Pekka Orponen. Vol. 6060. Lecture Notes in Computer Science. Springer, 2010, pp. 45–61. DOI: 10.1007/978-3-642-12476-1.

[112] J. Shane Culpepper and Alistair Moffat. “Efficient set intersection for inverted indexing.” In: *ACM Trans. Inf. Syst.* 29.1 (2010), 1:1–1:25. DOI: 10.1145/1877766.1877767.

[113] Sunghwan Kim, Taesung Lee, Seung-won Hwang, and Sameh Elnikety. “List Intersection for Web Search: Algorithms, Cost Models, and Optimizations.” In: *Proc. VLDB Endow.* 12.1 (2018), pp. 1–13. DOI: 10.14778/3275536.3275537.

[114] Charles Masson, Jee E. Rim, and Homin K. Lee. "DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees." In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2195–2205. DOI: 10.14778/3352063.3352135.

[115] Roger Weber, Hans-Jörg Schek, and Stephen Blott. "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces." In: *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Ed. by Ashish Gupta, Oded Shmueli, and Jennifer Widom. Morgan Kaufmann, 1998, pp. 194–205. URL: http://www.vldb.org/conf/1998/p194.pdf.

[116] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. "An Experimental Study of Bitmap Compression vs. Inverted List Compression." In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu. ACM, 2017, pp. 993–1008. DOI: 10.1145/3035918.3064007.

[117] Yann Collet and Murray S. Kucherawy. "Zstandard Compression and the 'application/zstd' Media Type." In: *RFC* 8878 (2021), pp. 1–45. DOI: 10.17487/RFC8878.

[118] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval.* Cambridge University Press, 2008. DOI: 10.1017/CBO9780511809071.

[119] Jens Axboe. *liburing: io_uring library for Linux.* Accessed: 2025-05-31. 2025. URL: https://github.com/axboe/liburing.

[120] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System." In: *IEEE Trans. Parallel Distributed Syst.* 33.6 (2022), pp. 1303–1320. DOI: 10.1109/TPDS.2021.3104255.

[121] Jeff Dean, Sanjay Ghemawat, and Steinar H. Gunderson. *Snappy: A Fast Compressor/Decompressor.* Accessed: 2025-05-31. 2011. URL: https://github.com/google/snappy.

[122] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994. URL: https://www-cs-faculty.stanford.edu//~knuth/gkp.html.

[123] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.* Vol. 4A. The Art of Computer Programming. Boston, MA: Addison–Wesley Professional, 2011.

[124] OpenAI. *ChatGPT.* Large language model [AI tool]. Accessed: 2025-06-17. June 2025. URL: https://chat.openai.com/chat.

[125] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. "Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation." In: *Proc. VLDB Endow.* 15.4 (2021), pp. 752–765. DOI: 10.14778/3503585.3503586.

[126] Uriel Feige, Vahab S. Mirrokni, and Jan Vondrák. "Maximizing Non-monotone Submodular Functions." In: *SIAM J. Comput.* 40.4 (2011), pp. 1133–1153. DOI: 10.1137/090779346.

[127] Edo Liberty and Maxim Sviridenko. "Greedy Minimization of Weakly Supermodular Set Functions." In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*. Ed. by Klaus Jansen, José D. P. Rolim, David Williamson, and Santosh S. Vempala. Vol. 81. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 19:1–19:11. DOI: 10.4230/LIPICS.APPROX-RANDOM.2017.19.

[128] Wouter Kool, Herke van Hoof, and Max Welling. "Stochastic Beams and Where To Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement." In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 3499–3508. URL: http://proceedings.mlr.press/v97/kool19a.html.

[129] Pavlos S. Efraimidis and Paul G. Spirakis. "Weighted random sampling with a reservoir." In: *Inf. Process. Lett.* 97.5 (2006), pp. 181–185. DOI: 10.1016/J.IPL.2005.11.003.

[130] SciPy community. *SciPy: Scientific Computing Tools for Python.* Function scipy.optimize.lsq_linear, accessed 2025-05-16. 2024. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.lsq_linear.html.

[131] Steven Diamond and Stephen P. Boyd. "CVXPY: A Python-Embedded Modeling Language for Convex Optimization." In: *J. Mach. Learn. Res.* 17 (2016), 83:1–83:5. URL: https://jmlr.org/papers/v17/15-408.html.