Foundations and Trends® in Databases
Vol. XX, No. XX (2020) 1–91

DOI: 10.1561/XXXXXXXXXX

now
the essence of knowledge

Principles, Tradeoffs, and Opportunities in Data Access Method Design

Manos Athanassoulis
Boston University
mathan@bu.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

Dennis Shasha
New York University
shasha@cs.nyu.edu

Contents

1 Introduction **2**
- 1.1 Access Methods Basics . . . 2
- 1.2 Tradeoffs in Access Method Designs . . . 3
- 1.3 From Read/Update to RUM: Memory & Space Costs . . . 7
- 1.4 RUM Performance Tradeoffs . . . 8
- 1.5 From RUM to PyRUMID . . . 10
- 1.6 Contributions: Design Space and Classification . . . 10
- 1.7 Related Work on Design Abstractions for Access Methods . 11

2 Performance Tradeoffs & Access Patterns **14**
- 2.1 The PyRUMID Tradeoff Space . . . 16

3 Design Principles: Dimensions of a Design Space **20**
- 3.1 Global Data Organization . . . 21
  - 3.1.1 No Organization . . . 21
  - 3.1.2 Temporal Organization . . . 22
  - 3.1.3 Range Partitioning . . . 23
  - 3.1.4 Sorted . . . 24
  - 3.1.5 Hash Partitioning . . . 25
  - 3.1.6 Radix Partitioning . . . 26
  - 3.1.7 Temporal Partitioning . . . 27

3.2 Search without Indexing 27
3.2.1 Full Scan 27
3.2.2 Binary Search 28
3.2.3 Direct Addressing 28
3.2.4 Data-Driven Search 29
3.3 Search with Indexing 29
3.3.1 Full Scan → Sparse Indexing 30
3.3.2 Binary Search → Binary, K-Ary Trees, B$^+$-Trees . . 31
3.3.3 Direct Addressing → Radix Trees, Hash Indexes . . 31
3.3.4 Data-Driven Search → Learned Indexes 32
3.3.5 Achieving Robust Performance with Search Trees . 33
3.4 Local-Global Hybrid Data Organization 34
3.4.1 Range - Sorted (Traditional B$^+$-Trees) 35
3.4.2 Range - Range (Fractal B$^+$-Trees) 35
3.4.3 Range - Temporal (Insert-Optimized B$^+$-Trees) . . 36
3.4.4 Range - Range - ... - Range (Bulk-Loaded B$^+$-Trees, Sorted, Adaptive Indexing, Cracking) 36
3.4.5 Range - Hash (Bounded Disorder) 36
3.4.6 Radix - Sorted (Radix Trees) 37
3.4.7 Hash - Temporal (Chained Hash Table) 37
3.4.8 Temporal - Sorted (LSM Trees) 37
3.4.9 Temporal - Radix/Hash (LSM Tries) 38
3.5 Update Policy: In-place vs. Out-of-place 38
3.5.1 In-place Updates 39
3.5.2 Deferred In-place Updates 39
3.5.3 Out-of-place Updates 40
3.5.4 Differential Out-of-place Updates 41
3.6 Buffering: Batching Requests 41
3.6.1 Buffering Reads 42
3.6.2 Cache Pinning as a Special Case of Read Buffering 43
3.6.3 Buffering Updates 43
3.6.4 Buffering Updates Locally 44
3.7 Contents Representation 44
3.7.1 Key-Record 44
3.7.2 Key-Row ID 45

3.7.3 Key-Pointer . . . 45
3.7.4 Key-Bitvector . . . 45
3.8 Adaptivity . . . 46
3.8.1 Adaptive Sorting . . . 46
3.8.2 Adaptive Vertical Partitioning . . . 48
3.8.3 Adaptive Update Merging . . . 48
3.9 Discussion . . . 48

4 Mapping Access Methods to the Design Space 50
4.1 Cost Model . . . 51
4.2 Reducing Read Amplification: Data Layouts & Scans . . . 51
4.2.1 Physical Design . . . 51
4.2.2 Adaptive Physical Design . . . 51
4.2.3 Membership Test Indexes . . . 54
4.3 Efficient Random Access: B^+-Trees and Variants . . . 54
4.3.1 B^+-Tree Designs . . . 55
4.4 Reducing Write Amplification: LSM Trees and Variants . . . 55
4.4.1 Leveled LSM Designs . . . 56
4.4.2 Tiered LSM Designs . . . 56
4.4.3 Hybrid LSM Designs . . . 56
4.4.4 Hash-based LSM Designs . . . 56
4.5 Balance Space and Read: Tries and Variants . . . 56
4.5.1 Trie Index Designs . . . 57
4.6 Reduce Space Amplification: Bitmap Indexing . . . 57
4.6.1 Bitmap Index Designs . . . 57
4.7 Making Access Methods Hardware-Friendly . . . 57
4.8 Other Access Methods . . . 57

5 Design Opportunities 60
5.1 The Design Space As A Design Advisor . . . 61
5.2 A Richer Design Space . . . 61
5.2.1 Concurrency . . . 61
5.2.2 Distributed Systems . . . 63
5.2.3 Privacy . . . 63
5.2.4 Caching: Using Space for Cheaper Reads . . . 63
5.2.5 Hardware-Oblivious Designs . . . 63

6 Summary **64**

References **65**

Appendices **87**

Dennis Suggestion **88**

.0.1 Inter-Block Organizations 89

.0.2 Time Measurement for Block Organization 89

.0.3 Organization within leaf nodes 91

Abstract

Access methods to support efficient search and modification are at the core of any data-driven system. Designing *access methods* has required a continuous effort to adapt to changing workload requirements and underlying hardware. In this article we outline the shared principles and *design dimensions* of access methods that facilitate efficient access to data residing at various levels of the storage hierarchy from durable storage (spinning disks, solid state disks, other non-volatile memories) to random access memory to caches (and registers), and we discuss the impact of using each with respect to the latency of different forms of searches and modifications, as well as space. We *classify* access methods based on both the design dimensions they use and the goal for which they are optimized for (reads, updates, or space). Finally, we discuss new *design opportunities* for building access methods that become possible because of the systematization of the *access method design space*.

now Publishers Inc.. *Principles, Tradeoffs, and Opportunities in Data Access Method Design.* Foundations and Trends® in Databases, vol. XX, no. XX, pp. 1–91, 2020.

DOI: 10.1561/XXXXXXXXXX.

1

Introduction

1.1 Access Methods Basics

Access Methods are the means by which executing programs store and obtain data. As such, access methods sit at the heart of all data-intensive computer systems. An access method consists of (1) the data, physically stored in some layout, (2) optional metadata to facilitate navigation over the data, and (3) algorithms to support storage and retrieval operations [106, 208, 119]. Other terms used in the literature for access methods include "data structures" and "data containers". This article uses the term *access method* to underscore the interplay between the design of a data storage component with the way data is accessed.

Data systems, operating systems, file systems, compilers, and network systems employ a diverse set of access methods. Our discussion in this paper draws examples primarily from the area of data systems in the sense that we consider secondary memory, but the core analysis and categorization we present applies generically to any application requiring the storage and retrieval of key-value pairs.

Problem Definition. An access method manages a collection of *key-*

value pairs. The value itself may have semantics ranging from a pointer or a reference in the base data collection, e.g., a row id in a relational system, a reference to a large object such as an image or video in a key-value store, or an arbitrary set of values that the application knows how to parse and use in a NoSQL system.

Under this general definition, an access method design can, for example, be used to describe the design of (i) metadata indexes used in file, network, and operating systems, (ii) base data layouts and indexes in relational systems [106], (iii) NoSQL and NewSQL data layout designs [174], and (iv) general data structures that assume two levels of storage with different performance and capacity, modeling the memory-disk or cache-main memory pairs.

Each application has a sweet spot with regards to the different operations it needs to perform over its data in terms of writes to inject new data, and reads to retrieve data. In addition, the amount of memory and persistent storage required (and can be afforded) are also critical parameters that shape the requirements of a given application. In this way, each access method design targets the requirements of a given application. For example, data management systems use access methods as the entry point in query processing, i.e., utilizing various forms of tree-based and hash-based indexes and various base data layouts such as column-oriented and row-oriented as well as hybrids. File systems manage file metadata and file contents using access methods optimized for frequent updates. Compilers typically use hash maps for managing variables during their life span, and abstract syntax trees to capture the overall shape of a program. Similarly, network devices have heavily specialized access method needs to efficiently store and access routing tables.

1.2 Tradeoffs in Access Method Designs

There is no perfect access method [120]. Every access method represents a particular performance tradeoff. For example, the more structure a data set has (either in the form of metadata or within the main data), the easier it is to search it, but the harder it is to insert or update

because the structure needs to be maintained (by re-organizing data or by updating metadata).

Hence, new designs arise that offer a new balance between read performance, update performance, and the memory and storage footprint. This happens either by optimizing an existing balance, that is, using engineering and sometimes algorithmic effort to improve performance, or by tilting it, that is, improving one aspect at the expense of another.

As applications and hardware evolve, they require the invention of new data structures. For example, the database community generalized the 2-3 tree to the B-tree [33] and then the B^+-Trees [89, 106] with its many variants. Over the course of more than four decades of relational data systems [17, 62], evolution of such systems is closely connected with the the development of new access methods. The same is true for NoSQL systems which heavily rely on specialized classes of data structures such as LSM-trees for data systems that support write-heavy applications [161, 182], B-trees for systems that target more read-optimized applications [179, 229], and hashing based systems that are the core of systems that support very write-heavy applications [54]. Similarly, Bloom filters [43] came from the networking community [48], and, today Bloom filters are used throughout computer science.

Overall, there are two drivers for the creation of access methods: (i) **new workloads and access patterns** dictate specialized designs, and (ii) **advances in hardware** (multi-cores, processors, caches, main memory, storage devices) impose new performance and cost tradeoffs. We discuss these two points in more detail below.

Workload-Driven Designs. As an example from data systems, the rise of analytical applications as of the early 2000s in which relational tables tended to be wide but only a few columns would be accessed favored a columnar layout [64] (as had been present in vector languages like APL [74] for some time) which eventually led to the development of a new column-oriented data system architecture, also called column-stores [1, 45, 76, 77, 82, 83, 113, 128, 140, 218, 247]. The column-store design works better for long analytical queries on few columns, while the row-store design for short selective queries that require most fields of each row. Thus, using a columnar instead of a row-oriented layout in

data management systems is effectively a new *access method* decision [30, 31, 137, 142, 143, 144, 145, 193, 194, 195]. Further, various hybrid approaches offer benefits from both worlds of row-stores and column-stores: (i) by nesting columnar data organization within data pages [5, 6], and (ii) by grouping multiple columns and offering specialized code for accessing groups of columns [9, 72, 76, 77, 97, 128].

Access methods have been invented that adapt according to their workload. For example, Database Cracking [114, 115, 116] exploits the read patterns to organize data accordingly, Bw-Tree [150] enables delta updates to minimize the updating cost, E-Tree [162] alternates between read-optimized tree nodes and write-optimized tree nodes based on the read/write pattern, and B^+-Trees [89] can be tuned with variable node sizes, and thus, fanout.

The rich ecosystem of new non-relational data management systems, typically categorized as NoSQL or newSQL [174], often employ a log-structured merge tree (LSM-Tree) [161, 182] design that amortizes the update cost by storing incoming data in immutable sorted files. LSM-based NoSQL designs also face access tradeoffs. For example, a different memory allocation strategy allows for a better read vs. update performance tradeoff [67, 68], and lazy and eager merging of sorted runs can optimize for write-intensive or read-intensive workloads respectively [136, 161]. Another approach, called lazy leveling, performs lazy merging throughout the tree, except from the last level and uses additional memory to balance the read costs when needed [69]. Using hashing instead of sorting can efficiently support workload with point access and no range queries [12, 29, 54, 70, 71, 211]. These are only a few examples of access method designs that are strongly tied to either expected or temporary workload patterns.

Memory-Driven And Storage-Driven Designs. As a complement to workload-based considerations, hardware changes create new challenges, needs, and opportunities in access method design. In the last decade, the *memory and storage hierarchy* has been enriched with devices such as solid-state disks, non-volatile memories, and deep cache hierarchies. In a storage hierarchy, the lower levels offer a lot of storage at low price but at high access latency, and as we move higher, that

is, closer to the processor, the storage is faster but smaller and more expensive per byte. In the storage/memory hierarchy there is always a level that is the bottleneck for a given application, which depends on the relative size of the data in question and the sizes of the levels of the hierarchy.

In data management systems, early access methods like B^+-Trees were optimized for disk accesses [89]. As the memory sizes grew, however, the bottleneck quickly moved higher to the main memory and non-volatile memory. This changed the tradeoffs dramatically. In particular, a key hardware trend has been the growing disparity between the CPU speed and the speed of off-chip memory, termed "the memory wall" [236]. Since the early 2000s, operating systems [171] and data management systems [124] have been carefully re-designed to account for the memory wall by optimizing for the increasing number of cache memories [5, 7, 44, 45, 59, 139, 164, 165, 166, 200, 243].

Additionally, secondary storage is already at a crossover point. Traditional hard disks have hit their physical limits [18], and new storage technologies like shingled disks and flash are now addressing this performance stagnation [109]. Shingled disks increase the density of storage on the magnetic medium, changing the nature of disks because the granularity of reads and writes is now different [109]. Flash-based drives offer significantly faster read performance than traditional disks, but may suffer from relatively poor write performance. Because flash-based drives are equipped with a complex firmware called Flash Translation Layer (FTL) which, when updated, can lead to drastic changes in performance. Thus, flash hardware performance changes both when hardware changes and when firmware gets updated, which may create the need to change access methods to exploit those changes.

New system and access method designs aim to exploit what the new hardware has to offer and hide its weaknesses [19, 21, 22, 70, 122, 127, 133, 151, 153, 176, 177, 199, 221]. For example, because writing certain kinds of flash devices requires writing full blocks at a time, access methods append incoming data without sorting until a flash erase block is full, at which point data is being organized in memory and re-written as one block. This minimizes write overhead, sacrificing

read performance (contents of a block are frequently not sorted) for write efficiency [4, 20, 24, 25, 61].

1.3 From Read/Update to RUM: Memory & Space Costs

Read vs. Update Captures Workload. In order to compare data structures and decide which one to use and under which conditions, we first need to define the appropriate metrics. The most common metrics quantify the tradeoff of each design between the read performance and update performance [47, 239, 240].

Read vs. Update vs. Memory Captures Storage. The common denominator of the lines of work that consider read and update performance as metrics is that they assume that disk capacity is cheap or even free in the ideal setting. This assumption comes from the time that disk was used as secondary storage and was so much cheaper than memory that the storage cost was considered insignificant and the main consideration was storage performance [96]. Since then, the storage hierarchy has been augmented with various devices including solid-state disks (SSDs), shingled magnetic recording disks (SMR), non-volatiles memories (NVMs) and other devices.

The new storage media offer tradeoffs: more expensive per byte and fast, or cheaper but slower performance [18]. Sometimes the higher performance comes at significant energy cost [212]. In addition, the data generation trends typically outpace the rate at which storage devices are delivered leading to a data-capacity storage gap [42, 108, 216].

Overall, the increasing use of storage with more expensive capacity and efficient random access has made the *memory vs. performance* (read/update cost) analysis an important factor of design and optimization of access methods [27, 73, 241]. The wildly different cost of secondary storage among the different storage technologies discussed above make space utilization and storage cost important factors of access method design.

Storage capacity cannot be considered abundant, and efficient access to storage is not cheap, hence the storage space and cost should also be included when judging the efficiency of an access method.

1.4 RUM Performance Tradeoffs

Now we outline the interplay between read performance, update performance, and space utilization, and how we can use them as a guide to design access methods and compare alternative designs. We define the following three quantities.

1. The **R**ead overhead, that is, the read amplification of every lookup operation (the ratio between the size of the total data and metadata read, and the actual size of the requested data).

2. The **U**pdate overhead, that is, the write amplification of every update operation (the ratio between the size of the data and metadata that were updated, and the exact size of the updated data).

3. The **M**emory (or storage) overhead, that is, the space amplification of the employed data structures (the ratio between the aggregate size of data, metadata, and lost space due to fragmentation, divided by the size of the base data alone).

These three overheads, (*RUM overheads* for short) form a three-way tradeoff space [27], extending the read vs. update overhead [47, 239, 240] and the read vs. memory overhead [103, 104, 227] analysis. The RUM tradeoffs manifest in every access method design. Figure 1.1 conceptually shows a broad classification of access method designs in the RUM tradeoff space.

A scan access method – even in a main-memory read-optimized column-store where it consists of an array of dense values for each attribute [1] – has high read cost, as it requires traversing the whole column even when the useful data is a small portion of the column. Hence, the read amplification is large, while the write amplification and the space amplification are minimal, since we can simply append new data and there is no need to maintain metadata.

On the other hand, a typical tree-structured index (like B^+-Trees) uses additional space for the index metadata to mitigate both the read and the update overhead. The non-leaf index nodes used to navigate

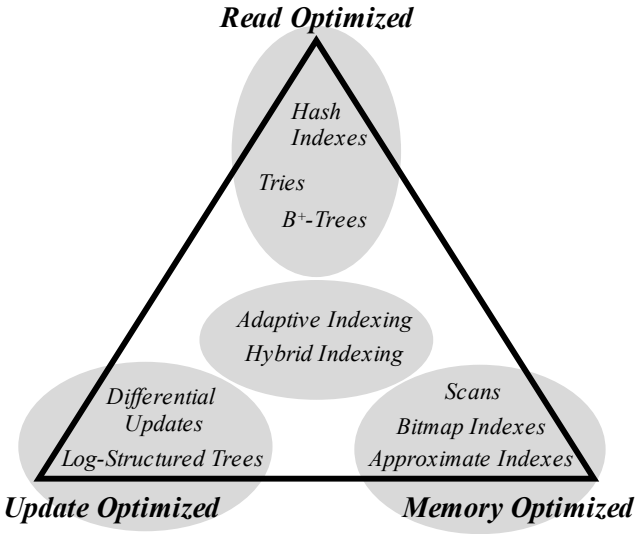

Figure 1.1: The RUM tradeoff space and a broad classification of access method designs according to the RUM balance they maintain.

to the desired partition of the data helps to read only the leaf which houses the desired data. In addition, the free slots available in a B^+-Tree (leading to an average fill factor of 67% [192]) increase the space consumed to guarantee that most inserts will update only one page and not a logarithmic number of pages (i.e., the entire path from root to leaf), effectively trading off space amplification for update amplification. For example, for the average case fill factor of 67% the overall tree size is increased by 50% compared to storing the data without free space. This factor of 50% has a profound impact on space utilization, and in practice such large constants dictate the access method to be used depending on the application constraints and hardware costs. Figure 1.1 shows similar design tradeoffs, which are common in access method designs [26, 27].

1.5 From RUM to PyRUMID

The RUM metrics help us quickly classify broad design classes of data structures. To get a more complete view, we need to go one level deeper and break down these metrics further. For example, read performance depends on the exact access pattern. A ***P****oint query* asking for a single item differs from a ***R****ange query* asking for a set of items based on a key interval. In fact, the optimal data structure designs for these two query types are entirely different. In the first case, we need a data structure that leads directly to a single page where the target item resides (e.g., a hash-based structure), while in the latter case we need a design that has a notion of ordering across pages of data such that it allows queries to select among those pages and traverse them in order (e.g., such as an ordered tree).

Similarly, for writes, an ***I****nsert* query that needs to insert a new value has different requirements than an ***U****pdate* or ***D****elete* query that needs to update or drop an existing value. In the first case, having a log-like structure is enough, e.g., appending new data all the time, while in the latter case we need the ability to quickly locate the target key-value pair. Overall, this makes for a complex tradeoff, which we call *PyRUMID overheads*, that is more demanding to reason about. Note that ***M*** stands for memory (or space).

1.6 Contributions: Design Space and Classification

Workload and technology trends as well as the various tradeoffs that manifest between access performance and metadata size create a complex problem space with respect to access method design. Understanding this space requires a systematic classification of ideas, tools, and concepts used to design access methods.

This classification, in turn, will enable us to answer frequent questions that come up when building a system. Which access method efficiently supports a given workload? What is the effect of adding a new design component to an access method and workload? Is it beneficial to get more or faster memory for a system? These are only a small sample of the vast set of design, research, and practical questions that

researchers and practitioners face on a daily basis.

In this survey we study how the low level fundamental design choices in data structures lead the final designs to occupy a particular balance in the overall PyRUMID tradeoff space. We work on classifying those design choices based on their effect on overall performance and provide the means for engineers and designers to reason about their final designs based on the design decisions.

Design Dimensions. To build the design space we survey numerous fundamental access method designs and distill them to come up with a number of *independent design dimensions* that identify a design. This small set of design dimensions can be used to describe any access method and to explain its behavior and properties:

- Data Organization
- Content representation
- Metadata for searching
- Buffering policy
- Update policy
- Adaptivity

Design Space. We use the above design dimensions to propose an *access method design space*. Each access method is a "point" in this space and designs that correspond to *families of access methods* cover a sub-space of this design space. We (i) describe the motivation behind each design dimension and its available options, (ii) characterize existing access methods as points in this space, and (iii) we study the impact of combining these design elements to create new access methods.

For every design dimension, and for every access method design we discuss its *performance tradeoffs* on the *PyRUMID overheads*.

1.7 Related Work on Design Abstractions for Access Methods

Early research on transforming static search data structures to dynamic ones (i.e., to support updates) led to the initial analysis of the read vs. update performance tradeoff [223, 224].

Prior Efforts on Classifying Access Methods. Graefe surveyed

different techniques used in modern B-Tree design [89]. Lehman et al. surveyed index structures for main memory over thirty years ago [148]. The classification proposed in this paper brings the new element of grouping existing designs under a small set of design dimensions which allows for a more structured way of arguing about design.

The goal of this paper is to propose a small expressive set of design dimensions, which when accordingly combined, can describe the huge space of access methods.

Design Abstractions. Hellerstein et al. [105] proposed the generalized search trees (GiST) as a new data structure that can easily be extended to operate like a B^+-Tree [89], an R-Tree [98, 167], an RD-Tree [105], or a variety of other variations of tree indexes (like partial sum trees [230], k-D-B-Trees [198], Ch-trees [131], hB-trees [158], V-trees [169], and TV-trees [154]). The core idea of GiST is that it has a general structure, a number of invariants, a number of methods that can be applied on the indexed keys, and a number of methods that are applied on the tree (upon searching, inserting, and deleting). The goal of GiST is to offer a single API that can implement different search trees, differing based on their applications and data domains. In spite of these differences, all these trees maintain the fundamental tree properties of a logarithmic cost of reading and updating. In this survey we seek to find the common design concepts among various access methods including – but not limited to – search trees.

Performance Abstractions. The read vs. update tradeoff [47, 239, 240] has been thoroughly studied as a follow-up on the initial work on *converting a static search structure to a dynamic one* [40, 184, 224]. Similar abstractions for trading *preprocessing time* off against *query time* and adding range variables have also been discussed [39, 203]. This work has two fundamental axis: the *decomposability* of search structures (which was inspired by the divide and conquer approach in algorithm design), and the *dynamization* of a static search structure [223]. The concept of decomposability naturally leads to the notion of partitioning as a key decision of access methods, and, inherently captures a read vs. update cost by suggesting that smaller blocks lead to better update

performance and worse read performance (and vice versa), assuming that an update has to physically replace a whole block.

This survey builds on our research efforts in classifying access methods [26, 119] and building new detailed abstractions [27, 112, 120], and expands on this line of work by discussing the different ways to offer efficient updates (dynamization) and how each technique affects each aspect of the PyRUMID performance tradeoff, both qualitatively and quantitatively in terms of exact or asymptotic behavior.

In the remainder of this paper, we first discuss in detail the performance tradeoffs with respect to access patterns, then we present the design dimensions one by one, and we give examples of access method designs and how they affect the performance tradeoffs (Table 4.1). Finally, we discuss open research questions with respect to both the design space and the individual designs themselves.

2

Performance Tradeoffs & Access Patterns

In this section we provide context and background that is needed for the rest of the paper. We review the key-value access method model, the memory/storage hierarchy and why it is essential for access method design, and we define in detail the core performance tradeoffs.

The Key-Value Model. In the key value model, arbitrary data values can be associated with unique keys. This is powerful because in principle we can store anything as a value. For example, this is how NoSQL data systems store arbitrary data as long as the higher level application knows how to parse the values. In this way, the key value model can be used to represent much more complex models including the relational model where each value effectively would be a whole row of a relational table and the key for each value is a concatenation of the database name, the table name, and the row ID. In addition, the access methods used within complex systems follow the key-valve model. For example, this is true for *database indexing*, like B^+-Trees, hash indexes, and bitmap indexes.

Memory Hierarchy. Access methods live across the memory hierarchy. That is, a sequence of memory devices that complement each other in terms of how fast we can read and write data from each one. At least

one level of the memory hierarchy is typically persistent so the data is not lost when power is down or the program terminates. Also one level of the memory hierarchy is large enough to hold all the data needed on a particular machine. Data is transferred back and forth across levels as requests for reads arrive or new data is inserted and old data is deleted or updated.

Because different types of memory (e.g. cache, RAM, solid state disks, etc.) work at vastly different speeds, when data is moved from a very slow memory level such as a hard disk, then the overall cost of using a particular access method is dominated by this cost. The cost of all other data movement at higher levels of the memory hierarchy is negligible. Similarly, due to the ever increasing performance gap between CPUs and data movement, CPU cost is negligible compared to the cost of bringing data from slow memory. For data-intensive applications, typically, the CPU will be using fewer cycles than its capacity due to waiting for data from disk or even random access memory.

Our modeling and design discussions in this paper take these performance properties into account and always assume two abstract levels of memory hierarchy as in the input-output (I/O) model [3]: one that can be treated as having essentially infinite capacity but constitutes the performance bottleneck, and one that is much faster but with limited capacity. This approach captures the *memory* and *disk* pair, but it can also capture any two levels of memory that have significant difference in access latency and cost (e.g., 1-3 orders of magnitude). In this way, in our cost models throughout this paper we measure and define performance with respect to the number of data blocks moved by an access method operation.

Overall, this performance metric holds true for the majority of operations in the key value-model and typical analytics. For example, this is true for file systems, network routers, NoSQL systems and SQL database systems. When designing access methods purely for an in memory environment or for an application with very high computational costs, e.g., due to a lot of iterative computations, then for optimal performance the design also needs to take into account computational costs as well as costs of moving data across more levels of the mem-

ory hierarchy. This is an emerging field of study that requires learned models combined with traditional cost models [120]. In addition as we move to non-volatile memories with wider buses and lack of mechanical constraints in secondary storage, the cell probe model [238] might be necessary.

2.1 The PyRUMID Tradeoff Space

We now define in detail the different tradeoffs we use to characterize the performance of an access method.

Read Cost. The read cost is defined as the *total data blocks* a query needs to move from disk (we will use disk as a shorthand for slow but infinite capacity storage) to random access memory. This includes the actual data blocks that the query needs, as well as any additional data blocks that we need to move and be read because of the design of the access method. For example, if the design of the access method includes a navigation portion/indexing, then we first read parts of the index which helps reduce the number of data blocks that we need to read. Similarly, if the design of the access method does not give direct access to the target data blocks, then we likely first need to read additional data blocks so we can filter the target values. All those components are included in the total read cost.

A central factor in determining the most suitable access method is to identify the *exact access patterns*, that is, the different part of the underlying data collection and auxiliary data that are accessed or modified during the execution of a workload. Different devices have different performance profiles for different access patterns. The workload pattern can suggest which access method design (or combination of designs) to use.

- **Point Query**: Read a specific object based on the key associated with this object. Point queries can use a number of auxiliary structures to enhance their performance. Membership tests (like Bloom filters), tree indexing, trie indexing are only some examples of data structures that can enhance point query performance. In addition, the exact data organization (sorting, partitioning) may

also affect the efficiency of a point query.

- **Range Query**: Read a collection of objects that can be found using a range of values on the key. Range queries can be characterized either as short range queries, long range queries, or full scans.

  - *Short range queries* may lead to similar behavior as point queries if the data is sorted, because accessing data requires reading the minimum granularity of data storage (e.g., a data page). Futhermore, sparse indexing optimization like zonemaps or column imprints can enhance the performance of short range queries. By contrast, a short range query cannot benefit from optimizations for point queries (like Bloom filters).

  - *Long range* queries need several data blocks and the exact organization of data and the efficiency of locating useful data matters. Optimizations like zonemaps and column imprints can enhance their performance. A critical factor that characterizes range queries is the selectivity, that is the fraction of the data blocks requested with respect to the total number of data blocks.

  - *Full scans* are effectively not affected by indexing. Instead they are affected only by the design choices for laying out the base data. In addition, performance can be enhanced by low-level engineering effort to consume the data in the fastest possible way, i.e., at the bandwidth of the level of the memory hierarchy upon which the data resides (disk bandwidth for disk-resident data, and memory bus bandwidth for memory-resident data).

Update Cost. When a new object is inserted or an existing object is updated, the update cost is given by the total number of blocks on slow storage (e.g., disks) that need to be read and written for the update operation to be persistent. Contrary to read cost, write cost includes

both data blocks that need to be read and written because for certain update operations we first need to locate the target data on the existing data set, and read this data before we can update it. Thus writes, are typically more expensive than reads in terms of total number of blocks. In addition, physical writes also consume energy and, when the writes take place to a non-volatile memory, they affect the durability of the device. Hence, by limiting the write cost of an operation, the benefit is multi-faceted: performance, energy, and durability.

Similarly to read cost, update cost can be broken down to individual costs with respect to the exact access pattern. Thus, again, different access patterns require different access method designs for optimal performance.

- **Insert** a new key-value pair. An issue with inserts is that it matters how the new key value pair is inserted, in order to not disrupt the existing data organization, which might in turn influence the performance. Another issue is that insert performance – or *ingestion rate* – can be enhanced by techniques that allow accumulating key-value pairs out-of-place and then including them into the core structure.

- **Delete** a key-value pair. Deleting a key may create some form of fragmentation, which in turn, warrants future work in order to reclaim the space occupied by deleted data. This delayed deletion may cause additional space amplification and also privacy challenges if the invalid data is not overwritten and kept on the device for long periods of time.

- **Update** the key of a key-value pair. An issue with updates is that the key-value pair might have to be moved to facilitate the data organization. An update can also be viewed under some conditions as a delete followed by an insert. The treatment of updates affects the efficiency of reads and potentially that of future inserts, deletes, or updates.

Memory Utilization. As storage becomes a more expensive resource, e.g., by moving to more expensive non-volatile memories and to the

cloud, the additional space occupied by auxiliary data, duplicate copies of the data, indexing, and buffer space increases the monetary cost of an access method design. On the other hand, using more memory for the design of an access method can lead to lower read or update time cost. For example, a logarithmic tree structure, or fractional cascading uses additional space as a way to speed up search. Similarly, buffer space to accumulate incoming data, or empty space in data containers like a B^+-Tree nodes uses additional space to facilitate updates.

Amplification as an Alternative Cost Definition. We defined above read and write costs in terms of total number of blocks of slow storage that have to be moved across the memory hierarchy. An alternative way to think about cost is in terms of amplification. For example, read cost can be defined as the excess number of data blocks an operation is forced to read on top of the actual data blocks that it needs. For example, a point query over a tree-like structure needs a single data block that contains the target key value pair. However, it is forced to read additional data blocks, in this case all the indexing blocks it needs to locate the target data. These additional blocks consist the *read amplification*. The definitions of *write amplification* and *memory amplification* are similar in that they define the excess data blocks we need to move during a write operation and the excess memory we need due to the design of the access method. Thinking about the costs in terms of amplification, as opposed to in terms of total number of blocks of slow memory, is often helpful as it gives a sense of how close a design is to the optimal.

3

Design Principles: Dimensions of a Design Space

We now present in detail the key design principles comprising the design space of access methods. We view access methods design as a six-step process where we make decisions about:

1. global organization, i.e., the way the data is organized in terms of how the different data blocks are laid out,
2. search method, i.e., the algorithm used to search over the data,
3. indexing, i.e., structure such as metadata, or models, that help accelerate access to individual data blocks,
4. local organization, i.e., the physical organization within each data block that creates combined with the global decision creates a global-local hierarchy,
5. update policy, i.e., the decision to update in-place vs. out-of-place,
6. buffering, i.e., the use of additional auxiliary space to group read and write requests before applying them.

An access method design includes decisions in each one of these categories even if the decision for example, that there is no indexing. In the

Data size (# elements (key-value pairs))	N
Page size (# elements that fit in a page)	B
Data size (# pages)	$N_B = N/B$
# partitions	P
index of searched key	i
index of the page of the searched key	$i_B = i/B$
index of the partition of the searched key	i_P
selectivity	s
radix length of the key domain	r

Table 3.1: Parameters

rest of this section, we discuss in detail the possible decisions in each step and how they affect the overall design in terms of the PyRAMID performance properties.

Cost Model. To quantify the PyRAMID costs throughout this section we use a simple cost model that uses the data set size in terms of number of elements N, the block size B that defines the access granularity, and the number of partitions P. Table 3.1 contains all parameters introduced in this section.

3.1 Global Data Organization

3.1.1 No Organization

Definition. The key value-pairs are physically stored without any particular order on the storage medium. As a result, any block my contain any key-value pair without constraints in order or relationship of keys that physically stored in close location.

Performance Implications. Since there is no underlying structure in the data every query (point, short range, long range) has the same cost $O(N_B)$, because it will have to sequentially scan the whole data collection. Inserting new entries simply needs to append at the current end of the collection having cost $O(1)$, while deleting and updating requires first to find the value to delete or update, hence there is an

expensive read cost followed by a single write operation (overwrite the deleted key, or swapping the updated keys).

Example. The most common example of *no organization* is traditional physical data organization of *heap files* employed by most relational database systems [192]. However, most access methods employ some data organization to facilitate either read queries or amortize the update cost. One example of the latter is the temporal organization which is discussed next.

3.1.2 Temporal Organization

Definition. Temporal organization ensures that all items are physically stored following exactly the arrival order. This ensures that every insertion creates minimal data movement.

Performance Implications. Incoming data items are simply appended to the existing data collection with cost $O(1)$. Effectively, we generate an ever growing log that absorbs data very quickly making it appropriate for write-heavy applications. When queries need to read data based on the arrival order, we can search the data quickly since it is maintained in that order. This is similar to searching over sorted keys. On the other hand, queries that search data based on a key need to pay the cost of scanning the whole collection, even when the result is a single key-value pair, or even when there is an empty query, having a total cost of $O(N_B)$. One can improve on that at a cost in space by creating a key-based index on top of the temporally stored items.

In the absence of an index, inserts may trade off space vs. time. For example, when a new data item arrives we can either append the item in the log as we described before, but then the dataset may contain duplicates entailing three extra costs: (1) memory amplification goes up for every duplicate key, (2) future queries need to resolve those duplicates, and (3) a background process may be needed to run periodically to remove duplicates by keeping only the most recent version of a key-value pair. The above approach sacrifices memory amplification and maintenance costs. The other extreme is to perform updates in place, as explained above.

Example. *Temporal organization* is used naturally when new objects are simply appended especially when the key is a timestamp. It is particularly useful in cases like data(time)-series [246].

3.1.3 Range Partitioning

Definition. Range partitioning physically organizes the data in P non-overlapping partitions based on different ranges of the key domain.

Performance Implications. The goal of any partitioning scheme is to create "groups" of data objects – called "partitions" or "buckets" – that have the same or about the same cardinality in order to divide the N items to P partitions each having N/P objects. This is not always possible; for example if there is no information about the data distribution. In addition, it is not always necessary in order to achieve good performance; for example, instead of equally distributing the data between partitions, the goal can be to equally distribute read or update access frequency. In the general case, having P partitions with approximately equal size allows for answering any point query by accessing only one partition, accessing, on average, $O(N_B/P)$ pages.

With respect to range queries, range partitioning allows to access only the relevant data partitions. There is a possibility that a partitioning will be accessed only to output one data item, but on average we expect that a range query with selectivity factor $s\%$ to read $s\% \cdot P$ partitions with a total I/O cost of $O(s\% \cdot N_B)$ pages. When the range query has a very short range, then the query will almost always read only one partition which will contain the interesting values in their entirety, so with range partitioning the cost of short range queries is the same as the cost of point queries, that is, $O(N_B/P)$. Point queries can be further optimized with *auxiliary data structures*, like membership test data structures, and sparse indexing, to avoid unnecessary accesses.

Modifications access the same number of blocks as searches, but must meet some constraints as to how an update policy can work [23]. If the data is range partitioned, then after an insert, update, or delete, they should remain range partitioned (otherwise we go to solutions that hurt read performance – for example, we can perform lazy updates that

future queries can resolve *adaptively*. Hence, a new object should be inserted in the correct position, which can be achieved by moving half of the data with cost $O(N_B/2)$ or by the less expensive ripple-insert algorithm [115] with cost $O(P/2)$. Updating and deleting requires to first locate an object, so there is a point query followed by the delete or update operation. As an additional consideration when deleting, the empty slot created can either remain in-place (and thus, gradually, deletes would cause fragmentation), it can be reclaimed and cause an increased write amplification, or it can be reclaimed periodically to amortize the write amplification caused.

Example. Range partitioning is frequently used either in its pure form when sharding data in different nodes, or extents based on key ranges of a column, or as one design decision for organizing data in non-overlapping collections (like leafs for B^+- Trees and their variations, and SST files for LSM-Trees).

3.1.4 Sorted

Definition. A sorted data organization guarantees that at all times, the data is sorted based on the key without requiring additional metadata, except the temporary scratch space needed when sorting the data.

Performance Implications. Having sorted data implies that point queries can be answered using binary search, regardless of the to key distribution, and has search complexity $O(log_2(N_B))$ in terms of disk block reads. Given some knowledge about the distribution of the data, the search complexity can be reduced further, using for example interpolation search [186, 225] or exponential search [41]. With respect to range queries, a sorted data organization allows us to quickly find the beginning of the qualifying range and then simply consume it sequentially. On the other hand, facilitating inserts, updates, and deletes can be expensive if we choose to maintain the strict sorting in a dense data organization. In general, inserting in a sorted collection would require to always insert in the correct location, that is, move key-value pairs "to their right" when a new value is inserted. This cost can be approximated as $O(N_B/2)$ because on average half of the data would

be moved for every insert. Similarly, deleting would require to consolidate the collection to avoid having holes and an update (from one key to another) would require swapping the key-value pairs. The cost of updating in-place can be mitigated by combining different data organizations with out-of-place update strategies using additional space for buffering.

Example. A *sorted organization* is an extremely common organization employed in key-value stores where data across all leaf level nodes in B^+-Trees [89] and data within each level of LSM-Trees [180] are sorted.

3.1.5 Hash Partitioning

Definition. Hash partitioning stores key-value pairs in partitions based on the hash of the key This approach creates non overlapping *hash-based partitions*.

Performance Implications. In contrast to the range partitioning approach, each hash-based partition may not contain a range of keys that is disjoint from the key ranges of other partitions. One of the basic properties of *good* hash functions [132, 134], is that they map the expected inputs as evenly as possible (ideally uniformly, but in practice within a factor of 2) to the output domain. As a result, organizing data using hash partitioning can easily create partitions with the similar sizes even in the absence of any information about the distribution of the hashed keys. In order to get many of the benefits of range partitioning in addition to the benefits of hashing, one can use an order preserving hash function [66] though this would typically mean that the output distribution will not be as close to uniform as with a non-order-preserving hash function, so the partition sizes may vary more. In this case, hash partitioning will be able to support range queries as well. A hash partitioning scheme supports a point query by retrieving only the relevant partition, so, if a hash partitioning scheme has P partitions, the I/O cost of a point query is $O(N_B/P)$ pages, similar to the best case for range partitioning. The fundamental difference is that for hash partitioning this is also the average case. predictable behavior for point queries, however, comes at the expense of the range query performance.

Effectively hash partitioning cannot accelerate range queries; they need to scan all key-value pairs. In specific cases of short range queries, and if the domain is discrete, a short range query can be replaced by a number of point queries, however, as the range grows this option becomes more expensive. As with any range partitioning scheme, modifications first perform searches and then perform the modification (insert, update, or delete).

Example. The most common example of hash-partitioning is partitioning phase of the hash-join algorithm and hash-index. They both organizes key in buckets based on their hash to ensure faster retrieval by searching smaller partitions. Note that, as in all partitioning schemes, the internal organization of the partition can be different.

3.1.6 Radix Partitioning

Definition. An interesting middle-ground between range and hash partitioning is radix partitioning, which uses the bitwise representation of the keys to partition the key-value pairs. In radix partitioning, a prefix (which can be statically or variable sized) of the bitwise representation of the key is used to map a range of values to a specific partition. Radix partitioning may create partitions of vastly different sizes because certain prefixes may occur far more often than others.

Performance Implications. While radix partitioning is very similar to range partitioning with respect to its efficiency on the average case of the uniform dataset, it relies on the bitwise representation to create partitions with more balanced entropy. This can be further augmented by the use of adaptive radix partitioning, where the length of the radix used depends on the data distribution [149]. Similarly to range partitioning, radix partitioning facilitates fast point queries and range queries. Inserts must go to the partition with the corresponding radix, possibly requiring restructuring.

Example. Radix partitioning has been used in modern design of join algorithms and indexing as it needs to compare fewer bits to decide which partitions to discard.

3.1.7 Temporal Partitioning

Definition. In addition to a strict temporal organization, data items can be accumulated into *epochs*, that each can have a different organization.

Example. Popular examples are the Log-Structured Merge Tree and similar hash-based access method designs [54, 160, 182], which allow for order-of-arrival ingestion and gradually organizes data based on its keys.

3.2 Search without Indexing

Now that we have covered the different data organization we can match each organization with the search algorithm that can be used and discuss the best choices for each one.

3.2.1 Full Scan

Applicable to: all data organizations for either a point or a range query.

Performance.. The performance of a full scan depend only on the data size and the query selectivity (due to predicate evaluation and writing the output [130]). In terms of I/O cost a full scan will always have to read $O(N_B)$ pages of data to find the selected object or objects. As a rule of thumb, a full scan should be used if (i) we have no information about the data organization, (ii) "no organization", (iii) a pure "temporal organization" is used, or (iv) the query will return a large percentage of the initial data collection, i.e., it is a long range query with high (i.e., large fraction) selectivity. The latter point is the topic of access path selection [130, 208].

Optimizations. While a full scan has to access the entire data collection, its simple access pattern can be significantly optimized by techniques that exploit modern hardware like parallelization (break the data collection in multiple chunks and use a different processors to scan each) and vectorization (exploit SIMD commands to increase the throughput of comparisons). In general, applying the same techniques

in the other search algorithms is more challenging because they lack the repetitive sequential accesses of the same data types.

3.2.2 Binary Search

Applicable to: all sorted, range partitioned, and can be combined with radix partitioned organizations for either a point or a range query.

Performance.. Binary search finds the desired value in a sorted collection after $O(log_2(N_B) + 1)$ accesses, or if the data is partitioned in P partitions, in $O(log_2(P) + N_B/P)$ accesses. If the query is a range query the binary search will be followed by sequential accesses to the subsequent values. In turn, the cost of the queries will depend on the selectivity factor of the queries in a linear fashion. A key benefit of binary search is that its complexity does not depend on the data distribution.

Optimizations. In addition to binary search, a sorted collection can be searched with a ternary search that creates three groups, or generally with a l-ary search with l groups, leading to $O(log_l(N))$ search cost. More complex algorithms like exponential [41] and interpolation search [186, 225] are treated as a special case, because while they can be employed on any sorted collection, they provide significant benefits depending on the data distribution.

3.2.3 Direct Addressing

Applicable to: hash and radix partitioned organizations for either a point or a range query.

Performance. When we have very accurate information about the data distribution we can implement direct addressing with cost $O(1)$. The classical example is a perfect hash function that gives the exact unique location of a key within a collection because of the uniformity of the hash function. In addition, a radix partitioning scheme also leads to direct addressing with respect to the partition or even the position of the block containing the desired key within the data collection. When direct addressing is possible it should be preferred. In some cases, like radix partitioning, both binary search and direct addressing are possible, however, the search algorithm of choice is usually direct addressing.

3.2.4 Data-Driven Search

Applicable to: all sorted, range partitioned, and can be combined with radix partitioned organizations for either a point or a range query.

Performance. A class of search algorithms that lie in between sorted search and direct addressing allow to exploit possible knowledge of the distribution, using it as a hint about how to direct the searching. Interpolation search [186, 225] effectively uses the information collected at the current step to discard not only half of the remaining data – as binary search is doing – but as much as possible, assuming that the distribution is uniform. By contrast, exponential search [41] allows for a very efficient search algorithm of an unbounded array, or conversely, of a very efficient search of a sorted array where the value we are looking for is in the beginning of the array. Exponential search performance depends on the final position of the searched item rather than the size of the overall data collection. If the desired element is in page i_B, the complexity is $O(log_2(i_B))$ compared to $O(log_2(N_B))$ for binary search. These data-driven search algorithms can be used in any collection that is sorted or organized based on the key.

3.3 Search with Indexing

In the previous subsection we covered in detail the different ways we can search a collection of keys depending on the employed data organization but without indexes. Instead, we can employ auxiliary metadata, called a navigable *index*, that allows us to find the item in question.

Memory Management. The discussion on indexing in this section assumes an ideal memory management of a limited number of blocks M. This ideal memory management is often approximated by buffer replacement policies that attempt to keep the most useful pages in memory. For example, the least recently used (LRU) replacement policy keeps always the M most recently used pages assuming that the future accesses will most probably be one of the most recent accesses. There is a wealth of buffer replacement policies which are out of the scope of this analysis. Here, we assume that we have an *oracle* that always allows

us to keep the M most useful pages. In practice, for tree indexing, this means that we can in memory the necessary pages to guarantee that the first l levels do not necessitate a disc access.

Similarly to any search algorithm on base data, the goal of an index is to locate the position(s) of entries with key k.[1]

Here we draw the parallels between the search algorithms described in Section 3.2 and the corresponding indexing approaches.

3.3.1 Full Scan → Sparse Indexing

Similarly to a full scan, a variety of sparse indexing approaches can be built over any data organization. In particular, for every physical data page, or in general, for an arbitrary collection of pages, we can store membership test data structures like Zonemaps [173], Column Imprints [214], and Bloom filters and its variants [38, 43, 75]. The core idea behind a membership test data structure is that it can identify at low cost whether it is safe to avoid accessing a portion of base data. For example, using Zonemaps, we create small summaries of each data block (or potentially a larger chunk) of a data collection, which does not have to be sorted or specifically organized. These small summaries contain the information for the minimum and maximum per chunk and allow to completely skip a chunk when the searched key fall outside its range. Similarly, Column Imprints store a lightweight histogram that allows for more accurate data skipping. In both and in general in sparse indexing approaches, there might false positives, that is some of the chunks that are not discarded will be fully scanned only to find no matches. Overall sparse indexing, reduces the *scan ratio*: the fraction of the data items that need to be scanned. Adapting a scan algorithm or any search algorithm to incorporate sparse indexing requires to simply check the appropriate filter before accessing each part of the data.

[1] In particular, the position(s), page(s), or partition(s) that entries with key k are stored.

3.3.2 Binary Search → Binary, K-Ary Trees, B$^+$-Trees

When a collection of keys is sorted, binary search leads to logarithmic cost of searching by exploiting the order and iteratively safely discarding half of the data in every iteration. With the aid of auxiliary metadata we are able to discard $P-1$ out of P partition of the data. In particular, storing the partition boundaries that create P equal partitions in size we can safely avoid reading $P-1$ partitions. In order to be able to do this iteratively, similarly to what binary search is doing we keep decomposing each of the partitions into P equal partitions, creating more, decreasing in size, layers of metadata, that allow us to safely discard $P-1$ out of P partitions at each level. Such a k-ary tree allows us to quickly search a sorted collection of keys and reduces the search time from logarithmic with base 2, to logarithmic with base P. When $P=2$ the k-ary trees perform exactly like a binary search, and take the shape of binary trees. While k-ary trees enable efficient searching, they suffer when the sorted collection of keys is updated. In order to facilitate dynamic workloads, both the sorted collection and the auxiliary metadata are organized in logical nodes and the resulting tree can be updated in-place without having to move large portions of the data. The k-ary trees, however, can be a case where less is more if the workload is read-intensive. The compact nature of k-ary trees with minimal metadata make them hardware-conscious because when reading a tree node in a cacheline, we can make full use of it, as it is the equivalent of searching the all the partitions created by the P fanout.

3.3.3 Direct Addressing → Radix Trees, Hash Indexes

Instead of scanning or iteratively searching, an alternative approach is to use the binary representation of the keys to directly locate where they are physically stored, through hashing or a radix search. The most common approach for direct addressing is hashing. Hashing is particularly good for highly skewed distributions. A hash index creates an unordered associative array and maps (hashes) the keys of a skewed distribution to a (close to) uniform distribution using an ideal hash function. Cryptographic hash functions offer good uniform distribu-

tions but are more computationally expensive. In order to ensure correctness, hash indexes employ various ways of handling collisions (key being hashed to the same hash value), in various ways including among others chaining, linear hashing, and open addressing.

Another approach is employing a radix tree. Radix trees use each bit of the binary representation of the keys to form a binary tree that upon searching we directly know which child to follow in order to locate the leaf corresponding to the key in question. To increase space efficiency and reduce random access during tree traversals, nodes that have valid values in only one child can be merged with the child. Further space efficiency can be achieved if two (or three) contiguous bits of the binary representation are always merged effectively creating a radix tree with four (or eight) children which would also fit in a cacheline and increase locality. Radix trees and their space-optimized variations efficiently support both point and range queries when the data distribution is not highly skewed.

Manos: Maybe some examples of decent hash functions and a picture.

3.3.4 Data-Driven Search → Learned Indexes

The metadata used by search trees attempt to map the real distribution of the data to a uniform distribution (by creating equi-depth histograms, or nodes with the same size). Hash indexes are particularly successful at this, often able to map a potentially highly skewed distribution to a nearly (within a factor of 2) uniform one. Data-driven search algorithms calculate the expected position of a key (assuming a uniform distribution). A natural next step is to *learn* the *actual* distribution of the data and subsequently use this *learned index* to *predict* the expected location of a key in a sorted collection [135]. The original learned indexes proposal employs machine learning models and the auxiliary metadata as model coefficients that are calculated after training. For example, FITing-Tree [84] uses linear models to replace each tree node, achieving a huge compression of the internal tree nodes. Essentially each tree node with fanout F, has F pointers and F value separators, while a learned index node that employs a linear model only

needs to store the two coefficients for the linear approximation. There are several interesting challenges in learned indexes, like bounding the indexing error and supporting updates, however, they are the first step towards the indexing counterpart of data-driven search algorithms.

Manos: a picture would be helpful here

Learned indexes also present a RUM tradeoff. For example, storing a model is generally space efficient, so it has minimum memory amplification compared to a classical (metadata-based) index. On the other hand, a learned index is data dependent, so it will require substantial maintenance effort during updates, leading to higher update cost, or to lazy updates that would lead to higher read cost.

3.3.5 Achieving Robust Performance with Search Trees

While performance is a key requirement for any data system and data access method, performance stability (predictability) is also expected. Dynamic search trees achieve both by ensuring low (logarithmic) search cost, and re-organizing their structure when updates cause an imbalance of the tree that would also affect the performance stability.

Logarithmic Method. Bentley proposed that most search algorithms can be decomposed [39] into (recursive) search algorithms in disjoint sets. This key observation is then used to prove that every decomposable search algorithm has logarithmic cost, which has been used as the core intuition behind both search algorithms (e.g., binary) and searchable data structures (e.g., k-ary trees and B^+-Trees).

Fractional Cascading. A k-ary search tree comprises of several levels that each contains a sorted collection of key separators. In order to reduce the search cost, each sorted level is connected with the following one with intermediate pointers to allow discarding increasing portions of the overall data collection. These pointers allow to navigate the search effort of one sorted level only to the useful (not yet search) part of the subsequent level. This way of optimizing iterative searches is called *fractional cascading* [57], and is the principle behind any the pointer connections between levels in any tree data structure.

Rotations and Rebalancing. Through the logarithmic method and

fractional cascading we build access method with optimal search time when data is static and organized as a pre-processing step. We already discussed that search trees can support dynamic insertions (or general updates including deletes) but this might affect the balanced structure and hence the worst-case latency of a query. In order to address this, dynamic search trees allow for rotations and rebalancing. Binary tree variations offer self-balancing through rotations [207]. AVL-Trees [2] and Red-Black-Trees [32] are two notable examples. New designs of self-rotating self-balancing binary trees are still under development in order to ensure the rotations will not affect the whole tree path, and they will have a minimal, localized to the updated region, cost [99]. For the dynamic search trees that have more than two children, that is, B^+-Trees and their variations, rebalancing is maintained with the process of splitting and merging nodes having an invariant that every node will be at least 50% full. A split (merge) might lead to a recursive split (merge) starting from the leaf all the way to the root in the worst case. In most cases, though, a local rebalancing between the last two levels is enough, and exponentially less frequently this propagates to higher levels. While this leads to an expensive worst-case insert (delete), the cost is amortized, and this approach ensures that the tree will always be balanced, hence offering the same (logarithmic) search cost for every part of the key domain. While the textbook algorithms include the merging of nodes as outline above, frequently, they are not performed because it has been shown that in general dynamic workloads merging is wasteful and nodes should be kept even when they are underutilized and removed if empty [125].

3.4 Local-Global Hybrid Data Organization

After deciding on the global data organization, the search methods, and the use of indexing to accelerate accessing the data in Sections 3.1, 3.2, and 3.3, the next decision to make is the local data organization, i.e., the organization of data within the blocks. In each block or partition, a different data organization decision can be made. A block or partition can have any of the available data organizations previously

described: none, temporal, sorted, range partitioned, hash partitioned, or radix partitioned. In this section, we focus on the interplay between a global and a local data organization, and the resulting hybrid data organizations.

Each data organization option discussed previously can be classified as a *terminal* or a *non-terminal* option. In particular, no organization, sorted organization, and temporal organization are terminal options because no further refinement is possible. On the other hand, all partitioning schemes (range, radix, hash, temporal) create groups of data objects – partitions – that each one internally can be organized in a different way. Below we describe hybrid data organizations that are already described in the literature and that can be derived by combining fundamental data organizations.

3.4.1 Range - Sorted (Traditional B$^+$-Trees)

The leaves of the traditional B$^+$-Trees [89] can be viewed as a hybrid data partitioning scheme that combines range partitioning at the high-level (per node) and sorted organization at the node level. Every time we want to perform a search we first need to find the appropriate partition (leaf) and then search within the sorted partition (either using binary search or a full scan of the partition). Finding the partition can be done using one of the above search algorithms or using additional metadata for faster indexing as discussed in Section 3.3.

3.4.2 Range - Range (Fractal B$^+$-Trees)

The design of B$^+$-Trees has been traditionally optimized for hard disks, and hence the node size – which can also be viewed as a partition – is tailored to the disk I/O unit size [87]. When a B$^+$-Tree is used across multiple levels of the memory hierarchy (disk, main memory, cache memories) it is important to access data chunks whose size is appropriate for each level of the memory hierarchy. For example, the fractal B$^+$-Tree organizes data in nodes sized according to the disk access unit, and internally each node is organized in a mini-tree following the unit size of cache memory [60], that is, a cache line. Effectively fractal B$^+$-Trees have a hybrid data organization with two levels of range

partitioning.

3.4.3 Range - Temporal (Insert-Optimized B$^+$-Trees)

Classical B$^+$-Tree designs assume that the nodes contain items (or nodes) that are sorted. This property can facilitate efficient searching. When nodes are written to a storage medium or a non-volatile memory, however, reorganizing the contents of the node every time can be expensive. To address this two approaches have proposed B$^+$-Trees with nodes that internally are not sorted. The unsorted B$^+$-Tree for non-volatile memories [61] aims at minimizing write amplification and the Bw-Tree design [150] aims at faster updates through logging. Both approaches effectively support range partitioning of the data to different leaf nodes, but unsorted data for each node as data arrives.

3.4.4 Range - Range - ... - Range (Bulk-Loaded B$^+$-Trees, Sorted, Adaptive Indexing, Cracking)

Recursive range partitioning creates a nesting organization strategy that leads to fully sorting data. Nesting range partitioning necessarily terminates when all items are sorted either as a bulk-loaded B$^+$-Tree [89] or as a sorted array. In addition to this extreme termination of the recursion, data can be recursively range partitioned up until the granularity that makes a difference (that is, a memory page size if data resides on disk, or a few cache-lines if data is already in memory). The actual values that will serve as range separators during the recursive partitioning can either be determined by the workload, as in traditional cracking approaches [110, 114, 116] *Dennis thinks we can't assume the reader knows what those traditional cracking approaches are*, or chosen to balance the partitioning effort evenly throughout the domain as in early adaptive indexing approaches [94], or a hybrid of the two [100, 118, 204].

3.4.5 Range - Hash (Bounded Disorder)

Starting from range partitioning as a first decision, we can also organize each partition by hashing. This scheme efficiently supports both point

queries and range queries. Point queries are directed to the appropriate range partition within which they can use direct addressing (hashing). Range queries will consume the corresponding range partitions similar to the way they can be executed in the case of pure range partitioning. A design that combines in that manner range and hash partitioning is the bounded disorder access method [155] which organizes keys in large ranges and within each range the keys are hashed forming a tight hash organization. *Dennis doesn't understand what "tight" means*

3.4.6 Radix - Sorted (Radix Trees)

Up to now we considered hybrid data organizations where the first decision was to range partition. A different decision could be to radix partition data. So the keys can be organized using a prefix of their radix representation. To address practical considerations like handling duplicates and maintaining good performance when there are areas of the domain that are sparse, objects that are partitioned together are typically kept sorted„ This then combines radix partitioning with sorted data organization. Adaptive radix tree [149] and Masstree [168] are the two classical such designs.

3.4.7 Hash - Temporal (Chained Hash Table)

Instead of radix partitioning the first decision in a hybrid choice can be hash partitioning. This creates buckets of values that are hashed together, so a second data organization decision needs to be taken for each partition. A typical decision is simply to append items on the same bucket as they arrive effectively using temporal organization. This hybrid between hash partitioning and temporal organization is used by the textbook chained hash table [192].*Dennis asks whether this is really called textbook chained hash table*

3.4.8 Temporal - Sorted (LSM Trees)

Another data organization that was introduced above was to temporally partition data. This allows for very cheap insert as incoming data are simply appended. However, as more data are being collected it is

becoming very expensive to search. A way to mitigate this problem is to reorganize each temporal partition (or epoch) in a way that it would be easier to search, for example by sorting. Sorting benefits both range and point queries. Taking this combination of temporal partitioning and sorting a step further by combining all epochs of equal size at every reorganization level, the Log-Structured Merge Trees (LSM-Trees) offer a very good balance between read and update performance [182]. *Dennis thinks this paragraph would be incomprehensible to a beginner without a picture*

3.4.9 Temporal - Radix/Hash (LSM Tries)

A different approach for reorganizing epochs is instead of sorting them to hash partition them. This approach will be able to efficiently support point queries – similarly to what hashing offers – but not range queries. Alternatively, each epoch can also be radix partitioned this would allow for direct addressing for point queries but at the added benefit of supporting range queries as well. These two hybrid data organizations have been proposed as a trie-based LSM-Tree and the LSM-Trie to support large data collections with many point queries having very low write amplification [235].

Manos says: in this section I will add a performance implication paragraph in each section

3.5 Update Policy: In-place vs. Out-of-place

For every organized data collection an incoming update request will have one of the two possible treatments. It will either change the current organization by placing the new item (or update an existing item) in its rightful place according to the organization, or it will be maintained as a pending update in a different place and the update will have to be reconciled with the base data at a later time. These two policies are termed *in-place* and *out-of-place* accordingly.

When employing an in-place update policy, the key-value pair of key k has always one version, which is the current one. On the other hand,

when employing an out-of-place update policy, there might exist several versions of the key-value pair for key k which have to be reconciled before the valid one is returned to the user.

Note that out-of-place updates and buffering updates (discussed in the next section) are similar, yet not identical concepts. Buffering updates refers to the space and time needed to keep an update in a temporary space prior to its final location. The final location of a new or an updated item may be either in-place, when it is stored in the same container with the rest of the data, or out-of-place, if it is stored in a new container.

3.5.1 In-place Updates

Following in-place updates, once we have an organized data collection and we receive a new item (or an update), the result of the insert (update) should be a new data collection with the same organization that includes that latest item. For example, if we have a sorted collection and we insert a new key k, it will have to be inserted in the position dictated by the sorting order. If the data collection was range partitioning, the new item will have to be placed physically in the corresponding partition. In-place updates is a common update policy used in several data structures including textbook B^+-Trees, hash indexes, bitmap indexes, sorted linked lists and others. In-place updates leave the data collection in a state that it can be very efficiently read by future access requests and has no extra memory consumption, however, it moves all the re-organization burden upon updates, hence the update performance suffers.

Impact on PyRUMID Costs. This is the default approach, assumed in the insert/update/delete costs above.

3.5.2 Deferred In-place Updates

A variation of in-place updates, when they are combined with buffering updates locally is the deferred in-place updates. These are updates that will eventually end up in the expected position according to the selected data organization, yet this will happen in a secondary step of

reconciliation. Deferred in-place updates are employed by approaches that do not want to eagerly reorganize data for each update, but want to quickly converge to the desired data organization like cracking [114], buffer repository trees [47] and fractal trees [136]. Deferred updates are temporarily stored as part of one partition of the access method and are quickly integrated in an in-place manner with follow-up read queries or a background operation. Another variation of deferred in-place updates creates additional metadata to indicate where the new update would physically go if it were to be merged [107]. This leads to very fast queries when when working on both the base data and the pending updates assuming that everything fits in memory. Deferred in-place updates require some buffering space, in order to ensure that read queries will be able to find close any recently inserted or updated element close to the base data, and further defer the re-organization effort in order to reduce the update cost.

Impact on PyRUMID Costs. When performing a deferred in-place updates, essentially every insert/update/delete operation has to pay the search cost to find the partition or block that it needs to be added to, without having to perform the local re-organization, rather using a local buffer.

3.5.3 Out-of-place Updates

Out-of-place updates completely avoid interfering with the current data organization and are stored in new data containers with their own organization, which is often the same but it could be potentially different. Contrary to deferred in-place updates, out of place updates have a single buffer space for the whole data collection, and have a more expensive reconciliation process. The key idea of out-of-place updates is that (a) they are stored sequentially because they are part of a new container – and as a result they are tailored for cases that random access is expensive, and (b) they do not affect the organization and the accessing process of the pre-existing data. The out-of-place paradigm is classically followed by Log-Structured Merge (LSM) Trees [182] and has since been used in a number of access methods to facilitate efficient updates. The update cost is minimal, because any new or updated en-

try is simply appended in the global buffer space. However, as a result, any read query has to go through the base data, and through the entire global buffer to provide an accurate answer. Space-wise out-of-place updates allow for duplication as an updated entry does not immediately update in-place leading to potentially several invalid entries. Overall, out-of-place updates aggressively minimize update cost at the expense of both increased read cost and increased space utilization.

Impact on PyRUMID Costs. Out-of-place updates avoid the search part altogether, leaving for future work both the work needed to find the appropriate partition or block and the local re- organization.

3.5.4 Differential Out-of-place Updates

A variation of out-of-place updates is the concept of differential out-of-place updates. The key idea is that when updating an existing entry, differential out-of-place updates have very small size because they store only the difference from the old value rather than not the whole value [209]. This allows for quick and space-efficient updating but has higher cost during reading because often the differential update is not enough and has to always be reconciled with the base data.

Impact on PyRUMID Costs. This is an optimization when compared to out-of-place updates only with respect to storing less metadata for each update.

3.6 Buffering: Batching Requests

Once the data organization, the use of auxiliary space for indexing, and the update policy are decided, an access method may use additional space to further accelerate read and writes requests in the form of buffering. Using buffering for reads and for writes have performance benefits for different reasons. Buffering read requests allows for better **scheduling**, while buffering incoming writes allows the data structure to apply them in one go. In essence, buffering allows a data organization to avoid repetitive unnecessary trips to the same physical location or partition.

Buffering reduces read (write) cost by using additional space to batch incoming read (write) requests, which can then be processed ***concurrently*** *and hence amortize their overall cost.*

In either case, buffering is a classical manifestation of the RUM conjecture [27] and follows the benefits of the buffer technique [14]. An access method design optimizes read performance (when batching read requests) by using additional space to buffer incoming read requests (hence the tradeoff is extra space for better read performance) and/or it optimizes write performance (when batching writes) by using additional space to buffer incoming writes (inserts, updates, or deletes). We further differentiate two types of batching updates, one globally for the whole domain, and one locally, for a single partition.

Orthogonally to this discussion, buffering can be beneficial to instruction cache locality. In particular, by allowing a specific piece of code to buffer its output as opposed to send it to the next logical operator and then be called again, we can keep the instruction caches warm and avoid instruction cache thrashing [101, 243].

3.6.1 Buffering Reads

When an access method buffers read requests, it can answer all of them (or a carefully selected subset) by a single access to the base data. A typical example of an access method that buffers reads has been proposed as shared scans (also known as cooperative scans) that are heavily employed by analytical data systems [16, 51, 85, 101, 102, 123, 170, 189, 191, 222, 248]. **The benefit of batching reads is that it allows for concurrent execution of queries without using more CPU resources and without requiring more data accesses**. On the contrary, a single access over the same data set – coupled with buffer space for read requests, and the metadata needed to keep track which items qualify for each query – is enough to answer all queries of the batch.

Impact on PyRUMID Costs. Buffering reads does not have direct impact on PyRUMID costs, other than the use of extra space in order to exploit to the maximum degree the available bandwidth of the

underlying storage.

3.6.2 Cache Pinning as a Special Case of Read Buffering

A special form of read buffering is **cache pinning**. While caching is not a design dimension of an access method, rather it is a system-wide feature, cache pinning can be thought of as a particular form of buffering. Consider that cache pinning would buffer in fast memory a small set of objects that are frequently read. So in this case, the access method can identify frequently read items and suggest to the cache that if they are pinned the overall number of slow accesses will be reduced dramatically.

Impact on PyRUMID Costs. Cache pinning is the explicit request to keep popular or useful pages in the cache, hence, in this respect it increases the memory cost to offer lower latency for reading the specified blocks.

3.6.3 Buffering Updates

Orthogonally to batching read requests, an access method can buffer updates and support batch processing for updates. Even a single update causes data movement and data reorganization, hence, by batching updates, an access method separates the access patterns, and updates do not interfere with other concurrent read operations. Access methods that target reads-intensive applications often buffer updates and apply them sequentially in batches. On the other hand, access methods that target transactional workloads typically apply each update independently. While buffering updates can be seen as an update policy, it is the tradeoff between memory and update efficiency that has more implications than only the out-of-place update policy, which makes it a design dimension of its own.

Impact on PyRUMID Costs. Buffering updates uses additional memory space in order to enable out-of-place updates. It exploits a tradeoff between memory usage and delaying the searching and the local reorganization work needed after an update.

3.6.4 Buffering Updates Locally

We differentiate between buffering updates and buffering updates locally, to showcase that when a data organization strategy has been selected, updates can be buffered in a per-partition basis. This allows the updates to be stored close to the targeted base data, and hence future read queries would need to merge only those updates. In addition, an access method tailored to the exact workload may buffer updates in one part of the domain, and opt to apply updates immediately in another part of the domain.

Impact on PyRUMID Costs. Buffering updates locally uses additional memory space in order to enable deferred in-place updates. It exploits a tradeoff between memory usage and delaying the local reorganization work needed after an update.

3.7 Contents Representation

For every access method design, another key decision is how to physically represent the indexed data. The exact choice of the contents representation indexed by an access method influences not only the access performance, but also the update process, the resource utilization, and the algorithms that can be used during query processing.

3.7.1 Key-Record

A natural way to store data is to store the indexing key along with the entire row. This model captures the traditional way relational systems store data in heap files, by treating the row ID as the key, and the whole tuple as the record. In general, this is also used by indexed files where a B^+-Tree on the key stores in its leaf nodes entire records, and by json objects, where the "value" of the key-value paradigm is the stored record. Under the key-record model, we also classify indexes that store a subset of the attributes of the relational tuple, but always have access to the same subset, that is, a secondary index with multiple attributes stored in the leaf nodes. The key-record contents representation approach is also termed alternative-1 data entry [192].

3.7.2 Key-Row ID

An alternative representation used in secondary indexing, and in some implementations of columnar storage is the key-row ID model, where for every existing key we store all the row IDs that this key exists. This model captures both the standard approach of how B^+-Trees are organized (alternative-2 and -3 data entry [192]), as well as the columnar storage implementation followed by MonetDB using Binary Association Tables (BATs) [113], where every column is a binary table, with the first attribute being the object identifier (the Row ID in our terminology) and the second attribute the actual stored value (the key in our terminology).

3.7.3 Key-Pointer

When the access method is purely in-memory or uses pointer swizzling [228] to convert row IDs to in-memory addresses [95], the contets of the access method are represented by the indexed key and the corresponding pointer(s). This approach is generally used by any access method operates purely in-memory and wants to ensure fast access to the original data location irrespectively of the exact data format followed in the base data.

3.7.4 Key-Bitvector

Finally, another alternative is an existence bitvector for every key of the domain. This approach captures all the bitmap indexing approaches [52] and offers a different storing, compressing, and execution model. Bitvectors can be aggressively compressed using variations of run-length encoding techniques, and data can be processed directly on the bitvectors for several operations including selection, projection, joins, and sorting. The bitvectors can further exploit efficient bitwise operations when in-memory and, hence, consume data a very high pace.

3.8 Adaptivity

Another essential design dimension is adaptivity. That is, access methods may react to workload patterns and adapt some of their core design dimensions to better match the performance requirements. In principle, it is possible to apply adaptivity on top of any design dimension discussed so far. The exact way to properly apply adaptivity as well as the performance effects may vary. The fundamental principle remains the same though: as the workload evolves the access method also evolves, e.g., by reorganizing the base data or indexes to better support the incoming requests. In fact the goal of adaptivity may be to do just enough to support the existing workload. Exactly because there is no perfect data structure, state-of-the-art designs "over-prepare" for multiple scenarios. For example, a sorted data set or a B^+-Tree can support any range query efficiently. However, a particular workload may need only certain partitioning which may be much faster to prepare and maintain.

Adaptivity is the process that treats every access as a hint as to how to change the current state of any of the previously mentioned design decisions of an access method.

Compared to the rest of the design dimensions, adaptivity has only recently gained attention as a promising design dimension in terms of: (i) the existence of applications that need adaptivity, and (ii) the viability of design and implementation of adaptive approaches. Applications that need adaptivity appear at an increasing pace because of more volatile workloads of user facing application and diverse hardware of cloud settings. Implementation viability improved due to the ability to hold hot data in large memories, which allows to avoid pushing every data reorganization action to disk.

3.8.1 Adaptive Sorting

The first design option for adaptivity is adaptive sorting. The primary design method for adaptivity is physical data reorganization. The key

concept is that every access to the dataset can be used as an opportunity to reorganize data. Under this design, the dataset is not organized a priori, rather, every access is piggy-packed with a reorganization step. Hence, adaptive sorting is gradually bringing the data collection closer to a sorted data collection. The region of the domain that is sorted can either be connected with the most recent query or not. Note that adaptive sorting is adaptively applying a data organization decision. As the data is being gradually more organized, a read query can be executed on average more efficiently, while an update query would increase the work needed to organize the data collection. In any case, adaptive sorting does not incur any space overhead, other than the temporary space needed to physically re-organize data in every step.

Query-Driven. The most common special case is the query-driven adaptive sorting. The key concept is that every reorganization step is driven by the most recently executed query (hence it would speed up a potential subsequent execution of the same query). This principle is used by approaches like Splay Trees [215] where nodes that were recently touched are moved up in the tree hierarchy. Another instantiation of the same principle is Database Cracking [114] which uses every range query as a hint to create a partition containing the data requested by the specific query by physically reorganizing the data.

Query-Independent. Another approach is to use the access of the query as an opportunity to re-organize, yet, not the accesses portion of the data. This design avoid over-fitting to the executed queries and gradually organizes the whole data collection. Such query-independent approaches have been proposed in the context of the Stochastic Cracking [100].

Hybrid. A third approach is the hybrid of the query-independent and the query-driven adaptive sorting. This approach used the query as a hint in some cases while organizing other parts of the data collection in the remaining re-organization steps [100]

3.8.2 Adaptive Vertical Partitioning

In addition to adaptively organizing the data based on the values, another action that can happen adaptively, is the decision how to vertically partition different data items that are part of the same key-value pair or row in a relational data system. While this decision is typically taken statically before any data operation (for example, a row-store or a column-store), adaptive vertical partitioning allows an access method to use hints from recent queries as to how to vertically partition the data collection. The system monitors workload execution and then materializes (that is, creates a new copy of) the data in a new layout continuing the workload execution. This technique has been proposed by the H_2O system [9] for read-only workloads and by the Peloton system [15] for mixed read-update workloads. Adaptive vertical partitioning is orthogonal to any adaptive sorting the system may employ on top, however, the two can be combined and potentially co-optimized, which still an open research question.

3.8.3 Adaptive Update Merging

The two prior subsections discuss read adaptivity only, as the process of gradually preparing data for future read queries. In general, for every access method design, we want updates to be correct and fast without interfering with our read queries.

In addition to that, the design of an access method can adapt as a result of an increased number of updates. For example, a B^+-Tree can have nodes that are read-optimized or write-optimized based on the ratio of reads and writes [162]. Thus, a fundamental change can be to switch update policy from in-place to out-of-place in order to absorb more incoming updates. Update adaptivity is much more recent direction which has more untapped potential especially in use-cases with update-intensive workload bursts.

3.9 Discussion

In this chapter we discuss the six fundamental design dimensions that can describe a large space of access methods. When designing an access

method more decisions have to be made with respect to performance tradeoffs. Notable examples are (i) how to trade off space for computation (e.g., the tradeoff of compression space efficiency and its computational cost), and (ii) how to allocate cache space for frequently read items, to avoid expensive retrieval from slow storage.

Adaptive Indexing. Splaying Trees can be considered as the first adaptive indexing approach [215]. The primary design concept is that data that has been recently touched moves up the tree. This means that when a query traverses the tree, it will likely find recently accessed data, higher in the tree and as such it will terminate the search faster. This same concept has also been applied in LSM-trees, with the work on Splaying LSM-trees [156] where every data item touched by a query is returned at the buffer of the LSM- tree. Effectively the idea of splaying organizes data based on a "recently accessed" policy which resembles the behavior of a cache. Indeed, the Splaying LSM-trees work showed that one of the primary benefits comes from the potential to avoid the need for a separate cache, thus minimizing memory requirements.

4

Mapping Access Methods to the Design Space

[Here we want to show how past access method designs can be easily captured but also how this classification can lead to new designs. For a number of classes of data structures (Trees, Tries, Hash index, Bitvectors, Zonemaps) we provide details with respect to how the designs are created when combining design decisions and what is the impact on design trafeoffs.]

[This will have a small overlap with the discussion about the dimensions in the previous paragraph. But we can make sure that the **survey** part is here!]

- Base Data, Data Layouts, and Scans, problem: fast query processing without updates, approach: column layouts+sparse indexing+parallel computation+workload-awareness+information theory, examples: column stores + fast scans + sharing+ cracking+byte slice ...

- B+ Trees, problem: point queries + updates, approach: fractional cascading, content representation, examples:

- LSM ...

- Tries, problem: not rely on dataset size, but fixed access latency, approach: tries, examples: ART
- Bitmap, problem: effcient access in low-cardinality columns, approach: data representation

4.1 Cost Model

[We may want to have a section here that introduces a unified cost model for a subset of data structures – we may reuse material from CIDR]

4.2 Reducing Read Amplification: Data Layouts & Scans

[Decisions about partitioning gives us the basic organization of data and columns. We can store the clustered, ordered, partitioned. Different decisions affect query, insert, and update performance. When combined with adaptivity, columnar storage yielded database cracking [114, 115, 116, 92, 90, 100, 91, 205, 188, 187].]

Challenges. [read (ideally) only relevant data] [what are the problems? point query? updates? distribution?]

Design Abstraction. [basically only data organization] [Here we connect with one of the design options discussed in Section 3 – there is no need to redefine the principle only to mention it and to introduce the abstraction]

4.2.1 Physical Design

[basically whether to sort. with adaptivity we got cracking]

4.2.2 Adaptive Physical Design

The classical access method that applies a physical design decision ("sorted") adaptively is database cracking [114], which sorts a data collection adaptively. Database cracking continuously reorganizes base data to match the query workload. Every query is used as an advice on how the data should be stored. Cracking does this by building and refining indexes partially and incrementally as part of query processing.

By reacting to every single query with lightweight actions, database cracking manages to adapt to a changing workload instantly. The more queries arrive, the more the indices are refined, the more performance improves, and eventually it reaches the optimal performance, i.e., the performance we would get from a manually tuned system.

The main idea in the original database cracking approach is that the data system reorganizes one column of the data at a time and only when touched by a query. In other words, the reorganization utilizes the fact that the data is already read and decides how to refine it in the best way. Effectively the original cracking approach overloads the select operator of a database system and uses the predicates of each query to determine how to reorganize the relevant column. The first time an attribute A is required by a query, a copy of the base column A is created, called the cracker column of A. Each select operator on A triggers the physical reorganization of the cracker column based on the requested range of the query. Entries with a key that is smaller than the lower bound are moved before the lower bound, while entries with a key that is greater than the upper bound are moved after the upper bound in the respective column. The partitioning information for each cracker column is maintained in an AVL- tree, the cracker index. Future queries on column A search the cracker index for the partition where the requested range falls. If the requested key already exist in the index, i.e., if past queries have cracked on exactly those ranges, then the select operator can return the result immediately. Otherwise, the select operator refines on-the-fly the column further, i.e., only the partitions/pieces of the column where the predicates fall will be reorganized (at most two partitions at the boundaries of the range). Progressively the column gets more "ordered" with more but smaller pieces.

Database cracking as a concept has been studied in the context of main-memory column-stores [114, 205]. The cracking algorithms have been adapted to work for many core database architecture issues such as: updates to incrementally and adaptively absorb data changes [115], multi-attribute queries to reorganize whole relations as opposed to only columns [116], to use also the join operator as a trigger for adaptation [110], concurrency control to deal with the problem that cracking effec-

tively turns reads into writes [91, 90], and partition-merge-like logic to provide cracking algorithms that can balance index convergence versus initialization costs [118]. In addition, tailored benchmarks have been developed to stress-test critical features such as how quickly an algorithm adapts [92]. Stochastic database cracking [100] shows how to be robust on various workloads, and [94] shows how adaptive indexing can apply to key columns. Finally, recent work on parallel adaptive indexing studies CPU-efficient implementations and proposes cracking algorithms to utilize multi- cores [11, 188] or even idle CPU time [187].

The database cracking concept has also been extended to provide adaptive indexes for time series data [244, 245] as well as for more broad storage layout decisions, i.e., reorganizing base data (columns/rows) according to incoming query requests [9], or even about which data should be loaded [8, 111]. Cracking has also been studied in the context of Hadoop [197] for local indexing in each node as well as for improving more traditional disk-based indexing which forces reading data at the granularity of pages and where writing back the reorganized data needs to be considered as a major overhead [93].

Adaptive Partitioning and Indexing. Similar to cracking, other systems have identified the upfront loading cost as a big overhead [178]. To address this cost, and support arbitrary queries that might have different nature both as time changes, and for different parts of the indexed files, Slalom proposes a new logical partitioning strategy which creates overlapping physical partitions and builds a different set of indexes for each partitioning depending on the queries that reach the specific partition. After deciding which should the partitions be, Slalom builds Bloom filters, B^+-Trees, Bitmaps, and Zonemaps for each partition depending on the type and the selectivity of the most frequent queries.

Adaptive Data Layouts. The same concept of adaptivity as discussed above for sorting and indexing, can also be applied on base data. That is, base data can change its shape to adapt to the workload. For example, the two fundamental data layouts for relational database systems are column-based and row-based storage. While the columnar layouts are better for more analytical queries and queries that require

few attributes from a table so that projection cost remains low, row-based layouts are better for transactional workloads that touch only few tuples and queries where many attributes are necessary. H_2O first proposed the idea that a system with adaptivity for base data can transition between these two layouts and their hybrids (i.e., organizing data in column groups) [9]. The idea is that the system monitors the workload over a period of time and then it makes a decision based on a model that tracks data accesses. It then materializes the new data layout and continues procession the workload. While H_20 focused on read only workloads, Peloton [15] focuses on both reads and writes while recent work also shows that the decision can be made with genetic algorithms [117]. Adaptive data layouts is orthogonal to any adaptive indexing the system may employ on top, although the two may be co-designed (this is an open topic).

4.2.3 Membership Test Indexes

[If we are scanning for a single value, or a small set of values, we can first have a membership test. This can happen with a whole [43]]

Bloom Filters. Bloom in Oracle: [13] Original paper: [43] Variations: [10, 201, 141, 159]

Cuckoo Filters. [75, 172, 58] Morton: [46]

Quotient Filters. [38, 185]

Zonemaps. [[80, 195, 219, 220, 237] and its variants can provide the information which rows belong to set (e.g., a range).]

Column Imprints. [A variation is Column Imprints [214] which stores a compact histogram per cache line.]

Information-Theoretic Scan Acceleration. [ByteSlice,BitWeaving] [78] [152]

4.3 Efficient Random Access: B$^+$-Trees and Variants

[A wealth of approaches about B-Trees [33, 34, 88, 89, 147] take into account the aforementioned dimensions one by one. With regards to the design dimensions B-Trees start as a combination of range partitioning and

logarithmic design. When adding the log-structured updates a new group of designs of variants of B-Trees was introduced. Typically these designs were also storage-friendly and memory-friendly [150, 127, 176]. – let's also classify [221, 126, 122] –Buffering is another aspect that augmented the design space of B-Trees. In particular, there are Tree index structures that buffer reads: Fractal Tree/BRT/COLA [36], LA-Tree, [4], IPL [146], BFTL [232], FlashDB [177] ADS [244]. Another approach is to buffer updates. This happens in a number of designs: IPLB [176], LA-Tree [4], and Positional Delta Tree [107]. Finally, another approach is to buffer requests and take into account the opportunity to execute in parallel some requests [199, 242, 14]. Another design dimension used by PDT, IPLB, LA-Tree and PBT [86] is differential updates. Finally, when combined with sparse indexing tree indexing resulted in approximate tree indexing: BF-Tree [19]. Bounded disorder access methods [155, 157] introduced the concept of a balance between range and hash partitioning which can happen with an tree-like index with hashing on the leaves. When B-Trees were combined with adaptivity we got adaptive indexing [118, 90, 91].]

[my current understanding is that we need subsection here about flash-friendly b-tree approaches where we would list how all different decisions would affect convert the trees to flash friendly]

Challenges. [quickly locate data we are looking for. update in place.]

Design Abstraction. [data organization + index + buffer +update-pocy + read-adaptivity for adaptive indexing]

4.3.1 B$^+$-Tree Designs

4.4 Reducing Write Amplification: LSM Trees and Variants

[The cornerstone idea is Differential files [209], and then the LSM paper [182]]

[LSM Trees [182] were proposed as a way to amortize the number of physical inserts in a write-intensive workload. They achieve that by naturally buffering updates in a first level and pushing them as a group to lower levels (each level is as a result an epoch of updates). Several variations were proposed hereafter [213, 235] that optimize specific aspects of the searching. With regards to the design dimensions LSM-Trees start as a

combination of time partitioning and logarithmic design. When combining LSM-Trees with fractional cascading as a design dimension new designs were proposed: FD-Tree[151], bLSM[206].

More about LSM:

LSM-Trie [235] merges less greedily by using T substructures in each level. That way it reduces write-amplification by a factor of T, at the expense of losing the ability to do range-queries. BigTable uses LSM-Trees with a fixed size ratio equal to ten between adjacent levels [55]. Bigtable also uses Bloom filters to avoid reading unnecessary levels. These Bloom filters are tuned statically and uniformly across levels. In addition, Cassandra [138], LOCS [226], VT-Tree [213], and bLSM [206] build on top of BigTable, LevelDB, and LSM-Trees to provide logarithmic access method design. LOCS [226] is a version of LevelDB that utilizes SSD parallelism. VT-Tree [213] avoids repeatidly writing sorted data by stitching non-overlapping runs. bLSM [206] restricts the number of levels, restricting read and write performance irrespectively of the workload.]

[and variants like stepped merge [121] and MaSM [24, 25] use time partitioning (e.g., epochs) and partially range partitioning (e.g., within each epoch) and combine this with native log-structured updates and buffering.]

Challenges. [initially random writes in HDD, also write-amplification, fragmentation due to splitting, space amplification due to fragmentation]

Design Abstraction. [data organization + metata + out-ofplace updates + mention that adaptivity is something discussed more work to be done]

4.4.1 Leveled LSM Designs

4.4.2 Tiered LSM Designs

4.4.3 Hybrid LSM Designs

4.4.4 Hash-based LSM Designs

4.5 Balance Space and Read: Tries and Variants

[Tries were introduced to decouple search time from dataset size (and couple it with the domain) [81, 175]. More recently a hardware-aware

implementation was proposed [149]. With regards to the design dimensions Tries start as a combination of radic partitioning and logarithmic design.]

Challenges.

Design Abstraction.

4.5.1 Trie Index Designs

4.6 Reduce Space Amplification: Bitmap Indexing

Challenges.

Design Abstraction.

4.6.1 Bitmap Index Designs

[A different category of data structures used for indexing is bitvectors, forming a bitmap index, by encoding each value of the domain with a bitvector which has one bit per row of the relation [231, 180, 183, 52, 53]. Bitmap Indexing is commonly used for a number of applications ranging from scientific data management [233] to analytics and data warehousing [56, 163, 202, 217, 218, 234]. Bitmap indexes are utilized by several popular database systems, including open-source systems like PostgreSQL and commercial systems like Oracle [210], SybaseIQ [163, 183], and DB2 [50]. When combined with buffering, differential updates, and adaptivity this resulted in the design of Upbit [28].]

4.7 Making Access Methods Hardware-Friendly

[Here we explain that some of the dimensions are typically used to make an access method hardware friendly and we will give some typical examples like: [107, 24, 19, 25, 151, 153, 93, 86, 199]]

4.8 Other Access Methods

[There are more access methods. Here we focus on mature classes of access methods that are used as integral parts of storage managers. However, if we dive in the details of data structures and in-memory data structures

there are more options. Two popular examples are Hash Indexes and Skip Lists, but also variations of trees, like binary trees, avl trees, red-black trees, and so on.]

Hash Indexes. [A way to organize and index data is through hashing. Here we will make a brief mention of hash indexing.]

Skip Lists. [Here we shall discuss the concept of skip lists [190] and the more recent variant that is lock free [79]]

dimensions → **design options** **per dimension →** **(RUM impact)** **access methods↓**	*Partitioning* *range (sorted)* (↓↑-) *radix/hash* (↓↑-) *general* (↓↑-) *epoch* (↑↓-)	*Searching metadata* *logarithmic-value* (↓↑↑) *logarithmic-radix* (↓↑-) *filters (BF, ZM, CI)* (↓↑↑) *fractional cascading* (↓↑↑)	*Update policy* *in-place (IP)* (↓↑-) *out-of-place (OOP)* (↑↓-)	*Contents* *RID* (↑-↓) *pointer* (↑-↓) *bitvector* (↓↑↓) *datum* (↓↓↑)	*Buffering* *reads* (↓-↑) *global updates* (-↓↑) *partition updates* (-↓↑)	*Recursion* *≠ (degree)*	*Caching* *of reads* (↓-↑)	*Adaptivity* *read-driven*
B$^+$-Tree [33, 34, 63, 89] **Partitioned B-Tree** [86]	range	log-val	IP	RID/ptr/datum	–	1	–	–
SB-Tree [181]	range	log-val	IP	RID/datum	–	1	–	–
fpB$^+$-Tree [59, 60]	range	log-val	IP	RID/ptr/datum	–	2	–	–
CSB-Tree [196]	range	log-val/frac. casc.	IP	RID/ptr/datum	–	1	–	–
Masstree [168]	radix/range	log-val/log-radix	IP	ptr/datum	–	2	–	–
BF-Tree [19]	range	log-val/filters (BF)	IP	RID/datum	–	1	–	–
Bw-Tree [150]	range	log-val	OOP	ptr/datum	partition updates	1	–	–
LA-Tree [4]	range	log-val	IP	datum	glob./part. updates	1	–	×
BRTree [49] **B$^\epsilon$-Tree** [37, 47, 136] **COLA** [36]	range	log-val	IP	ptr/datum	partition updates	1	–	–
PDT [107]	range/sorted	log-val	IP	RID	–	2	–	–
Array	–	–	IP	datum	–	1	–	–
Logging	epoch	–	OOP	datum	–	1	–	–
Cracking [114]	range	log-val	IP	RIDS	–	1	–	× (sorting)
Adaptive Merging [94]	range	log-val	IP	RIDS	–	1	–	× (merging)
Cracking w/ upd. [115]	range	log-val	IP	RIDS	partition updates	1	–	×
Hashing	hash	–	IP	RID/ptr/datum	–	1	–	–
Bitmap [52]	range/general	–	IP	bitvector	–	1	–	–
UpBit [28]	range/general	–	OOP	bitvector	partition updates	1	–	×
LSM [182]	epoch/sorted	–	OOP	RID/datum	global-updates	1	–	–
LSM (w/ B-Trees)[182]	epoch/sorted	log-val	OOP	RID/datum	global-updates	1	–	–
LSM [67]	epoch/sorted	filters (ZM, BF)	OOP	RID/datum	global-updates	1	–	–
LSM (levelDB)	epoch/sorted	filters (ZM, BF)	OOP	RID/datum	global-updates	1	–	×
bLSM [206]	epoch/sorted	log-val/filters (BF)	OOP	datum	partition updates	1	–	–
LSM-Trie [235]	epoch/hash	log-radix/filters (BF)	OOP	datum	partition updates	1	–	–
VT-Tree [213]	sorted/epoch	filters (QF)	OOP	datum	–	1	–	–
SILT [153]	epoch/hash/radix	log-radix	OOP	datum	–	3	–	–
FD-Tree [151]	range/sorted	log-val/frac. casc.	OOP	datum	–	1	–	–
MaSM [24, 25]	epoch/sorted	filters (ZM)	OOP	datum	global updates	1	–	–
Stepped-Merge [121]	epoch/sorted	log-val	OOP	datum	global updates	1	–	–
Stepped-Hash [121]	epoch/hash	–	OOP	datum	global updates	1	–	–
Skip List [190]	range	log-val (probabilistic)	IP	datum	–	1	–	–
Trie [81, 175]	radix	log-radix	IP	RID/ptr/datum	–	1	–	–
Prefix B-Tree [35]	range	log-radix	IP	RID/ptr/datum	–	1	–	–
ART [149]	radix	log-radix	IP	ptr/datum	–	4	–	–

Table 4.1: Design space and classification of access methods.

5

Design Opportunities

[Here we can specifically focus on the opportunities that are shown through the "design table" for new designs - and also mention some of out latest results. The idea is to showcase that (1) there is room for more research and (2) our classification helps to pinpoint to research directions.]

[This is kept as a separate section because it seems that it can be interesting for the reader to have easy access. If it is too small we can connect it back to the previous section]

The final part of the survey presents open problems. A recurring theme of this survey is that by studying and classifying access method designs we observe that there are strategies for achieving a specific optimization goal, e.g., optimize for read performance. In fact there are two fundamentally different ways to reach a static point in the read/update/memory design space: either by a static design, or by an adaptive design which eventually leads to the desired design point. A number of new research directions build upon this concept. For example, *how can we build access methods that support a dynamic optimization goal, which is reached either by a static or an adaptive to the workload design [27, 129]?* In addition, *can we build an "access method design optimizer" to help us choose the right access method given specific*

hardware and workload properties [27]? Similar directions are taken at the system level. For example, *can we build a declarative storage engine that does not have to chose between being either row-oriented or column-oriented [65, 72]?* In the final part of the tutorial, we discuss these open problems and we highlight opportunities for innovation.

5.1 The Design Space As A Design Advisor

[here we can discuss about the idea that one can use the clustering for structured design, i.e., to create tools etc that use this classification to help with design]

5.2 A Richer Design Space

[include additional dimensions that can be considered for the tradeoffs and have a more holistic discussion about future steps in further refining the design space and the analysis/classification further highlighting that this is going to be a constant effort]

[Are these 7+1 all the dimensions we would ever need? Are there more dimensions? The answer is that probably the design space would be more fine-grained in the future, so here we discuss a few potential new dimensions and introduce the concept of a modular design space (in addition to a modular access method design).]

5.2.1 Concurrency

[Concurrency can be discussed as a new dimension. Here we will discuss as an open research questions that if there is a way to add concurrency ad hoc to any design (or virtually any design) then we can definitely argue that concurrency is a new dimension. Here we can present a few concepts coming out of ongoing ideas.]

Let the number of items in a node be NodeItems. Let the total number of items be TotItems. Let the number of operations (insert, delete, update, search) that enter per second be NumProc. In a tree-like structure, each leaf node is hit by NumProc * NodeItems/TotItems per second. In a Log Structured Merge Tree (LSM) like structure, the root

node is hit NumProc times per second. Denote that as ProcPerNode.

The basic unit of cost of concurrency control is the time one process prevents other processes from accessing nodes times the number of such nodes. Holding a latch on a node for LatchTime thus costs LatchTime * max(0, (ProcPerNode - 1)). The subtraction of 1 is due to the fact that one process (the one holding the latch) locks the node. The max is due to the fact that the cost cannot be negative.

Further, the maximum throughput per node is 1/LatchTime. So if ProcPerNode > (1/LatchTime), then latching will be the bottleneck.

In a tree structure, holding a latch on a parent node having a fanout of f would have a cost of LatchTime * max(0, (f*ProcPerNode) - 1)). Denote this as ParentCost. Now let's consider a situation where we latch the parent and the child for a split (lock-coupling). If the time to modify the child is T1 and the time to modify the parent is T2 (normally T1 > T2, because data must be moved at the child level whereas only a pointer needs to be added at the parent level) then the total concurrency cost is (T1 + T2) * ParentCost. If the two latches occur separately, then the cost is LeafCost*T1 + T2*ParentCost. Thus, it is always better to lock just the two separately.

The downside to doing such decoupled locking is that certain searches will have to do extra work. Using a link technique, the extra work is (almost always) traversing a single link (though more is possible). Using a give-up technique, the extra work is (again, almost always) going back to the parent and then going to the next node. This happens only in the case that data has been moved from a node which is rare enough that it can be ignored in the average case.

We can use this model to compare data structures as well. For example, in a log-structured merge tree, the latch time per insert, delete, or update (summarized as an upsert for LSM trees) is short, because normally each upsert modifies only the in-memory root node. Nevertheless, the total throughput per second is bounded by 1/LatchTime if there is a single root node, because each upsert must access the root by itself. If that throughput is insufficient for the incoming traffic, then we might want a hybrid structure, e.g. a hash function that routes to k roots, for some number k, so that the single LSM tree becomes k LSM

trees.

5.2.2 Distributed Systems

[Here we discuss the question of distributing the system and the access method to multiple nodes. While this seems like an expected addition to the roster of dimensions the argumentation and modeling would require to add more levels of abstraction in order to include network latency and other costs that do not exist in the]

5.2.3 Privacy

[Privacy can be a new dimension] [Here we should actually connect with the delete element and the right-to-be-forgotten]

5.2.4 Caching: Using Space for Cheaper Reads

Caching: orthogonally to the buffering policies, access methods use additional space for caching of recent reads. This allows the access methods to facilitate faster reads at the expense of additional space.

5.2.5 Hardware-Oblivious Designs

[Cache-oblivious designs aim at removing from the programmer the need to take into account specific characteristics of the underlying memory (cache, in particular) hierarchy. There are several examples of interesting pieces of work towards this direction, however, it is a common intuition that cache-oblivious solutions succeed in having approaches with lower asymptotic complexity at the expense of higher constants which oftentimes makes them less practical. Here we will survey such cache-oblivious solutions and expand on the aforementioned tradeoff.]

6

Summary

[Since we already have a section for open research based on dimensions this should be a summary and outlook with higher-level open research problems.]

References

[1] D. J. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] G. Adelson-Velsky and E. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[3] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *Proceedings of the VLDB Endowment*, 2(1):361–372, 2009.

[5] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 169–180, 2001.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 266–277, 1999.

[8] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB in action: adaptive query processing on raw data. *Proceedings of the VLDB Endowment*, 5(12):1942–1945, 2012.

[9] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1103–1114, 2014.

[10] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable Bloom Filters. *Information Processing Letters*, 101(6):255–261, mar 2007.

[11] V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter. Main Memory Adaptive Indexing for Multi-Core Systems. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 3:1—-3:10, 2014.

[12] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, 2009.

[13] C. Antognini. Bloom Filters in Oracle DBMS. *Oracle White Paper*, 2008.

[14] L. Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica*, 37(1):1–24, 2003.

[15] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 583–598, 2016.

[16] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 519–530, 2010.

[17] M. M. Astrahan, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, V. Watson, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, and P. R. McJones. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[18] M. Athanassoulis. *Solid-State Storage and Work Sharing for Efficient Scaleup Data Analytics*. PhD thesis, EPFL, 2014.

[19] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. *Proceedings of the VLDB Endowment*, 7(14):1881–1892, 2014.

[20] M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. Flash in a DBMS: Where and How? *IEEE Data Engineering Bulletin*, 33(4):28–34, 2010.

[21] M. Athanassoulis, B. Bhattacharjee, M. Canim, and K. A. Ross. Path Processing using Solid State Storage. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 23–32, 2012.

[22] M. Athanassoulis, B. Bhattacharjee, M. Canim, and K. A. Ross. Querying Persistent Graphs using Solid State Storage. In *Proceedings of the Annual Non-Volatile Memories Workshop (NVMW)*, 2013.

[23] M. Athanassoulis, K. S. Bøgh, and S. Idreos. Optimal Column Layout for Hybrid Workloads. *Proceedings of the VLDB Endowment*, 12(13):2393–2407, 2019.

[24] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 865–876, 2011.

[25] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. *ACM Transactions on Database Systems (TODS)*, 40(1), 2015.

[26] M. Athanassoulis and S. Idreos. Design Tradeoffs of Data Access Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, 2016.

[27] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 461–466, 2016.

[28] M. Athanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016.

[29] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 123–136, 2009.

[30] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business Analytics in (a) Blink. *IEEE Data Engineering Bulletin*, 35(1):9–14, 2012.

[31] R. Barber, G. M. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer. In-Memory BLU Acceleration in IBM's DB2 and dashDB: Optimized for Modern Workloads and Hardware Architectures. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2015.

[32] R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.

[33] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141, 1970.

[34] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1(3):173–189, 1972.

[35] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems (TODS)*, 2(1):11–26, 1977.

[36] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-Oblivious Streaming B-trees. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 81–92, 2007.

[37] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. An Introduction to Bϵ-trees and Write-Optimization. *White Paper*, 2015.

[38] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[39] J. L. Bentley. Decomposable Searching Problems. *Information Processing Letters*, 8(5):244–251, 1979.

[40] J. L. Bentley and J. B. Saxe. Decomposable Searching Problems I: Static-to-Dynamic Transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[41] J. L. Bentley and A. C.-C. Yao. An Almost Optimal Algorithm for Unbounded Searching. *Information Processing Letters*, 5(3):82–87, 1976.

[42] W. A. Bhat. Bridging data-capacity gap in big data storage. *Future Generation Computer Systems*, 2018.

[43] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[44] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, 1999.

[45] P. A. Boncz, M. Zukowski, and N. J. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[46] A. Breslow and N. Jayasena. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.

[47] G. S. Brodal and R. Fagerberg. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, 2003.

[48] A. Z. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1:636–646, 2002.

[49] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On External Memory Graph Traversal. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 859–860, 2000.

[50] M. Cain and K. Milligan. IBM DB2 for i indexing methods and strategies. *IBM White Paper*, 2011.

[51] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment*, 2(1):277–288, 2009.

[52] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *ACM SIGMOD Record*, 27(2):355–366, 1998.

[53] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *ACM SIGMOD Record*, 28(2):215–226, 1999.

[54] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 275–290, 2018.

[55] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[56] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[57] B. Chazelle and L. J. Guibas. Fractional Cascading: A Data Structuring Technique with Geometric Applications. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 90–100, 1985.

[58] H. Chen, L. Liao, H. Jin, and J. Wu. The Dynamic Cuckoo Filter. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[59] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving Index Performance through Prefetching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–246, 2001.

[60] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2002.

[61] S. Chen, P. B. Gibbons, and S. Nath. Rethinking Database Algorithms for Phase Change Memory. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[62] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[63] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[64] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–279, 1985.

[65] P. Cudré-Mauroux, E. Wu, and S. Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.

[66] Z. J. Czech, G. Havas, and B. S. Majewski. An Optimal Algorithm for Generating Minimal Perfect Hash Functions. *Information Processing Letters*, 43(5):257–264, 1992.

[67] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–94, 2017.

[68] N. Dayan, M. Athanassoulis, and S. Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)*, 43(4):16:1–16:48, 2018.

[69] N. Dayan and S. Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–520, 2018.

[70] B. Debnath, S. Sengupta, and J. Li. FlashStore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[71] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 25–36, 2011.

[72] J. Dittrich and A. Jindal. Towards a One Size Fits All Database Architecture. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 195–198, 2011.

[73] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[74] A. D. Falkoff and K. E. Iverson. The Design of APL. *IBM Journal of Research and Development*, 17(5):324–334, 1973.

[75] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*, pages 75–88, 2014.

[76] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database: Data Management for Modern Business Applications. *ACM SIGMOD Record*, 40(4):45–51, 2011.

[77] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

[78] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 31–46, 2015.

[79] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 50–59, 2004.

[80] P. Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks*, 2011.

[81] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[82] C. D. French. "One size fits all" database architectures do not work for DSS. *ACM SIGMOD Record*, 24(2):449–450, 1995.

[83] C. D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 194–198, 1997.

[84] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1189–1206, 2019.

[85] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing One Thousand Queries with One Stone. *Proceedings of the VLDB Endowment*, 5(6):526–537, 2012.

[86] G. Graefe. Sorting And Indexing With Partitioned B-Trees. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[87] G. Graefe. The five-minute rule twenty years later. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, page 6, 2007.

[88] G. Graefe. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)*, 35(3), 2010.

[89] G. Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[90] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *Proceedings of the VLDB Endowment*, 5(7):656–667, 2012.

[91] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *The VLDB Journal*, 23(2):303–328, 2014.

[92] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *Proceedings of the TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems (TPCTC)*, pages 169–184, 2010.

[93] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 371–381, 2010.

[94] G. Graefe and H. A. Kuno. Adaptive indexing for relational keys. In *Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 69–74, 2010.

[95] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch. In-Memory Performance for Big Data. *Proceedings of the VLDB Endowment*, 8(1):37–48, 2014.

[96] J. Gray and F. Putzolu. The 5 Minute Rule for Trading Memory for Disc Accesses and the 5 Byte Rule for Trading Memory for CPU Time. *Tandem Computers - Technical Report*, 1986.

[97] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A Main Memory Hybrid Storage Engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010.

[98] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[99] B. Haeupler, S. Sen, and R. E. Tarjan. Rank-Balanced Trees. *ACM Transactions on Algorithms (TALG)*, 11(4):30:1—-30:26, 2015.

[100] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 5(6):502–513, 2012.

[101] S. Harizopoulos and A. Ailamaki. A Case for Staged Database Systems. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[102] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394, 2005.

[103] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a Model of Indexability and Its Bounds for Range Queries. *Journal of the ACM*, 49(1):35–55, 2002.

[104] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the Analysis of Indexing Schemes. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 249–256, 1997.

[105] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 562–573, 1995.

[106] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[107] S. Héman, M. Zukowski, and N. J. Nes. Positional Update Handling in Column Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 543–554, 2010.

[108] M. Hilbert and P. López. The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(6025):60–65, 2011.

[109] J. Hughes. Revolutions in Storage. In *IEEE Conference on Massive Data Storage*, Long Beach, CA, may 2013.

[110] S. Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, University of Amsterdam, 2010.

[111] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 57–68, 2011.

[112] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[113] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[114] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[115] S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–424, 2007.

[116] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–308, 2009.

[117] S. Idreos, L. M. Maas, and M. S. Kester. Evolutionary Data Systems. *CoRR*, abs/1706.0, 2017.

[118] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 4(9):586–597, 2011.

[119] S. Idreos, K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. The Periodic Table of Data Structures. *IEEE Data Engineering Bulletin*, 41(3):64–75, 2018.

[120] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 535–550, 2018.

[121] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 16–25, 1997.

[122] R. Jin, S. J. Kwon, and T.-S. Chung. FlashB-tree: A Novel B-tree Inex Scheme for Solid State Drives. In *Proceedings of the ACM Symposium on Research in Applied Computation (RACS)*, pages 50–55, 2011.

[123] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To Share or Not to Share? In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 351–362, 2007.

[124] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 24–35, 2009.

[125] T. Johnson and D. Shasha. Utilization of B-trees with Inserts, Deletes and Modifies. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 235–246, 1989.

[126] M. V. Jorgensen, R. B. Rasmussen, S. Šaltenis, and C. Schjonning. FB-tree: a B+-tree for flash-based SSDs. In *Proceedings of the Symposium on International Database Engineering & Applications (IDEAS)*, pages 34–42, 2011.

[127] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim. mu-tree: an ordered index structure for NAND flash memory. In *Proceedings of the ACM/IEEE International Conference on Embedded Software (EMSOFT)*, pages 144–153, 2007.

[128] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, 2011.

[129] O. Kennedy and L. Ziarek. Just-In-Time Data Structures. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[130] M. S. Kester, M. Athanassoulis, and S. Idreos. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 715–730, 2017.

[131] W. Kim, K.-C. Kim, and A. G. Dale. Indexing Techniques for Object-Oriented Databases. In *Object-Oriented Concepts, Databases, and Applications*, pages 371–394. ACM Press and Addison-Wesley, 1989.

[132] G. D. Knott. Hashing Functions. *The Computer Journal*, 18(3):265–278, 1975.

[133] I. Koltsidas and S. D. Viglas. Data management over flash memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1209–1212, 2011.

[134] A. G. Konheim. *Hashing in Computer Science: Fifty Years of Slicing and Dicing*. John Wiley & Sons, Inc., 2010.

[135] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 489–504, 2018.

[136] B. C. Kuszmaul. A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. *Tokutek White Paper*, 2014.

[137] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle Database In-Memory: A Dual Format In-Memory Database. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2015.

[138] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[139] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, 1991.

[140] A. Lamb, M. Fuller, and R. Varadarajan. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[141] H. Lang, T. Neumann, A. Kemper, and P. A. Boncz. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *Proceedings of the VLDB Endowment*, 12(5):502–515, 2019.

[142] P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL server column store indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1177–1184, 2011.

[143] P.-A. Larson, E. N. Hanson, and S. L. Price. Columnar Storage in SQL Server 2012. *IEEE Data Engineering Bulletin*, 35(1):15–20, 2012.

[144] P.-A. Larson, E. N. Hanson, and M. Zwilling. Evolving the Architecture of SQL Server for Modern Hardware Trends. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2015.

[145] P.-A. Larson, R. Rusanu, M. Saubhasik, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, and S. Rangarajan. Enhancements to SQL server column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1159–1168, 2013.

[146] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 55–66, 2007.

[147] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[148] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 294–303, 1986.

[149] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.

[150] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 302–313, 2013.

[151] Y. Li, B. He, J. Yang, Q. Luo, K. Yi, and R. J. Yang. Tree Indexing on Solid State Drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.

[152] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013.

[153] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13, 2011.

[154] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-Tree: An Index Structure for High-Dimensional Data. *The VLDB Journal*, 3(4):517–542, 1994.

[155] W. Litwin and D. B. Lomet. The Bounded Disorder Access Method. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 38–48, 1986.

[156] T. Lively, L. Schroeder, and C. Mendizábal. Splaying Log-Structured Merge-Trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1839–1841, 2018.

[157] D. B. Lomet. A simple bounded disorder file organization with good performance. *ACM Transactions on Database Systems (TODS)*, 13(4):525–551, 1988.

[158] D. B. Lomet and B. Salzberg. The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Transactions on Database Systems (TODS)*, 15(4):625–658, 1990.

[159] G. Lu, B. Debnath, and D. H. C. Du. A Forest-structured Bloom Filter with flash memory. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, 2011.

[160] C. Luo and M. J. Carey. LSM-based Storage Techniques: A Survey. *CoRR*, abs/1812.0, 2018.

[161] C. Luo and M. J. Carey. LSM-based Storage Techniques: A Survey. *The VLDB Journal*, 2019.

[162] G. Lustiber. E-Tree: An Ever Evolving Tree for Evolving Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Student Research Competition*, pages 13–15, 2017.

[163] R. MacNicol and B. French. Sybase IQ Multiplex - Designed For Analytics. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1227–1230, 2004.

[164] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 191–202, 2002.

[165] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, 2002.

[166] S. Manegold, P. A. Boncz, and N. Nes. Cache-Conscious Radix-Decluster Projections. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 684–695, 2004.

[167] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Advanced Information and Knowledge Processing. Springer, 2006.

[168] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 183–196, 2012.

[169] M. R. Mediano, M. A. Casanova, and M. Dreux. V-Trees - A Storage Method for Long Vector Data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 321–330, 1994.

[170] M. Mehta, V. Soloviev, and D. J. DeWitt. Batch Scheduling in Parallel Database Systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 400–410, apr 1993.

[171] D. S. Milojicic and T. Roscoe. Outlook on Operating Systems. *IEEE Computer*, 49(1):43–51, 2016.

[172] M. Mitzenmacher, S. Pontarelli, and P. Reviriego. Adaptive Cuckoo Filters. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47, 2018.

[173] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 1998.

[174] C. Mohan. Tutorial: An In-Depth Look at Modern Database Systems. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, page 674, 2014.

[175] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[176] G.-J. Na, B. Moon, and S.-W. Lee. IPLB+-tree for Flash Memory Database Systems. *Journal of Information Science and Engineering (JISE)*, 27(1):111–127, jan 2011.

[177] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. *ACM/IEEE IPSN*, 2007.

[178] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing. *Proceedings of the VLDB Endowment*, 10(10):1106–1117, 2017.

[179] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 183–191, 1999.

[180] P. E. O'Neil. Model 204 Architecture and Performance. In *Proceedings of the International Workshop on High Performance Transaction Systems (HPTS)*, pages 40–59, 1987.

[181] P. E. O'Neil. The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. *Acta Informatica*, 29(3):241–265, 1992.

[182] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[183] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. *ACM SIGMOD Record*, 26(2):38–49, 1997.

[184] M. H. Overmars and J. van Leeuwen. Worst-Case Optimal Insertion and Deletion Methods for Decomposable Searching Problems. *Information Processing Letters*, 12(4):168–173, 1981.

[185] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 775–787, 2017.

[186] Y. Perl, A. Itai, and H. Avni. Interpolation Search—A log logN Search. *Communications of the ACM*, 21(7):550–553, 1978.

[187] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

[188] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. L. Kersten. Database cracking: fancy scan, not poor man's sort! In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 1–8, 2014.

[189] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing Data and Work Across Concurrent Analytical Queries. *Proceedings of the VLDB Endowment*, 6(9):637–648, 2013.

[190] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.

[191] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory Scan Sharing for Multi-core CPUs. *Proceedings of the VLDB Endowment*, 1(1):610–621, 2008.

[192] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2002.

[193] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 430–441, 2002.

[194] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. *The VLDB Journal*, 12(2):89–101, 2003.

[195] V. Raman, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, L. Zhang, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, and S. Liu. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

[196] J. Rao and K. A. Ross. Making B+-trees Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 475–486, 2000.

[197] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in Hadoop. *The VLDB Journal*, 23(3):469–494, 2013.

[198] J. T. Robinson. The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[199] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.

[200] K. A. Ross. Selection Conditions in Main Memory. *ACM Transactions on Database Systems (TODS)*, 29:132–161, 2004.

[201] C. E. Rothenberg, C. Macapuna, F. Verdi, and M. Magalhaes. The deletable Bloom filter: a new member of the Bloom family. *IEEE Communications Letters*, 14(6):557–559, jun 2010.

[202] P. Russom. High-Performance Data Warehousing. *TDWI Best Practices Report*, 2012.

[203] H. W. Scholten and M. H. Overmars. General Methods for Adding Range Restrictions to Decomposable Searching Problems. *Journal of Symbolic Computation*, 7(1):1–10, 1989.

[204] F. M. Schuhknecht, J. Dittrich, and L. Linden. Adaptive Adaptive Indexing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 665–676, 2018.

[205] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *Proceedings of the VLDB Endowment*, 7(2):97–108, 2013.

[206] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[207] R. Sedgewick. Balanced Trees. In *Algorithms*. Addison-Wesley, 1983.

[208] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[209] D. G. Severance and G. M. Lohman. Differential files: their application to the maintenance of large databases. *ACM Transactions on Database Systems (TODS)*, 1(3):256–267, 1976.

[210] V. Sharma. Bitmap Index vs. B-tree Index: Which and When? *Oracle White Paper*, 2005.

[211] J. Sheehy and D. Smith. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. *Basho White Paper*, 2010.

[212] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner. United States Data Center Energy Usage Report. *Ernest Orlando Lawrence Berkeley National Laboratory*, LBNL-10057, 2016.

[213] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 17–30, 2013.

[214] L. Sidirourgos and M. L. Kersten. Column Imprints: A Secondary Index Structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 893–904, 2013.

[215] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[216] Spectra. Digital Data Storage Outlook. *Spectra White Paper*, 2017.

[217] K. Stockinger. Bitmap Indices for Speeding Up High-Dimensional Data Analysis. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pages 881–890, 2002.

[218] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[219] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained Partitioning for Aggressive Data Skipping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1115–1126, 2014.

[220] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented Partitioning for Columnar Layouts. *Proceedings of the VLDB Endowment*, 10(4):421–432, 2016.

[221] R. Thonangi, S. Babu, and J. Yang. A Practical Concurrent Index for Solid-State Drives. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1332–1341, 2012.

[222] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. *Proceedings of the VLDB Endowment*, 2(1):706–717, 2009.

[223] J. van Leeuwen and M. H. Overmars. The Art of Dynamizing. In *Proceedings of Mathematical Foundations of Computer Science*, pages 121–131, 1981.

[224] J. van Leeuwen and D. Wood. Dynamization of Decomposable Searching Problems. *Information Processing Letters*, 10(2):51–56, 1980.

[225] P. Van Sandt, Y. Chronis, and J. M. Patel. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 36–53, 2019.

[226] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 16:1–16:14, 2014.

[227] Z. Wei, K. Yi, and Q. Zhang. Dynamic external hashing: the limit of buffering. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 253–259, 2009.

[228] P. R. Wilson. Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware. *SIGARCH Computer Architecture News*, 19(4):6–13, 1991.

[229] WiredTiger. Source Code. *https://github.com/wiredtiger/wiredtiger*.

[230] C. K. Wong and M. C. Easton. An Efficient Method for Weighted Sampling Without Replacement. *SIAM Journal on Computing (SICOMP)*, 9(1):111–113, 1980.

[231] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit Transposed Files. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 448–457, 1985.

[232] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3), 2007.

[233] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. J. Otoo, V. Perevoztchikov, A. Poskanzer, O. Rübel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W.-M. Zhang. FastBit: interactively searching massive data. *Journal of Physics: Conference Series*, 180(1):012053, 2009.

[234] M.-C. Wu and A. P. Buchmann. Encoded Bitmap Indexing for Data Warehouses. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 220–230, 1998.

[235] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 71–82, 2015.

[236] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[237] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2013.

[238] A. C. Yao. Space-Time Tradeoff for Answering Range Queries (Extended Abstract). In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*, pages 128–136, 1982.

[239] K. Yi. Dynamic indexability and lower bounds for dynamic one-dimensional range query indexes. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 187–196, 2009.

[240] K. Yi. Dynamic Indexability and the Optimality of B-Trees. *Journal of the ACM*, 59(4):21:1—-21:19, 2012.

[241] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1567–1581, 2016.

[242] J. Zhou and K. A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 405–416, 2003.

[243] J. Zhou and K. A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 191–202, 2004.

[244] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1555–1566, 2014.

[245] K. Zoumpatianos, S. Idreos, and T. Palpanas. ADS: the adaptive data series index. *The VLDB Journal*, 25(6):843–866, 2016.

[246] K. Zoumpatianos and T. Palpanas. Data Series Management: Fulfilling the Need for Big Sequence Analytics. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2018.

[247] M. Zukowski and P. A. Boncz. Vectorwise: Beyond Column Stores. *IEEE Data Engineering Bulletin*, 35(1):21–27, 2012.

[248] M. Zukowski, S. Héman, N. J. Nes, and P. A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 723–734, 2007.

Appendices

Dennis Suggestion

Access methods support the dictionary operations of:

Point Search – given a key value k, find the value associated with k.

Range Search – given a range of keys k1 and k2 with k1 *le* k2, find all the key-value pairs for keys k between k1 and k2 inclusive.

Update – given a key k and value v, replace the value of k in the data structure by v.

Insert – insert a key k and value v into the structure if k is not already present.

[and maybe bulk load]

Delete – given a key k, remove k and any associated value in the data structure.

We measure performance by average case time complexity. With large Random Access Memories (RAMs) and much larger secondary storage memories, space is less of a consideration unless it impacts on time. This impact may manifest for example if an access method has so much space overhead that it pushes data from a faster memory (e.g. cache) to a slower memory (e.g. RAM).

The extreme differences in speed in the memory hierarchy suggest

that the only important cost is the cost at the slowest level of that hierarchy. For example, if a tree-based access method stores its first few levels in RAM and the last levels on secondary storage, all that matters is the cost at those last levels.

.0.1 Inter-Block Organizations

Array – contiguous sequence of blocks in some combination of primary or secondary memory. Arrays can be sorted or unsorted.

Linked list – a set of blocks, linked together by pointers. Linked lists can be sorted or unsorted.

Logarithmic tree structure – a set of blocks with a root in a tree-like structure. Suppose that for block b, the lowest key is denoted low(b) and the greatest key is denoted high(b). In this structure high(b) $<$ low(b') where b' is to the right of b. Major parameters are block size and fanout and whether the tree is balanced or not.

Hash structure – Usually, this means open hashing in which a key is mapped by a hash function into a target value and some portion of that target value is used as an address leading to a linked list of blocks.

Trie (each node is a block) – [I'm thinking this is a special case of a logarithmic tree structure for our purposes]

Multi-copy structure e.g. LSM and BW-tree (each node is a block) – Structures in which the same key k can appear multiple times and where the currently valid value associated with a key k (i.e. the logical value associated with k) is the value associated with the last insert on k.

.0.2 Time Measurement for Block Organization

We partition the time into secondary memory cost, RAM cost, and cache cost. Cache cost applies only to within block accesses. If the size of the data structure is larger than the size of RAM, then secondary memory cost dominates. Otherwise, RAM cost dominates.

In what follows, we assume we are storing N items each of size s in bytes. Available memory space of this data structure M bytes Each block has b bytes. Each range query requires r items.

Unordered Array structures

If N * s > M, then on average, each point search, insert, delete, and update will require secondary memory cost: ((N*s) - M)/2b block accesses, because one will have to search half the blocks in secondary memory. Range queries have a far greater chance to require scanning the entire structure, because there is no order, so the secondary memory cost is likely to be (((N*s) - M)/b) block accesses.

If N*s <= M, then each each point search, insert, delete, and update will require RAM cost: ((N*s)/2b) block accesses. Range queries will require RAM cost: ((N*s)/b) block accesses.

Ordered Array structures

The analysis here is the analysis of the logarithmic tree structure with fanout of 2, except that insert costs are much higher. Ordered arrays are never useful.

Logarithmic tree structures

fanout F, items per leaf L.

First we calculate how many levels of blocks will be within memory. M/b blocks will be in memory. This translates to $1 + log_F(M/b)$ levels assuming top levels stay in memory which would be the case for most modern block replacement schemes. Call this number InMem.

There are N/L leaves. Assuming a reasonably large fanout (e.g., 5 or greater), virtually all nodes are leaf nodes, so the structure is completely within memory if $N/L < M/b$.

If $N/L < M/b$, point search/insert/update/delete have RAM cost: InMem. Range queries have RAM cost: InMem + r/b because the items are all ordered.

If N/L > M/b Point Secondary: $(1 + floor(log_F N/L) - InMem)$ secondary memory block accesses Range search has secondary cost the point search cost + r/b.

Splits occur for 1/L of the inserts and cost an extra block access. This is inconsequential provided L is 10 or more.

Hash Structures

A well-designed hash structure should require one access to a block for any point search, insert, delete, or update. That is: if N*s > M, then secondary memory cost: (1 - M/(N*s)) because on the average the fraction M/(N*s) of the blocks will be available in RAM. If N*s <= M, then RAM memory cost: 1

Range queries in general will require accessing min(r, (N*s/b)) blocks either in secondary memory or primary memory, because every block will have to be searched for each element in the range.

Multi-copy Structures

insert/delete/update is RAM: 1 block access.

Point and range search operations require accessing potentially every block, but in practice (thanks to Bloom filters) requires only one secondary access. So that means that if N*s > M, then secondary cost is 1. If N*s <= M, then RAM: 1 block.

.0.3 Organization within leaf nodes

A leaf node may consist of k blocks, each of b bytes. We assume that blocks are in memory when they are accessed. So the time here is all RAM time and cache time assuming the blocks can fit into cache.

A range search takes RAM: k access + Cache: O(kb)

Unordered – for all other (non-range) operations, RAM: k access + Cache: O(kb)

Cracked – for all other (non-range) operations, RAM: 1 access + Cache: O(b)

Sorted – for (non-range) search, update, marking delete, RAM: 1 access + Cache: O(log(b))

Hash – For all other operations RAM: 1 access + Cache: O(1)

Log-structured block – for all insert, delete, update RAM: 1 access + Cache: O(1). For all point and range searches, RAM: k access + Cache: O(kb).