Foundations and Trends® in Databases

Data Structures for Data-Intensive Applications: Tradeoffs and Design Guidelines

Suggested Citation: Manos Athanassoulis, Stratos Idreos and Dennis Shasha (2023), "Data Structures for Data-Intensive Applications: Tradeoffs and Design Guidelines", Foundations and Trends® in Databases: Vol. 13, No. 1-2, pp 1–168. DOI: 10.1561/1900000059.

Manos Athanassoulis
Boston University
mathan@bu.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

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

now
the essence of knowledge
Boston — Delft

Contents

Data Structures for Data-Intensive Applications: Tradeoffs and Design Guidelines

Manos Athanassoulis[1], Stratos Idreos[2] and Dennis Shasha[3]

[1] *Boston University, USA; mathan@bu.edu*
[2] *Harvard University, USA; stratos@seas.harvard.edu*
[3] *New York University, USA; shasha@cs.nyu.edu*

ABSTRACT

Key-value data structures constitute the core of any data-driven system. They provide the means to store, search, and modify data residing at various levels of the storage and memory hierarchy, from durable storage (spinning disks, solid state disks, and other non-volatile memories) to random access memory, caches, and registers. Designing efficient data structures for given workloads has long been a focus of research and practice in both academia and industry.

This book outlines the underlying design dimensions of data structures and shows how they can be combined to support (or fail to support) various workloads. The book further shows how these design dimensions can lead to an understanding of the behavior of individual state-of-the-art data structures and their hybrids. Finally, this systematization of the *design space* and the accompanying guidelines will enable you to select the most fitting data structure or even to invent an entirely new data structure for a given workload.

Manos Athanassoulis, Stratos Idreos and Dennis Shasha (2023), "Data Structures for Data-Intensive Applications: Tradeoffs and Design Guidelines", Foundations and Trends® in Databases: Vol. 13, No. 1-2, pp 1–168. DOI: 10.1561/1900000059.

1

Introduction

1.1 Data Structures Are Foundational

Data structures are the means by which software programs store and retrieve data. This book focuses on key-value data structures, which are widely used for data-intensive applications thanks to the versatility of the key-value data model. Key-value data structures manage a collection of *key-value entries*, with the property that a given key maps to only one value but the same value can be associated with many keys. The value part of a data entry may have arbitrary semantics. For example, it may be a record in a relational database or a Pandas DataFrame, or an arbitrary set of fields that the application knows how to parse and use in a NoSQL system. In some settings, such as when systems manage data for social networks, the value may contain a reference to a large object such as an image or video.

Physically, a key-value data structure 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 (Hellerstein *et al.*, 2007; Selinger *et al.*, 1979; Idreos *et al.*, 2018a). Other terms used in the literature for data structures include "access methods," "data containers," and "search structures."

Data systems, operating systems, file systems, compilers, and network systems employ a diverse set of data structures. This book draws examples primarily from the area of large-volume data systems which require secondary storage devices, but the core analysis and design dimensions apply to purely in-memory systems as well, where access to RAM (random access memory) is far slower than access to the cache. In fact, the analysis applies to any setting in which there are two or more levels in the memory/storage hierarchy.

Given the wealth of applications that can be modeled using key-value data, such data structures have enormous general utility. For example, a particular data structure can be used to describe (i) metadata access in files, networks, and operating systems (Bovet and Cesati, 2005; Rodeh, 2008), (ii) data access in relational systems (Hellerstein *et al.*, 2007), (iii) data access in NoSQL and NewSQL systems (Idreos and Callaghan, 2020; Mohan, 2014), and (iv) feature engineering and model structures in machine learning pipelines (Wasay *et al.*, 2021).

Each application, or *workload*, can be represented as a mixture of key-value operations (point queries, range queries, inserts, deletes, and modifications) it supports over its data. In addition, the amount of memory and persistent storage required, along with their cost, shape the requirements of a given application. For example, file systems manage file metadata and contents using data structures optimized for frequent updates. Compilers typically use hash maps to manage variables during the variables' lifespan and use abstract syntax trees to capture the overall shape of a program. Similarly, network devices require specialized data structures to efficiently store and access routing tables.

As data-intensive applications emerge and evolve over time, using efficient data structures becomes critical to the viability of such applications, sometimes resulting in a three orders of magnitude performance change, as shown by Chatterjee *et al.* (2022). The reason is that data movement is the major bottleneck in data-intensive applications. Data movement is largely governed by the way data is stored, i.e., by the data structure. Thus, we expect that there will be an ongoing need for new data structures as new applications appear, hardware changes and data grows. Currently, research in academia and industry produces several new data structure designs every year, and this pace is expected

to grow. At the same time, with a growing set of new data structures available, even the task of choosing from an off-the-shelf data structure, that is, one that can be found in textbooks, has become more complex.

This book aims to explain the space of data structure design choices, how to select the appropriate data structure depending on the goals and workload of an application at hand, and how the ever-evolving hardware and data properties require innovations in data structure design. The overarching goal is to help the reader both select the best existing data structures and design and build new ones.

1.2 Tradeoffs in Data Structure Design

Every data structure represents a particular workload- and hardware-dependent performance tradeoff (formalized by Athanassoulis *et al.*, 2016 and Idreos *et al.*, 2018b). In order to choose an existing data structure or to design a new data structure for a particular workload on particular hardware, you should understand the possible design space of data structure design clearly and formally. That is the focus of this book. To motivate that discussion, let us look at a few examples of designs and tradeoffs when considering the workload (Section 1.2.1) as well as the underlying hardware (Section 1.2.2) and how they both evolve over time.

1.2.1 Workload-Driven Designs

Optimizing for a Workload. Consider a workload that consists of a small number of inserts and updates together with a large number of point and range queries. In order to balance the read and the write cost, many applications employ a B^+-tree, originally proposed by Bayer and McCreight (1972) and later surveyed by Graefe (2011). B^+-trees have a high node fanout, so that traversing from root to leaf requires few secondary memory accesses, and their top levels are cached in the faster levels of the memory hierarchy (Section 4.1). Further, a B^+-tree supports range queries by maintaining all the keys sorted in the leaf nodes and by connecting the leaf nodes in a linked list. As the number of insertions and updates increase, however, leaf nodes must be reorganized and maybe even split, which can become a performance bottleneck.

To address workloads having many inserts, a completely different approach is taken by a data structure called the log-structured merge-tree (LSM-tree). LSM-trees were originally introduced by O'Neil *et al.* (1996), and their many variants were surveyed by Luo and Carey (2020). As we will see in Section 4.8, LSM-trees place all updates in a common memory buffer which is flushed to disk when it becomes full. As more buffers accumulate, they are merged to form larger sorted data collection. This design employs an *out-of-place* policy of handling modifications, surveyed in detail by Sarkar and Athanassoulis (2022), in which there can be many key-value pairs in the structure having the same key. (For a given key k, the most recently inserted key-value pair for k has the current value.)

Thus two different workloads – one with more read queries and one with more insert operations – suggest different data structures.

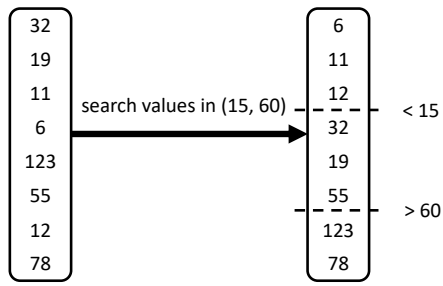

Figure 1.1: Adaptive data organization using the most recent search as a hint. In this example, a range query for values 15 to 60 leads to partitioning the base data in three non-overlapping partitions, one with values less than 15, one with values between 15 and 60, and one with values greater than 60.

Adapting to a Workload. Because the central theme of this book is to design a data structure given the expected workload, we also consider designing data structures that gradually adapt to the ideal design. To illustrate this point, consider that the B$^+$-tree and the LSM-tree, as originally designed, impose a sorted order within disk-resident nodes in order to answer any point or range query. An adaptive data structure may start with one or more unsorted nodes, but sorts them gradually in an opportunistic way, as shown in Figure 1.1. Chapter 5 will explain the concept of database cracking, proposed by Idreos *et al.* (2007a)

and further expanded by Idreos *et al.* (2007b) and Idreos *et al.* (2009). Intuitively, cracking uses the access patterns of incoming queries to continuously and incrementally physically reorganize the underlying data with the goal of improving the performance of future queries.

1.2.2 Memory-Driven And Storage-Driven Designs

As a complement to workload-based considerations, hardware advances create new challenges, needs, and opportunities in data structure design. Over the years, the *memory and storage hierarchy* has been enriched with devices such as solid-state disks, non-volatile memories, and deep cache hierarchies. Here, we discuss a few key hardware considerations for data structures, and we expand on them in Chapters 6 and 7.

Optimizing for the Storage/Memory Hierarchy. In a storage hierarchy, the lower levels offer a lot of storage at a low price but at high access latency, and as we move higher, that is, closer to the processor(s), 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 size of the application data relative to the storage capacity available at the different levels of the hierarchy.

Originally, B^+-trees tried to minimize disk accesses by maximizing the fanout. As the memory sizes grew, however, much of the data could fit into random access memory or non-volatile secondary memory. This changed the tradeoffs dramatically. For example, in-memory B^+-trees perform best with small fanout, as shown by Kester *et al.* (2017).

Memory Wall. While the memory hierarchy is expanding with technologies like high-bandwidth memory as outlined by Pohl *et al.* (2020), a key hardware trend for several decades has been the growing disparity between processor speed and the speed of off-chip memory, termed *the memory wall* by Wulf and McKee (1995). Since the early 2000s, operating systems, as discussed by Milojicic and Roscoe (2016), and data management systems, as discussed by Johnson *et al.* (2009), have been carefully re-designed to account for the memory wall by optimizing the use of cache memories.

Storage Devices Evolve. In addition, secondary storage itself has reached a crossover point. Traditional hard disks have long since hit their physical speed limits (Athanassoulis, 2014), and have largely been replaced by shingled disks and flash-based devices (Hughes, 2013). Shingled disks increase the density of storage on the magnetic medium, changing the performance properties of disks because the granularity of reads and writes changes (Hughes, 2013). Flash-based drives offer significantly faster read performance than traditional disks, but suffer from relatively poor write performance. Further, flash-based drives are equipped with a highly functional firmware layer called the Flash Translation Layer (FTL), which can be rewritten to yield dramatic changes in performance. Thus, flash hardware performance depends on both hardware and firmware. Such changes create a need for new data structures to optimize for different hardware/firmware combinations.

1.3 Audience & Prerequisites

This book aims to be used as part of graduate-level classes on designing complex data structures. It assumes at least an undergraduate-level familiarity with data structures.

Specifically, we assume that the reader is already familiar with basic data structures like arrays, linked lists, binary trees, queues, stacks, heaps, and hash tables, all taught in introductory courses for data structures and algorithms (Cormen *et al.*, 2009). Such basic data structures sit at the core of the more complex designs we outline. Most of the use cases and the presented designs are motivated by data-intensive systems, hence building on classical data structures used in database systems like B^+-trees (Bayer and McCreight, 1972), open hashing (Ramakrishnan and Gehrke, 2002), and LSM-trees (O'Neil *et al.*, 1996).

Other than that, the book is self-contained. We will describe all algorithms used in navigating complex data structures and outline how to combine the various design decisions we introduce.

1.4 Learning Outcomes

After reading this book, you will be able to reason about which existing data structure will perform best given a workload and the underlying hardware. In addition, you will be able to design new and possibly hybrid data structures to handle workloads with different composition, locality, and access patterns.

1.5 Overview of the Book

Here is the outline of our book. We recommend that you read the chapters sequentially.

- Chapter 2 introduces the fundamental performance metrics for data structures with respect to the most important key-value operations and hardware properties.
- Chapter 3 presents the core set of design principles, primarily based on workload characteristics, that largely guide the design of key-value data structures.
- Chapter 4 starts by explaining the design and performance characteristics of traditional data structures based on the design principles. We then discuss how to use the design principles to design new data structures for arbitrary workloads.
- Chapter 5 discusses the need for and design principles underlying adaptive data structures. We also illustrate use cases in which adaptivity leads to greatly improved performance.
- Chapter 6 discusses how data structures are utilized in big data applications, including databases, file systems, and machine learning.
- Chapter 7 discusses additional design considerations that can influence the detailed deployment of data structures ranging from deploying data structures in a setting with concurrent execution, in the context of distributed systems, to new hardware, new type workloads, and new application requirements.

2

Performance Metrics and Operational Tradeoffs

We now describe the core performance metrics we use to characterize key-value data structure designs. Because the metrics depend partly on the underlying hardware, we start by reviewing the memory/storage hierarchy and its importance for data structure design.

2.1 Memory Hierarchy

The memory hierarchy is a sequence of memory devices or layers, from fastest – e.g., registers and cache – to slowest – e.g., memory and disk (Manegold, 2009). They complement each other in terms of size, cost and speed of access. At least one layer of the memory and storage hierarchy is typically persistent (especially for file systems and data systems) so the data will not be lost when power fails or a machine is shut down. In addition, one layer of the memory hierarchy is large enough to hold all the data needed for a particular application. Data is transferred back and forth across layers as requests for data arrive, new data is inserted, and old data is deleted or updated.

Because different types of memory work at vastly different speeds, when data is moved towards the processing unit from a very slow memory layer, such as a disk, the overall cost of using a particular data structure

is dominated by this transfer cost. However, large transfer sizes from the slower memories to the faster memories partly mitigate the difference in speed. For example, the unit of transfer from memory to a fast cache memory is a few bytes, while the transfer unit from disk to main memory may be several kilobytes. The larger size potentially reduces the need for many disk-to-memory transfers. The cost of all other data movement at higher (i.e., faster) layers of the memory hierarchy is negligible by comparison. Typically, a modern processor running at more than 1 GHz speed is under-utilized because it is waiting for data from random access memory (which may take tens of nanoseconds) or disk (which may take thousands or millions of nanoseconds). Thus, design choices for data structures seek to minimize the number of accesses to the slowest layer.

For this reason, the modeling and design discussions in this book take these performance properties into account and always assume two abstract layers of memory hierarchy (Aggarwal and Vitter, 1988): one that is slow but can be treated as having essentially infinite capacity and one that is much faster but with limited capacity. This approach captures the *random access memory* and *disk* pair, but it can also capture any two layers of memory that have a significant difference in access latency and cost (e.g., 1-3 orders of magnitude).

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

In order to compare data structure designs and decide which one to use under particular conditions, we first need to define the appropriate metrics. The most common metrics quantify the read performance and update performance (Brodal and Fagerberg, 2003; Yi, 2009; Yi, 2012).

Read vs. Update. The Read cost defines how fast we can retrieve data while the Update cost describes how fast we can insert new data or change existing data. In this way, these costs collectively describe the end-to-end performance that a data structure design provides for a given workload. Such costs can be measured in terms of (expected) response time or more typically in terms of the amount of data we need to move to complete an operation. Reducing either of these often also increases throughput (the number of operations per second).

Read vs. Update vs. Memory. Data size is an additional metric that is crucial in practice. This metric is important because the disk or memory space to hold data structures is neither cheap nor unlimited.

Historically, data structure design research considered only read and update performance and ignored data size. 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 (Gray and Putzolu, 1986). 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 can be either expensive per byte and fast, or cheap but slow (Athanassoulis, 2014). Sometimes the higher performance comes at significant energy cost (Shehabi *et al.*, 2016). Flash-based SSDs exhibit a read/write asymmetry, where writes are typically one to three times more expensive than reads. At the same time, such devices can sustain a number of concurrent accesses before saturating the device bandwidth (Papon and Athanassoulis, 2021a; Papon and Athanassoulis, 2021b). In the current sensor-rich world, data generation outpaces the rate at which storage devices are delivered leading to a data-capacity storage gap (Bhat, 2018; Hilbert and López, 2011; Spectra, 2017).

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 in the design and optimization of data structures (Athanassoulis *et al.*, 2016; Dong *et al.*, 2017; Zhang *et al.*, 2016).

Storage and memory are not free, hence the footprint and cost of a data structure should also be considered when judging its efficiency.

2.3 RUM Performance Costs

We now formally define the Read, Update, and Memory costs. We first define three quantities that help capture the individual costs. These are defined as the *cost amplification* over the minimum conceivable.

1. **R**ead cost is defined as the read amplification of every lookup operation. This is the ratio between the size of the metadata (typically, in an index) plus the data an operation needs to touch divided by the actual size of the needed data. In other words, this reflects the extra data that is read when using a particular data structure design compared to the minimum conceivable cost a system could achieve in an ideal world where the system could directly retrieve the result without any additional search effort.

2. **U**pdate cost is defined as the write amplification of every update operation. That is the ratio between the size of the data and metadata that were accessed and the size of the updated data itself.

3. **M**emory (or storage) cost is defined as the space amplification of the employed data structure. That is the ratio between the aggregate size of data, metadata, and lost space due to fragmentation, divided by the size of the base data.

Amplification as an Alternative Cost Definition. The most common cost metric in terms of data structures is typically the total number of pages that have to be moved between slow and fast storage. The above definitions of the three overheads (RUM overheads for short) present *amplification* as a data-size-sensitive metric. For example, consider that, in an ideal world, a point query needs to access only the single data page that contains the target key-value entry. However, when using a tree-like structure, the query traverses indexing pages to locate the target data. The ratio between the number of all accessed (index and data) pages and the page(s) holding the data constitutes the *read amplification*. The definitions of *write amplification* and *memory amplification* are similar in that they capture the excess data pages an operation must access during a write operation and the excess memory required by the data structure. Thinking about the costs in terms of amplification, as opposed to in terms of the total number of pages of slow memory that are accessed, is often helpful as it gives a sense of how close a design is to the optimal.

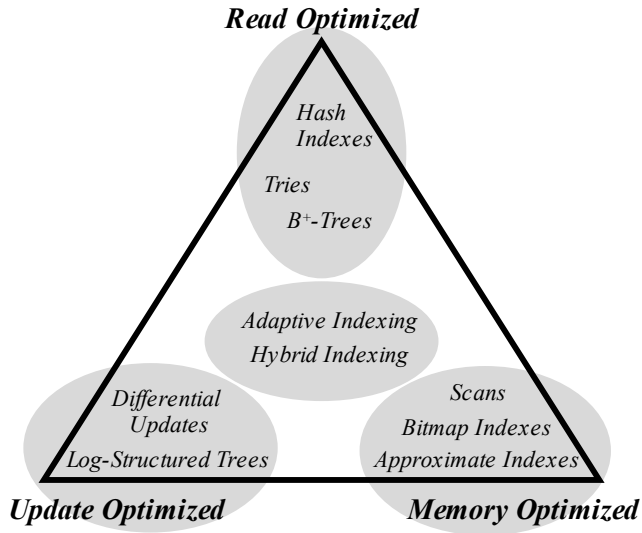

Figure 2.1: The RUM tradeoff space and a broad classification of data structure designs according to the RUM balance they maintain. The primitives underlying these data structures are discussed in Chapter 3 and the full structures are described in Chapter 4.

The three *RUM overheads* expose a three-way tradeoff (Athanassoulis *et al.*, 2016), between read, update, and memory overhead (Brodal and Fagerberg, 2003; Yi, 2009; Yi, 2012; Hellerstein *et al.*, 2002; Hellerstein *et al.*, 1997; Wei *et al.*, 2009).

Figure 2.1 conceptually shows a broad classification of data structure designs in the RUM tradeoff space. For example, a scan operation has high read cost, as it requires traversing an entire table – or at least a column (Abadi *et al.*, 2013) – to return relatively little data. Hence, the read amplification is large. By contrast, space amplification is minimal or even zero, because there is no need to maintain metadata.

Write amplification is also minimal when we can simply append or prepend new data (otherwise, it can be large if writes must locate previous instances of an input key through a scan).

By contrast, a typical tree-structured index (such as a B^+-tree) utilizes extra space for the index metadata to reduce read and update costs. The index nodes are used to navigate to the leaf node(s) that

house the desired data. In addition, it is also typical to trade space for efficient updates. For example, free slots available in leaf nodes of a B$^+$-tree, leading to an average fill factor of 67% (Ramakrishnan and Gehrke, 2002), increase the space consumed. This guarantees that most inserts will update only one page and not require splitting a page. Effectively this trades space amplification for update amplification. Figure 2.1 shows similar design tradeoffs, which are common in data structure designs as outlined by Athanassoulis and Idreos (2016) and Athanassoulis *et al.* (2016).

Overall, during the design of data structures, every step is a tradeoff consideration. Every new piece of metadata or any structure imposed on the base data has direct side effects on read, update, or memory amplification.

2.4 From RUM to PyRUMID

The RUM costs offer a good high-level metric of how broad design classes of data structures behave. To get a more complete view, which is necessary for hardware- and workload-conscious data structure design, we need to refine these metrics further. To do that we consider all possible operations that may form a *workload* of a data structure. Specifically, read performance depends on the exact access pattern. A ***P**oint query* asking for the value associated with a single key differs from a ***R**ange query* asking for a set of key-value entries ranging from a low key to a high key. Similarly, for writes, an ***I**nsert* operation has different requirements from an ***U**pdate* or ***D**elete*. While RUM classifies all modifications as Updates, their costs may differ in different data structures. For example, in a B$^+$-tree, an ***I**nsert* needs to store a new key-value pair which may trigger a sort operation on a leaf node, while an ***U**pdate* might change a value without disturbing the key order and a ***D**elete* might simply mark a key as logically deleted.

Overall, this makes for a complex tradeoff, which we call *PyRUMID tradeoff* from **P**oint queries, **R**ange queries, **U**pdates, **M**emory, **I**nserts, and **D**eletes. PyRUMID is more complex to reason about than RUM because there are more metrics but it offers a more accurate mapping of data structure designs to actual costs.

Next, we define in detail the costs of the different workload operations captured by the PyRUMID tradeoff. As before, we define all costs in terms of the total number of pages that need to be moved from slow memory to fast memory.

- **Point Query**: A point query needs to retrieve a single data entry based on a given key. The total cost in terms of pages includes not only the page that contains the target data entry but also any indexing data pages that we need to read because of the design of the data structure. Similarly, if the design of the data structure does not give direct access to the target data pages, then we likely first need to read additional data pages so we can locate the target data entry. These components are included in the total read cost.

- **Range Query**: A range query needs to retrieve all key-value entries whose key falls between specified low and high values. As with point queries, the total cost includes all indexing pages that need to be read. A critical factor that characterizes range queries is the selectivity, that is the fraction of the data pages requested with respect to the total number of data pages. Thus, there are three types of range queries, having different performance properties.

  - *Short range queries* have similar performance characteristics to point queries if the data is sorted, because accessing data requires reading the minimum granularity of data storage, typically a single data page. See below a more detailed discussion on how to reason about what range constitutes a short range query.
  - *Long range* queries read several data pages and the exact organization of data and the efficiency of locating useful data matters.
  - *Full scans* need to access all data pages. They might benefit from a special type of indexing, termed *filtering* (Section 3.3.2), which allows for partial data skipping. Further, they are affected by the design choices for laying out the base data, and can be enhanced by low-level engineering optimizations

to consume data in the fastest possible way. Such optimizations include vectorization (Boncz *et al.*, 2008), pre-fetching (Ramakrishnan and Gehrke, 2002) or using SIMD (single instruction/multiple data) instructions (Polychroniou *et al.*, 2015; Willhalm *et al.*, 2009).

- **Insert**: An insert operation adds a new key-value entry to the data structure. As with read operations, the cost is given by the total number of pages on slow storage (e.g., disks) that need to be read and written for the insert operation to be persistent. The cost includes both data pages that need to be read and those that need to be written. For example, an insert on a B^+-tree needs to locate an appropriate page for the new data entry. Further, any new key-value entry should not disrupt the existing data organization because that would degrade the performance of future operations. Insert performance – or *ingestion rate* – can be enhanced by techniques that allow accumulating key-value entries out-of-place and then reorganizing (e.g., re-sorting) later. That preserves the sorted organization of the base node without requiring expensive maintenance operations. Such a strategy allows multiple instances of the same key, generally with different values. The correct value to be associated with a key is the value of the most recent insert or update.

- **Delete**: A delete operation removes a key-value entry given an input key. Deleting a key-value entry may create fragmentation, which in turn, requires a mechanism 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 physically erased.

- **Update**: An update operation modifies the value of a specific key-value entry. Updates will often take less time than inserts or deletes, because the key organization does not change.

Characterizing Short Range Queries. In order to classify a range query as a short range query, we consider the random vs. sequential

access tradeoff. Specifically, we consider that a point query accesses approximately a single data page with a random access, while a short range query would spend about the same amount of time sequentially accessing data. Following the analysis by Kester *et al.* (2017), we consider that the latency of random access of a page with size P is L time units and that the sequential read bandwidth is BW. Assuming that a point query performs one random access, the latency of that query is L, a range query would be considered to be short if it has a similar cost.

The amount of data we can read sequentially in L time units is $BW \cdot L$, which occupies $\frac{BW \cdot L}{P}$ pages. For example, assuming a 4KB page size which is prevalent in contemporary systems, when operating in a memory that offers 100GB/s read bandwidth and 180ns access latency, a short range query would be one that accesses $^{(100\text{GB/s}) \cdot 180\text{ns}}/_{4\text{KB}} \approx 4.7$ memory pages (in 180ns), or $^{(100\text{GB/s}) \cdot 180\text{ns}}/_{64\text{B}} \approx 300$ cache lines (of 64 bytes each). If we consider an SSD with 1GB/s read bandwidth, 50µs latency, and 2KB page size, the corresponding short range is $^{(1\text{GB/s}) \cdot 50\mu\text{s}}/_{2\text{KB}} \approx 26$ pages (in 50µs). Finally, for a traditional hard disk drive with 150MB/s read bandwidth, 4ms access latency, and 4KB page size, the short range is $^{(150\text{MB/s}) \cdot 4\text{ms}}/_{4\text{KB}} \approx 150$ pages in 4ms. As devices evolve, one can use the $\frac{BW \cdot L}{P}$ rule of thumb to quantify how much data (or memory units, like pages, cache lines, etc.) can be accessed sequentially in the same time as needed for a point query. Thus, that amount of data characterizes a short range query.

Memory Utilization. When storage is expensive, e.g., within non-volatile memory or the memory supplied by cloud providers, the additional space occupied by auxiliary data, duplicate copies of the data, indexing, and buffer space increases the monetary cost of a data structure design. Using more memory for access methods can, however, lead to lower read or update time cost. So there is often a money/time tradeoff.

2.5 Chapter Summary

This chapter starts by characterizing the memory hierarchy from slow abundant memory to fast expensive memory. Next, it defines the notion of *cost amplification* as the ratio of the data that an operation needs

to touch to locate and access the relevant data compared to the size of the relevant data itself. For example, if a read of a record r must scan an entire file to find record r, the amplification would be the file size divided by the size of r. Finally, the chapter discusses the most important amplification issues that apply to each operation (point query, range query, insert, delete, update).

2.6 Questions

1. How does an update operation compare to a read operation in terms of how much data movement each needs to do, assuming the data is on slower memory (e.g., disk)?
   Answer Sketch: *We assume that the relevant data is on a single page. Both a read operation and an update operation need to read the data page. The update operation, however, may have to reorganize the data page. In addition, the updated page must eventually be written back to disk so the update persists.*

   (More advanced answer related to later chapters.) Both read and update operations may amortize their costs across many operations. For example, for update operations, the write to disk may be delayed by putting the update in a memory buffer so that several updates on the same page can be written to disk at once and share the write cost. However, if there is a crash before the updates make it to disk, the data structure may need to be re-populated with the aid of a higher level recovery process (e.g., a write-ahead log).

2. In moving from RUM performance metrics to the PyRUMID metrics, we distinguish among inserts, deletes, and updates. How might those different modifications be handled differently in a sorted array?
   Answer Sketch: *Inserts may need to move data within a sorted array to maintain the sorted order (e.g., if the key being inserted lies in the middle of the key range of that page). Deletes also may require data movement unless they are processed simply by marking a key-value pair as not valid (logical deletion). Updates*

may require little data movement if the replacing value is no larger in size than the replaced value. Thus the different modifications may require different analyses.

3. For a sorted array, do short range queries and point queries have similar compute and data movement cost characteristics? What about for unsorted arrays?
   Answer Sketch: *Any sorted data structure will have similar costs for point and range queries because it is likely that the entire short range will be present in the same section of the array (and therefore the same page). For an unsorted array, a short range query may require scanning the entire array.*

2.7 Further Readings

Data Management on new Storage Hardware. In this chapter, we discuss the interplay of data access methods with hardware and, specifically, with the memory hierarchy. Further understanding can be gained by studying tutorials and surveys that discuss the implications of new storage devices on data management and indexing targeting flash devices, by Koltsidas and Viglas (2011) and the broader class of non-volatile memories by Viglas (2015). Moreover, storage devices can be tailored to support specific workload characteristics by co-designing them with the application, as outlined in the tutorial by Lerner and Bonnet (2021).

Examples. Athanassoulis and Ailamaki (2014), Na *et al.* (2011), Roh *et al.* (2011), and Thonangi *et al.* (2012) proposed tree indexes that exploit the underlying storage device parallelism, while Jin *et al.* (2011), Li *et al.* (2010), and Papon and Athanassoulis (2023) propose ways to address the read/write asymmetry of flash storage devices, and Kang *et al.* (2007) aim to aggressively exploit locality.

Further, Athanassoulis *et al.* (2015), Debnath *et al.* (2010), Lim *et al.* (2011), and Nath and Kansal (2007) proposed key-value data stores that are tailored to flash storage by exploiting parallelism and efficient random accesses, while respecting limitations regarding the write cost and the higher cost compared to traditional disks. More examples of data management on flash devices have been discussed by Athanassoulis *et al.* (2010).

Data Management on new Compute Hardware. In addition to the evolution of storage hardware, compute hardware is also dramatically shifting. Surveys on multi-cores and deep memory hierarchies (Ailamaki *et al.*, 2017; Ross, 2021), GPUs for data management (Paul *et al.*, 2021), FPGA-based specialized hardware for data management (Fang *et al.*, 2020; István *et al.*, 2020), and storage systems in the RDMA (Remote Direct Memory Access) era (Ma *et al.*, 2022) can help understand the implications of new hardware trends on data access methods, storage, and data management.

Examples. Ailamaki *et al.* (1999) found that row-oriented systems face a very high number of cache memory data stalls. Based on this finding, Ailamaki *et al.* (2002) presented a new weaved page layout that reduced the cache stalls by grouping together elements belonging to the same column. Boncz *et al.* (1999) and Boncz *et al.* (2005) proposed a new vertically fragmented physical data layout to address the memory bottleneck. Further, Chen *et al.* (2001) proposed cache prefetching as a technique to optimize indexes for main memory, Ross (2004) improved selection conditions by avoiding expensive if statements, and Manegold *et al.* (2002a) built a cost model for cache-resident execution. More work on buffering, block-based accesses, and other optimizations follows similar principles (Lam *et al.*, 1991; Manegold *et al.*, 2002b; Manegold *et al.*, 2004; Zhou and Ross, 2004). The overarching goal is to keep the useful data at the fastest level of the memory hierarchy.

3

Dimensions of the Data Structure Design Space

We now present the dimensions of the data structure design space. Choosing the values of these dimensions constitutes the *fundamental design decisions* about how key-value data is physically stored and accessed. We present eight dimensions that collectively describe both well known state-of-the-art data structure designs as well as still unspecified designs that can be characterized by the design space. The eight dimensions are as follows:

1. **Global Organization** (Section 3.1): the assignment of keys to data pages. For example, a given page may contain a particular sub-range of the keys or a subset dictated by a hash function.

2. **Global Search Method**:

   (a) when there is no index (Section 3.2): the algorithm used to search over the data pages, e.g., binary search for sorted data pages.

   (b) using an index (Section 3.3): metadata that accelerates access to the target set of data pages for a given query, e.g., a hierarchical organization such as a B^+-tree.

3. **Local Organization** (Section 3.4): the physical key-value organization within each data page. For example, each page could be sorted by key.

4. **Local Search Method** (Section 3.5): the algorithm used to search within a data page or a partition. For example, binary search or hash lookup.

5. **Update Policy** (Section 3.6): in-place vs. out-of-place. For example, in an out-of-place list, the same key k may be found several times. The logical value to be associated with key k is the value stored with the first instance of k when traversed from the root of that list.

6. **Buffering** (Section 3.7): the decision to use auxiliary space to store read and write requests and then to apply the requests later to the data structure. For example, modifications to a key-sorted node may be buffered separately from the sorted contents. Later, the updates can be applied as a batch at once and thus require only one sort for the batch instead of one shift per insert or delete. Searches would first look at the buffered updates and then the sorted node.

7. **Key-value Representation** (Section 3.8): whether the access method stores keys whose values are entire records, only a record ID, a pointer to a record or a bitvector representation of the records corresponding to a key.

8. **Adaptivity** (Chapter 5): that is, whether the design uses queries as hints to gradually reorganize a structure to accelerate future queries. For example, a query on an unordered node may perform a range partitioning within the node. We treat adaptivity as a meta-dimension, since it changes the design to *gradually* reach a specific end-point which can be described by a value in each of the above seven design dimensions. For that reason, we discuss the seven design dimensions in this chapter and adaptivity in Chapter 5.

Cost Model: Quantifying Design Impact. In the rest of this chapter, we present the possible decisions for each dimension and how they affect the overall design in terms of PyRUMID performance.

To quantify the PyRUMID costs we utilize a cost model with the parameters shown in Table 3.1. This is an I/O model that captures the number of disk pages moved. As discussed in the previous chapter, we focus on data movement as this is the primary factor that affects end-to-end performance for data system applications where computation is light (e.g., simple comparisons).

Table 3.1: Parameters capturing the various dimensions of the data set and the data organization that affect the cost of accessing keys in a particular data structure.

Key-value entry size (# bytes)	E
Key size (# bytes)	K
Page size (# entries that fit in a page)	B
Data size (# entries)	N
Data size (# pages)	$N_B \geq \lceil N/B \rceil$
Memory size (# pages that fit in main memory)	M
# partitions	P
index of searched key	i
index of the page of the searched key	i_B
index of the partition of the searched key	i_P
selectivity (fraction of entries retrieved)	s
radix length of the key domain	r

Note to the reader: This chapter is the longest in the book as it gives all fundamental principles. If you are new to the field, we propose reading this chapter in two or more sessions. For example, you might choose to read through the end of "Search When Using an Index" (Section 3.3) in the first sitting, let that soak in, and then read the rest of the chapter in a second sitting. Every subsection is also self-contained, describing one design decision at a time.

3.1 Global Data Organization

The first decision for an access method is how to organize data across pages or partitions, or the *global data organization*. Below, we introduce the main global data organization options, and we accompany each with a simple example for illustration. We further differentiate between *key-level* organizations and *partition-level* organizations. Key-level organizations specify how all the keys in a data structure are organized (e.g., sorted, hashed, or unordered), while partition-level organizations specify the organization in a nested fashion: first how to organize keys across partitions, and second how to organize the keys within a partition (further discussed in Section 3.4).

3.1.1 Key-level Organizations

Notation: In each example below, data pages are separated by ";", partitions are indicated using "(...)", and the overall dataset is enclosed by "[...]". In the absence of "(...)," there are no partitions. Note that, while a partition typically consists of multiple pages, having multiple partitions within a page or one partition per page is also possible.

No Organization. The key-value pairs are stored in pages without any structure or order enforced. As a result, any particular key-value pair may appear in any page.
Example (showing keys in pages only):
[242 2000 1002; 200 49 2304; 25 230 1500]

Sorted. Key-value pairs are sorted based on their keys (note we represent only the keys and not their associated values).
Example (showing keys in pages only):
[25 49 200; 230 242 1002; 1500 2000 2304]

Hashing. Key-value pairs are stored based on the hash of the key in the hash table.
Example (using as hash function $h(x) = x \bmod 9$), hash value in $\{\cdot\}$:
[{0}: 2304, {1}: 200, {2}: 2000; {3}: 1002, {4}: 49, {5}: 230; {6}: 1500, {7}: 25, {8}: 242]

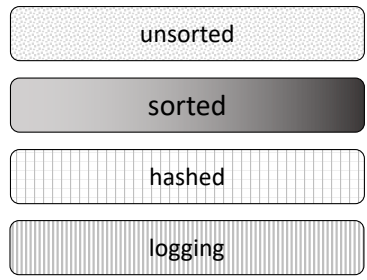

Figure 3.1: The four fundamental key-level global data organizations (without sub-partitions). Key-value pairs may be stored without regard to the key values (unsorted); in key-sorted order (sorted); based on a hash function on keys (hashed); and in time order of entry (logging).

Logging. Key-value pairs are physically stored according to their *arrival order* forming a log[1].
Example:
[230 2000 1002; 200 49 1500; 25 242 2304]
Figure 3.1 summarizes the key-level data organization options using a visual representation that we use in the remainder of the book to describe data structure designs.

3.1.2 Partition-Level Organizations

Range Partitioning. There are P non-overlapping partitions. Each key-value pair (k, v) belongs to the single partition whose key range contains k. For example, if there are three partitions with key ranges $[1, 220)$, $[220, 1200)$ and $[1200, 3000]$, the collection of our running example could be partitioned as follows. In this example, each partition occupies exactly one page. Further, note that within each partition, the keys need not be organized in any particular way.
Example:
[(49 25 200); (1002 230 242); (2000 2304 1500)]
Radix Partitioning. A prefix of the bit representation of the key k of a given key-value pair (k, v) is used to map (k, v) to a specific partition. For

[1]Note that the use of the term "logging" here refers to the notion of appending data in a time-ordered manner to form a log and does not relate to database recovery (Bernstein *et al.*, 1987).

example, we can use the first two bits of the 12-bit binary representation of the keys in our example data set: 25 = 0*b*000000011001, 49 = 0*b*000000110001, 200 = 0*b*000011001000, 230 = 0*b*000011100110, 242 = 0*b*000011110010, 1002 = 0*b*001111101020, 1500 = 0*b*010111011100, 2000 = 0*b*011111010000, and 2304 = 0*b*100100000000.
This would result in the following (uneven, because most of the keys are small) partitioning:
[(0*b*00: 49 25 242; 1002 200 230); (0*b*01: 2000 1500); (0*b*10: 2304)]

Hash Partitioning. Each key-value pair (k, v) goes into a partition $h(k)$ based on a hash function $h(\cdot)$. For example, assuming for ease of presentation a simple hash function h(k) = k *mod* 3, the data collection in our example will be partitioned as follows.
Example, hash value in $\{\cdot\}$:
[({0}: 1002 1500 2304); ({1}: 49 25); ({2}: 230 2000 200 242)]

A special case of hashing is *order-preserving* hashing, which creates non-overlapping partitions with increasing value ranges (Fox *et al.*, 1991; Hutflesz *et al.*, 1988; Robinson, 1986; Sabek *et al.*, 2022).

Partitioned Logging. As an alternative to pure logging, partitioned logging partitions key-value pairs based on disjoint time intervals called *epochs*, where the ith epoch is marked as e_i. The same key may be present in multiple epochs, resulting in memory amplification.
Example:
[(e_1: 230 2000 1500; 200); (e_2: 49 1500 25); (e_3: 242 200 2304)]
Figure 3.2 summarizes the partition-level data organization options. Note that once a partitioning-based global data organization is selected, the various partitions may have different local data organizations, which we discuss in Section 3.4.

3.2 Global Search Algorithms When Not Using an Index

Having discussed the fundamental global data organizations, we now discuss the algorithms to search for a single key or a key range in each data organization. For each of the choices, we provide definitions, examples, and the order of magnitude time cost. The decisions and their impact are summarized in Tables 3.2 and 3.3. In order to handle the

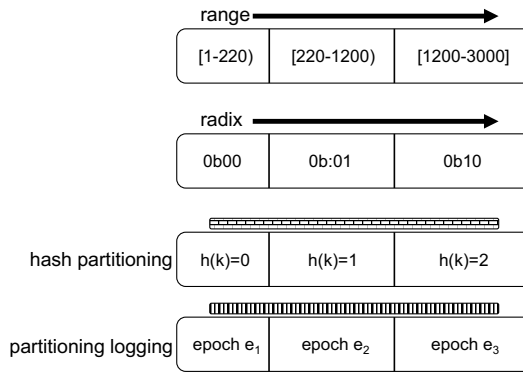

Figure 3.2: The four partition-level data organizations. The assignment of a key-value (k, v) pair to a partition can be based on which range of values key k lies (range partitioning); based on the bit representation of k (radix); the value of the hash on k; or the time interval when (k, v) arrived.

worst case, we assume that our algorithms cannot benefit from caching, so all data is assumed to reside in the slowest level of the memory hierarchy considered (e.g., disk). We first present the fundamental *algorithm classes for searching data when not using an index.*

3.2.1 Full Scan

A full scan accesses all N_B pages regardless of the data organization as shown in Figure 3.3.

Applicable to: all data organizations for point or range queries.

Point and Range Query Performance. The performance of a full scan depends only on the data size and the query selectivity which together affect the overheads of predicate evaluation and writing the output (Kester *et al.*, 2017). In terms of I/O cost, a full scan will always have to read $O(N_B)$ pages of data to find the required key-value pair(s). As a rule of thumb, a full scan should be used if (i) the search/update/delete has no information about the data organization, (ii) the data is stored using *no organization*, (iii) the data is stored based on *logging* but the key is not time-monotonic (sequential), or (iv) the query will return a large fraction of the initial data collection.

Optimizations. While a full scan has to access the entire data collection, it can be significantly accelerated by techniques that exploit hardware. The two main techniques are (i) parallelization, which breaks the data collection into disjoint chunks and use a different processor to scan each, and (ii) vectorization, which exploits SIMD commands to increase the throughput of comparisons.

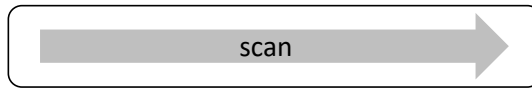

Figure 3.3: A full scan can be applied to any data organization strategy

3.2.2 Binary Search

A binary search uses the order in the data to avoid accessing all pages. Specifically, a binary search needs to access only a logarithmic (base 2) number of pages for point queries as shown in Figure 3.4. For example, if there are a million pages, then binary search will read only 20 pages.

Applicable to: sorted, range-partitioned, order-preserving hash, and radix-partitioned organizations, for point as well as range queries.

Point Query Performance. Binary search finds the desired value in a sorted collection after $O(\lceil log_2(N_B) \rceil)$ accesses. If the data is partitioned in P partitions, the search must first find the correct partition and then, in the worst case, search all pages in that partition, requiring $O(\lceil log_2(P) \rceil + \lceil N_B/P \rceil)$ accesses.

Range Query Performance. If the query is a range query, the binary search for the first key will be followed by sequential accesses to subsequent pages. The cost of a range query includes a component that depends linearly on the fraction of the data retrieved ($s\%$). In the case of a sorted data organization, the number of matching pages is $\lceil s\% \cdot N_B \rceil$. Including the search for the first key, the cost of a range query on a sorted data organization is $O(\lceil log_2(N_B) \rceil + \lceil s\% \cdot N_B \rceil)$. In the case of range partitioning, a range query will read $k = \lceil s\% \cdot P \rceil$ partitions. Thus, the cost of a range query (when there is no order within partitions) is $O(\lceil log_2(P) \rceil + k \cdot \lceil N_B/P \rceil)$ pages.

Optimizations. In addition to binary search, a sorted collection can be searched using an m-ary search with m groups, leading to $O(log_m(N))$ search cost (Schlegel *et al.*, 2009). The main idea of this algorithm is that we divide the search space (initially the entire array) into m parts. At the first step, an m-ary search uses SIMD commands to load and compare the values at m positions of the array to identify which of the m parts we should look into. An m-position search is performed recursively within the selected part.

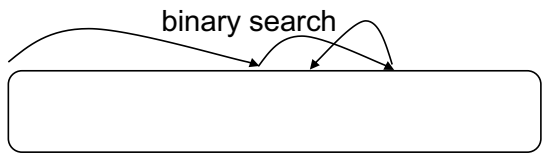

Figure 3.4: A sorted or a range partitioned data organization can benefit from binary search (and even more from an m-ary search).

3.2.3 Direct Addressing

Direct Addressing is the strategy of mapping a key directly to a location, perhaps after some transformation. For example, a hash structure based on a function H will map a key-value pair (k, v) to a block whose location is $H(k)$. By contrast, binary search trees, B-trees, and other such structures map keys to locations based on a series of comparisons.

Applicable to: hash and radix partitioned organizations for point and range queries.

Point Query Performance. When there are few collisions, a hash index needs to read only one data block to answer a point query as shown in Figure 3.5. Radix partitioning maps a prefix of the bit representation of the key to a partition ID. In disk-based implementations, the hash or the radix of the key points to a unique position of an array that hosts a partition ID and its offset. Note that, as in every partitioning strategy, the entire partition containing the desired key might have to be read if there is no internal organization.

Range Query Performance. Both radix-based data organizations and order-preserving hashing support range queries. Specifically, a range

query will first perform a constant time lookup to find the beginning of the range of the qualifying entries, and then it will scan the qualifying entries leading to total cost $O(1 + k \cdot \lceil N_B/P \rceil)$ where k is the number of partitions read for the range and N_B/P is the average number of pages per partition.

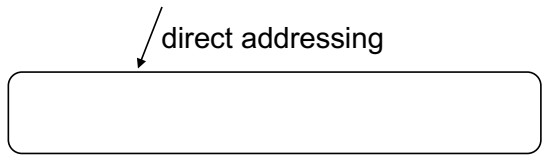

Figure 3.5: Direct addressing is able to access the desired data in a point query without an expensive search if the underlying data organization is hash or radix based.

3.2.4 Data-Driven Search

Data-driven search consists of a class of search algorithms that lie between sorted search and direct addressing. Data-driven search uses available knowledge of the data distribution as a hint about how to navigate the data, as shown in Figure 3.6.

Applicable to: sorted, range partitioned, and radix partitioned organizations for point and range queries.

Performance. In contrast to binary search which discards roughly half the data at each step, data-driven search potentially discards a much higher fraction of the data by using information about the data distribution. For example, if the key range is between 10 and 20 million, the keys are close to uniformly distributed, and the key is 18,542,123, then interpolation search (Perl *et al.*, 1978; Van Sandt *et al.*, 2019; Yao and Yao, 1976) will perform a first search at about the 85th percentile of the array. The time complexity is calculated to be on average $O(log_2(log_2(N_B)))$ for a uniform dataset (Perl *et al.*, 1978; Yao and Yao, 1976). Other generalized search algorithms (Kraska *et al.*, 2018) improve upon binary search in the face of a non-uniform data distribution by storing some characterization of that distribution.

Exponential search (Bentley and Yao, 1976) starts by searching for k at a position $bound = L$. If the value v at position L is less than k, then it searches at $bound = 2 \cdot bound$. Otherwise, it initiates a binary search between the beginning of the array and the current *bound*. Overall, exponential search has time complexity $O(log_2(i_B))$, where i_B is the page index where the search key is located in the data collection, leading to $O(log_2(N_B))$ worst-case performance. In practice, this algorithm can be much faster than simple binary search when the search key is located at the beginning of the array, but it may be slower (due to the additional cost of re-initiating binary searches) when the key in question is near the end of the array.

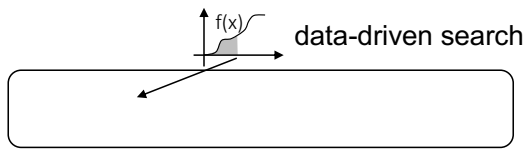

Figure 3.6: Data-driven search uses information about the data distribution to reduce the number of search steps taken. This can be much faster then binary search and sometimes faster than m-ary search.

3.2.5 Summary of Access Costs without Indexing

Table 3.2 presents the search cost for a point query for each of the possible combinations of data organizations on the one hand (first column), and search algorithms on the other hand (columns 2-5). These costs assume that none of the data is in memory when the query begins. Note that a missing entry (marked as "–") indicates that the combination of data organization and search algorithm is not feasible.

Table 3.3 presents the search cost for a range query for each possible combination of data organizations and search algorithms, again assuming that none of the data is in memory when the query begins. Note that we add one more row to refer to order-preserving hash partitioning (*OP hash part.*).

Table 3.2: Costs for point queries. All organizations support scans, but some organizations can be much faster using other search strategies (binary, direct, and data-driven). M-ary changes the base of the logarithm from 2 to m (not shown).

Search Algor. → **Organization** ↓	Scan	Binary	Direct	Data-driven
none	$O(N_B)$	–	–	–
sorted	$O(N_B)$	$O(\lceil log_2(N_B) \rceil)$	–	$O(log_2(log_2(N_B)))$
logging	$O(N_B)$	–	–	–
range part.	$O(N_B)$	$O(\lceil log_2(P) \rceil + N_B/P)$	–	–
hash part.	$O(N_B)$	–	$O(N_B/P)$	–
radix part.	$O(N_B)$	–	$O(N_B/P)$	–
part. logging	$O(N_B)$	–	–	–

Table 3.3: Costs for range queries. Scanning always works, but other methods may work faster especially when the hardware supports random access page access that is as fast as sequential page access.

Search Algor. → **Organization** ↓	Scan	Binary	Direct	Data-driven
none	$O(N_B)$	–	–	–
sorted	$O(N_B)$	$O(\lceil log_2(N_B) \rceil + \lceil s\% \cdot N_B \rceil)$	–	$O(log_2(log_2(N_B)))$
logging	$O(N_B)$	–	–	–
range part.	$O(N_B)$	$O(\lceil log_2(P) \rceil + k \cdot \lceil N_B/P \rceil)$	–	–
hash part.	$O(N_B)$	–	–	–
OP hash part.	$O(N_B)$	–	$O(1 + k \cdot \lceil N_B/P \rceil)$	–
radix part.	$O(N_B)$	–	$O(1 + k \cdot \lceil N_B/P \rceil)$	–
part. logging	$O(N_B)$	–	–	–

3.3 Search When Using an Index

An *index* is auxiliary metadata[2] that accelerates any search structure operation by rapidly locating the position of the relevant key. This applies to searches, but also to modifications, because, in many common data structures (e.g., binary search trees, hash indexes and B$^+$-trees), insert, delete, and update operations must first find the location of the target key-value pair to be modified.

In the previous subsection, we covered the different ways to search a data set depending on the employed data organization but without using any indexes. In this section, we discuss the different index design

[2]Note that in classical database systems literature, indexes may also be the main file organization as discussed by Ramakrishnan and Gehrke (2002), however, even in this case, the index metadata is in addition to the base data the resides in the leaf nodes of the index as part of the value of the key-value pairs.

options to accelerate the search process for any key-value operation. Before that, we present our memory management assumptions.

3.3.1 Memory Management

In this section, we assume the availability of M page frames, which can be used to store the highest levels of an index (those near the root) in fast memory while the remaining index pages and all data pages reside in slower memory.

Achieving this ideal memory utilization requires a buffer replacement policy that attempts to keep the most useful pages in memory. The widely-used least recently used (LRU) replacement policy keeps the M most recently used pages, assuming that future accesses will most probably be similar to recent accesses. However, this approach does not necessarily keep the upper levels of the index in memory. For example, when a search on a B^+-tree accesses a particular leaf node, LRU would retain that leaf node, even though that leaf node may not be accessed again for a long time. It would be better to use that memory for an interior node of the index. For that reason, a wealth of buffer replacement policies improve on LRU in the sense of tending to retain the highest level nodes of the index (i.e., those closest to the root) have been proposed by Chan *et al.* (1992), Graefe and Kuno (2010b), Johnson and Shasha (1994), Megiddo and Modha (2004), O'Neil *et al.* (1993), Sacco (1987), and Stonebraker (1981). Because buffer replacement is beyond the scope of this book, we assume the use of an idealized strategy that keeps the nodes of the top levels in fast memory for purposes of our cost model.

In the rest of this section, we relate the search algorithms described in Section 3.2 to the corresponding indexing approaches.

3.3.2 Full Scan *improved by* Filter Indexing

A scan is perhaps the most fundamental operation because it is used when there is no order in the data, either globally or locally.

To speed up a scan, a filter index can be used. A filter index is an in-memory index that can indicate that a certain disk-resident page or partition is irrelevant to a given query and, therefore, need not be

searched. We will refer to the portion of data summarized as a "chunk" since the granularity can be one of a partition, a page, or a group of pages. Overall, the goal of filter indexes is to reduce the *scan ratio*, that is the fraction of pages that need to be scanned. Figure 3.7 visualizes filter indexing.

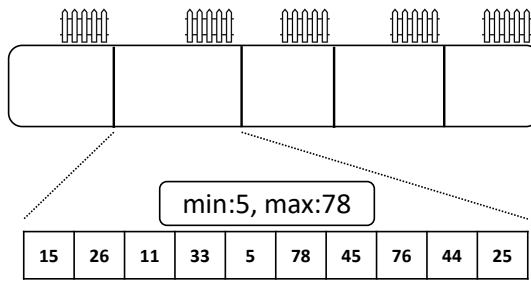

Figure 3.7: Filter indexes incorporate per-partition summaries that allow scan implementations of key searches to skip some partitions. In the example, the partition has no particular local data organization, and the filter stores the minimum and the maximum keys in the partition.

There are a variety of such structures: Zonemaps (Moerkotte, 1998), Column Imprints (Sidirourgos and Kersten, 2013), as well as Bloom filters (Bloom, 1970) and its variants (Bender *et al.*, 2012; Breslow and Jayasena, 2018; Deeds *et al.*, 2020; Fan *et al.*, 2014; Lang *et al.*, 2019; Pandey *et al.*, 2021). A filter index rapidly determines whether it is safe to avoid accessing a partition of base data. For example, a Zonemap summarizes each partition using that partition's minimum and maximum key. These small summaries allow a search, update, or delete to completely skip a partition (or potentially a page within a partition) when the search key falls outside its range. Similarly, Column Imprints store a lightweight histogram per page such that if the search key does not lie within that histogram, then that page can be skipped.

While the above methods are most useful when there is range-partitioning, Bloom filters use hashing to index each element of a chunk by setting a number of bits of a bitvector as shown in Figure 3.8. Bloom filters can allow a point query to completely skip accessing an entire chunk when the bitvector indicates that the query's argument cannot be in that partition. Note that a positive result from a Bloom filter

indicates merely that the query argument could be in the partition. There is some probability of a false positive. This probability is *tunable*: devoting more space to Bloom filters by using more bits to index the same number of elements will reduce the false positive probability.

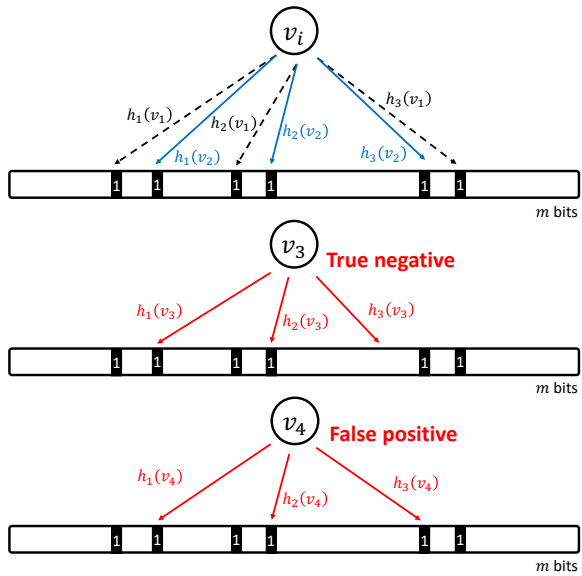

Figure 3.8: A Bloom filter stores membership information of a set of n elements using a bitvector of m bits. The accuracy of a Bloom filter depends on the ratio of bits per key ($BPK = \frac{m}{n}$). In order to insert a key, we hash it with k (three in the example) different hash functions that map keys to the range $0..k-1$, and we set the corresponding bits to 1, as shown in the top part of the figure. Every other bit remains 0. When searching for a key, the same process is followed. When we search for a key that was not inserted in the Bloom filter, then either one or more of the tested bits will be set to 0 (middle part of the figure), leading to a true negative, or all k bits will be set to 1 due to the insertions of other keys (bottom part of the figure), leading to a false positive.

Search Cost. Suppose the page size in bytes is $E \cdot B$ (that is, the number of entries per page $\times$ entry size). The total size of a Zonemap filter index is $O(2 \cdot N_B \cdot K)$, where K is the key size. A Column Imprint that uses histograms of H bytes per page has a total size of $O(N_B \cdot H)$ bytes. A collection of Bloom filters per partition (or a single Bloom filter for the entire data set) has total size $O(N_B \cdot B \cdot BPK/8)$ bytes,

where B is the number of entries per page, and BPK is the number of bits per key for the Bloom filter. By contrast, a full index is much larger. Consider for example an in-memory hash index with load factor l_f (with a common value of 90%-95%). The total size of such a hash index is $O(N_B \cdot B \cdot E/l_f)$.

Assuming that the memory buffer has enough space to hold this small amount of metadata, the cost of a point query using filter indexes over sorted data can be as low as $O(1)$, and over range partitioned data as low as $O(N_B/P)$, while for other data organizations, it depends on the data distribution per partition (Moerkotte, 1998; Sidirourgos and Kersten, 2013).

Filters vs. Sparse Indexing. *Filter indexing* is different from *sparse indexing* (Dong and Hull, 1982; Ramakrishnan and Gehrke, 2002). Consider a point query on key k. A sparse indexing approach will point to at most a single data page that might contain k. By contrast, a filter indexing approach may indicate that multiple pages may contain k. The key space must be partitioned for sparse indexing: sorted for B$^+$-tree style indexes or partitioned based on a function for hash indexes.

A filter index benefits from partitioning but does not require the key space to be partitioned. For example, if the dataset is not sorted and a key k falls in the range of several pages, then Zonemaps would indicate that all those pages would have to be searched, but might still filter out the vast majority of pages. If the data is sorted, the value of a point query will always belong to the range of a single page. In the sorted case, a filter index that lies in main memory has the same I/O cost as a sparse index.

3.3.3 Binary Search *implemented as* Binary, m-Ary, B$^+$-trees

As we have seen before, when the data is sorted, a binary search leads to a logarithmic base 2 cost of searching by safely discarding half of the data in every iteration. That yields a cost of $O(log_2(N_B))$ page reads.

Indexes on sorted data effectively gain in read performance at the cost of memory. Specifically, the index increases the base of the logarithm from 2 to hundreds or more. This can have an enormous impact on performance. For example if N_B is a billion, $log_2(N_B) = 30$, but

$log_{1000}(N_B) = 3$. To achieve this, we would need to design tree-based indexes where every node has a fanout of about 1000, implying that each search on a node discards 999/1000 of the remaining possible nodes. The levels of the tree are connected via pointers that help to point a search to a limited number of memory/disk accesses per level. The concept of pointers between levels is called *fractional cascading* (Chazelle and Guibas, 1985) and is the defining principle of tree indexes. Figure 3.9 visualizes a search tree index.

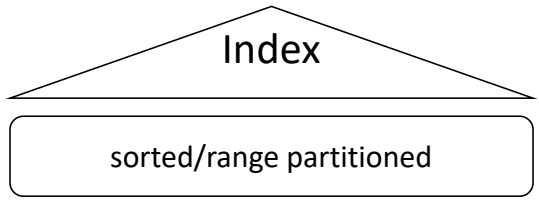

Figure 3.9: Sorted search can be accelerated with a search tree. The underlying data should be sorted or at least range-partitioned to benefit from the search tree.

Search Cost: $O(log_m(N_B))$.
Tradeoffs: While m-ary trees for large fanout values of m enable fast searching, such trees must also be maintained when the sorted collection of keys is modified due to new data or updates. In order to facilitate dynamic workloads, both the sorted collection and the auxiliary metadata are organized in logical nodes, often having some empty space. The resulting tree can be modified in place (e.g., by inserting new entries into the empty space of a single leaf node) without having to move large amounts of data. Empty space, however, comes at a cost. Interior tree nodes with empty space will not only occupy more total disk space but will also have a smaller fanout than full nodes. Less fanout will, in certain cases, imply an increase in the number of levels of the tree.

3.3.4 Direct Addressing *implemented as* Hash Indexes

Instead of scanning or iteratively searching, an alternative approach is to use the binary representation of a key k to directly locate where k and its associated value are physically stored through hashing or radix

search. Hashing is particularly good for highly skewed distributions because it assigns nearly equal keys to different partitions. For example, the simple hash function $h(k)=k \ mod \ 101$ will hash 100 consecutive keys to 100 different partitions. Figure 3.10 shows a visualization of hash indexing.

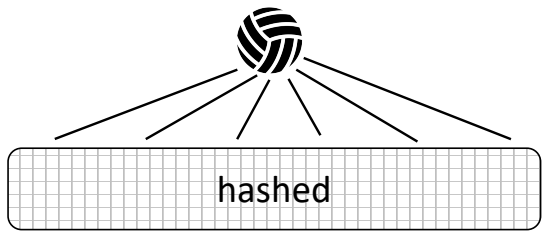

Figure 3.10: The ball represents a function that takes a key and maps that (or "kicks" it) to an address of a partition where the key is present if that key is anywhere in the data structure. Thus, hash indexing can lead a point query directly to the location of the desired data.

Search Cost: $O(N_B/P)$ on average. Note that if the partitions have different sizes, the worst-case cost depends on the size of the largest partition.

3.3.5 Direct/Partial Key Addressing *implemented as* Radix Trees

Another approach that can be seen as a hybrid of classical search trees and direct addressing is a search tree with compressed internal nodes, or a *radix tree*. Though the first applications of Radix trees were to store strings, Radix trees are now frequently used to store arbitrary key-value pairs using the bit-wise (radix) representation of the key when searching. Radix trees (also called prefix trees or tries) support searches by associating portions of a key with each edge in the tree as shown in Figure 3.11 (Morrison, 1968). The common prefix of all key strings is stored in the root node. Navigating a radix tree entails matching bits to each edge. In the context of data management, the design of Radix Trees has been refined to reflect hardware considerations (Mao *et al.*, 2012) and data properties (Leis *et al.*, 2013).

Radix trees have other applications as well. For example, Weiner (1973) proposed suffix trees. Suffix trees take as input a set of key

strings and form the radix tree of all suffixes of those strings. This data structure enables efficient substring searching, along with searches of consecutive substrings, longest repeated substrings, longest common substrings, and other similar searches within the indexed collection of strings.

A classical search tree (e.g., B^+-tree) has internal nodes that use values from the search domain as "separators" to create the internal node hierarchy. Since all these values come from the same domain, each node needs to store only the differentiating information. Now suppose that we have a binary representation of the domain (or a radix) that needs r bits to represent each indexed value.

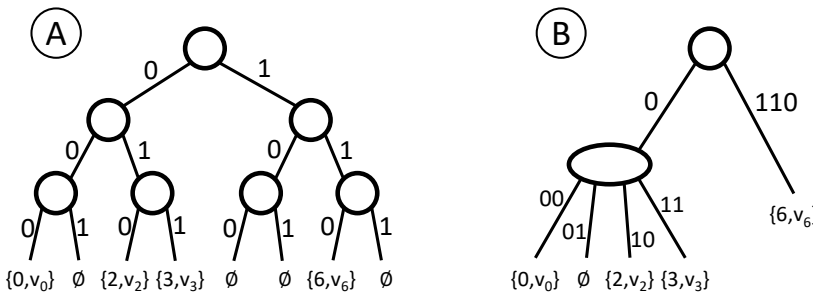

Figure 3.11: A simple (A) full radix tree design, and (B) the adaptively compressed counterpart. In the left subtree of tree B, three nodes having two children have been merged into one node having four children. In the right subtree of tree B, edges pointing to nulls are eliminated.

Given a key, a search using a basic radix tree will start at the root and branch based on each bit of the key, in turn from left to right. To improve 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 by collapsing an entire sub-tree to a single node. A simple example is given in Figure 3.11. Radix trees can be compressed in two ways. The first is *node merging*, where binary nodes are fused together with their parent nodes – shown in the bottom left part of Figure 3.11(B). Node merging can be applied recursively across multiple levels. The second is *subtree removal*, where an entire subtree can be removed if there are no keys in its leaf nodes –

shown in the rightmost part of Figure 3.11(B), where the subtree for keys with prefix 10 is completely removed because neither 100 and 101 exists. Subtree removal compression is particularly effective for skewed data. When the data is not skewed, all or most subtrees will have a key in one of their leaf nodes.

A practical design is to use 4 or 8 bits of the radix in a single node that would have 16 or 256 child nodes (Leis *et al.*, 2013), respectively. Such a radix-tree node fits in a few cache lines and maintains high locality while offering effective navigation in the key domain. Radix trees and their space-optimized variants efficiently support both point and range queries.

Search Cost: $O(\lceil log_{2^S}(\mathcal{U})\rceil) = O\left(\left\lceil \frac{log_2(\mathcal{U})}{S}\right\rceil\right)$, where $\mathcal{U}$ is the maximum value of the domain, so $\lceil log_2(\mathcal{U})\rceil$ is the number of bits needed for that maximum value, and S is the number of radix bits used per node. For example for a 32-bit domain, if we use a radix tree with nodes storing 8 bits of the radix, the height of the tree would be $log_2 2^{32}/8 = 32/8 = 4$. Note that the cost for a point search query on a radix tree does not depend on the data size, but rather on the maximum possible value of the domain. An optimization that would allow the search cost to exploit the data distribution is to tune S dynamically to be different in different parts of the domain (Leis *et al.*, 2013) and to collapse paths of the trees that lead to only one leaf node, as illustrated in Figure 3.11.

3.3.6 Model-Driven Search *evolves into* Learned Indexes

All indexes presented so far that target both point and range queries maintain a sorted, range-based, or radix-based version of the data and create meta-data for navigation purposes. The meta-data is based on the data organization. Operations over the data structure may navigate this metadata to reach the target data entries directly without having to read the rest of the data. On the other hand, this navigation metadata may require a lot of memory space as it involves explicitly maintaining key order.

An alternative index design option is to maintain key order as a model instead of explicit keys. The basic idea of model-driven search algorithms is to calculate the expected position of a data entry using

knowledge of the key distribution. For example, the original learned index vision (Kraska *et al.*, 2018) proposes to sort all data entries and *learn* the distribution of the data by building a machine learning model. Searches over the data structure then use the model to *predict* the expected location of a data entry given its key. Given that prediction, a query may have to search for the exact slot of the data entry in the data page. However, as long as the model points to the right data page, the overall I/O cost is still reading a single page (e.g., for a point query). This is illustrated in Figure 3.12.

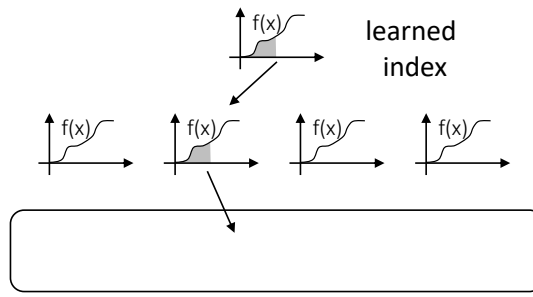

Figure 3.12: A hierarchy of models captures the data distribution in a learned index design.

One approach to creating a learned index is to build a single model to represent all data. For example, the model for a dense uniform distribution might say that key k will be on page $1000 + k/100$. This means that the whole index is replaced by a single model. However, this is an extreme approach, and the core idea may be further refined by partitioning the space like a classical tree index and then making further design decisions on how to navigate the additional structure.

Creating the last indexing level of a learned tree. State-of-the-art learned index designs make the decision to first partition the sorted data into many contiguous parts, called segments, each of which contains one or more pages. Within each segment, a separate model can be created that tries to achieve a desired error bound. This is necessary for good model prediction and, thus, I/O performance. One way to think about the segments and the models representing those segments is that they represent the second to last level of a classical tree (i.e., the last level of internal nodes of a B^+-tree).

Recall that in a classical multi-way tree, each interior node with fanout F, has F pointers, one per target page, and $F-1$ key separators. By contrast, a learned index node that employs a linear model approximates the relationship between the $F-1$ values and the F pointers that have to be followed when searching using a linear approximation of the form $pos = value \cdot \alpha + \beta$. Hence, instead of having to store $F-1$ keys and F pointers, it needs to store only the two coefficients α and β, the starting key of the segment as well as a single pointer/offset to the segment.

By contrast, the leaf nodes of a learned index take the same space as the leaf of a B$^+$-tree. Thus the main advantage is that the interior nodes take much less space, as illustrated in Figure 3.13.

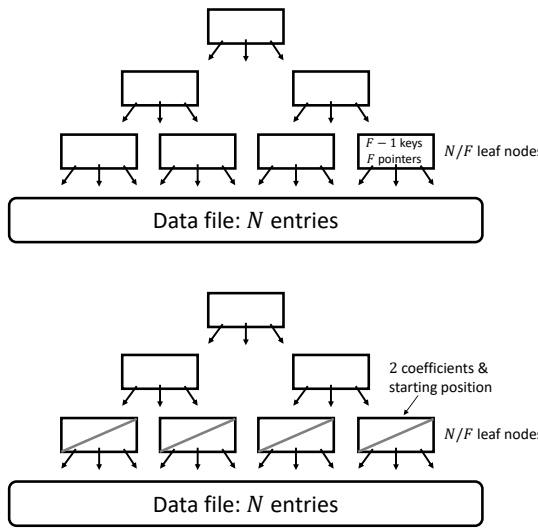

Figure 3.13: Every non-leaf level in a B$^+$-tree (top) stores the key and the pointer for every child node. By contrast, for a learned index (bottom), the nodes above the leaf level need to store only a coefficient and the starting position of each segment in the level below.

In a learned index, each node uses a variation of a local interpolation search. Instead of using the first and the last pairs of *(key, position)* to perform the interpolation, a learned index finds the maximal size of a segment that allows efficient search using an interpolated position. The structure can even provide error guarantees with respect to position.

Figure 3.14 shows an example of a two-segment approximation in a learned index to match the data distribution (blue line) in comparison with a simple interpolation search that might significantly misrepresent the data distribution (red line).

Creating the rest of the indexing levels. Once the segments have been created, approaches differ in how the rest of the navigation structure is created. For example, Ferragina and Vinciguerra (2020) propose the PGM index that recursively creates more models in order to index the segments themselves, thus, creating a full learned tree of models. On the other hand, Galakatos *et al.* (2019) propose FITing-Tree that uses a classical B^+-tree to index each segment. The latter approach creates a hybrid tree that blends classical nodes and models for the last level, i.e., the most space-consuming level of an index and the one that really "points" to the data.

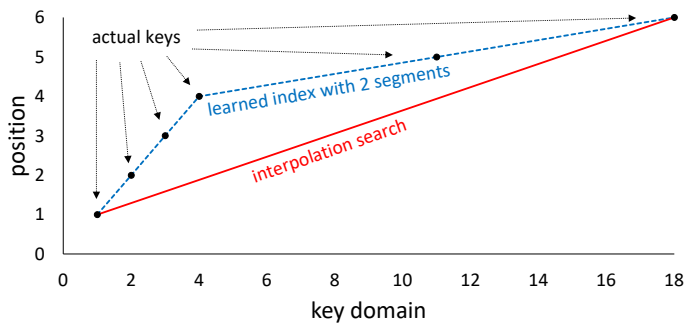

Figure 3.14: The node of a learned tree index approximates the positions using linear regression. In contrast to an interpolation search that would simply use the first and the last point of the dataset (red line), a learned index node creates one or more regression lines (blue lines) whose offsets need not be the lowest key in a subtree. In general, using multiple segments with distribution-dependent length allows a learned index to better match the distribution. In this example, if we had to use a single segment (red line) it would not accurately capture the distribution. Breaking it into two segments captures the distribution almost perfectly.

Performance Tradeoffs. Learned indexes reduce the size of the index at the expense of potential indexing errors in which a search accesses an incorrect child node and must therefore perform additional reads to correct such errors. In addition, learned indexes may require substantial

maintenance effort during updates. If an update changes the distribution of keys substantially, then the data structure needs to retrain its model(s) which can be time-consuming (Ding *et al.*, 2020b). If updates avoid retraining models, then future searches can be very slow because the estimated location of data entries may be highly inaccurate.

Search Cost: The ideal cost of a learned index is $O(1)$ since it directly calculates which page to access using very small metadata that can easily fit in memory and then only one page needs to be read. However, this ideal cost may be unobtainable if the underlying key distribution has several subranges of keys with duplicates or with different distributions, in which case a regression model will be inaccurate and there will be position errors, requiring accesses to many leaf pages. Even when the key distribution is friendly, learning the distribution may require significant time.

Overall, there are three different cost components to be considered: the index access cost before the learning is completed, the cost when it is completed, and the cost to retrain the models after modifications. Empirically, the search cost is $O(1)$ for point queries and $O(k)$ for a range query where k pages are required, but the learning cost has no fixed upper bound and depends on the complexity of the models involved as well as the data properties and the frequency of updates (Kraska *et al.*, 2018).

3.3.7 Summary of Access Costs with Indexing

Here we summarize the cost of point queries and range queries when using indexing for searching, along with the cost to insert new entries in each of the indexing strategies. We assume that we have M pages of available memory. When M is larger than the size of a filter index (e.g., when $M > 2 \cdot N_B \cdot K$ for Zonemaps) then the filter index can be stored in memory and offer the maximum possible benefit when searching. Similarly, the coefficients from learned indexes have a small memory footprint so can be kept in memory. For B$^+$-trees and their variants, the top levels (as many as will occupy M pages) will reside in memory, assuming an ideal page-replacement algorithm.

Table 3.4: Search cost for point queries when using an index. Each row corresponds to one data organization decision ("Org."), and each column corresponds to an indexing decision. Note that some combinations are impossible, that is, the specific index type would not help find a key given that data organization (marked as "–").

Index → **Org. ↓**	Filter Index	Search Tree	Hash Index	Radix Tree	Learned Index
none	$O(N_B)$	–	–	–	–
sorted	$O(1)$	$O(\lceil log_B(N_B) \rceil - l)$	–	$O(\lceil log_S(\mathcal{U}) \rceil - l)$	$O(1)$
logging	$O(N_B)$	–	–	–	–
range part.	$O(N_B/P)$	$O(\lceil log_B(P) \rceil - l + N_B/P)$	–	$O(\lceil log_S(\mathcal{U}) \rceil - l)$	$O(N_B/P)$
hash	$O(N_B)$	–	$O(1)$	–	(open)
radix	$O(N_B/P)$	$O(\lceil log_B(N_B) \rceil - l)$	–	$O(\lceil log_S(\mathcal{U}) \rceil - l)$	(open)
part. logging	$O(N_B)$	–	–	–	–

Table 3.5: Search cost for range queries with selectivity $s\%$ when using an index. The table follows the same structure as Table 3.4.
*Note that the range query using a hash index used the number of entries N and can work only in the case of a discrete domain.
**Further, note that order-preserving hashing will allow for the near-optimal cost of only reading the relevant data pages sequentially.

Index → **Org. ↓**	Filter Index	Search Tree ($PQ + \dots$)	Hash Index	Radix Tree ($PQ + \dots$)	Learned Index
none	$O(N_B)$	–	–	–	–
sorted	$O(\lceil s\% \cdot N_B \rceil)$	$O(\lceil s\% \cdot N_B \rceil)$	–	$O(\lceil s\% \cdot N_B \rceil)$	$O(\lceil s\% \cdot N_B \rceil)$
logging	$O(N_B)$	–	–	–	–
range part.	$O(\lceil s\% \cdot N_B \rceil)$	$O(\lceil s\% \cdot N_B \rceil)$	–	$O(\lceil s\% \cdot N_B \rceil)$	$O(\lceil s\% \cdot N_B \rceil)$
hash	$O(N_B)$	–	$O(\lceil s\% \cdot N \rceil)^*$	–	–
OP hash	$O(N_B)$	–	$O(\lceil s\% \cdot N_B \rceil)^{**}$	–	–
radix	$O(\lceil s\% \cdot N_B \rceil)$	$O(\lceil s\% \cdot N_B \rceil)$	–	$O(\lceil s\% \cdot N_B \rceil)$	(open)
part. logging	$O(N_B)$	–	–	–	–

Table 3.4 shows the cost of a point query for each possible indexing methodology given the data organization. Table 3.5 shows the cost for a range query with selectivity $s\%$. When a query uses a search tree or a radix tree, the cost is calculated by adding the corresponding point query cost from Table 3.4. Further, note that the hash index cost depends on the number of entries N and not on the number of pages N_B, and is applicable only when the domain is discrete and we can map a range query to multiple point queries. This mapping approach quickly becomes inefficient even for very selective queries. On the other hand, in case an order-preserving hashing approach can be used, the range query will have a near-optimal cost.

Finally, in the case of learned indexes, we have identified a few combinations that are possible but currently, they are the subject of active research, and we mark them as "(open)".

Table 3.6: Index insertion cost.

Index	Cost
Sparse Index	$O(1)$
Search Tree	$O(\lceil log_B(N_B) \rceil - l)$
Hash Index	$O(1)$
Radix Tree	$O(\lceil log_S(\mathcal{U}) \rceil - l)$
Learned Index	(open)

Last but not least, we discuss the insert cost in the basic classes of index structures in Table 3.6. Recent literature contains some approaches for updating learned indexes (Ding *et al.*, 2020b), however, the question of how to perform efficient updating is still open as of this writing.

3.4 Local Data Organization

After deciding on the global data organization, the search methods, and the use of indexing to accelerate the location of the correct page in Sections 3.1, 3.2, and 3.3, we must now decide on the local data organization, i.e., how to organize each partition. A given data structure may even organize different partitions differently. Below, we present the five local data organizations which are visualized in Figure 3.15.

In addition to the general goal of creating metadata to help find the sought-for data items fast, local search introduces new goals. One is that local data organizations should fit into cache lines to eliminate accesses to RAM. On certain hardware, the local organization is designed to accommodate SIMD instructions better. Further, operations in local organizations often try to avoid a new memory allocation at every insert or update.

Logged. Entries to a partition are appended in time order to an array, forming a log[3].

[3]Please recall that the term "logged" here does not refer to a data structure that supports recovery from failure (Bernstein *et al.*, 1987), but rather to data being appended in a time-ordered manner.

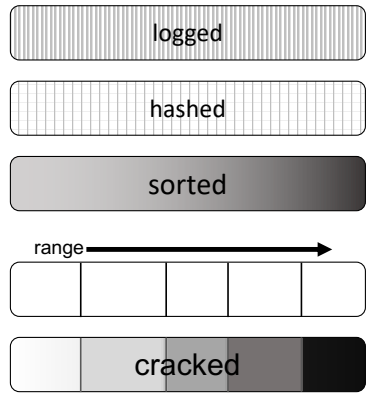

Figure 3.15: The five local data organizations.

Hashed. The partition uses a hash structure internally to store (and locate) the key-value pairs within that partition.

Sorted. The contents of the partition are maintained in sorted order by key.

Range. The contents of the partition are maintained in a range-partitioned manner by key, possibly with a non-sorted organization with those sub-partitions.

Cracked. The contents of the partition are cracked. That is, they are initially logged, but whenever a query (typically a range query) is executed, the data may be *adaptively* organized to become sorted or range-partitioned. Chapter 5 discusses this method and its generalizations in detail.

Performance Implications. Each local data organization has similar performance implications at a local level as the corresponding global data organization without indexes have at the global level. For example, binary search is possible if the local data organization is sorted. The key difference is that local data often fit in memory. In addition, making different global and local decisions may result in hybrid cost models. For example, a data organization may combine partitioned logging at the global level with hashing, sorting, or cracking at the local level.

3.5 Local Search

Once we decide on the local data organization, we must choose an exact local search algorithm, that is, the algorithm for searching within each partition.

Sequential Search. Any local data organization for the contents of a single partition or page can be searched using sequential search, which examines all key-value entries.

Hashing. If the data is organized locally using a hash structure, then answering point queries through hashing is the fastest approach.

Sorted Search. When the data within a partition is sorted, a sorted search algorithm can be employed such as binary search, m-ary search, or some data-driven search algorithm. The considerations are analogous to those described in Section 3.2 for a sorted global organization.

Hybrid Search. Finally, for hybrid local data organizations like range-partitioned, a hybrid sorted-sequential search can be employed. For example, binary search might locate the correct mini-partition, which would then be scanned.

3.6 Modification Policy: In-place vs. Out-of-place

For every organized data collection, an incoming modification (that is update, insert, or delete) on key k will either:

- Perform an ***in-place*** modification: on insert, place the pair (k,v) somewhere in the structure if k is nowhere present; on update, replace the value associated with k by a new value; on delete, remove k and its associated value. The net result will be to preserve the **single-copy** invariant that there is always at most a single instance of k in the structure. This invariant holds for historically familiar data structures such as hash structures and B-trees.

- Perform an ***out-of-place*** modification: every modification will manifest as an append of a pair (k,v) in some previously empty position. The key k may therefore occur many times in the structure, leading to a **multi-copy** data organization. A search that follows

a modification on k must determine which key-value pair for k was the most recent and return the associated value (which will be a "tombstone" when k is deleted, indicating that there is no value associated with k). Out-of-place modifications are typically followed by **periodic eliminations** of duplicate and invalid items to reduce read and space amplification.

3.6.1 In-place Modifications

An in-place modification normally conforms to whatever global and local data organization was present before the modification occurred. For example, if we have a sorted collection and we insert a new key-value pair (k, v), it will have to be inserted in the position dictated by the rank of k in the sorting order as shown in Figure 3.16. If the data organization is range partitioning, the new item should be placed physically in the corresponding partition. In-place modification is the policy used in basic data structures including textbook B$^+$-trees, hash indexes, sorted linked lists, and others. In-place modifications on such data structures enjoy high read performance but may incur a reorganization burden for modifications.

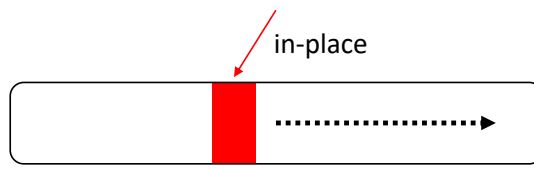

Figure 3.16: In-place modifications insert a new key-value pair (k, v) in the position that corresponds to k in the current data organization. This may require moving several existing entries to "make room" for the entry. If the modification is an update, the value is updated and subsequent entries have to move if the value has a different size from the previous value associated with k. A delete may require contracting entries unless an implementation chooses to mark an entry as deleted.

Impact on PyRUMID Costs. The cost of an in-place modification is equal to the cost of a point search to find the key-value pair to modify and at least one more write access to ensure that the updated key-value pair is on disk. The cost of this approach is presented in Table 3.6.

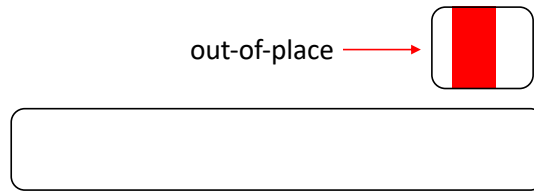

Figure 3.17: Out-of-place updates are directed to new empty positions without disturbing the existing data organization. This leads to potentially multiple copies (when modifications update existing entries), and the need to *garbage collect* invalidated entries periodically to reduce space wastage.

3.6.2 Out-of-place Modifications

An out-of-place modification on key k avoids interfering with the current data organization. Instead, the modification may be stored separately from any other instances of k in the data structure. Following the paradigm of the Log-Structured Merge (LSM) Trees originally proposed by O'Neil *et al.* (1996) and surveyed by Luo and Carey (2020), out-of-place modifications are (a) logged as part of an in-memory container, so modifications require *zero* disk accesses and (b) they do not affect the organization or accessing strategy of the pre-existing data, as shown in Figure 3.17. In an LSM-tree, any new or modified entry is simply appended in a memory-resident buffer. When the buffer becomes full, it is pushed to a container on disk as a set of key-value pairs. As a result, any read query may have to search through multiple data containers to provide the final answer, as shown in Figure 3.18. Space-wise, out-of-place modifications allow for duplication because several key-value pairs may exist for a given key k.

Consolidation. Accumulated modifications may create invalid data entry either through deletion or through updating. These invalid copies, in turn, increase the space amplification of the data structure and the read amplification, leading to worse read performance since more data needs to be sifted through to find the desired data entry. To address this, out-of-place modifications are typically followed by periodic consolidations that *merge* existing data. For example, in an LSM-tree setting, different sorted runs are merged into a single sorted run. During this consolidation phase – which is also termed *compaction*, *garbage collection*,

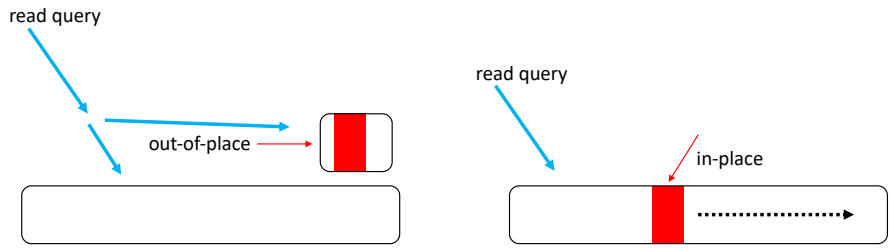

Figure 3.18: The modification policy affects the way we execute read queries. An out-of-place modification allows multiple versions of the same key in different portions of the data, so both the base data and the out-of-place buffer have to be searched. On the other hand, in-place modifications maintain only a single valid copy per key in the base data organization.

or simply *merging* – duplicates and invalid entries are discarded. Note that consolidation necessarily re-writes data previously appended, hence it increases the write amplification, however, it reduces the number of sorted runs that need to be accessed, thus benefiting read performance. This is a classical manifestation of the three-way tradeoff between read, update, and storage costs as outlined in Section 2.3.

Impact on PyRUMID Costs. Out-of-place modifications aggressively minimize the modification cost at the expense of both increased read cost and increased space utilization. When an out-of-place modification appends to a buffer in main memory, it requires zero disk accesses. Searches, however, must find the appropriate partition or page and may perform local re-organization. Practical designs of systems that employ out-of-place modifications, like LSM-based key-value stores, reduce the space and read amplification by merging sorted runs to form fewer longer sorted runs that have fewer or no duplicates and are easier to search. A basic design for an LSM-tree is shown in Figure 3.19. The LSM-tree data structure is discussed in more detail in Section 4.8.

A Practical Comparison of In-Place vs. Out-Of-Place. When compared with an in-place approach, out-of-place modifications significantly reduce the write amplification when ingesting data. For example, Sears and Ramakrishnan (2012) point out that B^+-trees may have a write amplification of up to 1000, while out-of-place modifications in LSM-based key-value stores have significantly smaller write amplifica-

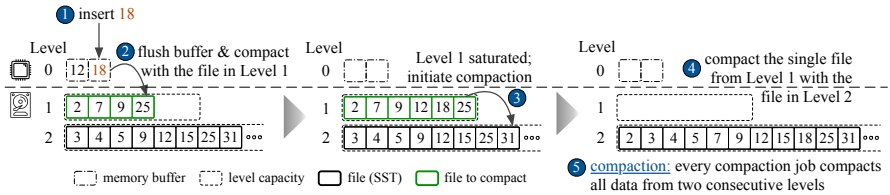

Figure 3.19: An LSM-tree comprises a memory buffer and multiple levels with sorted runs in storage that have exponentially increasing sizes. Once the buffer is full, the entries are sorted and flushed to the next level. If the level already contains data, then the current data and the incoming data are merged in a process called *compaction*, and if after the compaction, the level reaches its maximum capacity, it is flushed to the next level where the same process will be repeated.

tion, typically between 10 and 40, leading to a 25×-100× more efficient ingestion (Sarkar *et al.*, 2021; Dong *et al.*, 2017). In earlier work, Sears and Ramakrishnan (2012) showed that for an ingestion-only workload on a (then) contemporary SSD, bLSM, their LSM-based system, achieved 32K op/s as opposed to 2K op/s for baseline InnoDB, which was the B^+-tree-based storage engine of MySQL. Though technology continually evolves, the reduced write amplification of LSM-trees and their variants compared with in-place data structures gives LSM structures an advantage for insert-heavy workloads.

3.6.3 Differential Out-of-place modifications

A variation of out-of-place modifications is the concept of differential out-of-place modifications. When updating the value associated with a key k, differential out-of-place modifications store key k and the difference from the old value associated with k rather than storing the entire new value (Severance and Lohman, 1976), potentially saving space. This allows for space-efficient updating but may incur higher costs during read queries because it may be necessary to reconstruct the value associated with k from several key-value pairs. This differential approach is most useful when the values are large (e.g., image or video files) and the modifications can be expressed succinctly. The differential approach would not, however, be useful for values consisting of integers or floating point numbers or modestly-sized strings.

Impact on PyRUMID Costs. Differential out-of-place modifications reduce space consumption at the expense of more expensive searches because they require reconstructing the current logical value from a base and possibly several differential values.

3.7 Buffering

In addition to using main memory to index metadata, main memory can be used to buffer read and write requests. The goal is to apply multiple requests (typically called a "batch") in one go.

3.7.1 Buffering Read Requests

A typical goal of buffering read requests is to apply several or all such requests together on each data partition. Such batching of requests will normally cause an increase in individual query response time (because read requests don't return any value while they are buffered) but may result in higher overall throughput. The improvement in throughput comes from the ability to bring a partition into memory and then process all the requests of the buffer on that partition.

Examples. Analytical data systems employ shared scans (also known as cooperative scans) (Arumugam *et al.*, 2010; Candea *et al.*, 2009; Giannikis *et al.*, 2012; Harizopoulos and Ailamaki, 2003; Harizopoulos *et al.*, 2005; Johnson *et al.*, 2007; Mehta *et al.*, 1993; Psaroudakis *et al.*, 2013; Qiao *et al.*, 2008; Unterbrunner *et al.*, 2009; Zukowski *et al.*, 2007), which batch multiple long running queries to exploit common data accesses to achieve concurrent execution of long-running queries with low CPU and disk utilization. A single access over the data set is enough to answer all the queries of the batch provided there is sufficient buffer space for the read requests and the bookkeeping to keep track of which items qualify for which request.

Impact on PyRUMID Costs. Buffering reads increases the read throughput at the cost of increased response time and additional space to store the batched reads.

3.7.2 Global Buffering of Modifications

The typical goal of buffering modification requests is to apply several or all such requests together on each piece of data at one time, including any needed reorganizations. This form of buffering and batching penalizes reads that must scan the buffered modifications but increases modification throughput because the page or partition can be reorganized just once for the entire batch as opposed to once per modification.

Examples include cracking (Idreos *et al.*, 2007a), buffer repository trees (Brodal and Fagerberg, 2003) and fractal trees (Kuszmaul, 2014). Another variant of buffering modifications creates additional metadata in main memory to indicate where the new modification(s) will eventually go in the base data structure (Héman *et al.*, 2010). This leads to fast query performance on both the base data and the pending modifications, assuming that all the pending modifications fit in memory.

Impact on PyRUMID Costs. Every modification is associated with the partition to which it belongs, but is placed in a buffer. The partition is later reorganized with respect to the modifications in the buffer. This reduces reorganization overhead overall but may increase search time because the buffer may have to be scanned by reads to determine the most recent value associated with a search key.

3.7.3 Local Buffering of Modifications

Further, buffered modifications can be stored locally on a per-partition basis as opposed to in a global buffer for the entire data structure. Examples include the B^{ϵ}-Tree (Bender *et al.*, 2015) which resembles a B^{+}-tree but also employs a buffer per internal node to accumulate incoming modifications before forwarding them to the next level of the tree (and, ultimately, to the corresponding leaf node). We will discuss B^{ϵ}-trees in detail in Section 4.3.

Localized buffering makes it easy to process each partition separately. Choosing which partition to process may depend on how many modifications have been buffered per partition (how "write-hot" each partition is) and on search traffic to the partition.

Impact on PyRUMID Costs. Similar to global buffering, local buffering uses additional space in order to apply per-partition modifications. Processing a batch of modifications to a single partition tends to be more efficient when using local buffering than when using global buffering.

3.7.4 Cache Pinning as a Special Case of Buffering

A special form of buffering is **cache pinning**. While caching is not a design dimension of an access method, cache pinning can be considered a particular form of buffering. Instead of buffering requests, cache pinning aims to ensure that frequently accessed data is kept in fast memory. Access methods can identify frequently read items and pin them in the cache. The net effect is to speed access to those items, sometimes dramatically.

Impact on PyRUMID Costs. Cache pinning is the explicit request to keep popular or useful pages in the cache. It guarantees lower latency for pinned page frames but leaves less cache memory available for the remaining pages.

3.8 Key-Value Representation

Independently of the index design, we have to decide how to physically lay out keys and values. This is called *key-value representation*. Below, we present the different options for content representation.

Key-Value Separation. The first decision is whether to store keys and values physically close to one another. The default approach of most data structures is to collocate them. However, because the value is not used when searching, separating the key from the value can increase the speed of searching (Lu *et al.*, 2016).

3.8.1 Key-Record

A natural way to index a table is to associate an entire record with each key. That is, a record constitutes the value in a key-value pair. This approach is common in NoSQL data stores and heap files. The Key-

Record representation is termed the alternative-1 data representation in (Ramakrishnan and Gehrke, 2002).

This representation is used when the primary goal is to avoid additional disk accesses for the value after locating the key. It's especially useful when most or all of the associated record is needed. When only part of the record is typically needed, one can choose to store just those parts of the record with the key.

3.8.2 Key-Pointer (Offset)

When the access method is purely in-memory or uses pointer swizzling (Wilson, 1991) to convert offsets to in-memory addresses (Graefe *et al.*, 2014), the contents of the access method are represented by the indexed key and the corresponding pointer(s). The Key-Pointer content representation is useful when the record associated with a key does not move and having the pointer readily available accelerates data access.

3.8.3 Key-RowID

When the records are stored in a separate file, then we can use a row ID to find the position of the record associated with a key. This approach is frequently used when the base data is stored in a separate heap file in a database system, or when the records are stored in a separate container like a log. This approach is termed alternative-2 or alternative-3 (Ramakrishnan and Gehrke, 2002) and corresponds to the classical database system design for secondary indexes. Note that record IDs can be logical (e.g., primary keys), or physical (e.g., page ID and slot ID).

The Key-RowID content representation works well for queries that do not care about a row's contents (e.g., for count queries). It is also useful when the workload issues frequent updates that change the size of the row, which would cause significant reorganization cost if pointers to records were stored in the index.

3.8.4 Key-Bitvector

A substantially different alternative is to associate a bit vector with each distinct key value. This applies to cases where there are few distinct

key values (e.g., gender, days of week) and relatively many records. The set of bit vectors constitutes a bitmap (Chan and Ioannidis, 1998).

For example, suppose that we have three unique keys k_1, k_2, k_3 and overall we have five entries. Each key will be associated with a 5-bit long bitvector. Now consider that the first entry is equal to k_1, the second and the fourth equal to k_2, and the third and the fifth equal to k_3. The bitvector for k_1 will be `10000`, the bitvector for k_2 will be `01010`, and the bitvector for k_3 will be `00101`, as shown in Figure 3.20.

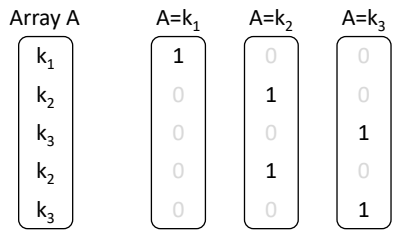

Figure 3.20: A bitmap index representation of an array of values.

Bitvectors are typically aggressively compressed using various encoding schemes like Byte-aligned Bitmap Compression (Antoshenkov, 1995), Word-Aligned Hybrid (Wu *et al.*, 2006), Position List Word Aligned Hybrid (Deliège and Pedersen, 2010) and others (Chan and Ioannidis, 1999; Colantonio and Di Pietro, 2010). A key benefit of most bitvector encoding schemes is that data can be processed directly on the bitvectors for several operations, including selection, projection, joins, and sorting. Bit vectors can further exploit machine-efficient bitwise operations to process data very fast (Ding *et al.*, 2020a).

3.9 Summary of the Design Space Dimensions

This chapter has described the design space of access methods using eight fundamental design dimensions, each representing a key decision made when designing an access method. Table 3.7 shows the options in each dimension.

Table 3.7: Design space of access methods.

(A) Global Data Organization (§3.1)	(B) Search w/o Indexing (§3.2)	(C) Search w/ Indexing (§3.3)	(D) Local Data Organization (§3.4)
no org.	full scan	filter indexing	logging
sorted	binary search	search trees	sorting
logging	direct addressing	hash indexes	hashing
range partitioning	data-driven search	learned indexes	range recursive
hash partitioning			cracked
radix partitioning			
partitioned logging			

(E) Local Search (§3.5)	(F) Modification Policy (§3.6)	(G) Buffering (§3.7)	(H) Contents Representation (§3.8)
sequential search	in-place	buffering reads	key-record
hashing	out-of-place	buff. mod. globally	key-pointer
sorted search	diff. out-of-place	buff. mod. locally	key-rowID
hybrid search		cache pinning	key-bitvector

3.10 Data Structure Design *Expert Rules*

We now present a collection of *expert* rules that can help a designer choose or invent an access method as a *function of the workload and the available resources*. We find that the three most important factors are: (i) the presence of range queries, (ii) the importance of modification performance, and (iii) whether space amplification is tolerable.

1. *Range-prevalent:* If **range queries** are prevalent, then employ *sorted, order-preserving hashing, or range/radix partitioned* data organizations (Section 3.1).

2. *Point-only:* If a workload has **only point queries** then *hash partitioning* will often be faster than sorted or range partitioned global data organizations (Section 3.1), and a *hashed* local data organization will be preferable to a sorted local data organization (Section 3.4).

3. *Selective queries:* If point or range queries are selective, i.e., access a small fraction of the entries, then *indexing* (as described in Section 3.2) and *global data partitioning* are helpful (Section 3.3).

3.a *Small RAM:* If the **index does not fit entirely in main memory** and **indexing** is used, then the fan-out should be high for the parts of the index on disk storage. So, a *B^+-tree or hash index* is preferable to a low-fanout structure, like a binary search tree (Section 3.3.3 and Section 3.3.4).

4. *Size-agnostic search time:* When data size increases and we want to decouple the search time from the data size, we can employ direct addressing (Section 3.2.3) in the form of *hash indexing* (Section 3.3.4) if our workload has only point queries, or *radix-based indexing* when the underlying data is sorted and organized based on radix partitioning (Section 3.3.5). Radix trees (Section 3.3.5) are particularly beneficial when indexed data is highly skewed because the tree itself can take less space than when keys are more nearly uniformly distributed over the key space.

5. *Modification-frequent:* If **inserts/deletes** are frequent, then we differentiate between *moderate* and *intensive* modification traffic. Specifically, when modification traffic is heavy, we may want to employ *logging*, *partitioned logging*, or *hashing* at the global level (Section 3.1). When modification traffic is moderate, then there can be more flexibility at the global level (e.g., B^+-tree style may be fine), but the local level can use a *hashed* or *logged* local data organization (Section 3.4). We also note the following provisions:

   5.a *Batching can replace local logging:* A design that employs a *logged* local data organization to support modification traffic can instead employ a highly organized (e.g., sorted) local data organization and then *batch modifications* either globally (Section 3.7.2) or locally (Section 3.7.3).

   5.b *Overlapping partitions require filters:* A design that employs *partitioned logging* as a global data organization should also employ *light-weight filter indexing* (such as Bloom filters) to eliminate at least some unnecessary page accesses (Section 3.3.2).

6. *Query bursts:* If there are **bursts of read queries**, *batching* them together reduces redundant data accesses. The benefit is maximized

when queries are grouped together based on key-locality, i.e., by batching together queries that target the same partition (or group of partitions). Batching read queries increases overall read throughput, but will also potentially increase the response time of individual operations (Section 3.7) that must wait until their batch is processed.

7. *Update bursts:* If there are **bursts of updates** on sorted data, then *update batching* should be employed in order to reduce overall reorganization costs (Section 3.7.2).

8. *Scan-only:* If the **queries are full scans or large range** queries, then *no indexing* is required (Section 3.2.1 and Section 6.1.2). Large range scans can be optimized using Zonemaps, Bloom filters, and other forms of *filter indexing* (Section 3.3.2).

9. *Scans with logging:* If the **queries are full scans or large range** queries, and **are based on the time of insertion**, then time-ordered *logging* should be employed (Section 3.1). *Filter indexing* (such as Zonemaps, in this case) should also be used (Section 3.3.2).

10. *Locality vs. Flexibility: The decision regarding key-value representation is orthogonal to the other dimensions.* When data **locality** is the prime goal, **keys and values should be stored together**. On the other hand, when building an index, typically a **secondary index**, on a set of records that is already organized, each key of that index should be paired with a pointer to a record or a rowID rather than duplicating the records. Further, if a workload has updates that might **drastically affect the size of values, then keys and values should be stored separately** in order to avoid the data movement reorganization entailed by those updates.

In the next chapter, we discuss how to use these rules both to choose among existing designs for a given workload and to propose new designs if no existing design satisfies all requirements.

3.11 Chapter Summary

This chapter has described the eight major design dimensions of access methods. The first four dimensions concern the structure at both the global and local levels. The next three dimensions concern choices in how to implement operations. The final dimension describes various key-value layout options. The chapter ends with a presentation of expert rules that map workloads to design choices.

3.12 Questions

1. Give an example of a workload for which a sorted global and sorted local data organization would work well. Give an example of a possibly different workload for which a range partitioned global but hashed local organization would work well.
   Answer Sketch: A workload with many range queries would benefit from a sorted global data organization. This makes it easy to locate the end points of a requested range. The sorted local organization would be particularly helpful when the partitions extend over many pages and the range queries are short (so a query could fetch only the relevant pages). On the other hand, a range partitioned global and hashed local organization would be a good fit for a workload that has a mix of point queries and long range queries. In the hashed local organization setting, long range queries would skip some partitions thanks to the global range organization but would have to perform complete scans of the unskipped partitions. Maintaining the local hash organization is likely to be less expensive than maintaining a sorted local organization.

2. We have treated response time and memory costs as different criteria to minimize. Describe some ways in which increasing the size of memory can decrease response time.
   Answer Sketch: Increasing the size of memory may allow putting parts of a data structure in memory or even expanding the data structure design with additional in-memory components. For example, in a B^+-tree design, we may put all internal nodes in

memory. This would mean that a point query over the tree needs just a single I/O operation, giving it the same I/O cost as a hash table.

In addition, increasing memory can also allow the inclusion of filtering indexes such as Bloom filters to avoid accessing data pages when the relevant data is certain to be absent. This is commonly used in key-value stores systems where the LSM-tree is the core structure.

3. How does the distribution of the indexed key affect the size of B^+-trees and radix trees?
   Answer Sketch: *The distribution of the indexed key does not significantly affect the size of a B^+-tree, which is mostly governed by the number of inserted elements (though prefix compression does benefit from skewed data). By contrast, radix trees can benefit from specific data distributions. Highly skewed data allow for aggressive subtree removal, yielding good compression of the radix tree.*

4. When a data structure must hold data that is large enough that most must reside on disk, which is likely to have more impact in terms of end-to-end data system performance: local or global organization? Answer this question for each of the five major operations: point query, range query, insert, update, and delete.
   Answer Sketch: *If much of the data is on disk, then global organization will have by far a bigger effect because that can change the time complexity of disk operations from linear in the size of the data to logarithmic or even constant time for any operation (whether point, range, insert, update, or delete) which must access a single key. In comparison to disk operations, local node operations are so inexpensive as to be considered free.*

   By contrast, if the data fits in memory, then local organization will have a relatively larger influence on overall cost, because local node costs are comparable to the costs of traversing from one node to another.

5. Buffering of updates can be applied at different levels of a B^+-tree structure. Suppose the buffering will be in memory, and all internal levels of the B^+-tree will be in-memory except for the leaf level which will be on disk. Describe the relative benefits of applying buffering at the leaf level versus at any other level of the tree.
   Answer Sketch: *Buffering is most useful if it allows an overall reduction in disk I/O. For this reason, for any level of the tree structure that may be in memory, buffering is only minimally helpful. Therefore, in the setting described in the question, buffering would be most useful at the leaves.*

6. How is the policy of out-of-place updates impacted by workload properties with respect to the order of operations in the workload? Consider the following three workloads in which reads may be point or range queries: (i) all insert operations arrive before all read queries; (ii) all read queries arrive before the inserts; and (iii) reads and inserts are fully mixed in the workload. Present a design for each of these three workload patterns.
   Answer Sketch: *For workload (i), batch the inserts locally on fully sorted nodes and then apply them all at once before the reads. For workload (ii), sort the nodes to support the queries and then batch the inserts and apply when done. For workload (iii), provided most reads are point queries or large range queries, use global range partitioning, but local hashing to support point queries. Range queries might have to read entire nodes, but they would have to do so anyway. If all or most reads are short range queries, then it might be good to refine the range partitioning to within-node range partitioning. Regardless of the form of read traffic, avoid a sorted organization, because maintaining that sort would entail large costs on inserts.*

7. How does introducing an index affect the PyRUMID costs? For example, consider both the increase and reduction of PyRUMID costs resulting from introducing a tree index over sorted data.
   Answer Sketch: *Indexes take up space so increase memory costs. They nearly always reduce read costs, however. They can also*

reduce update, insert, and delete costs, especially for in-place (single-copy) structures, because they lead to a quick identification of where the relevant key(s) is(are).

8. Consider a sorted and unsorted global partitioned data organization. For which organization would Zonemaps be more useful? For which organization would Bloom filters be more useful?
   Answer Sketch: Zonemaps are far more useful for a sorted data organization. For an unsorted organization, the zone ranges would be very large. Bloom filters help most for unsorted data organizations.

3.13 Further Readings

Modeling: The readings by Brodal and Fagerberg (2003), Hellerstein *et al.* (2002), Yi (2009), and Yi (2012) provide a more in-depth view of detailed modeling for diverse operations in data structures.

Transformation-based Design: The following series of works rely on decomposing workloads and treating every workload operation as a potential to change/transform a static data structure design given a set of transformation rules (Bentley, 1979; Bentley and Saxe, 1980; Scholten and Overmars, 1989; Overmars and Leeuwen, 1981; Leeuwen and Overmars, 1981; Leeuwen and Wood, 1980). This way one can generate a tailored design for a given workload.

Extensible Designs via Abstractions: Hellerstein *et al.* (1995) proposed generalized search trees (GiST) as a data structure abstraction. GiST offers an API and a code template that can implement different search trees, to tailor the design to a specific application. For example, GiST can behave like a B^+-tree (Graefe, 2011), an R-Tree (Guttman, 1984; Manolopoulos *et al.*, 2006), an RD-Tree (Hellerstein *et al.*, 1995), or a variety of other variations of tree indexes (like partial sum trees (Wong and Easton, 1980), k-D-B-Trees (Robinson, 1981), Ch-trees (Kim *et al.*, 1989), hB-trees (Lomet and Salzberg, 1990), V-trees (Mediano *et al.*, 1994), and TV-trees (Lin *et al.*, 1994)).

Data Structure Grammar: Another approach to data structure design is from first principles, i.e., from the smallest possible design decisions involved in a full design. The Data Calculator (Idreos *et al.*, 2018a; Idreos *et al.*, 2018b) presents such a grammar of data structures which consists of design principles and design rules. It shows that there exist more than 10^{100} possible data structure designs for key-value data structures. In addition, this work enables a more fine-grained classification of designs and tradeoffs (Athanassoulis and Idreos, 2016; Athanassoulis *et al.*, 2016), as well as providing data structure designs and code automatically by utilizing mathematical modeling and machine learning-enhanced search algorithms (Idreos *et al.*, 2019b).

Rotations and Rebalancing. When data is modified in place and the data organization is augmented with an index structure, this index structure has to be updated as well to reflect the new positions of each element. In order to preserve the benefits of tree-based indexing in the face of inserts, deletes, and updates, dynamic search trees require reorganization to maintain their sub-linear performance. Binary tree variations offer self-balancing through rotations (Sedgewick, 1983). AVL-Trees (Adelson-Velsky and Landis, 1962) and Red-Black-Trees (Bayer, 1972) are two notable examples. New designs of self-rotating self-balancing binary trees strive to ensure that rotations will affect only a local region (Haeupler *et al.*, 2015).

Rebalancing B^+-trees. Dynamic search trees that have many more children, such as B^+-trees and their variations, maintain balance by splitting and merging nodes to support an invariant that every node will be at least 50% full (Graefe, 2011; Ramakrishnan and Gehrke, 2002). The rebalancing happens mostly at the bottom level, and exponentially fewer times as we move towards the root, leading to low amortized insert cost. Further, this approach ensures that the tree will always be balanced, offering the same (logarithmic) path length for every part of the key domain (Bayer and McCreight, 1970; Comer, 1979).

Deletion in B$^+$-trees. While the textbook algorithms for B$^+$-trees include the merging of nodes as outlined above, workloads in which there are more inserts than deletes (even slightly more) will tend to split shortly after merging. In such cases, rather than merging a node n when n has more than 50% empty space, it is better to wait until n is empty and then simply remove it. This approach gives close to the same utilization at far less maintenance cost (Johnson and Shasha, 1989).

Multi-version Indexing. In the discussion of the modification policy in Section 3.6, we note that out-of-place modifications create a multi-copy data organization because older key-value versions of a given key remain in the data structure even after new ones enter. While the multi-copy property is primarily treated as an overhead leading to increased space amplification, as discussed by Dong *et al.* (2017), this artifact of out-of-place modifications provides (unintentional) partial support for a *multi-version* data organization that aims to store, index, and provide access to multiple historic versions of values associated with a given key. Log-like structures that explicitly support multiple versions LHAM by Muth *et al.* (2000) or multi-version B$^+$-trees (Becker *et al.*, 1996; Varman and Verma, 1997). Other multi-version indexes for modern hardware and HTAP applications include (Gottstein *et al.*, 2014; Riegger *et al.*, 2017; Riegger *et al.*, 2019; Riegger *et al.*, 2020), which capitalize on the design of LSM-trees (O'Neil *et al.*, 1996) and Partitioned B-Trees (Graefe, 2003).

4

From Workloads to Data Structures

We now use the framework presented in Chapter 3, specifically the design rules from Section 3.10, to show how we can derive both existing and novel data structure designs starting from the knowledge of target workloads, environmental information (e.g., the available hardware), and the properties of data (e.g., size).

In Chapter 2, we introduced the five fundamental operations that a search structure supports: point queries, range queries (that can further be classified as short range queries, long range queries, and full scans), inserts, updates, and deletes. We classify a workload based on the following three fundamental dimensions:

1. the type and relative frequencies of read queries: point, short range, long range, scans,
2. whether queries focus on data inserted over a certain time range (e.g., data inserted from two hours ago to one hour ago) and
3. whether the workload is modification-heavy or read-mostly.

4.1 Point and Range Queries, Modifications, but Rare Scans (Traditional B$^+$-trees and Learned Tree Indexes)

When the workload is read-mostly and includes range queries but very few scans (what database practitioners call an on-line transaction processing setting), the rules from Section 3.10 suggest the use of a sorted or a range partitioned data organization (rule #1: *range-prevalent*). In addition, when the data does not fit in RAM, indexing would work best with high fanout (rule #3.a: *small-RAM*). Following these rules, we get a data structure design that maps to B$^+$-trees.

The B$^+$-tree design. A traditional B$^+$-tree (Graefe, 2011) uses range partitioning across leaves and internal nodes. All key-value pairs are stored in the leaves. Internal nodes help in navigation. Each leaf node is typically fully sorted. To find the leaf in question, a search navigates the tree. Each modification happens in place, possibly triggering a re-organization of a leaf to maintain sortedness and, relatively rarely, a change to the node structure of the tree through splits or merges. The leaf nodes of a B$^+$-tree store keys and row IDs or keys and entire records. The B$^+$-tree design is visualized in Figure 4.1 and summarized in Table 4.1.

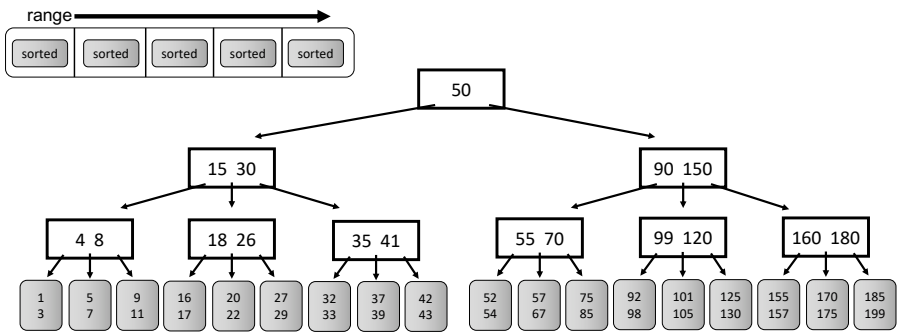

Figure 4.1: A B$^+$-tree design in terms of its design decisions (top left) and its data structure design (main figure). Internal nodes have key separators and pointers to guide a search query toward the leaf that contains the desired data. Every node is sorted, and searching for a key in a node can be done using binary search, though sequential search can sometimes be more efficient (e.g., when the node size is small).

Table 4.1: Traditional B$^+$-trees

Global Data Organization:	Range partitioning
Global Searching:	Indexing
Local Data Organization:	Sorted
Local Searching:	Sequential/Sorted
Modification Policy:	In-place
Batching Requests:	None
Content Representation:	Key-Record, Key-RowID

Learned B$^+$-trees. The recursive model index (RMI) proposed by Kraska *et al.* (2018) implements a tree-based index where the internal navigational nodes are replaced by *models*, i.e., local linear regressions. Ding *et al.* (2020b) augmented this design to make it modification-friendly. While learned indexing is a brand new area of research, in terms of design principles, the two indexes (classical and learned B$^+$-trees) are very similar, having a few different key decisions specifically on how to *search*. The learned B$^+$-tree design is visualized in Figure 4.2 and summarized in Table 4.2.

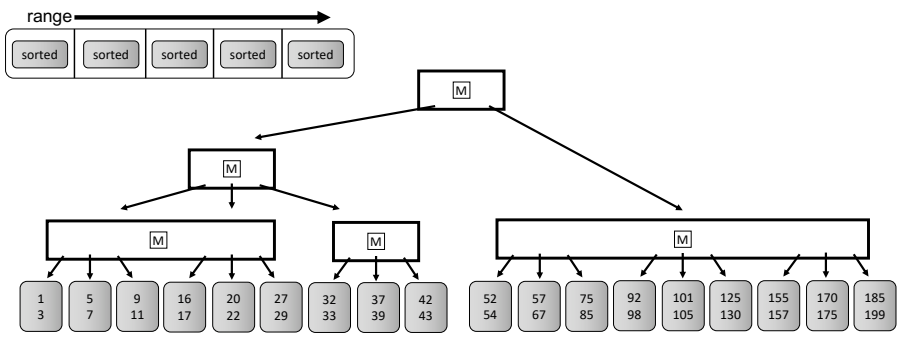

Figure 4.2: A learned B$^+$-tree design in terms of its design decisions (top left) and its data structure design (main figure). Instead of key separators in an internal node, a learned B$^+$-tree has models that predict the position of the pointer that should be followed to find the key in question. The number of entries in an internal node depends on the distribution of the indexed data.

Table 4.2: Learned tree indexes

Global Data Organization:	Range partitioning
Global Searching:	Recursive Models
Local Data Organization:	Sorted
Local Searching:	Interpolation search
Modification Policy:	In-place
Batching Requests:	None
Content Representation:	Key-Record, Key-RowID

4.2 Similar Workload With a Working Set That Fits in Memory (Fractal B$^+$-trees)

We now consider a workload that has a working set that, most of the time, fits in main memory. By following rule #3.a: *small-RAM*, we arrive at a design whose internal nodes have large fanout, but where each internal node consists of many small nodes (based on the unit size of cache memory) having small fanout. These "fractal B$^+$-trees" optimize for different levels of the memory hierarchy with a single design, while efficiently supporting point and range queries as do classical B$^+$-trees. A fractal B$^+$-tree behaves like a traditional B$^+$-tree for disk-resident nodes, but improves in-memory performance. Though they gain in efficiency because of their better locality both on disk and in memory, they require a more complex implementation than classical B$^+$-trees.

The Fractal B$^+$-tree Design. In contrast to traditional B$^+$-trees, fractal B$^+$-trees are tailored for *both* the disk I/O unit size, and the cache memory unit size. A fractal B$^+$-tree node has the size of a disk page, but it contains cacheline-sized mini-pages (Chen *et al.*, 2002). Effectively fractal B$^+$-trees support a hybrid data organization with two levels of range partitioning. The design is visualized in Figure 4.3 and summarized in Table 4.3.

4.3 Point and Range Queries, Rare Scans, More Modifications (Insert-Optimized Search Trees)

For workloads that include point and range read queries as well as a significant number of insertions, rule #5: *modification-frequent* suggests employing logging at each node instead of re-sorting the node

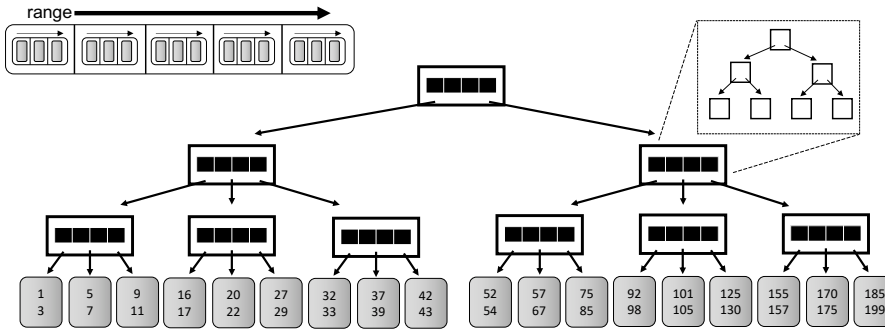

Figure 4.3: A fractal B$^+$-tree shares the same design principles as the B$^+$-tree with an additional level of abstraction where each local organization is range partitioned itself (top left). This is essentially a three-level style of data organization, which is implemented by storing a mini-tree within each non-leaf node (top right). Overall a fractal B$^+$-tree shares the same high-level design as a classical B$^+$-tree (main figure).

Table 4.3: Fractal B$^+$-trees

Global Data Organization:	Range partitioning
Global Searching:	Indexing (Search Tree)
Local Data Organization:	Range
Local Searching:	Sequential/Sorted
Modification Policy:	In-place
Batching Requests:	None
Content Representation:	Key-Record, Key-RowID

after each modification. Such an approach imposes less reorganization overhead on inserts, but requires reads to scan the log at each node instead of using binary search. For disk-resident nodes, this decision has negligible impact because loading the node into memory is vastly more expensive than scanning it. By contrast, for memory-resident nodes, reorganization is expensive relative to simple access, so this decision can improve performance substantially in modification-heavy environments. For example, Levandoski *et al.* (2013) reported speedups between 5.8× and 18.7× over a traditional B$^+$-tree when applying in-memory appending to absorb modifications, combined with a careful latch-free implementation of tree maintenance operations. Note, however, that using the appending strategy for internal nodes would slow down all tree traversals, so appending is employed only at leaf nodes.

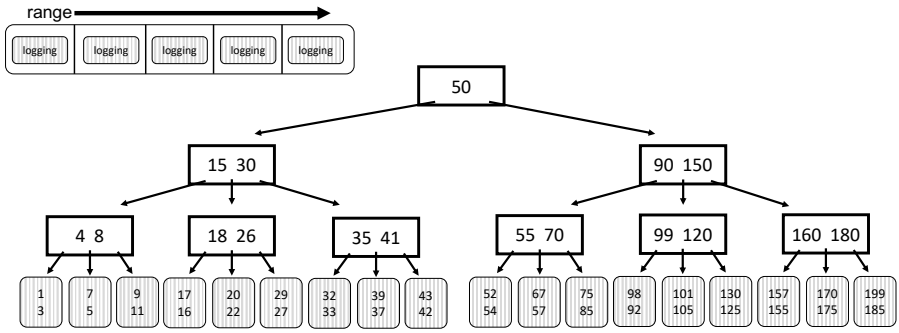

Figure 4.4: The various insert-optimized B$^+$-tree designs maintain the same internal node structure as traditional B$^+$-trees, but instead of maintaining all leaf nodes sorted, they append incoming changes. The contents of the leaf nodes are only lazily sorted, typically at split time or adaptively at query time.

The Insert-Optimized B$^+$-tree Design. Two approaches have proposed B$^+$-trees whose nodes are not sorted internally. The unsorted B$^+$-tree for non-volatile memories (Chen *et al.*, 2011) aims to minimize write amplification and the Bw-Tree design (Levandoski *et al.*, 2013) aims at faster updates through time-ordered logging. Both approaches support range partitioning of the data across leaf nodes but unsorted data within each node. The class of insert-optimized B$^+$-tree designs is visualized in Figure 4.4 and summarized in Table 4.4.

Table 4.4: Insert-optimized B$^+$-trees

Global Data Organization:	Range partitioning
Global Searching:	Indexing (Search Tree)
Local Data Organization:	Logging
Local Searching:	Sequential + Binary Search
Modification Policy:	Deferred In-place
Batching Requests:	Batching Modifications via Local Buffering
Content Representation:	Key-Record, Key-RowID

The B$^\epsilon$-tree Design. Brodal and Fagerberg (2003) proposed a significantly different insert-optimized B$^+$-tree variation, called a B$^\epsilon$-tree. The core idea is that instead of logging incoming key-value data only at the leaf level, a buffer is attached to all internal nodes, following rules

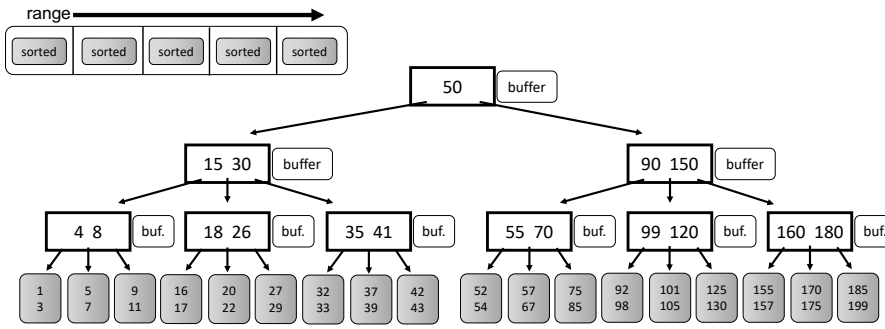

Figure 4.5: Another way to absorb incoming data efficiently is to employ buffering. The B$^\epsilon$-tree design follows the same overall decision as classical B$^+$-trees, with the addition of per-node buffering across all levels to allow for lazy insertions that are only gradually propagated toward the leaf nodes.

#5.a: *batching can replace local logging* and #7: *update-bursts*. Thus, incoming data is buffered both globally and locally. The unbuffered data is maintained sorted following rule #1: *range-prevalent*. Searching for a key entails searching both first through the buffered data and then through the unbuffered data at each level.

The B$^\epsilon$-tree borrows the overall structure of the classic B$^+$-tree. The difference is that every internal node n, in addition to the classic pivot-pointer structure to navigate the domain of the indexed key, is augmented with a buffer to store incoming data. That data will eventually be propagated to appropriate leaf nodes of n. Leaf nodes store fully sorted key-value pairs as in a classic B$^+$-tree. Incoming data are stored in the buffer of the root node (global buffering), and once the buffer is full, the data entries are propagated to buffers of the children with the corresponding key range (local buffering). Every buffer maintains entries in sorted order, and there is no overlap between sibling nodes. Having the data sorted in the buffers allows for the quick selection of which entries to push to the next level of the tree. Overlap may exist at different levels of a single root-to-leaf path. In that case, an internal node may buffer a key-value pair (k, v) that invalidates a key-value pair (k, v') that is at the leaf level in the spirit of LSM-trees.

A query for a specific key navigates the tree as in a B$^+$-tree except that, at each level L, it first performs a binary or other sorted search

Table 4.5: B^{ϵ}-trees

Global Data Organization:	Range Partitioning
Global Searching:	Indexing (Search Tree)
Local Data Organization:	Sorted
Local Searching:	Sorted Search + Sorted Search
Modification Policy:	Out-of-place
Batching Requests:	Batching Modifications via Global and Local Buffering
Content Representation:	Key-Record, Key-RowID

in the buffer at L before following the pointer in the node at level L towards a child node. This design is tunable by a parameter ϵ, which controls the fraction of the internal nodes dedicated to the pivot-pointers vs. the buffer. Because of the combination of global and local buffering, the B^{ϵ}-tree can absorb a small burst of updates, inserts, and deletes at negligible cost while offering read performance similar to a B^{+}-tree (Bender *et al.*, 2015). For large bursts of modifications, the cost of continuous merging becomes more pronounced. Further, answering a range query requires merging multiple sources (buffers from each level and leaf nodes). The B^{ϵ}-tree design is visualized in Figure 4.5 and summarized in Table 4.5.

The SA B^{+}-tree design. Raman *et al.* (2023) has proposed a variation of an ingestion-optimized B^{+}-tree, which aims to benefit from partial data pre-sortedness. The design follows the same overall principles of the above insert-optimized indexes, and focuses specifically on the rule #7: *update-bursts* to employ global buffering for the incoming data.

The intuition of this design is that indexes can be thought of as a means to add structure (*sortedness*) to an otherwise unsorted data collection. As a result, if the data entries are inserted in-order or in a *near-sorted* fashion, the ingestion should be more efficient. In fact, when data is fully ordered, they can be very efficiently bulk loaded. However, when data is near-sorted, then most indexes would treat the data in the same way as if the data entries were scrambled. By contrast, the sortedness-aware B^{+}-tree (SA B^{+}-tree) uses a *sortedness buffer* to absorb near-sorted data, which are appended as they arrive, and at

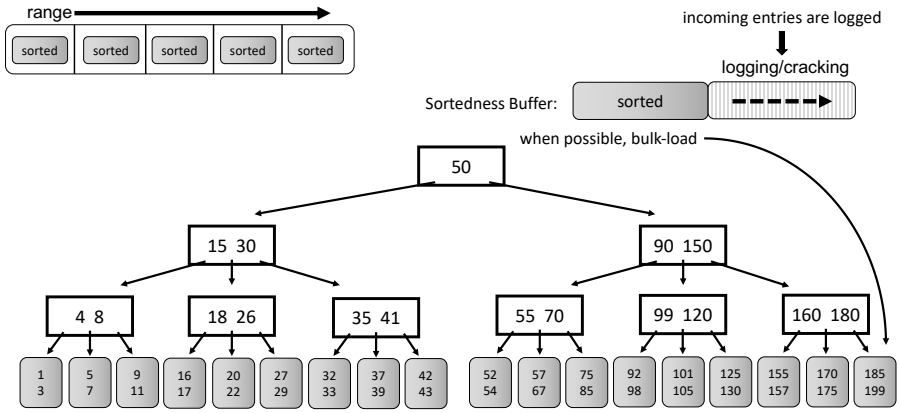

Figure 4.6: With a global buffering scheme, an SA B$^+$-tree aims to absorb update bursts, specifically targeting near-sorted incoming data. The global *sortedness buffer* allows the tree to maximize the fraction of entries that can be bulk-loaded when they arrive near-sorted. The cracking approach used in the buffer, allows read queries to exploit any data sorting effort done for previous queries.

query time, they are cracked (discussed more in Chapter 5) as needed. Once the buffer is full, the cracked data is fully sorted, so if there is no overlap with the data already in the tree, that data is bulk loaded. Otherwise, the data is inserted starting from the root. SA B$^+$-tree outperforms a traditional B$^+$-tree by up to 5× for mixed read/write workloads, due to achieving a 8.8× improvement in ingestion efficiency for near-sorted data. On the other hand, when the data is completely scrambled, SA B$^+$-tree pays a cost of 10%-20%. Overall, any workload with a non-negligible degree of sortedness would benefit from the SA B$^+$-tree design, which is visualized in Figure 4.6 and summarized in Table 4.6.

4.4 Mixed Workload With No Short Range Queries (Hybrid Range Trees)

For a workload having point queries, long range queries, and no short range queries, the combination of rules #1: *range-prevalent* and #2: *point-only* give us a new hybrid design that employs range partitioning at a coarse level and hashing within each partition. That way, point

Table 4.6: SA B$^+$-trees

Global Data Organization:	Range Partitioning
Global Searching:	Search Tree + Filters + Zonemaps
Local Data Organization:	Sorted
Local Searching:	Sorted Search + Sorted Search + Scan
Modification Policy:	Out-of-place
Batching Requests:	Batching Modifications via Global Buffering
Content Representation:	Key-Record, Key-RowID

queries can be quickly answered following a shallow tree access (to the relevant partition) and a hash search. Long-range queries can be answered by accessing only the relevant partitions.

Since we assume only a moderate number of modifications, we use a hashed local data organization following rule #5: *modification-frequent*. Thus, the overall design maintains a hybrid range-partitioned tree structure with a cheaper-to-update local data organization in the face of modifications because entries are hashed in place. As a result, this design also enjoys low space overhead.

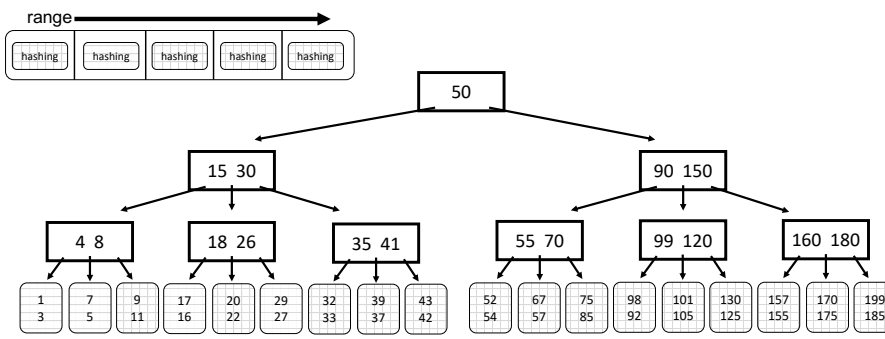

Figure 4.7: The bounded disorder access method employs a tree-based range partitioning global organization with hashing for local data organization (top left). Essentially, this design borrows the internals of a classical B$^+$-tree with hash buckets at the leaf level.

The Bounded Disorder Access Method Design. A design that combines range and hash partitioning in this way is the bounded disorder access method proposed by Litwin and Lomet (1986), which organizes

Table 4.7: Bounded Disorder Access Method

Global Data Organization:	Range partitioning
Global Searching:	Indexing (Search Tree)
Local Data Organization:	Hashing
Local Searching:	Hashing
Modification Policy:	In-place
Batching Requests:	None
Content Representation:	Key-Record, Key-RowID

keys in large range partitions, and within each partition, the keys are hashed in a dense hash table. Thus, a point query is directed to the appropriate range partition within which they can use direct addressing (hashing). Range queries will consume the hashed partitions in much the same way as they consume data in the case of pure range partitioning. The bounded disorder access method offers better point query performance and more efficient maintenance after modifications than classical B^+-trees, but worse support for short range queries. The design is visualized in Figure 4.7 and summarized in Table 4.7.

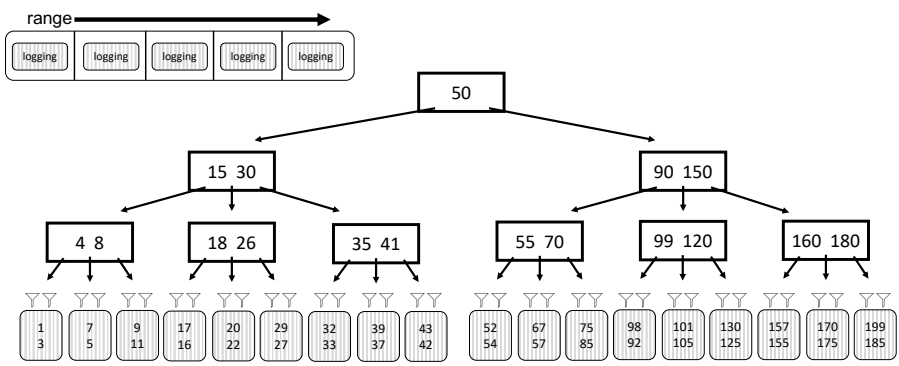

Figure 4.8: The BF-Tree employs a tree-based range partitioning global organization with logging for local data organization (top left). The design borrows the internals of an insert-optimized B^+-tree with the difference that each range partition typically consists of multiple pages (as opposed to a single leaf page in B^+-tree-variants). Each page is equipped with a local Bloom filter, so at point query time, the vast majority of accesses to unnecessary pages are avoided, while maintaining a significantly smaller data structure.

Table 4.8: BF-Trees

Global Data Organization:	Range partitioning
Global Searching:	Indexing (Search Tree)
Local Data Organization:	None/Logging
Local Searching:	Scan/Bloom filters
Modification Policy:	In-place
Batching Requests:	None
Content Representation:	Key-Record, Key-RowID

The BF-Tree Design. A variation of the bounded disorder access method that allows the leaf nodes to have a lightweight local data organization (*none* or *logging*) and can support similar workloads is the Bloom filter tree (BF-Tree), proposed by Athanassoulis and Ailamaki (2014). BF-Tree has the same range-partitioning as the bounded disorder access method at the global level, however, the key-value pairs of each partition are indexed in multiple Bloom filters, one per page of the partition they belong. That way, a point query is directed to the appropriate range partition, within which, they probe all Bloom filters to find which pages contain the desired key. Long range queries consume entire partitions and discard unnecessary entries at the first and last partitions of the range. Like the bounded disorder access method, BF-Trees do not target short range queries, but they offer efficient point and long range queries, along with substantial space savings. The design is visualized in Figure 4.8 and summarized in Table 4.8.

4.5 Mixed Workload, With Ever Increasing Data Size (Radix Trees)

The access cost of a B^+-tree depends on the total number of elements inserted in the index. While this is a logarithmic cost, it still increases with data size. As a result, when we have a mixed workload and we want to decouple the access cost from data size, we can use the radix of the domain to index the incoming entries following rule #4: *size-agnostic search*. The mixed workload includes range queries implying that rule #1: *range-prevalent* still applies. Because the data size exceeds RAM size, following rule #3.a: *small-RAM*, one possible design is a tree based

on radix partitioning that uses groups of several radix bits in each interior tree node, leading to a high fanout.

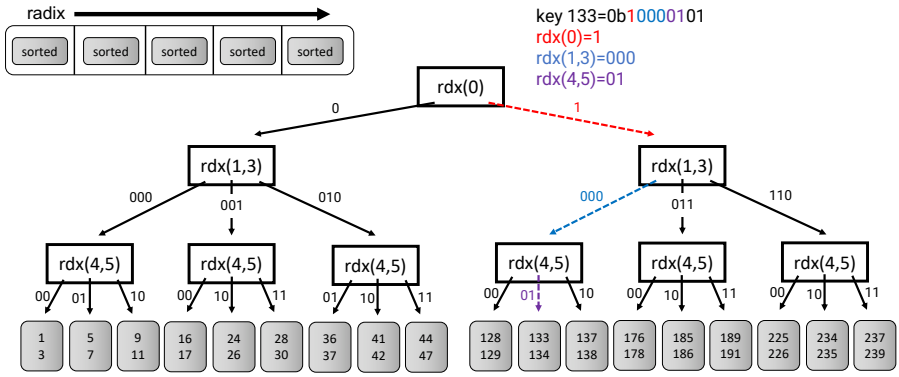

Figure 4.9: Radix trees employ a radix global data organization with sorted data locally (top left). The internals of the tree use only parts of the binary representation (radix) of the key. Consider key 133. Its binary representation is $0b10000101$. In the sample radix tree, the root node corresponds to the first bit of the key, while the second level to the following three bits. Note that a node does not always have all possible children since a subtree can be removed if no keys exist in that part of the domain. The tree's third level corresponds to the radix's fourth and fifth bit. Overall, following the arrows for $rdx(0) = 1$, $rdx(1,3) = 000$, and $rdx(4,5) = 01$, we arrive at a leaf node that stores all 8-bit keys having a prefix of 100001, viz, $0b10000100$ (132) through $0b10000111$ (135).

The Radix Tree Design. Putting everything together, radix trees use radix global partitioning and sorted local data organization. The radix tree design is visualized in Figure 4.9 and summarized in Table 4.9. Masstree (Mao *et al.*, 2012) and the adaptive radix tree (Leis *et al.*, 2013) are two such designs.

4.6 Point Queries, Inserts, and Some Modifications (Static Hashing with Overflow Pages)

Since this workload has no range queries, hash partitioning at the global level can be very effective (rule #2: *point-only*). In fact, for both point queries and modifications, hash partitioning allows navigating directly to the appropriate partition. Using hash partitioning is consistent with rule #5: *modification-frequent*. Further, following the same rule, we

Table 4.9: Radix Trees

Global Data Organization:	Radix partitioning
Global Searching:	Indexing (Compressed Search Tree)
Local Data Organization:	Sorted
Local Searching:	Sorted Search
Modification Policy:	In-place
Batching Requests:	Batching Modifications via Local Buffering
Content Representation:	Key-Record, Key-RowID

employ a logged data organization when there is no space for in-place modifications. This design allows for efficient point queries and insertion, but point query latency may increase because of the need to search the local logs.

The Static Hashing with Overflow Pages Design. Another alternative is a global organization based on hash partitioning. This creates partitions of keys whose hash values are the same. A typical organization of those partitions is simply to append items in the same bucket as they arrive, effectively employing logging. This hybrid between hash partitioning and logging is used by the textbook hash index having chained overflow pages (Ramakrishnan and Gehrke, 2002). It is visualized in Figure 4.10 and is summarized in Table 4.10.

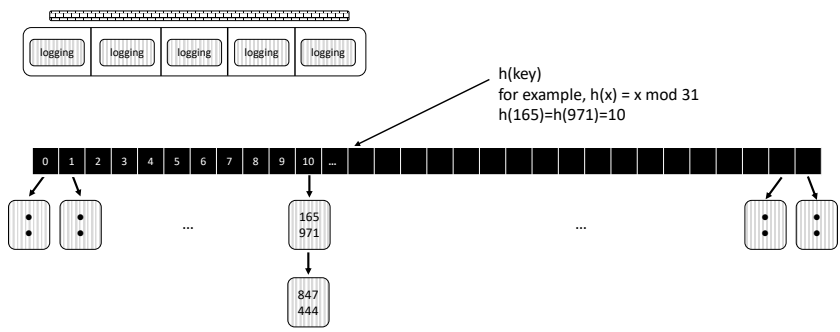

Figure 4.10: Static hashing with overflow pages uses hash partitioning for global partitioning and logging for local partitioning (top left). Any insert or query uses the selected hash function to find the corresponding bucket. Each read query reads the appropriate chained bucket sequentially. Inserts are appended at the end of the appropriate chain.

Table 4.10: Static Hashing with Overflow Pages

Global Data Organization:	Hash partitioning
Global Searching:	Direct Addressing (via hashing)
Local Data Organization:	Logging
Local Searching:	Sequential
Modification Policy:	In-place
Batching Requests:	None
Content Representation:	Key-Record, Key-RowID

4.7 Read-mostly With Long Range Queries (Scans with Zonemaps)

Consider a workload consisting of many long range queries based on time intervals and few modifications that mostly consist of appending new data. Following rule #8: *scan-only*, we do not need to enforce any key-based data organization. Inserts can simply be appended to a linear data structure. In order to reduce the cost of large range searches, order-based filter indexing, like Zonemaps, can help.

Scans with Zonemaps. Full scans and long range queries do not benefit from classical indexing since they have to access most data anyway (Kester *et al.*, 2017; Selinger *et al.*, 1979). Zonemaps help such scans as lightweight metadata that help to skip some unnecessary pages (Moerkotte, 1998). Such a design is frequent in systems for analytical data processing both at the disk level (Francisco, 2011) and in memory (Lang *et al.*, 2016; Qin and Idreos, 2016). It is visualized in Figure 4.11 and summarized in Table 4.11.

Table 4.11: Scans with Zonemaps

Global Data Organization:	None/Logging
Global Searching:	Full Scan (Filter Indexing)
Local Data Organization:	N/A
Local Searching:	N/A
Modification Policy:	In-place
Batching Requests:	N/A
Content Representation:	Key-Record

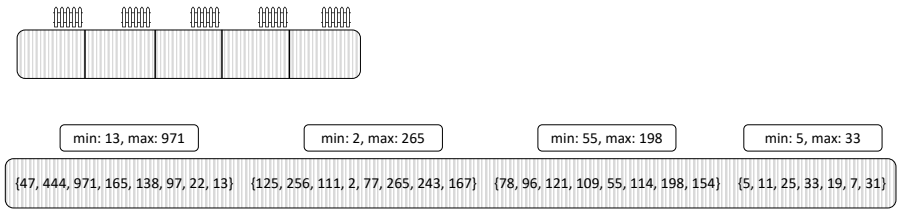

Figure 4.11: Zonemaps store the minimum and maximum key of a block of data which can be at the granularity of a single page or several pages. They constitute a low-cost method for reads to avoid reading a data block when there is no particular global data organization in a data structure design. Zonemaps are especially attractive when keys are sequential (i.e., later inserts have greater keys than earlier ones).

4.8 Modification-intensive With Point and Range Queries (LSM-tree)

For a workload having many modifications, rule #5: *modification-frequent* suggests employing partitioned logging, where each level partitions the key space, but different levels may have keys in common. In addition, rule #5.b: *partitioned logging benefits from filters* suggests employing lightweight metadata within each level to avoid unnecessary data accesses. Finally, rule #1: *range-prevalent* suggests sorting every level for the sake of read query performance. Overall, this design offers high modification performance as modifications are immediately buffered upon arrival. It does so at the expense of both query performance (because even point queries may access blocks at different levels) and space amplification. Both B$^\epsilon$-trees and LSM-trees buffer incoming data globally and gradually push them towards a bottom layer of data (leaf) nodes. A key difference lies in the fact that LSM-trees can select any subset of the data to be propagated when a level is full, while B$^\epsilon$-tree must select the data that correspond to the buffer that just became full, thus reducing the flexibility allowed for performance optimizations.

The LSM-tree Design. Log-structured merge (LSM) based data structures use buffering with periodic sorting and merging of potentially overlapping levels (or, in some cases, partitions within a level). New entries are logged to an in-memory structure and then spilled to disk as needed. In order to offer competitive read performance, LSM-trees

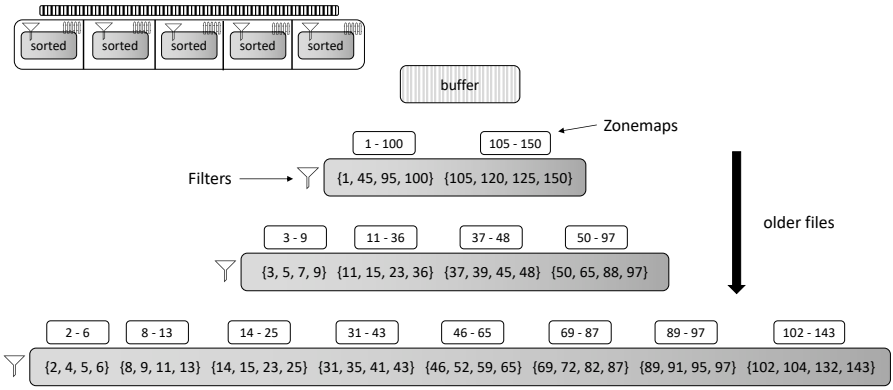

Figure 4.12: LSM-trees employ partitioned logging for their global data organization and each level is sorted. They also employ auxiliary data structures; Zonemaps to help with range queries, and filters to help with point queries (top left). An LSM-tree consists of multiple potentially overlapping sorted levels organized from newer data at or near the root to older data farther down in the tree. A query starts from the most recent data and uses the auxiliary data structures to skip unnecessary nodes.

organize the data that is spilled to disk in various ways, the simplest being to sort the spilled data and then to store it as immutable files (Luo and Carey, 2020; O'Neil *et al.*, 1996). The design is visualized in Figure 4.12 and summarized in Table 4.12.

Table 4.12: LSM-trees

Global Data Organization:	Partitioned Logging
Global Searching:	Filter Indexing (with index)
Local Data Organization:	Sorted
Local Searching:	Sorted Search
Modification Policy:	Out-of-place
Batching Requests:	Batching Modifications via Global Buffering
Content Representation:	Key-Record, Key-RowID

LSM-trees can be further optimized in orthogonal ways. For example, various memory allocation strategies can change the read vs. update performance tradeoff (Dayan *et al.*, 2017; Dayan *et al.*, 2018). Lazy and eager merging of sorted runs can optimize for write-intensive or read-intensive workloads respectively (Kuszmaul, 2014; Luo and Carey, 2020). Another approach, called lazy leveling, performs lazy merging

throughout the tree, except at the last level and uses additional memory to reduce the read costs when needed (Dayan and Idreos, 2018). Finally, a scheme with different merging frequency per level (doubly exponentially increasing size ratios) allows LSM-trees to further reduce write costs (Dayan and Idreos, 2019). Each of the above designs targets specific workloads. Overall, they allow the LSM-tree data structure to navigate a better performance front. For a more detailed discussion on the variations of LSM designs and layouts, the interested reader is directed to surveys, tutorials, and interactive demos by Sarkar *et al.* (2021), Sarkar and Athanassoulis (2022), Sarkar *et al.* (2022), and Sarkar *et al.* (2023).

4.9 Modification-intensive With Point Queries Only (LSM-hash)

Similarly to the previous design, we employ partitioned logging following rule #5: *modification-frequent*. In addition, since we have only point queries, rule #2: *point-only* suggests that we should employ a hashed local data organization.

The LSM-hash Design. To maintain efficient ingestion, we store incoming data using a partitioned logging approach: each partition uses hashing for its local data organization, and the updates are applied out-of-place in the log of each partition. This approach can efficiently support point queries – similarly to what hashing offers – but not range queries, for which it will have to scan the entire data set. Note that order-preserving hashing (e.g., a learned hash function as proposed by Sabek *et al.*, 2022) could help LSM-based data structures to efficiently support range queries, too. However, such an approach has not been fully worked out as of this writing. The derived design is termed LSM-hash[1] and has inspired a family of data structures (Andersen *et al.*, 2009; Badam *et al.*, 2009; Chandramouli *et al.*, 2018; Debnath *et al.*, 2010; Debnath *et al.*, 2011; Sheehy and Smith, 2010; Wu *et al.*, 2015). The design is visualized in Figure 4.13 and summarized in Table 4.13.

[1]Some approaches use the term "LSM-trie" (Wu *et al.*, 2015), however, the disk-component data organization is hashing, so we denote this strategy as "LSM-hash" to avoid confusion with radix partitioning.

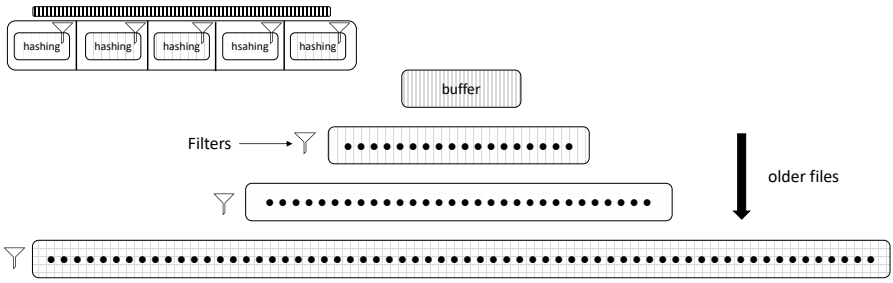

Figure 4.13: LSM-hash also employs partitioned logging as the global data organization, but since it targets only point queries, the local organization selected is hashing with filters. The most recent modifications are at the root.

Table 4.13: LSM-hash

Global Data Organization:	Partitioned Logging
Global Searching:	Filter Indexing (with index)
Local Data Organization:	Hashing
Local Searching:	Hashing
Modification Policy:	Out-of-place
Batching Requests:	Batching Modifications via Global Buffering
Content Representation:	Key-Record, Key-RowID

4.10 When to Design Heterogeneous Data Structures

This chapter's discussion until now has assumed the workload is uniform across the data structure (i.e., across the key space). That would imply that every partition should use the same local strategy. We now consider workloads that differ on different partitions of the data.

Insert-heavy with point queries that target older data. Consider a workload that is insert-heavy, but older data is read-only. If all queries are point queries, the older data should be stored in a hash structure (rule #2: *point-only*). For newly inserted data, however, we can employ a simple logging approach to reduce the insertion cost.

Insert-heavy with range queries on older data and point queries on recent data. In this case, we can further differentiate the decision per partition. For example, the newest partition should employ a hash

local data organization to facilitate point queries (rule #2: *point-only*), while older partitions can maintain the data sorted to answer range queries (rule #1: *range-prevalent*).

In general, heterogeneous workloads suggest a system that makes local decisions per partition and employs different indexing strategies for different partitions. Extending the example above, one partition may employ hashing, another partition may use a local B$^+$-tree, a third partition may be organized as a log to accept many inserts, and a fourth partition may employ a local learned index because its data distribution is roughly linear and seldom receives updates. Moreover, any partition that supports sorted data can use a log buffer to amortize data reorganization costs when encountering bursts of inserts (rule #7: *update-bursts*). A system that supports such heterogeneity should also have the mechanisms to decide which index to build in each partition as proposed by Olma *et al.* (2020).

Note to the reader: Heterogeneity can be decided at design time as discussed above. However, it can also be determined and maintained dynamically if we allow the design to *adapt* using incoming workload information as we discuss in Chapter 5.

4.11 Data Structures in Practice

Adoption by Production Systems. Many of the designs we present in this chapter were first proposed as part of research efforts and were later implemented in production systems.

- B$^+$-trees were originally proposed by Bayer and McCreight (1970) to store index pages for large random-access files. Today B$^+$-trees are part of virtually every relational database system. Systems like PostgreSQL, MySQL, Oracle, IBM DB2, Microsoft SQL Server, SQLite, and others all use B$^+$-trees as one of the core data structures used for secondary indexing (Ramakrishnan and Gehrke, 2002). In addition to database systems, several file systems like Apple's HFS+ and APFS, Microsoft's NTFS, and Linux's Ext4 use B$^+$-trees.

- The case for Learned Indexes was first made by Kraska *et al.* (2018). As of this writing, learned indexes have attracted both research and industrial interest, including that of large database vendors like Google (Abu-Libdeh *et al.*, 2020) and Microsoft (Ding *et al.*, 2020b).
- Insert-optimized B^+-trees were developed as part of research projects carried out by database system vendors like Microsoft and HP (Chen *et al.*, 2011; Levandoski *et al.*, 2013).
- The B^ϵ-tree design was proposed by Brodal and Fagerberg (2003) as a theoretical result, and it was ultimately used as the core data structure of TokuDB (Bender *et al.*, 2015) and BetrFS (Jannen *et al.*, 2015).
- The bounded disorder access method was originally proposed by Litwin and Lomet (1986), and it has inspired research in hybrid data structures that combine different seemingly incompatible data organization decisions.
- The radix tree design is one of the oldest and most widely used designs (Morrison, 1968). Since its inception, it has been a core data structure for file systems, database systems, and operating systems.
- Hash indexing is also a core data structure ubiquitously used in database systems to support point queries (Ramakrishnan and Gehrke, 2002).
- Scans with Zonemaps was originally developed as a research prototype by Moerkotte (1998) and has since been implemented in virtually every column-store analytical data system (Abadi *et al.*, 2013).
- The LSM-tree design was proposed by O'Neil *et al.* (1996). Since its inception it has been used as the core data structure of a variety of production storage engines and systems, including Google's BigTable (Chang *et al.*, 2006) and LevelDB[2], Facebook's RocksDB

[2] https://github.com/google/leveldb/

(Dong *et al.*, 2021), Alibaba's X-Engine (Huang *et al.*, 2019), Apache HBase[3], Cassandra[4], and AsterixDB (Alsubaiee *et al.*, 2014).

- The LSM-hash design was invented as part of various research projects (Andersen *et al.*, 2009; Badam *et al.*, 2009). The LSM-hash design has since been implemented in production environments, like GitHub's metadata infrastructure at Microsoft (Chandramouli *et al.*, 2018).

Programming Language Support. In order to implement data structures for practical systems, we need programming language constructs that allow for low-level control of memory and storage management. Programming languages like C, C++, and Rust allow the developer to make fine-grained decisions regarding memory allocation and layout, so they are appropriate. By contrast, high-level languages like Python, Java, and Go make memory and storage decisions for the developer, so they are not appropriate. Even in low-level languages, libraries may make different assumptions about and may even automate memory and storage management, so they should be used with caution.

4.12 Chapter Summary

The design and implementation of data structures remains an art rather than an "exact science". However, this book, in general, and this chapter, in particular, presents an effort to systematize data structure design decisions, following the vision towards "a calculus of data structures" by Tarjan (1978). The ultimate goal, elusive as of yet, is to build an algebra of components that can be easily synthesized using a high-level description while offering all the benefits of a carefully crafted implementation with low-level optimization (Idreos *et al.*, 2018b).

This chapter showed how state-of-the-art data structure designs for particular workloads can be derived from the design dimensions and the design guidelines. It also showed both the need and the design of hybrid

[3] https://hbase.apache.org/
[4] https://cassandra.apache.org/

data structure designs given a heterogeneous workload description. Hybrid designs are especially promising for the increasingly dynamic workloads and hardware contexts on the cloud. Hybrid designs have started to appear in the research literature (Chatterjee *et al.*, 2022; Huang and Ghandeharizadeh, 2021).

The next chapter introduces and analyzes adaptivity as a core design dimension that spans the design space. The following chapter discusses how the design considerations apply once a data structure becomes part of a larger complex system. In that setting, performance is not only about one data structure at a time but rather depends on how a complex system's numerous data structures interact with one another.

4.13 Questions

1. Why are Bloom filters particularly useful for LSM-trees in which the global organization for the on-disk data is based on the order of arrival? How useful are Bloom filters for B^+-trees?
   Answer Sketch: Consider searching for a key on such a structure. The key could be on one or more of the many unchanging on-disk pages. Bloom filters will help avoid accessing many such pages. At another extreme, in a B^+-tree, a given key may be found in only one page, so Bloom filters might eliminate only one access, thus will be less useful.

2. When might a radix tree work better than a B^+-tree? And vice versa?
   Answer Sketch: For both structures, the data has a global sorted organization. If the pages are laid out contiguously, then the radix index can fit entirely within memory because the key leads directly to the page. That works well if the data doesn't change. If it does change, then laying out the data contiguously is infeasible, so a B^+-tree would be better.

3. When might an interpolation-based method work better than a B^+-tree? And vice versa?
   Answer Sketch: An interpolation-based method that locates keys based on an assumed distribution uses far less space per internal

node than a B-tree (two or three values along with corresponding pointers vs. thousands of separators), so an interpolation-based method is likely to use less space. Further, almost all of the pointers can be avoided if we assume fixed-sized entries and contiguous levels. On the other hand, the interpolation-based method can lead a search to an incorrect page, so that it may require more accesses to slower memory. This inaccuracy problem is likely to become worse if there are many inserts and deletes.

4. Suppose your data structure design keeps all key-value data at the leaf pages of a tree-like design as in a B^+-tree. Suppose further that each leaf is sorted by fixed-length keys but varying-length values. How might you lay data out in the leaves and adjust your algorithm to do as few sorts as possible?
   Answer Sketch: You might separate the keys and values within a page so that the fixed-length keys are in one array and the values are in the rest of the page. If you leave gaps in the key array, then inserts have a good chance of occupying those gaps. Deletes enlarge the gaps. Updates don't affect the key array at all.

5. Suppose that a workload has a short burst of inserts followed by a read-only period of point queries and that this pattern repeats. What kind of data structure might you use?
   Answer Sketch: You might use modification batching for the inserts on top of a hash structure.

6. What kind of data structure might you want to use for a sequential key (one that is monotonic with the time of insertion) in which there is a mix of inserts and time-based range queries?
   Answer Sketch: A B^+-tree sorted by time in which nodes split only when they are full. Alternatively, an append-only logging data organization with Zonemaps would be able to efficiently answer these range queries.

7. Which workloads would be well supported by LSM-trees that are partitioned based on key range at each level of tree? In such a setting, would Zonemaps or Bloom filters be more useful and why?

Answer Sketch: *An insert-heavy workload with frequent range queries. Zonemaps would be useful because the key range of each page would be small, so Zonemaps would filter well.*

8. Which workloads would be well supported by LSM-trees that are partitioned based on a hash function? In such a setting, would Zonemaps be useful and why?
Answer Sketch: *Insert-heavy workloads with point queries. Zonemaps would not be useful, because the range indicated by a Zonemap for a given page would be large.*

9. We have considered so far point queries, range queries, inserts, deletes, and updates. Some other query types include prefix queries (e.g., all names that start with 'Sm') and extremal queries (e.g., minimum and maximum). Which local and global organizations would work well for such queries?
Answer Sketch: *Prefix queries and extremal queries benefit from a sorted data structure organization. So, they benefit from any structure that works well for range queries. Hashing (except for order-preserving hashing) doesn't help at all for prefix or extremal queries.*

10. Suppose we extend the set of operations to include occasional time-travel queries. To support such queries, assume that we associate each key-value pair (k, v) with the time when it was inserted in the structure. A time-travel query might then ask what the value associated with key k was at a certain time in the past. The vast majority of queries, however, will be interested in the most recent value associated with a given key (i.e., simple point queries). Assume that the workload consists only of inserts, point queries, and time-travel queries. What would be a good data structure for such a workload?
Answer Sketch: *We can support this workload by employing a hashing global data organization which will always point us to the collection of (k, v) pairs that have the same k and different insertion timestamps. Note that if we physically maintain the pairs ordered in descending order by timestamp, the hashing will always*

point us to the most recent pair for key k, so normal point queries will find their answer fast. Time-travel queries will have to traverse (either sequentially, or via a sorted search on the timestamp) the list of pairs with the same key until they find the pair that was valid at the desired time.

Another approach is to use partitioned logging globally and hashing for local data organization. Time-travel queries will first identify the appropriate global partition based on time and then probe the hash structure.

11. Now suppose the workload also includes "deep deletes" where `deepdelete(`k`)` would physically remove all the pairs (k, v_i) (over all time) associated with a given key k in a pre-determined time threshold. How would you design a data structure to support this operation?
Answer Sketch: In order to perform deep deletes quickly, ideally there would be one copy of each key, accessible by hash or a B^+-tree. On the other hand, if the ingestion rate is high and that leads to a design in which modifications are buffered, then those buffers should be mutated to remove invalid entries on a regular basis to satisfy the timeliness requirements of deep deletes.

4.14 Further Readings

Surveys: Graefe (2011) surveyed different techniques used in B$^+$-tree designs. Lehman and Carey, 1986 surveyed index structures for main memory. Vitter (2001) surveyed algorithms and data structures for external memory (disk). Luo and Carey (2020) surveyed the variations of LSM-trees, and Sarkar and Athanassoulis (2022) dissected the LSM design space and their optimizations. Huang *et al.* (2022) surveyed indexes for persistent memories. Zhang *et al.* (2015) surveyed in-memory data management and indexing. These surveys and tutorials will help the reader understand individual data structure designs and properties.

5

Adaptivity: Evolving Data Structures to a Workload

The core philosophy underlying adaptivity is that a data organization should automatically evolve towards an organization that achieves the best performance as the workload changes. By contrast, non-adaptive designs fix a data organization and maintain it, regardless of the workload.

For example, a non-adaptive design of a B^+-tree might require a total order of the data in each node. This is a good design if the vast majority of queries are reads including short range queries. By contrast, an adaptive design may evolve to a partially sorted organization on part of the tree to absorb workloads in which there are a substantial fraction of modifications. A partially sorted organization is easier to maintain on each such modification. For reads, an adaptive design that uses rough partitioning of the data will be nearly as efficient as a B^+-tree with sorted nodes.

Overall, adaptivity leads to designs that evolve with the workload. In this chapter, we will discuss past and current work that creates data structures that are autonomously adaptive and reactive to every key-value (read or write) operation.

The benefits of adaptivity include:

- Data structures require time to be built. For interactive workloads on large data files that span the memory hierarchy, users may need to query the data immediately upon load and before there is enough time to build an index. An adaptive approach builds the data structure incrementally as queries arrive. Thus, one core performance benefit is that adaptivity allows data to be queried immediately instead of waiting several minutes or hours for traditional indexes to be built.

- Data structures, by definition, impose a particular organization on the data that has to be maintained when the data changes in any way, which may reduce the performance of workloads having frequent updates. Adaptivity can help by creating data structures that impose a rough (just enough) order on the data to reduce the overhead of updates while still providing adequate support for reads. In addition, organizations that may be slightly sub-optimal for pure read workloads can substantially reduce the overhead of concurrency control when there are more inserts than deletes (Section 7.1). Thus, another core performance benefit is that adaptivity can dynamically create bespoke data organizations that fit the current workload with respect to the read/write balance.

- The workload may vary on different parts of a data structure. For example, if the key is temporal (meaning the key is monotonic with insertion time), then all insertions will touch only the rightmost leaf of a B^+-tree. This implies that different parts of a file may benefit from different index structures (heterogeneity as discussed in Section 4.10). Further, the workload on different parts of a file may change over time. Adaptivity algorithms can monitor access patterns on different parts of the file and incrementally build appropriate indexes and discard unnecessary indexes as the local workload demands. The decision of which index to build may depend both on the access pattern for a given part of the data and the storage platform for the index, e.g., a small fanout for fast memory (node sizes of a cache line) and a large fanout for solid

state disks (node sizes of several kilobytes). Thus, another core performance benefit is that adaptivity creates variable bespoke data organizations that fit different workloads on different parts of the structure. This may bring a very large speed-up (e.g., three orders of magnitude in NoSQL systems as shown in Chatterjee *et al.*, 2022).

Adaptivity has gained in importance because (i) increasingly, applications create exploratory workloads (e.g., in data science and machine learning), (ii) larger data sizes make it harder to index all data, (iii) applications have become more fluid in the sense that they might frequently encounter different workloads, e.g., when introducing new features, and (iv) one memory technology may work better with one data layout while another memory technology may work better with another one (e.g., the fanout example above), especially in a context where hardware evolves rapidly.

The reasons above suggest that fixed data layout choices can limit the performance of an application, while an adaptive approach can improve performance by adjusting a data structure design as data, queries, or hardware change.

Preliminary Observation: The Need for Deep Design Changes. Trying to achieve adaptivity outside the data structure leads to impractical solutions. For example, if an adaptive method is designed and implemented over a non-adaptive data structure, it would need to observe the workload for some number of queries (or time) and then perform a rewrite of all or part of the data structure. This will result in infrequent and expensive transformations that do not satisfy the agile needs of diverse, dynamically changing workloads. As such, the design principles we will see below entail deeply embedding adaptivity in the core design and implementation of a data structure with the goal of changing the data structure efficiently and rapidly as the workload changes.

5.1 Design Dimension: Reorganization Aggressiveness

Physical reorganization of the core data and index components of a data structure are the main mechanisms of adaptivity. Performing aggressive reorganization leading to significant changes in a data structure (e.g., completely sorting a partition that is so far unordered) may allow a data structure to adapt more quickly to a new workload. On the other hand, such an approach might require more work on the part of the reorganization, which might slow down queries while the adaptation happens. Aggressive adaptivity may also push data structures into a state that may be hard to maintain (e.g., if many inserts arrive soon after a node has been sorted).

Reorganization Triggers and Scheduling. At its core, aggressiveness can be measured based on two quantities:

1. *how much* reorganization is applied in every adaptivity step, and
2. *how often* reorganization is activated during a query workload.

Understanding what *how much* and *how often* mean depends on the nature of the data structure and the workload. We discuss these with examples below.

5.2 Adaptivity for Frequently Accessed Data

Data structures can keep the most frequently accessed data in fast memory and require minimal read amplification. The rationale is that data that has been frequently accessed will likely be accessed soon again in the future. If we keep such data in a location that makes it immediately accessible, that will likely improve future queries.

The Splay Tree (Sleator and Tarjan, 1985) was the first such design. It is effectively a binary tree that moves recently accessed data to the root of the tree. Doing so, however, comes with some cost: there are tradeoffs across all PyRUMID metrics. For example, moving recently accessed data to the top of the tree means that accessing other data may become significantly slower as that data has been pushed away from the root.

A similar adaptivity concept can apply to LSM-trees. LSM-trees, by default, maintain the key-value pairs reflecting the most frequent modifications near the root. This benefits queries on such keys but pushes other keys farther from the root. Splay LSM-trees (Lively *et al.*, 2018) move frequently read keys towards the root. This can bring significant benefits to skewed workloads where a small set of keys are accessed repeatedly. While in other data structures the same effect can be achieved with a cache that sits outside the data structure at the system level, a Splay Tree or a Splay LSM-tree adapts to such workload patterns directly without space amplification, by reorganizing existing data along with new data, using a single data structure design.

Similarly, data structures that allow navigational flexibility can adapt to workload hot spots. For example, a data structure may allow multiple navigation paths to speed access to popular data items at some cost in maintenance. This is illustrated in Skip Lists (Pugh, 1990), which do not have a fixed navigation scheme, but rather allow access to the same data via multiple paths. For example, Skip Lists can offer short paths to frequently accessed data items, including those having large keys.

Adaptivity side-effects. As we have discussed several times in this book, there is no free lunch with respect to data structure design decisions. Every design decision can bring certain benefits but can also engender performance costs depending on the workload. The case of adaptivity in LSM-trees as described above illustrates such a cost-benefit tradeoff. While splayed LSM-trees can boost skewed workloads by keeping recently accessed data high in the tree, they also entail forcing data to go through a merge process multiple times, increasing both computation and I/O costs.

Whether this design decision is a "good" one or not depends on the application properties and requirements. Adaptivity simply enables a workload-dependent extension to the design space.

5.3 Adaptivity for Value-Organized Data

When data needs to be organized by value, adaptively moving data around as described above may violate that order. We cannot just move

the qualifying data to a new physical location without affecting the rest of the data organization. For example, if data is currently sorted, then arbitrary data moves would often destroy the sort order.

Further, there is no good way to treat adaptivity as an additional step that happens only after resolving a query. For example, suppose we start with an unordered array and the goal is to end with a sorted array. A strategy might be to scan all data to answer, say, a point query. Then, in a second pass, re-scan the data to impose a new order. This implies the requirement of going through (nearly) all the data twice, a high overhead.

For the above reasons, adaptive approaches change the organization of the data as part of query processing with the goal of gradually evolving towards the desired key-ordering. We describe below state-of-the-art approaches which we order with respect to how aggressively they organize data.

Aggressiveness of Adaptivity during Querying. A lightweight approach to adaptivity is to reorganize key-value pairs based on each query. For example, assume a node is currently an array of unordered integer keys and the goal is to end up with a sorted array. If a query asks for all keys less than 15, an adaptive approach can reorganize all data such that all keys smaller than 15 are at the first part of the array. If another query then asks for all keys less than 90, the adaptive approach will split the larger partition in two partitions as shown in the middle part of Figure 5.1. Finally, assume that a third query asks for keys between 10 and 30. This query can exploit the existing partitioning (and skip the last partition altogether), and it can also split the two overlapping partitions at the query range ends as shown the last part of Figure 5.1. Every subsequent query can utilize this incremental but useful range-partitioned information to focus on only the relevant partition(s).

The above description is how Database Cracking works (Idreos *et al.*, 2007a). Further, every future query will improve the organization of the data by refining the partitioning. For example, if a later range query seeks all key-value pairs for keys between 20-40, then Database Cracking will choose to partition the first part of the array into three subparts.

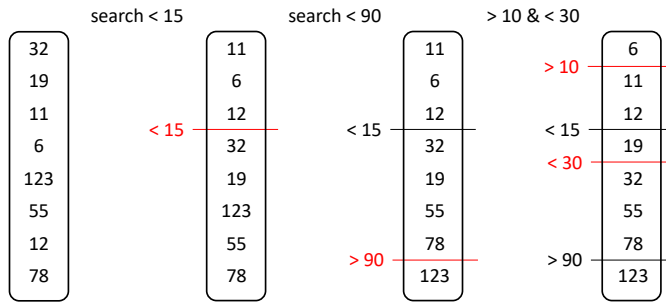

Figure 5.1: A basic adaptivity approach uses the executed queries as hints on how to physically partition the data incrementally. Gradually the data organization is approaching a sorted array.

For efficiency, everything is done in a single pass, meaning once the reorganization is done, the qualifying data is all in a continuous area of the array. Short range queries will touch small portions the array while the pieces become increasingly smaller as more queries arrive due to the additional partitioning on the "hot" area of the data.

In this way, performance steadily improves as more queries arrive in a way that is adaptive to the incoming queries. For example, using this approach in its extended version for multi-column cracking (Idreos *et al.*, 2009) adaptivity can bring a significant benefit by matching the results of a perfect index created manually for all TPC-H queries for Scale Factor 1 using MonetDB. However, in order to create the perfect index for each query, we need to have full and accurate workload knowledge that this type of query is an important one for the workload, and we also need to invest time in constructing the index. In the experiments by Idreos *et al.* (2009) it takes up to 30 minutes to create all indexes needed for TPC-H while with adaptivity enabled MonetDB could finish processing all queries in a matter of milliseconds and quickly converged to the optimal response times automatically without human database administrator intervention.

Increasing Aggressiveness for Robustness. The basic approach described above, where adaptivity uses only the query range to guide reorganization, may result in a data structure that behaves poorly as it depends on the details of the past workload. This can happen

when cracked queries do not change the data structure in a useful way. For example, if we have a billion data items and a query results in creating two partitions, one with two items and one with all others, subsequent queries will still likely have to go through the massive partition.

To improve the robustness of an adaptive data structure for such workloads, Stochastic Cracking (Halim *et al.*, 2012) performs random reorganization operations as it processes the data to apply the reorganization based on the range requested by the query. For example, a random reorganization operation picks a random pivot to partition a subset of the data instead of using only the value range requested by the query. These additional random operations "balance" the partitioning of the array allowing future queries to access increasingly smaller data parts, achieving good adaptive properties. Progressive and Predictive Indexing allows for further control on how much investment in reorganization to apply along with every query (Holanda *et al.*, 2019; Teixeira *et al.*, 2018).

Exploiting Hardware Properties to Increase Aggressiveness. Modern hardware offers the ability to run numerous operations in parallel with multi-core CPUs, some of which may be idle. Holistic Indexing (Petraki *et al.*, 2015) monitors CPU cycles and increases the number of random reorganization operations when there are CPU cycles to spare. Effectively, the system is executing a series of "fake queries with random select operators" to trigger reorganization operations which improve the index for the sake of real workload queries to come.

5.4 Aggressiveness of Adaptivity during Initialization

The first few reorganization operations after initialization (i.e., when data is first loaded) are critical for two reasons. First, they are likely to be the most expensive reorganizations since they are partitioning larger segments of data. Second, from the point of view of user experience, these operations may lead to query response times that are slow not only compared to an optimal design but even compared to a full scan of the data. Thus, it is crucial for the first few queries and reorganization operations to be as efficient as possible and ideally not worse than a full scan.

Parallelize Reorganization to Match Fast Scans. The first technique to accelerate the first reorganizations is to parallelize them. This entails making maximum use of modern multi-core hardware and SIMD capabilities (Petraki *et al.*, 2015). This strategy takes advantage of data and processing parallelism in every step of the reorganization algorithm with the result that the first reorganization incurs the same cost as a highly optimized multi-core scan. The foundational insight is that effectively a reorganization operation is very similar to a scan operation as it needs to go through all data linearly and do a series of comparisons per value. Thus, the techniques for advanced parallel scans can be modified slightly to apply to reorganization operations.

Balance Reorganization Investment/Benefits. The second technique is to make the most of the time investment in reorganizing data. Graefe and Kuno (2010a) first observed that early adaptive indexing strategies read numerous data pages from the disk, then apply a single reorganization operation, and then wrote all pages back to the disk to store the new organization. Because computation is much cheaper than reading and writing data, the original adaptive indexing techniques incurred a massive data movement cost. Graefe and Kuno proposed a new technique where every page that is brought from disk to memory is fully sorted in memory before being written back to disk, making maximum use of the data movement investment.

Subsequent work has observed that there is a spectrum of reorganization choices the first few queries can do. Different amounts of reorganization yield different system performance properties. The spectrum of aggressiveness ranges from a full sorting of individual pages at the most aggressive to a much less aggressive series of incremental partitioning steps (Idreos *et al.*, 2011; Schuhknecht *et al.*, 2018).

5.5 Partial Adaptive Indexing

All adaptive designs described so far apply partitioning over all relevant data. For example, if a query over a data set of a key A asks for $10 < A < 30$, adaptive indexing techniques will go ahead and apply reorganization operations across all of A. However, what if the future workload never asks for data $A > 30$ or for data $10 > A$? The net result would be a lot of wasted work and memory resources.

Partial adaptive indexing, such as partial database cracking (Idreos *et al.*, 2009), avoids these problems by creating an adaptive index only on the portion of the data that is relevant to the workload. That is, not only does the organization of the data structure evolve over time as queries arrive, but also the contents of the data structure evolve by discarding data from the index (but keeping it in the base data) when that data becomes irrelevant to queries. The result is to reduce both the work and the storage overhead of the adaptively maintained index. In fact, with partial adaptive indexing, it is possible to fix a storage budget. Whenever the footprint of the index exceeds the budget, certain contents of the index can be discarded to make space for new content based on a policy that considers access frequency.

5.6 Adaptive Modifications

Until now we have discussed adaptive indexing techniques for read queries: as read queries arrive, we apply physical reorganization operations, ideally focusing the reorganization operations on the data relevant to the query only. A complementary strategy can apply to modifications. That is, an adaptive data structure should do as little work as possible to process a modification until a read query asks for this data. Specifically, when a write (insert, delete, or update) arrives, it can be logged "outside" the core data structure as a pending write. This concept is similar to local buffering of modifications (discussed in Section 3.7.3) but with one important difference. In non-adaptive data structures, pending modifications are merged when certain "structural" conditions are met, e.g., when K pending modifications have been batched. By contrast, in adaptive indexing, modifications are merged only when and if relevant read queries arrive. That is, modifications could be left as pending forever if no relevant query arrives (Idreos *et al.*, 2007b).

In addition, adaptive indexing does not need to ensure all data is fully structured. This gives a lot of room for flexibility in how merges are designed. In particular, adaptive indexing techniques for modifications will move data from the main data structure to the pending modifications area to make space for "hot" modifications that have been queried recently. This allows adaptive indexing to absorb/merge new modifica-

tions using local physical reorganization operations without affecting the rest of the data structure. For example, if two specific pending inserts need to be placed in a particular partition of a contiguous array, a pending technique can move two "non-hot" values to the pending modifications area to make space for the hot ones. When data is stored as a contiguous array, this greatly reduces the reorganization overhead while still preserving the benefits of keeping the data contiguous for read queries.

A special case of this idea is to require that the data items that adaptivity moves from the base structure to the pending modifications area have greater keys than the keys that are merged in. This ensures that the pending modifications area will tend to become smaller because, eventually, all data items will be merged into the main structure (Idreos *et al.*, 2007b). This property follows from the fact that merging two ordered sequences works from smaller to bigger keys. An open topic is to allow for bi-directional merges of pending modifications, which can result in less data movement and can be done in parallel. Bi-directional means that data could be merged both from smaller to bigger keys and from bigger to smaller keys depending on which direction results in less data movement for each merge operation.

5.7 Adaptivity and Concurrency

Because adaptivity entails reorganization, it gives rise to concurrency control issues. A modification that merely appends data requires only local concurrency control. However, read operations may change several data parts as a side-effect of data reorganization. This would require concurrency control protocols to exclusively lock large portions of the data. Fortunately, this is not the case.

For example, concurrency control has been studied in the context of adaptivity in column-stores (Graefe *et al.*, 2012) with the main idea being to distinguish between data changes and structure changes while still ensuring concurrent correctness. Data changes require full concurrency control as normally done in a database system. However, reorganization operations following a read query do not change the content of a data structure. They change only the organization. For that reason, such changes can be protected with short-term latches

instead of full database locks. These observations allow for techniques that allow high concurrency during adaptation.

5.8 Adaptivity Metrics

Measuring the cost and benefits of an adaptive data structure requires observing PyRUMID metrics as they evolve over time. The reason is that the performance of adaptive data structures evolves as the workload evolves. For this reason, measuring the performance of any PyRUMID metric becomes a function of "when" the measurement takes place in the life-cycle of a workload. That is, it depends on which and how many operations have appeared before the current operation. By contrast, regardless of timing, non-adaptive data structures have the same PyRUMID costs. Benchmarks created for adaptive data structures use such evolution patterns of the PyRUMID metrics to compare adaptive designs (Graefe *et al.*, 2010; Halim *et al.*, 2012).

Furthermore, adaptive data structures require the following additional metrics which summarize properties with respect to performance behavior during workload evolution (Graefe *et al.*, 2010):

1. *Data to query time:* This metric defines how much time is needed from when the data is loaded until the first query can return results. For example, if no data structure is created and the processing method is to simply scan all data, then the data to query time is close to zero as we can access the data immediately. If any structure needs to be put in place, though, then the data to query time includes this construction cost. Ideally, a query is processed as soon as it arrives (whether the data structure exists or not at that point) and returns results as quickly as possible.

2. *Convergence to optimal:* The second time-based metric is how long it takes to converge to the optimal organization for the sake of subsequent query performance. Optimal query performance is defined as the performance of the fully adapted data structure to the whole workload. For example, if the ideal data structure for a given workload is for all the data to be sorted, then converge-to-optimal measures the number of queries that have to be processed

until all the data is fully sorted. Since each query in an adaptive data structure performs small, incremental changes to the data, it takes potentially many queries to reach the optimal state. At the same time, precisely because an adaptive data structure does not perform expensive reconstruction steps, queries are processed faster end-to-end. Ideally, the total cost over all queries performing adaptive reorganization of converging to the optimal is not larger than the cost of constructing a fully organized data structure and then processing all queries.

3. *Number of queries slower than scan:* The third time-based metric is a measure of how long it takes until queries return results with a response time that is better than a full scan. The rationale of this metric is that a full scan would be the default option if no index were present. By contrast, queries that support adaptivity incur some cost to reorganize data to benefit future queries. Ideally, the number of queries slower than a scan is zero. In practice, this is not possible if an application needs immediate access to the data. In these cases, a good adaptive data structure will minimize the number of queries slower than a scan as well as the amount of extra time they take.

5.9 Open Topics

There are numerous open research opportunities in the design of adaptive data structures.

First, the design principles of adaptivity have not been applied broadly to all classes of data structures, leaving an opportunity to improve those structures. For example, in LSM-trees, adaptivity can be applied to one level of the tree at a time or even across levels, affecting how and when data moves across levels.

Second, regarding the larger goal of tailoring adaptivity to a given workload, one can ask how much organization is the right amount. For example, if the modification load is high, a constrained organization (such as fully sorted) entails significant overhead per modification to ensure that the organization conforms to that constraint. With such

a workload, it may never be good to fully sort all data because of the overhead on future modifications to maintain that organization. By contrast, it may be useful to have rough partitions. An exciting goal of future work would be to automatically decide the most appropriate level of organization an adaptive design should aim for on the data such that the overall balance of read and write cost is within acceptable user bounds for a projected workload. Existing designs aggressively follow the workload and do create custom designs, e.g., with rough partitions instead of sorting, but do not do so by taking into account the projected workload. Instead, they continuously adapt. This means that they can get trapped into suboptimal intermediate states or they can cycle between designs but without necessarily reaping the benefits of each intermediate design as the next reorganization might occur too soon. Thus an overall promising direction for future approaches is to more holistically take into account workload samples and to invest in predicting the workload (e.g., with machine learning enhanced approaches).

5.10 Chapter Summary

This chapter has described strategies that have been invented to help access methods adapt to workloads as they change. The fundamental ideas are (i) to design read operations to perform some data reorganization to make subsequent operations faster and (ii) to buffer modifications in order to defer reorganization costs to when they are needed. Adaptivity is a nearly continuous design dimension in that it can be done aggressively or not. The optimal aggressiveness should be determined largely by the frequency, as well as the spatial and temporal locality of modifications.

Adaptivity has become increasingly relevant due to the more dynamic nature of modern data-intensive applications. Open-source systems such as MonetDB ship with adaptive techniques, which can be enabled through optimizer knobs. In addition, industrial research is increasingly moving in the direction of adaptive data layouts (Ding *et al.*, 2021) and overall adaptive behavior. Cloud systems are a natural environment for adaptivity to flourish due both to the diverse hardware

and the diverse workloads employed. Recent research on cloud systems is moving in this direction with adaptive layouts (Chatterjee *et al.*, 2022; Huang and Ghandeharizadeh, 2021). Overall, we expect adaptivity to be pervasive in future system stacks, creating opportunities for novel designs and concepts.

5.11 Questions

1. Consider a workflow consisting of a repeated pattern of bursts of inserts followed by periods of range queries. Suppose that these patterns differ on different parts of the value domain. What would be a good adaptive approach to use?
   Answer Sketch: Buffering the inserts would keep up with the high insert traffic. Instead of sorting before the range queries appear, cracking the page would be better, thus creating non-overlapping sub-partitions that are unordered within the sub-partitions. Over the course of several queries, the page will likely approach a sorted ordering.

2. Compare your adaptive approach to the previous question with a non-adaptive B^+-tree in which (a) leaf nodes are never sorted, or, by contrast, (b) leaf nodes are always sorted.
   Answer Sketch: If the leaf nodes are never sorted, then inserts can be placed directly in the nodes without buffering, but short range queries might be slow because they would have to read the entire node. If leaf nodes are always sorted, then every insert could require substantial data movement within the page.

3. Suppose the workload consists mostly of point reads and range queries but with, say, 20% modifications. One possible organization is a rough partitioning, another is cracking, and another still is a fully sorted organization along with buffered modifications. Please experiment to determine which is best in terms of total execution latency. In order to explain your observations you can use additional metrics including CPU utilization, and memory or disk bandwidth utilized.
   Answer Sketch: This is a programming assignment.

4. Describe how you might use adaptivity in a workload without range queries, i.e., point queries, interspersed with bursts of inserts, deletes, and updates. Start with a hash structure having many buckets.
   Answer Sketch: If the insert/delete/update traffic is particularly heavy, then each such modification might require an access to slow memory on a conventional hash structure. It might therefore be better to buffer the modifications in memory and then periodically to move them into hash data buckets.

5. Consider a data structure of stock trades that maps a composite key of day and stockid to a time-ordered vector of trades for a given day. The insert workload consists of trades entering for the current day. There are no (or very few) modifications to trades from previous days. Queries ask about one or more trades of a given stock over one or more days including the current day. How would you design an adaptive structure for such a workload?
   Answer Sketch: The historical (not including the current day) global organization is sorted or hashed based on stockid. The organization for the current day could be an LSM-tree with hash partitioning at the non-root levels based on stockid.

6

Data Structures for Specific Application Domains

This book has so far focused on the design of individual data structures. In this chapter, we discuss data structures as components of complex real-world systems. A complex system in general may contain numerous data structures to support the diverse set of functionalities that the system's application requires. In addition, systems often offer multiple data structure options to support the same functionality (e.g., B^+-trees and binary search trees to support search on sorted data). Workload considerations may suggest one data structure rather than another. In such a setting, the design of each individual data structure and how each data structure contributes to end-to-end system performance are both important.

The examples we present in this chapter come from systems that require secondary memory: database systems, file systems, and machine learning pipelines.

6.1 Data Structures in Relational Database Systems

Relational database systems must make several decisions around the data structures they employ, regarding (i) how to represent tabular data, (ii) how to index data to accelerate data access and which index a

given query should use, (iii) how to store intermediate results of queries, and (iv) how to adapt to fluctuating workloads.

6.1.1 Data Structure Design for Base Data Organization

The relational model organizes data into tables (or *relations*). Such tables are conventionally laid out row-by-row (or *record-by-record*), a so-called row-oriented organization. An index might map a key (such as a social security number) to a value (such as an employee record). The rows may be partitioned into different locations, a technique called *horizontal partitioning*, while maintaining this basic key-to-record data structure.

An alternative is to partition the table by column. For example, if the access pattern of a read query requires accessing a few columns of a relational table (e.g., only the employee salary column to calculate the average) and each column is separately stored, then scans need to access only the columns required by that query. A table that is *column-partitioned* is said to follow a *column-store* or *columnar* data organization.

Applications of column-store approaches came to the fore in the 1980s when analytical scenarios led to the creation of wide relational tables (i.e., tables with many columns) in which queries accessed only a few columns (Copeland and Khoshafian, 1985) at a time. This led to the development of columnar data system architectures (Abadi *et al.*, 2013; Boncz *et al.*, 2005; Färber *et al.*, 2011; Färber *et al.*, 2012; French, 1995; French, 1997; Idreos *et al.*, 2012; Kemper and Neumann, 2011; Lamb *et al.*, 2012; MacNicol and French, 2004; Stonebraker *et al.*, 2005; Zukowski and Boncz, 2012) (as had already been present in vector languages like APL (Falkoff and Iverson, 1973) since their invention). Using a columnar instead of a row-oriented layout in data management systems is a data organization decision that has been extensively studied (Barber *et al.*, 2012; Barber *et al.*, 2015; Lahiri *et al.*, 2015; Larson *et al.*, 2011; Larson *et al.*, 2012; Larson *et al.*, 2015; Larson *et al.*, 2013; Ramamurthy *et al.*, 2002; Ramamurthy *et al.*, 2003; Raman *et al.*, 2013).

In addition to pure column-store organizations, one can store groups of columns together. For example, one can store the age and the salary of

an employee together, because queries may want to calculate the average salary per age bracket. Various hybrid approaches may offer benefits depending on the workload: (i) by nesting columnar data organization within data pages (Ailamaki *et al.*, 2002; Ailamaki *et al.*, 2001), and (ii) by grouping multiple columns and offering specialized code for accessing groups of columns (Alagiannis *et al.*, 2014; Dittrich and Jindal, 2011; Färber *et al.*, 2011; Färber *et al.*, 2012; Grund *et al.*, 2010; Kemper and Neumann, 2011).

Overall, the column-store design works better for analytical queries each of which depends on a few columns (say five to ten) of tables having many columns (hundreds or thousands). By contrast, the row-store design works well for queries that require all or most fields of each row. A columnar data organization enjoys an administrative benefit because it makes schema changes easier: columns can be added to or removed from a table without affecting the rest of the data. In a row-store, by contrast, dropping or adding a column would normally entail reading and rewriting the whole table.

Finally, a common design approach is to replicate the data where each replicate has a different data structure design. For example, systems that need to support streaming data along with analytical queries may employ (i) an LSM-tree row-based design to absorb data quickly and (ii) a second copy in a columnar layout. This supports diverse workload patterns at the expense of additional storage costs and some delay in moving updates from the insert-optimized copy to the analytical copy.

6.1.2 Data Structure Design for Access Path Selection

While indexing entails the overhead of metadata, it is beneficial for the majority of selective queries that access a small fraction of the data. The choice of when to use an index is the topic of access path selection (Kester *et al.*, 2017; Selinger *et al.*, 1979), which weighs multiple factors including query selectivity (the fraction of the entries retrieved by a query), whether data is sorted by key, the number of keys per page, and the difference in cost between sequential and random access.

If the underlying data is sorted by the same key as an index, then the index should be used regardless of the selectivity.

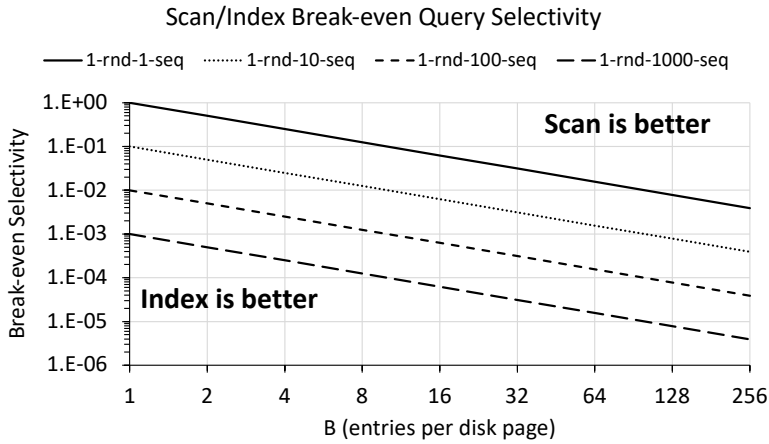

Figure 6.1: When comparing indexing vs. scanning when the base data is *not* organized based on the key, the number of entries per disk page B plays a central role. If random and sequential accesses have the same cost, then the break-even selectivity is $1/B$. For example, if a selection will match $1/1000$ of the records and there are 100 records per page (B), then only about $1/10$th of the pages will be accessed when using an index, so the index is helpful. On the other hand, if there are $10,000$ records per page, then the index will access nearly every page, roughly 10 times each, so scanning would be better. For devices for which random access is slower than sequential access by a factor of R, the break-even selectivity is further reduced, being equal to $\frac{1}{B \cdot R}$. For example, if the system can transfer 10 pages sequentially in the time it takes to read one page randomly (dotted line) and there are 128 entries per page, and at least $1/1000$ (1.E-03 on the vertical axis) of the records match a query, then a scan is at least as fast as an index search. In the figure, different line textures indicate different R values.

When the data is not organized based on the key of the index, there will be one disk access for each qualifying entry (Ramakrishnan and Gehrke, 2002). Figure 6.1 shows the relative benefits of scanning and index probing as a function of (i) query selectivity and (ii) the degree to which sequential search is faster than random search. The figure shows several comparison points for the considered parameters.

For example, if $B = 1$ (every page contains only one entry) and a random access is as efficient as a sequential access, then we should always employ an index (break-even selectivity is 100%). By contrast, the larger the cost difference between random and sequential access (shown by the different lines), the less advantageous is indexing. The reason is

that for a non-clustering index, indexing entails random accesses while scanning is sequential. For example, if scanning 100 pages costs the same as accessing one page randomly and the query will return more than 1/100th of the data, then scanning will be better.

When to Sort the Output of an Index. When using an index to access base data, the query plan often assumes (or specifies) that the index returns the data sorted based on this indexed attribute. For example, a query plan may use an index to select a subset of the rows and then calculate a group by. The group by operator might rely on receiving the rows in sorted order based on the grouping attribute to give the correct answers. If, however, the overall plan for a query does not require the entries to be sorted based on the indexed attribute (e.g., it performs a selection and then calculates the average), repeated accesses to the same page can be avoided by sorting the entries based on their row ID instead. This ensures that each physical data page will be accessed only once, potentially making the use of an index much more attractive compared to a scan.

Adaptive Data Structure Selection. If it is not possible to sort the output of an index based on the rowID then it may be that using an index is actually worse in terms of performance compared to scanning all data as discussed by Borovica-Gajic *et al.* (2015). This effect may seem counter-intuitive at first, however, it is due to data movement. If the system is trying to access data given an unordered set of rowIDs, then it will end up moving the same page multiple times to access this data in the proper order. In that case, the index access can be multiple times slower than a full scan. The smooth scan operator proposed by Borovica-Gajic *et al.* (2015) solves this problem by adaptively switching from an index-based access to a scan-based access as it processes any given query. The core idea is that in each step of an index access, when the system brings in a data page, it will check not just for one rowID, but will instead "scan" the whole page for all relevant rowIDs.

Shared Scans vs Indexes. Finally, advanced scan algorithms may enjoy an advantage over indexes if several queries can be answered in a single scan pass as this can dramatically reduce data movement as shown by Psaroudakis *et al.* (2013), Giannikis *et al.* (2013), Candea *et al.*

(2011), and Zukowski *et al.* (2007). However, advanced data structures can still provide benefits for highly selective queries even in memory. Compared to more traditional approaches, modern optimizers take into account not only selectivity but also query concurrency to make the best possible decision on which access path to select for each query (or set of queries) as discussed by Kester *et al.* (2017).

6.1.3 Data Structure Design for Intermediate Results

When using a single data structure we typically perform a single key-value access at a time. Queries in a relational database system, however, might perform millions of data accesses to one or more tables. The output of one operation in a query plan may be the input to one or more other operations in the plan. When systems process big data, these intermediate results tend to be of substantial size. Thus, reading and writing that data becomes a critical part of the overall cost of executing a query. As a result, in relational data systems, it is not only important to choose data structures to store the base tables but also to decide how intermediate results should be stored and accessed, as these two decisions interact.

For example, the intermediate results in columnar systems must identify which rows of the data have survived at least some of the query's filters (e.g., conditions such as A = 5 and B < 5). Representing the surviving rows can be done in at least two ways.

First, one might use a position list, which is a dense array in which each value represents an offset to the base table data. When many rows qualify, a position list must store and move one value per row which for bigger data can be a significant overhead. A second design is to use a bitvector in which every row of the base table is associated with a "0" or "1" depending on whether it qualifies or not. Such a representation stores one bit per row of the original base table as opposed to the worst-case scenario for position lists which is one integer (e.g., 64 bits) for each row that has survived the filter. Roughly speaking, if fewer than 1/64th of the rows qualify, position lists will perform better than bit vectors (when each position is encoded as a 64-bit integer)[1]. Otherwise,

[1]The calculation goes as follows. Assuming N rows, the bitvector will have 1 bit

bit vectors are better. Both approaches implement a version of *late materialization* (Abadi *et al.*, 2007) that allows the system to minimize the data movement until it is absolutely necessary.

In summary, relational data systems need to make dynamic data structure design decisions that depend on the context of a particular query and the overall state of the system.

6.1.4 Data Structure Design for Adaptivity in Data Systems

Data layouts have a large influence on the performance of queries. For that reason, allowing data layouts to adapt to workloads may offer some benefits.

For example, consider a columnar database system where every column of a relational table is stored as a dense fixed-width array. To accelerate queries on specific attributes, conventional column-store systems create *projections*. Each projection replicates all or a subset of the columns of a table, organized in the sorted order of a single or a few columns of that subset. The sorted column(s) that is (are) said to be *leading*.

This is the standard design of all column-stores and results in substantial storage and update overhead because it may require a column-store projection for every single column (as the leading column). Alternatively, a designer might fix a certain selection of k column-store projections and replicate the data once for each such projection.

Reorganizing just the columns of interest to a given query would cause columns to be out of alignment with one another (i.e., the ith element of one column would not belong to the same logical row as the ith element of another column). In such a setting, maintaining the logical row structure of different columns can be done by holding positional references in every single column. That is, every column is augmented with what is effectively an additional column to hold the position of each value in a common reference position across all table columns (e.g., the first value in this column corresponds to row 13,

---

ber row, hence, N overall bits. The size of the position list is $N \cdot s\% \cdot 64$ bits where $s\%$ is the selectivity of the filter. In order for the position lists to be better we need $N \cdot s\% \cdot 64 < N \Rightarrow s\% < 1/64$.

the second to row 2954, etc). The problem with this approach is that it causes a lot of random access for the common case of queries that express conditions on several attributes.

Idreos *et al.* (2009) address this problem with sideways cracking by gradually propagating the reorganization actions (e.g., a sort) performed on a single column across the other columns of the table as shown in Figure 6.2. Once the propagation is complete, position i of one column will correspond to the same logical row as position i of any other column. This eliminates the need for positional references and random access.

To avoid overburdening an individual cracking query with the task of reorganizing multiple columns, sideways cracking does this propagation adaptively. Assume a relational table consists of 100 attributes $C1, C2, ..., C100$. If a query arrives with a selection on $C1$ and an aggregation $C2$, adaptive indexing will reorganize $C1$ based on the selection (cracking) and propagate this order on $C2$ but it will not touch the other 98 columns. If future queries with a (potentially different) selection on $C1$ need to utilize one or more of the other columns, they will then create a copy of these other columns to propagate the revised order. In order to achieve this incremental organization, sideways cracking maintains the history of reorganization operations for every column in a table, so these operations can be "replayed" in the future as needed. Replaying operations means that exactly the same data reorganization operations are applied in the new columns and in exactly the same order as the columns already reorganized. Overall, this design adaptively builds column-store projections and achieves the performance impact of a well-tuned column-store projection but without the need for a priori workload knowledge about which column should be the leading one and without expending too much time or storage.

Finally, adaptivity has been utilized to organize the base data layout of database systems by dynamically creating data layouts that are hybrids of row-stores and column-stores to bridge the requirements of different queries. Alagiannis *et al.* (2014) and Arulraj *et al.* (2016) use queries as hints on how data should be stored. Then those hints are realized through incremental reorganization actions.

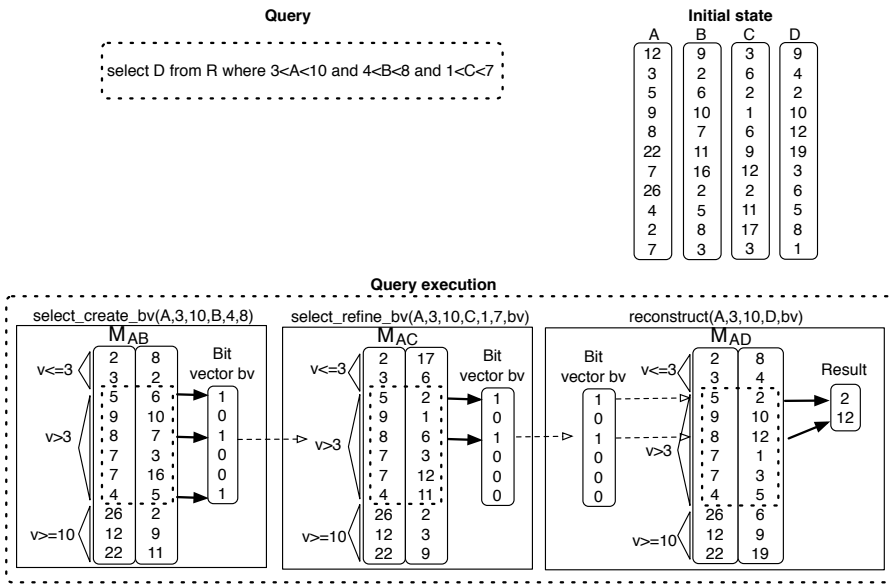

Figure 6.2: Example of sideways cracking: adaptively reorganizing pairs of columns by using incoming queries as advice on how data should be stored to incrementally improve data access speed.

6.2 File Systems & Memory Management

A file system offers an application programming interface (API) that operating systems use to store and access data. File systems take care of data management under the hood to make sure that data can be accessed as fast as possible in a hardware-conscious way as well as to ensure that data is consistent (i.e., accesses to a file appear to execute in a serial fashion consistent with their order of arrival). File systems deal with massive amounts of data and the performance properties they offer impact any application running on top of operating systems.

File system access patterns typically involve dealing with pages of data, e.g., storing or retrieving pages from a file. When reading file pages the combination of an `fseek` and an `fread` command finds the location of interest in the file and then reads a contiguous blob of bytes. Note that the actual file pages may not be contiguous on the storage medium, hence the core access pattern in file systems may be point or range queries, much as in a database system.

File systems have traditionally relied on B^+-tree data structures. As discussed in Chapter 4, B^+-trees provide a good balance between point query, range query, and update requests, thus, several file systems including Ext3, ZFS, and ResierFS use B^+-trees to represent files and directories (Menon *et al.*, 2003; Rodeh, 2008; Sweeney *et al.*, 1996).

Another class of file systems includes log-structured-based systems which were designed over LSM-tree-like data structures. The motivation is that most applications must support frequent insertions and mostly need to access recent data. LSM-tree style designs work well for such applications. In fact, the similarity between journaling, LSM-tree compaction, and garbage collection, has led to the development of approaches that embed the LSM design within the physical device (Dayan *et al.*, 2021; Kang *et al.*, 2019).

Managing memory requires various data structures to efficiently maintain and retrieve resources, e.g., via memory address translations. For example, the page cache and the translation lookaside buffer (TLB) traditionally employ a radix tree to index all the translations from logical to physical memory addresses (Bovet and Cesati, 2005).

New data structures can benefit the internals of operating systems either in terms of performance or space utilization. For example, Margaritov *et al.* (2018) suggested that a learned tree index can replace radix trees and increase the space efficiency of the translation index.

6.3 Data Structures in Machine Learning Pipelines

All modern machine learning pipelines entail storing, moving, and analyzing large amounts of data. As such, data structures, their performance properties, and tradeoffs are crucial to the overall efficiency and even feasibility of machine learning pipelines.

Typically, a machine learning pipeline consists of several sub-systems, each corresponding to a single pipeline step. Sub-systems include systems to handle data injection, cleaning, transformations, labeling, modeling, serving, monitoring, and caching. Each one of those sub-systems needs to deal with large amounts of data and so each one of those subsystems needs to make data structure design decisions.

In many applications, machine learning is applied to structured relational data. The reason is that most businesses and scientific research groups manage their data using relational data systems. In addition, most modern machine learning libraries store data in a column-store. The operations performed in machine learning pipelines are a superset of those performed in relational systems, but the data structure considerations are largely the same: caching and management of intermediate results (Wasay *et al.*, 2017), compressing data to improve data movement (Jain *et al.*, 2018), sharing execution of complex computations over the same data (Wasay *et al.*, 2020), and utilizing parallelism and memory management when data is large (Narayanan *et al.*, 2021).

When the input is not in relational form, state-of-the-art machine learning libraries nevertheless use columnar storage for the underlying data representations as this is often more efficient in terms of performing operations on individual data entities. Complex data types such as images or text are also transformed into arrays/columns for efficient processing. The reason is that Machine Learning algorithms typically take as input arrays or matrixes of floating point numbers with the task of performing a series of mathematical operations to discover possible patterns in these input numbers. Thus, before utilizing a given ML algorithm (e.g., from a specific library), a data scientist first transforms their input data into a standard numerical form as an array or matrix or set of matrices of floating point numbers. The underlying representation to store arrays and matrices in most libraries (e.g., Pandas) is column-oriented as this provides the most flexibility.

In addition to using a columnar layout, many applications require fast data ingestion. Sometimes, high-volume incoming data needs to trigger the execution of machine learning models. In these "streaming" cases, incoming data is pushed onto an LSM-based storage engine. From there the machine learning pipeline will extract the relevant features for the models that need to be triggered. For these reasons, LSM-trees play an important role in the inference response time of machine learning pipelines.

In addition to other tree data structures, decision trees represent a large portion of machine learning models used in practice due to the simplicity of interpreting their results. This happens as follows: every

node of a decision tree represents a decision based on a feature of the model. The fanout of a node represents the possible values for this feature. Performing a prediction based on a tree model means traversing the tree starting from the root and ending at the leaf. The leaf represents the suggested prediction (e.g., the predicted class of an object such as the price range of a requested taxi ride or the risk range of a mortgage request). For complex problems, machine learning pipelines use tens, hundreds, or even thousands of decision trees in a random forest. The final prediction result for a given query is based on averaging the results of all trees in the forest. In such applications, machine learning pipelines traverse numerous decision trees for every single prediction request.

Overall, data structures play a dominant role during numerous phases of machine learning pipelines. They influence performance and they are also at the center of critical open challenges in the field. For example, how do we maintain end-to-end model properties during data and model drift (Yuan *et al.*, 2023), how do we maintain high-performance inference during phases of high data injection rates, and how can we include additional features in a model to improve quality without significantly increasing inference and retraining time? For all those challenges, efficient and application-tailored data structures are critical, especially since the quality of Machine Learning models requires ever larger data.

6.4 Cross-System Design Considerations and Tradeoffs

Even though each system class has its own design elements, there are also certain design considerations and tradeoffs that are common across system classes.

For example, most systems have some form of cache. The cache should ideally be used to store recently used data pages or individual data entries to minimize access to the disk or remote machines. Because the cache consumes memory, this implies that caching gives rise to a performance tradeoff: how much memory should be given to the cache and how much to the various data structures of the system? In fact, this is an open research problem across all forms of systems. As of now, all systems expose the size of the cache as a tuning parameter and then use

what is left for internal data structures. However, given a workload, one could co-design the cache size and the overall design of the numerous data structure of the system to achieve a given performance goal.

Another design consideration that creates tradeoffs across multiple system components is how to utilize multiple processing cores. For example, multiple cores may perform independent processing on different data parts, but that will make those CPU cycles unavailable for other purposes.

Hardware-sensitive design must also consider the high cost of accessing data on remote sites. This may inspire strategies such as adding filtering data structures on local sites (similar to the filters discussed in Section 3.3.2) to avoid accessing a remote site. Another strategy may be to keep an index on a local site to data on a remote site in order to minimize the number of accesses to that site. Such design decisions need to consider the ever-evolving underlying hardware. For example, as RDMA (Remote Data Memory Access) becomes more prevalent, accessing remote data will be less costly. This changes overall system and data structure design considerations. For example, this may suggest distributing large indexes as the cost of data movement decreases.

Overall, there are numerous open design tradeoffs that, combined with application requirements, influence data structure design decisions. An open research challenge is to automate such decisions across the entire tradeoff space.

6.5 Chapter Summary

This chapter describes how data structure design considerations change when designing systems for diverse applications, ranging from relational databases to machine learning. Each such application can be characterized by a family of workloads. The workloads and hardware considerations suggest design choices for data structures. Because any design entails tradeoffs, system designers need to consider the end-to-end impact of each data structure design decision on overall system performance.

6.6 Questions

1. Consider a network application in which each router must forward messages towards their destinations at high speed and collect monitoring information such as the number of messages passing through that router, the sizes of those messages, and the lengths of delays. Each of the routing and monitoring functionalities might need one or more data structures. Try to identify them. Which ones should be kept in memory?
   Answer Sketch: The routing tables map a destination node to the next node (or nodes) to send to. These would be well served by an in-memory hash structure. The monitoring information such as the aggregate size of messages passing through the router does not need any special data structure. If one wanted to keep track of sender-destination pairs and messages between them, then a hash structure whose keys are sender id-destination id pairs and whose values are message information would be appropriate. This could be an on-disk structure.

2. Consider a forecasting application for stock market data that must collect streaming time-ordered trade data of the form (stockid, quantity, price) and perform per-stock price forecasting over the next few seconds. The per-stock forecasting could use any standard univariate (history per stock id) forecasting method. What would be an appropriate data structure for this application?
   Answer Sketch: Because the forecasting is per stock id, a hash structure per stock id with values stored in time-sorted order would work well.

3. In many machine learning applications, there is a preliminary step known as feature selection in which only a (possibly small) subset of possible features are considered in a machine learning task. How might feature selection influence data organization?
   Answer Sketch: A feature is an attribute of some entity, corresponding to a column in a relational model. When only a few will be needed, a columnar store will enable access to exactly the data from the features that are needed.

6.7 Further Readings

System Surveys: Every system category comes with a lot of design complexity rooted to the particular class of design or application. Surveys that go into greater detail on particular systems classes can be very useful to further understand design implications of data structures in these systems. Examples of further readings include surveys and tutorials on: relational database systems (Hellerstein *et al.*, 2007), columnar database systems (Abadi *et al.*, 2013), in-memory data systems (Idreos *et al.*, 2012; Faerber *et al.*, 2017), hardware conscious analytical system (Barber *et al.*, 2012), key-value store systems (Idreos and Callaghan, 2020), LSM-based systems (Luo and Carey, 2020; Sarkar and Athanassoulis, 2022).

Data Systems Grammar: As we discussed in this section, systems consist of numerous data structures. An approach to overall system design is to create a grammar that supports the co-design of all data structures with respect to the desired end-to-end system performance behavior. Such a grammar has been devised for the Cosine Systems Calculator (Chatterjee *et al.*, 2022) for key-value storage engines. It consists of design principles and rules for design which allow for 10^{36} unique system designs. The overall design space is much larger as it consists of numerous data structures, but it has been reduced dramatically thanks to the construction of the design continuum (Idreos *et al.*, 2019a), which allows fast search over the best possible designs.

7

Challenging Design Considerations

A recurring theme of this book is that there are one or more points in the space defined by the design dimensions that can satisfy a given workload. We have also discussed methods that allow heterogeneous treatment of different parts of a data structure (e.g., if the key is sequential, do not split the rightmost leaf of a B^+-tree in half; instead, let that node be full and simply introduce a new empty node for further inserts). We have also described adaption strategies as the workload changes. Though the design space so far shown is rich, there are still other dimensions that research to date has explored in isolation, but not as part of an integrated design method. For this reason, we believe these dimensions require further research.

- How to design concurrency control mechanisms for each option of the design dimensions? How does concurrency control interact with those dimensions?
- How to support data structures that span a set of processing nodes across a distributed system?
- How to make a data structure adapt to new workloads?

- How should data structure design exploit hardware features?
- What is the interplay between caching and data structure design?

7.1 Concurrency

Preliminaries. In this brief discussion about concurrency, we use the notion of locks that protect the access of an object (e.g., an index node, or a file page) when this object is altered. We assume two types of locks: a read lock that allows for multiple readers but no writers, and a write lock that allows for one writer and no readers.

To understand how concurrency might interact with the other dimensions of data structure design, consider the problem of maintaining a sorted order among the keys of a node.

An incorrect approach would allow the sort to execute while searches and perhaps even modifications execute concurrently. This can lead to modifications being missed and to searches incorrectly returning that keys are not found.

A naive correct solution might be to exclusively lock the node during each insert or delete in order to establish a sorted order. However, if that node is accessed by many concurrent threads, those other threads must wait until the sort completes.

A better correct solution might be to simply put the inserts or deletes into a buffer associated with the node, e.g., see the work of Levandoski *et al.* (2013), discussed in Section 4.3. Then, each subsequent operation would look first at the buffer and then at the node whose keys are sorted. Periodically, preferably when there are few accesses to the node, the node would be locked and re-sorted using the modifications from the buffer. Thus, we see that buffering (Section 3.7) increases possible concurrency because write locking the whole node occurs only during the relatively rare full reorganizations.

Another interaction of concurrency control and design has to do with concurrency control in hierarchical structures. Holding a lock, especially a write lock, on an interior node m of a B^+-tree interferes with other operations far more than holding a lock on a leaf n, because any operation that needs to access a leaf below m would be blocked. For

that reason, it is important that locks on interior nodes be held for a short time, during which time locked nodes should be in memory (Kornacker *et al.*, 1997; Lehman and Yao, 1981; Lomet and Salzberg, 1992; Sagiv, 1986; Shasha and Goodman, 1988). If that is not possible, one might consider data structure designs that have no internal nodes such as order-preserving hashing. Thus, concurrency control considerations can suggest new data structure designs.

Concurrency considerations might even suggest changing existing algorithms to reduce reorganization overhead, and therefore concurrency bottlenecks. For example, and as we have observed in Chapters 3 and 4, in virtually any setting where there are more inserts than deletes, merging B^+-tree nodes when they become less than half full (merge-at-half) hurts performance, because merges will soon be followed by splits. Such a merge-then-soon-split pattern entails substantial reorganization overhead and causes concurrency control-induced delays. Instead, it is better to allow a node to become empty before freeing it. When there are more inserts than deletes, free-at-empty achieves page utilization nearly as high as merge-at-half (Johnson and Shasha, 1989) while increasing throughput.

The possibility of concurrency bottlenecks also suggests using highly concurrent data structure algorithms, some of which have been automatically verified (Krishna *et al.*, 2021). Some authors have suggested using lock-free algorithms often inspired by the text *The art of multiprocessor programming* (Herlihy *et al.*, 2020). Lock-free algorithms allow searches to proceed while nodes are reorganized, thus greatly reducing blocking overhead, but such algorithms often do speculative work followed by a compare-and-swap operation. When the compare-and-swap fails, the speculative work has been wasted. Nevertheless, a combination of minimal locking approaches and lock-freedom has shown great promise (Levandoski *et al.*, 2013).

Another promising avenue is to avoid the need for locking by designing approaches that take advantage of naturally serialized machine instructions as done, for example, in Masstree by Mao *et al.* (2012).

As multi-threaded systems are now commonplace, workloads are getting increasingly complex and hybrid, systems support more users, and new data systems architectures are proposed, much more research is

needed to understand the interactions of concurrency with other aspects of data structure design.

7.2 Distributed Systems

In a distributed system, a data structure may be partitioned and even partly replicated over many processing sites. Since accessing a local site is much less expensive than accessing a remote one, the replication/partitioning choice can have a big influence on performance. This was recognized in some of the earliest distributed data structures which targeted both point queries with hash-based approaches (Litwin, 1980; Litwin *et al.*, 1996) and range queries (Litwin *et al.*, 1994).

To understand these issues, consider a distributed and partially replicated B^+-tree. The root will rarely be modified. For that reason, it might be worthwhile to replicate the root and perhaps other high-level nodes across the processing sites. This might be particularly attractive in a shared-nothing architecture. In that way, a search or modification can always start at the local site before being forced to go to a remote site. Ideally, only the leaf access may entail an access to a remote site.

Replicating information comes at the cost of having to modify all the replicas when any one is modified. This in turn affects concurrent operations.

A second issue for a distributed data structure is how to balance the load when one site's portion of the data structure receives a high workload or when a new processing site is introduced (Karger *et al.*, 1997). A further question is how often to rebalance the load with all the concurrent overhead that implies (Luo and Carey, 2022).

7.3 Emerging Workload Types

In Section 2.4 we introduce seven workload operations[1] that data structures frequently have to serve. As new applications and new (legal) requirements emerge a set of new workload operations also emerges (Athanassoulis *et al.*, 2016; Kennedy and Ziarek, 2015).

[1] Point queries, short range queries, long range queries, full scans, inserts, deletes, and updates.

We have seen recently the emergence of "Deep Deletes" that require a deletion to provide guarantees that the data has been completely erased to protect an individual's privacy (Sarkar *et al.*, 2020; Athanassoulis *et al.*, 2022). Similarly, we can consider a "Deep Update" which can be thought of, functionally, as a deep delete followed by an insert. In addition, both deep delete and update may be associated with a deadline by which time the purging of the older data has to be completed. In addition to deep deletes, many systems also employ a deletion of older data or an "Aged Delete" mainly for storage and cost reasons. Last but not least, many applications require "Time-travel queries" for both application and compliance reasons. This rich set of new workload operations is largely unexplored and requires more research.

7.4 Hardware Considerations in Data Structure Implementation

Implementation-related optimizations require intimate knowledge of the computing hardware and the facilities of the underlying processor(s).

Even after the global and local design of a data structure has been established, the overall efficiency of a data structure depends on the quality of its implementation. Good implementation can take advantage of many hardware features. Here are some of the main examples:

- Exploit SIMD instructions to group multiple accesses on consecutive memory locations (Polychroniou *et al.*, 2015; Raman *et al.*, 2008; Willhalm *et al.*, 2009).
- Employ cache-aware designs that access portions of memory equal to one (or a few) cache-lines at a time. There is no benefit in accessing fewer bytes than the entire cache-line because the unit of transfer from the main memory to the cache is a full cache-line (Ailamaki *et al.*, 2002; Boncz *et al.*, 2005; Rao and Ross, 2000).
- Prefetch memory chunks when the chunks that will be accessed are known. A prefetch request can be concurrent with the execution of some other logic (Chen *et al.*, 2001; Chen *et al.*, 2002; Smith, 1978).

- When we have consecutive condition checks (for example, in a sequential scan that selects a subset of the data), we can re-write the code in a way that avoids if statements by employing predication, which reduces the cost of branch mispredictions from the processor (Ross, 2021).

- Exploit NUMA (non-uniform memory accesses) which is now commonplace in production servers by careful placement of data in computing units (sockets) forming performance *islands* (Ailamaki *et al.*, 2014; Ailamaki *et al.*, 2017; Porobic *et al.*, 2014).

7.5 Chapter Summary

The eight design dimensions of Chapter 3 constitute the major choices a data structure designer must make given an application (workload). This chapter discusses considerations on the frontier of current research that are important when designing practical data structures: concurrency, scaling out to distributed settings, new workloads, and the evolution of hardware.

7.6 Questions

1. When there are frequent inserts on a globally sorted organization that is continually maintained, in the absence of batching, which of these two organizations would incur the most concurrent contention: (i) sorted local organization or (ii) hashed local organization?
   Answer Sketch: Local sorting would suffer from the most contention, because each insert would lock the whole partition.

2. How might batching modifications onto a buffering array mitigate the concurrent contention that a locally sorted organization would otherwise suffer?
   Answer Sketch: Each modification would perform a very local lock (or even a lock-free compare-and-swap) in order to add a description of that modification to the buffering array. Thus, each modification would not conflict with searches. By contrast, sorting

for each modification would conflict with searches. Of course, processing the batches will disrupt searches, but only once per batch as opposed to once per modification.

3. What considerations might influence the design of an adaptive distributed B$^+$-tree that determines how many of the top levels of the tree to replicate across processing nodes to benefit searches at the cost of invalidation when nodes in those top levels are updated (say due to a split)?
   Answer Sketch: Intuitively, if each of the distributed sites is responsible for some portion of the key space, then the replicated part of the B^+-tree should guide the search to the correct site. Provided the replicated part is seldom invalidated, the benefit of replication outweighs the cost of relatively rare invalidations. Thus, the overall consideration for what to replicate is based on an estimate of the frequency and cost of invalidation versus the benefit to searches of replication.

4. What would be a good local organization of an LSM-tree that should support time-travel queries for particular keys, but no range queries?
   Answer Sketch: Because there are point but no range queries, local hashing will work better than a sorted data structure. So, hash at each non-root level. In order to support time travel, no key-value pairs can be deleted, so the hash structures of non-root nodes won't be modified unless nodes are merged.

7.7 Further Readings

Data-series and Multi-dimensional Data Structures. The book you're currently reading focuses on data structures having a uni-dimensional key. These are prevalent in almost any type of computing system. Building data structures for multi-dimensional data and data (and time) series face the same challenges plus the additional challenge of the curse of dimensionality (Chen, 2009).

The interested reader can read more about spatial multi-dimensional indexes (Lu and Ooi, 1993; Ooi *et al.*, 1993) including their recent learned counterparts (Al-Mamun *et al.*, 2020), and separately data series indexing and management (Jensen *et al.*, 2017; Zoumpatianos and Palpanas, 2018).

8

Summary

This book has decomposed the art of key-value data structure design into a set of dimensions having to do with global organization, local organization, batching, and placement of modifications. Each dimension has a set of possible values (e.g., hash-, range-, radix-partitioning for global organization). Certain combinations of dimension decisions work well for some workloads and less well for others.

Given a workload, this book has provided a set of guidelines on how to create data structures that support this workload by suggesting the values for each dimension. The ability to think about the design process as a set of design dimensions supports (i) choosing among well-known data structures and (ii) exploring new designs that emerge as possible combinations of various design dimension values.

Because the workload can vary both in time and on different parts of the key space, a good design may incorporate various kinds of adaptivity: (i) Different parts of the data structure may reflect different dimension decisions because, for example, modifications hit only one part of the data structure. (ii) Temporal variations in the workload may cause parts of the data structure to be less organized because modifications are frequent, whereas other parts are sorted to support point and range queries.

In addition to discussing individual data structures, we discuss (Chapter 6) how different data-intensive applications (e.g., databases, file systems, and machine learning systems) use multiple data structures for storage and workflow processing.

We also discuss issues that impact data structure design but have been so far studied more or less in isolation (Chapter 7). These issues influence the design space in a still-to-be-researched way.

Data structure design remains an art. We hope a deeper understanding of its dimensions will make the new data structures you design both beautiful and efficient.

Acknowledgments

The authors would like to thank the anonymous reviewers of the now publishers' Foundations and Trends in Databases publication, Alberto Lerner, and Viktor Sanca, for their careful reading and their insightful feedback. We also thank Zhehu Yuan, Konstantinos Karatsenidis, Jinqi Lu, and all BU DiSC lab members for their careful proofreading. The authors would also like to thank the following funding bodies for their financial support:

(Athanassoulis): U.S. National Science Foundation grant 2144547, a Facebook Faculty Research Award, a Meta gift, and a Red Hat Research Incubation Award.

(Idreos): U.S. Department of Energy Early Career Award grant DE-SC0020200.

(Shasha): U.S. National Science Foundation grants 1840761 A002, 1934388, 1840761, U.S. National Institutes of Health 1R01GM121753-01A1, and NYU Wireless.

References

Abadi, D. J., P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. (2013). "The Design and Implementation of Modern Column-Oriented Database Systems". *Foundations and Trends in Databases*. 5(3): 197–280. DOI: 10.1561/1900000024.

Abadi, D. J., D. S. Myers, D. J. DeWitt, and S. R. Madden. (2007). "Materialization Strategies in a Column-Oriented DBMS". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 466–475. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4221695.

Abu-Libdeh, H., D. Altınbüken, A. Beutel, E. H. Chi, L. Doshi, T. Kraska, Xiaozhou, Li, A. Ly, and C. Olston. (2020). "Learned Indexes for a Google-scale Disk-based Database". In: *Proceedings of the Workshop on ML for Systems at NeurIPS*. URL: http://mlforsystems.org/assets/papers/neurips2020/learned_abu-libdeh_2020.pdf.

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

Aggarwal, A. and J. S. Vitter. (1988). "The Input/Output Complexity of Sorting and Related Problems". *Communications of the ACM*. 31(9): 1116–1127. DOI: 10.1145/48529.48535.

Ailamaki, A., D. J. DeWitt, M. D. Hill, and M. Skounakis. (2001). "Weaving Relations for Cache Performance". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 169–180. URL: http://dl.acm.org/citation.cfm?id=645927.672367.

Ailamaki, A., D. J. DeWitt, M. D. Hill, and D. A. Wood. (1999). "DBMSs on a modern processor: Where does time go?" In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 266–277. URL: http://128.105.2.28/pub/techreports/1999/TR1394.pdf.

Ailamaki, A., E. Liarou, P. Tözün, D. Porobic, and I. Psaroudakis. (2014). "How to stop under-utilization and love multicores". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 189–192. DOI: 10.1109/ICDE.2015.7113419.

Ailamaki, A., E. Liarou, P. Tözün, D. Porobic, and I. Psaroudakis. (2017). *Databases on Modern Hardware: How to Stop Underutilization and Love Multicores. Synthesis Lectures on Data Management*. Morgan & Claypool Publishers. DOI: 10.2200/S00774ED1V01Y201704DTM045.

Ailamaki, A., D. J. DeWitt, and M. D. Hill. (2002). "Data Page Layouts for Relational Databases on Deep Memory Hierarchies". *The VLDB Journal*. 11(3): 198–215. DOI: 10.1007/s00778-002-0074-9.

Alagiannis, I., S. Idreos, and A. Ailamaki. (2014). "H2O: A Hands-free Adaptive Store". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1103–1114. DOI: 10.1145/2588555.2610502.

Alsubaiee, S., Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. (2014). "AsterixDB: A Scalable, Open Source BDMS". *Proceedings of the VLDB Endowment*. 7(14): 1905–1916. DOI: 10.14778/2733085.2733096.

Andersen, D. G., J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. (2009). "FAWN: A Fast Array of Wimpy Nodes". In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 1–14. DOI: 10.1145/1629575.1629577.

Antoshenkov, G. (1995). "Byte-aligned Bitmap Compression". In: *Proceedings of the Conference on Data Compression (DCC)*. 476–476. URL: http://dl.acm.org/citation.cfm?id=874051.874730.

Arulraj, J., A. Pavlo, and P. Menon. (2016). "Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 583–598. DOI: 10.1145/2882903.2915231.

Arumugam, S., A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. (2010). "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*. 519–530. URL: http://dl.acm.org/citation.cfm?id=1807167.1807224.

Athanassoulis, M. (2014). "Solid-State Storage and Work Sharing for Efficient Scaleup Data Analytics". *PhD thesis*. EPFL. DOI: http://dx.doi.org/10.5075/epfl-thesis-6032.

Athanassoulis, M. and A. Ailamaki. (2014). "BF-Tree: Approximate Tree Indexing". *Proceedings of the VLDB Endowment*. 7(14): 1881–1892. URL: http://www.vldb.org/pvldb/vol7/p1881-athanassoulis.pdf.

Athanassoulis, M., A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. (2010). "Flash in a DBMS: Where and How?" *IEEE Data Engineering Bulletin*. 33(4): 28–34. URL: http://sites.computer.org/debull/A10dec/athanassoulis.pdf.

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

Athanassoulis, M. and S. Idreos. (2016). "Design Tradeoffs of Data Access Methods". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*.

Athanassoulis, M., M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. (2016). "Designing Access Methods: The RUM Conjecture". In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 461–466. URL: http://dx.doi.org/10.5441/002/edbt.2016.42.

Athanassoulis, M., S. Sarkar, T. I. Papon, Z. Zhu, and D. Staratzis. (2022). "Building Deletion-Compliant Data Systems". In: *IEEE Data Engineering Bulletin*. 21–36.

Badam, A., K. Park, V. S. Pai, and L. L. Peterson. (2009). "HashCache: Cache Storage for the Next Billion". In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 123–136. URL: http://dl.acm.org/citation.cfm?id=1558977.1558986.

Barber, R., 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. (2012). "Business Analytics in (a) Blink". *IEEE Data Engineering Bulletin*. 35(1): 9–14. URL: http://sites.computer.org/debull/A12mar/blink.pdf.

Barber, R., G. M. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer. (2015). "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)*.

Bayer, R. (1972). "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms". *Acta Informatica*. 1: 290–306. DOI: 10.1007/BF00289509.

Bayer, R. and E. M. McCreight. (1970). "Organization and Maintenance of Large Ordered Indices". In: *Proceedings of the ACM SIGFIDET Workshop on Data Description and Access*. 107–141. DOI: 10.1007/BF00288683.

Bayer, R. and E. M. McCreight. (1972). "Organization and Maintenance of Large Ordered Indices". *Acta Informatica*. 1(3): 173–189. DOI: 10.1007/BF00288683.

Becker, B., S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. (1996). "An Asymptotically Optimal Multiversion B-Tree". *The VLDB Journal*. 5(4): 264–275. DOI: 10.1007/s007780050028.

Bender, M. A., M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. (2015). "An Introduction to Bε-trees and Write-Optimization". *White Paper*. URL: http://supertech.csail.mit.edu/papers/BenderFaJa15.pdf.

Bender, M. A., M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. (2012). "Don't Thrash: How to Cache Your Hash on Flash". *Proceedings of the VLDB Endowment*. 5(11): 1627–1637. URL: http://dl.acm.org/citation.cfm?id=2350229.2350275.

Bentley, J. L. (1979). "Decomposable Searching Problems". *Information Processing Letters*. 8(5): 244–251. DOI: 10.1016/0020-0190(79)90117-0.

Bentley, J. L. and J. B. Saxe. (1980). "Decomposable Searching Problems I: Static-to-Dynamic Transformation". *Journal of Algorithms*. 1(4): 301–358. DOI: 10.1016/0196-6774(80)90015-2.

Bentley, J. L. and A. C.-C. Yao. (1976). "An Almost Optimal Algorithm for Unbounded Searching". *Information Processing Letters*. 5(3): 82–87. DOI: 10.1016/0020-0190(76)90071-5.

Bernstein, P. A., V. Hadzilacos, and N. Goodman. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Bhat, W. A. (2018). "Bridging data-capacity gap in big data storage". *Future Generation Computer Systems*. DOI: https://doi.org/10.1016/j.future.2017.12.066.

Bloom, B. H. (1970). "Space/Time Trade-offs in Hash Coding with Allowable Errors". *Communications of the ACM*. 13(7): 422–426. URL: http://dl.acm.org/citation.cfm?id=362686.362692.

Boncz, P. A., M. L. Kersten, and S. Manegold. (2008). "Breaking the Memory Wall in MonetDB". *Communications of the ACM*. 51(12): 77–85. DOI: 10.1145/1409360.1409380.

Boncz, P. A., S. Manegold, and M. L. Kersten. (1999). "Database architecture optimized for the new bottleneck: Memory access". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 54–65. URL: http://www.vldb.org/conf/1999/P5.pdf.

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

Borovica-Gajic, R., S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser. (2015). "Smooth Scan: Statistics-Oblivious Access Paths". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 315–326. DOI: 10.1109/ICDE.2015.7113294.

Bovet, D. P. and M. Cesati. (2005). *Understanding the Linux Kernel*. 3rd Editio. O'Reilly Media, Inc.

Breslow, A. and N. Jayasena. (2018). "Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity". *Proceedings of the VLDB Endowment*. 11(9): 1041–1055. DOI: 10.14778/3213880.3213884.

Brodal, G. S. and R. Fagerberg. (2003). "Lower Bounds for External Memory Dictionaries". In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 546–554. URL: http://dl.acm.org/citation.cfm?id=644108.644201%20http://www.cs.au.dk/~gerth/papers/alcomft-tr-03-75.pdf.

Candea, G., N. Polyzotis, and R. Vingralek. (2009). "A scalable, predictable join operator for highly concurrent data warehouses". *Proceedings of the VLDB Endowment*. 2(1): 277–288. URL: http://dl.acm.org/citation.cfm?id=1687627.1687659.

Candea, G., N. Polyzotis, and R. Vingralek. (2011). "Predictable Performance and High Query Concurrency for Data Analytics". *The VLDB Journal*. 20(2): 227–248. URL: http://dl.acm.org/citation.cfm?id=1969331.1969355.

Chan, C. Y., B. C. Ooi, and H. Lu. (1992). "Extensible Buffer Management of Indexes". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 444–454. URL: http://www.vldb.org/conf/1992/P444.PDF.

Chan, C.-Y. and Y. E. Ioannidis. (1998). "Bitmap index design and evaluation". *ACM SIGMOD Record*. 27(2): 355–366. DOI: 10.1145/276305.276336.

Chan, C.-Y. and Y. E. Ioannidis. (1999). "An efficient bitmap encoding scheme for selection queries". *ACM SIGMOD Record*. 28(2): 215–226. DOI: 10.1145/304181.304201.

Chandramouli, B., G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. (2018). "FASTER: A Concurrent Key-Value Store with In-Place Updates". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 275–290. DOI: 10.1145/3183713.3196898.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. (2006). "Bigtable: A Distributed Storage System for Structured Data". In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 205–218. URL: http://dl.acm.org/citation.cfm?id=1267308.1267323.

Chatterjee, S., M. Jagadeesan, W. Qin, and S. Idreos. (2022). "Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine". In: *In Proceedings of the Very Large Databases Endowment*. DOI: 10.14778/3485450.3485461.

Chazelle, B. and L. J. Guibas. (1985). "Fractional Cascading: A Data Structuring Technique with Geometric Applications". In: *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*. 90–100. DOI: 10.1007/BFb0015734.

Chen, L. (2009). "Curse of Dimensionality". In: *Encyclopedia of Database Systems*. Ed. by L. Liu and T. Özsu. Boston, MA: Springer US. 545–546. DOI: 10.1007/978-0-387-39940-9{_}133.

Chen, S., P. B. Gibbons, and T. C. Mowry. (2001). "Improving Index Performance through Prefetching". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 235–246. DOI: 10.1145/375663.375688.

Chen, S., P. B. Gibbons, T. C. Mowry, and G. Valentin. (2002). "Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 157–168. URL: http://dl.acm.org/citation.cfm?id=564691.564710.

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

Colantonio, A. and R. Di Pietro. (2010). "Concise: Compressed 'N' Composable Integer Set". *Information Processing Letters*. 110(16): 644–650. DOI: 10.1016/j.ipl.2010.05.018.

Comer, D. (1979). "The Ubiquitous B-Tree". *ACM Computing Surveys*. 11(2): 121–137. DOI: 10.1145/356770.356776.

Copeland, G. P. and S. Khoshafian. (1985). "A Decomposition Storage Model". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 268–279. URL: http://dl.acm.org/citation.cfm?id=318898.318923.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press. URL: http://mitpress.mit.edu/books/introduction-algorithms.

Dayan, N., M. Athanassoulis, and S. Idreos. (2017). "Monkey: Optimal Navigable Key-Value Store". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94. DOI: 10.1145/3035918.3064054.

Dayan, N., M. Athanassoulis, and S. Idreos. (2018). "Optimal Bloom Filters and Adaptive Merging for LSM-Trees". *ACM Transactions on Database Systems (TODS)*. 43(4): 1–16. DOI: 10.1145/3276980.

Dayan, N. and S. Idreos. (2018). "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*. 505–520. DOI: 10.1145/3183713.3196927.

Dayan, N. and S. Idreos. (2019). "The Log-Structured Merge-Bush & the Wacky Continuum". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 449–466. DOI: 10.1145/3299869.3319903.

Dayan, N., Y. Rochman, I. Naiss, S. Dashevsky, N. Rabinovich, E. Bortnikov, I. Maly, O. Frishman, I. B. Zion, Avraham, M. Twitto, U. Beitler, E. Ginzburg, and M. Mokryn. (2021). "The End of Moore's Law and the Rise of The Data Processor". *Proceedings of the VLDB Endowment*. 14(12): 2932–2944. URL: http://www.vldb.org/pvldb/vol14/p2932-dayan.pdf.

Debnath, B., S. Sengupta, and J. Li. (2010). "FlashStore: high throughput persistent key-value store". *Proceedings of the VLDB Endowment*. 3(1-2): 1414–1425. URL: http://dl.acm.org/citation.cfm?id=1920841.1921015.

Debnath, B., S. Sengupta, and J. Li. (2011). "SkimpyStash: RAM space skimpy key-value store on flash-based storage". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 25–36. DOI: 10.1145/1989323.1989327.

Deeds, K., B. Hentschel, and S. Idreos. (2020). "Stacked Filters: Learning to Filter by Structure". *Proceedings of the VLDB Endowment*. 14(4): 600–612. DOI: 10.14778/3436905.3436919.

Deliège, F. and T. B. Pedersen. (2010). "Position list word aligned hybrid: optimizing space and performance for compressed bitmaps". In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 228–239. DOI: 10.1145/1739041.1739071.

Ding, B., S. Chaudhuri, and V. R. Narasayya. (2020a). "Bitvector-aware Query Optimization for Decision Support Queries". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2011–2026. DOI: 10.1145/3318464.3389769.

Ding, J., U. F. Minhas, B. Chandramouli, C. Wang, Y. Li, Y. Li, D. Kossmann, J. Gehrke, and T. Kraska. (2021). "Instance-Optimized Data Layouts for Cloud Analytics Workloads". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 418–431. DOI: 10.1145/3448016.3457270.

Ding, J., U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska. (2020b). "ALEX: An Updatable Adaptive Learned Index". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 969–984. DOI: 10.1145/3318464.3389711.

Dittrich, J. and A. Jindal. (2011). "Towards a One Size Fits All Database Architecture". In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. 195–198. URL: http://cidrdb.org/cidr2011/Papers/CIDR11_Paper25.pdf.

Dong, J. and R. Hull. (1982). "Applying Approximate Order Dependency to Reduce Indexing Space". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 119–127. DOI: 10.1145/582353.582375.

Dong, S., M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. (2017). "Optimizing Space Amplification in RocksDB". In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. URL: http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf.

Dong, S., A. Kryczka, Y. Jin, and M. Stumm. (2021). "RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications". *ACM Transactions on Storage (TOS)*. 17(4): 26:1–26:32. DOI: 10.1145/3483840.

Faerber, F., A. Kemper, P.-Å. Larson, J. J. Levandoski, T. Neumann, and A. Pavlo. (2017). "Main Memory Database Systems". *Foundations and Trends in Databases*. 8(1-2): 1–130. DOI: 10.1561/1900000058.

Falkoff, A. D. and K. E. Iverson. (1973). "The Design of APL". *IBM Journal of Research and Development*. 17(5): 324–334. DOI: 10.1147/rd.174.0324.

Fan, B., D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. (2014). "Cuckoo Filter: Practically Better Than Bloom". In: *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 75–88. DOI: 10.1145/2674005.2674994.

Fang, J., Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee. (2020). "In-memory database acceleration on FPGAs: a survey". *The VLDB Journal*. 29(1): 33–59. DOI: 10.1007/s00778-019-00581-w.

Färber, F., S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. (2011). "SAP HANA Database: Data Management for Modern Business Applications". *ACM SIGMOD Record*. 40(4): 45–51. DOI: 10.1145/2094114.2094126.

Färber, F., N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. (2012). "The SAP HANA Database – An Architecture Overview". *IEEE Data Engineering Bulletin*. 35(1): 28–33. URL: http://sites.computer.org/debull/A12mar/hana.pdf.

Ferragina, P. and G. Vinciguerra. (2020). "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds". *Proceedings of the VLDB Endowment*. 13(8): 1162–1175. DOI: 10.14778/3389133.3389135.

Fox, E. A., Q. F. Chen, A. M. Daoud, and L. S. Heath. (1991). "Order-Preserving Minimal Perfect Hash Functions and Information Retrieval". *ACM Transactions on Information Systems (TOIS)*. 9(3): 281–308. DOI: 10.1145/125187.125200.

Francisco, P. (2011). "The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics". *IBM Redbooks*. URL: http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf.

French, C. D. (1995). ""One size fits all" database architectures do not work for DSS". *ACM SIGMOD Record*. 24(2): 449–450. DOI: 10.1145/568271.223871.

French, C. D. (1997). "Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 194–198. URL: http://dl.acm.org/citation.cfm?id=645482.653422.

Galakatos, A., M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. (2019). "FITing-Tree: A Data-aware Index Structure". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1189–1206. DOI: 10.1145/3299869.3319860.

Giannikis, G., G. Alonso, and D. Kossmann. (2012). "SharedDB: Killing One Thousand Queries with One Stone". *Proceedings of the VLDB Endowment*. 5(6): 526–537. URL: http://dl.acm.org/citation.cfm?id=2168651.2168654.

Giannikis, G., D. Makreshanski, G. Alonso, and D. Kossmann. (2013). "Workload optimization using SharedDB". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1045–1048. DOI: 10.1145/2463676.2463678.

Gottstein, R., R. Goyal, S. Hardock, I. Petrov, and A. P. Buchmann. (2014). "MV-IDX: indexing in multi-version databases". In: *Proceedings of the Symposium on International Database Engineering & Applications (IDEAS)*. 142–148. DOI: 10.1145/2628194.2628911.

Graefe, G. (2003). "Sorting And Indexing With Partitioned B-Trees". In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. URL: http://www-db.cs.wisc.edu/cidr/cidr2003/program/p1.pdf.

Graefe, G. (2011). "Modern B-Tree Techniques". *Foundations and Trends in Databases*. 3(4): 203–402. URL: http://dx.doi.org/10.1561/1900000028.

Graefe, G., F. Halim, S. Idreos, H. Kuno, and S. Manegold. (2012). "Concurrency control for adaptive indexing". *Proceedings of the VLDB Endowment*. 5(7): 656–667. URL: http://dl.acm.org/citation.cfm?id=2180918.

Graefe, G., S. Idreos, H. Kuno, and S. Manegold. (2010). "Benchmarking adaptive indexing". In: *Proceedings of the TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems (TPCTC)*. 169–184. URL: http://dl.acm.org/citation.cfm?id=1946050.1946063.

Graefe, G. and H. Kuno. (2010a). "Self-selecting, self-tuning, incrementally optimized indexes". In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 371–381. URL: http://dl.acm.org/citation.cfm?id=1739041.1739087.

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

Graefe, G., H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch. (2014). "In-Memory Performance for Big Data". *Proceedings of the VLDB Endowment*. 8(1): 37–48. DOI: 10.14778/2735461.2735465.

Gray, J. and F. Putzolu. (1986). "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*. URL: http://db.cs.berkeley.edu/cs286/papers/fiveminute-tr1986.pdf.

Grund, M., J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. (2010). "HYRISE: A Main Memory Hybrid Storage Engine". *Proceedings of the VLDB Endowment*. 4(2): 105–116. DOI: 10.14778/1921071.1921077.

Guttman, A. (1984). "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 47–57. DOI: 10.1145/602259.602266.

Haeupler, B., S. Sen, and R. E. Tarjan. (2015). "Rank-Balanced Trees". *ACM Transactions on Algorithms (TALG)*. 11(4): 30:1–30:26. DOI: 10.1145/2689412.

Halim, F., S. Idreos, P. Karras, and R. H. C. Yap. (2012). "Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores." *Proceedings of the VLDB Endowment*. 5(6): 502–513. URL: http://vldb.org/pvldb/vol5/p502_felixhalim_vldb2012.pdf.

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

Harizopoulos, S., V. Shkapenyuk, and A. Ailamaki. (2005). "QPipe: A Simultaneously Pipelined Relational Query Engine". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 383–394. URL: http://dl.acm.org/citation.cfm?id=1066157.1066201.

Hellerstein, J. M., E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. (2002). "On a Model of Indexability and Its Bounds for Range Queries". *Journal of the ACM*. 49(1): 35–55. DOI: 10.1145/505241.505244.

Hellerstein, J. M., E. Koutsoupias, and C. H. Papadimitriou. (1997). "On the Analysis of Indexing Schemes". In: *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*. 249–256. DOI: 10.1145/263661.263688.

Hellerstein, J. M., J. F. Naughton, and A. Pfeffer. (1995). "Generalized Search Trees for Database Systems". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 562–573. URL: http://dl.acm.org/citation.cfm?id=645921.673145.

Hellerstein, J. M., M. Stonebraker, and J. R. Hamilton. (2007). "Architecture of a Database System". *Foundations and Trends in Databases*. 1(2): 141–259. DOI: 10.1561/1900000002.

Héman, S., M. Zukowski, and N. J. Nes. (2010). "Positional Update Handling in Column Stores". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 543–554. URL: http://dl.acm.org/citation.cfm?id=1807167.1807227.

Herlihy, M., N. Shavit, V. Luchangco, and M. Spear. (2020). *The Art of Multiprocessor Programming (Second Edition)*. Morgan Kaufmann. DOI: 10.1016/C2011-0-06993-4.

Hilbert, M. and P. López. (2011). "The World's Technological Capacity to Store, Communicate, and Compute Information". *Science*. 332(6025): 60–65. DOI: 10.1126/science.1200970.

Holanda, P., S. Manegold, H. Mühleisen, and M. Raasveldt. (2019). "Progressive Indexes: Indexing for Interactive Data Analysis". *Proceedings of the VLDB Endowment*. 12(13): 2366–2378. URL: http://www.vldb.org/pvldb/vol12/p2366-holanda.pdf.

Huang, G., X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. (2019). "X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 651–665. DOI: 10.1145/3299869.3314041.

Huang, H. and S. Ghandeharizadeh. (2021). "Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 749–763. DOI: 10.1145/3448016.3457297.

Huang, K., Y. He, and T. Wang. (2022). "The Past, Present and Future of Indexing on Persistent Memory". *Proceedings of the VLDB Endowment*. 15(12): 3774–3777. URL: https://www.vldb.org/pvldb/vol15/p3774-wang.pdf%20https://www2.cs.sfu.ca/~tzwang/pmem-index-tutorial-slides.pdf.

Hughes, J. (2013). "Revolutions in Storage". In: *IEEE Conference on Massive Data Storage*. Long Beach, CA. URL: http://storageconference.us/2013/Presentations/Hughes.pdf.

Hutflesz, A., H.-W. Six, and P. Widmayer. (1988). "Globally Order Preserving Multidimensional Linear Hashing". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 572–579. DOI: 10.1109/ICDE.1988.105505.

Idreos, S. and M. Callaghan. (2020). "Key-Value Storage Engines". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2667–2672. DOI: 10.1145/3318464.3383133.

Idreos, S., N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. (2019a). "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)*.

Idreos, S., F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. (2012). "MonetDB: Two Decades of Research in Column-oriented Database Architectures". *IEEE Data Engineering Bulletin*. 35(1): 40–45. URL: http://sites.computer.org/debull/A12mar/monetdb.pdf.

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

Idreos, S., M. L. Kersten, and S. Manegold. (2007b). "Updating a Cracked Database". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 413–424. DOI: 10.1145/1247480.1247527.

Idreos, S., M. L. Kersten, and S. Manegold. (2009). "Self-organizing Tuple Reconstruction in Column-Stores". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 297–308. DOI: 10.1145/1559845.1559878.

Idreos, S., S. Manegold, H. Kuno, and G. Graefe. (2011). "Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores". *Proceedings of the VLDB Endowment*. 4(9): 586–597. URL: https://www.vldb.org/pvldb/vol4/p586-idreos.pdf.

Idreos, S., K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. (2018a). "The Periodic Table of Data Structures". *IEEE Data Engineering Bulletin*. 41(3): 64–75. URL: http://sites.computer.org/debull/A18sept/p64.pdf.

Idreos, S., K. Zoumpatianos, S. Chatterjee, W. Qin, A. Wasay, B. Hentschel, M. S. Kester, N. Dayan, D. Guo, M. Kang, and Y. Sun. (2019b). "Learning Data Structure Alchemy". *IEEE Data Engineering Bulletin*. 42(2): 47–58. URL: http://sites.computer.org/debull/A19june/p47.pdf.

Idreos, S., K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. (2018b). "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*. 535–550. DOI: 10.1145/3183713.3199671.

István, Z., K. Kara, and D. Sidler. (2020). "FPGA-Accelerated Analytics: From Single Nodes to Clusters". *Foundations and Trends in Databases*. 9(2): 101–208. DOI: 10.1561/1900000072.

Jain, A., A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. (2018). "Gist: Efficient Data Encoding for Deep Neural Network Training". In: *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 776–789. DOI: 10.1109/ISCA.2018.00070.

Jannen, W., J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. (2015). "BetrFS: A Right-optimized Write-optimized File System". In: *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 301–315. URL: http://dl.acm.org/citation.cfm?id=2750482.2750505.

Jensen, S. K., T. B. Pedersen, and C. Thomsen. (2017). "Time Series Management Systems: A Survey". *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. 29(11): 2581–2600. DOI: 10.1109/TKDE.2017.2740932.

Jin, R., S. J. Kwon, and T.-S. Chung. (2011). "FlashB-tree: A Novel B-tree Inex Scheme for Solid State Drives". In: *Proceedings of the ACM Symposium on Research in Applied Computation (RACS)*. 50–55. DOI: 10.1145/2103380.2103390.

Johnson, R., S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. (2007). "To Share or Not to Share?" In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 351–362. URL: http://dl.acm.org/citation.cfm?id=1325851.1325894.

Johnson, R., I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. (2009). "Shore-MT: a scalable storage manager for the multicore era". In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 24–35. URL: http://dl.acm.org/citation.cfm?id=1516360.1516365.

Johnson, T. and D. Shasha. (1989). "Utilization of B-trees with Inserts, Deletes and Modifies". In: *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*. 235–246. DOI: 10.1145/73721.73745.

Johnson, T. and D. Shasha. (1994). "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 439–450. URL: http://dl.acm.org/citation.cfm?id=645920.672996.

Kang, D., D. Jung, J.-U. Kang, and J.-S. Kim. (2007). "mu-tree: an ordered index structure for NAND flash memory". In: *Proceedings of the ACM/IEEE International Conference on Embedded Software (EMSOFT)*. 144–153. URL: http://dl.acm.org/citation.cfm?doid=1289927.1289953.

Kang, Y., R. Pitchumani, P. Mishra, Y.-S. Kee, F. Londono, S. Oh, J. Lee, and D. D. G. Lee. (2019). "Towards building a high-performance, scale-in key-value storage system". In: *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR)*. 144–154. DOI: 10.1145/3319647.3325831.

Karger, D. R., E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. (1997). "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Annual ACM Symposium on the Theory of Computing (STOC)*. 654–663. DOI: 10.1145/258533.258660.

Kemper, A. and T. Neumann. (2011). "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)*. 195–206. DOI: 10.1109/ICDE.2011.5767867.

Kennedy, O. and L. Ziarek. (2015). "Just-In-Time Data Structures". In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. URL: http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper9.pdf.

Kester, M. S., M. Athanassoulis, and S. Idreos. (2017). "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*. 715–730. DOI: 10.1145/3035918.3064049.

Kim, W., K.-C. Kim, and A. G. Dale. (1989). "Indexing Techniques for Object-Oriented Databases". In: *Object-Oriented Concepts, Databases, and Applications*. ACM Press and Addison-Wesley. 371–394.

Koltsidas, I. and S. D. Viglas. (2011). "Data management over flash memory". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1209–1212. DOI: 10.1145/1989323.1989455.

Kornacker, M., C. Mohan, and J. M. Hellerstein. (1997). "Concurrency and Recovery in Generalized Search Trees". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 62–72. DOI: 10.1145/253260.253272.

Kraska, T., A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. (2018). "The Case for Learned Index Structures". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 489–504. DOI: 10.1145/3183713.3196909.

Krishna, S., N. Patel, D. Shasha, and T. Wies. (2021). "Automated Verification of Concurrent Search Structures". *Synthesis Lectures on Computer Science*. 9(1): 1–188. DOI: 10.2200/S01089ED1V01Y202104CSL013.

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

Lahiri, T., 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. (2015). "Oracle Database In-Memory: A Dual Format In-Memory Database". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

Lam, M. S., E. E. Rothberg, and M. E. Wolf. (1991). "The Cache Performance and Optimizations of Blocked Algorithms". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 63–74. DOI: 10.1145/106972.106981.

Lamb, A., M. Fuller, and R. Varadarajan. (2012). "The Vertica Analytic Database: C-Store 7 Years Later". *Proceedings of the VLDB Endowment*. 5(12): 1790–1801. URL: http://dl.acm.org/citation.cfm?id=2367518.

Lang, H., T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. (2016). "Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. DOI: 10.1145/2882903.2882925.

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

Larson, P.-A., C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. (2011). "SQL server column store indexes". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1177–1184. DOI: 10.1145/1989323.1989448.

Larson, P.-A., E. N. Hanson, and S. L. Price. (2012). "Columnar Storage in SQL Server 2012". *IEEE Data Engineering Bulletin*. 35(1): 15–20. URL: http://sites.computer.org/debull/A12mar/apollo.pdf.

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

Larson, P.-A., R. Rusanu, M. Saubhasik, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, and S. Rangarajan. (2013). "Enhancements to SQL server column stores". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1159–1168. DOI: 10.1145/2463676.2463708.

Leeuwen, J. van and M. H. Overmars. (1981). "The Art of Dynamizing". In: *Proceedings of Mathematical Foundations of Computer Science*. 121–131. DOI: 10.1007/3-540-10856-4{_}78.

Leeuwen, J. van and D. Wood. (1980). "Dynamization of Decomposable Searching Problems". *Information Processing Letters*. 10(2): 51–56. DOI: 10.1016/S0020-0190(80)90073-3.

Lehman, P. L. and S. B. Yao. (1981). "Efficient Locking for Concurrent Operations on B-Trees". *ACM Transactions on Database Systems (TODS)*. 6(4): 650–670. DOI: 10.1145/319628.319663.

Lehman, T. J. and M. J. Carey. (1986). "A Study of Index Structures for Main Memory Database Management Systems". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 294–303. URL: http://www.vldb.org/conf/1986/P294.PDF.

Leis, V., A. Kemper, and T. Neumann. (2013). "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 38–49. DOI: 10.1109/ICDE.2013.6544812.

Lerner, A. and P. Bonnet. (2021). "Not your Grandpa's SSD: The Era of Co-Designed Storage Devices". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2852–2858. DOI: 10.1145/3448016.3457540.

Levandoski, J. J., D. B. Lomet, and S. Sengupta. (2013). "The Bw-Tree: A B-tree for New Hardware Platforms". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 302–313. DOI: 10.1109/ICDE.2013.6544834.

Li, Y., B. He, J. Yang, Q. Luo, K. Yi, and R. J. Yang. (2010). "Tree Indexing on Solid State Drives". *Proceedings of the VLDB Endowment*. 3(1-2): 1195–1206. URL: http://dl.acm.org/citation.cfm?id=1920841.1920990.

Lim, H., B. Fan, D. G. Andersen, and M. Kaminsky. (2011). "SILT: A Memory-Efficient, High-Performance Key-Value Store". In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 1–13. URL: http://dl.acm.org/citation.cfm?id=2043556.2043558.

Lin, K.-I., H. V. Jagadish, and C. Faloutsos. (1994). "The TV-Tree: An Index Structure for High-Dimensional Data". *The VLDB Journal*. 3(4): 517–542. URL: http://www.vldb.org/journal/VLDBJ3/P517.pdf.

Litwin, W. (1980). "Linear Hashing: A New Tool for File and Table Addressing". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 212–223.

Litwin, W. and D. B. Lomet. (1986). "The Bounded Disorder Access Method". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 38–48. URL: http://dl.acm.org/citation.cfm?id=645471.655390.

Litwin, W., M.-A. Neimat, and D. A. Schneider. (1994). "RP*: A Family of Order Preserving Scalable Distributed Data Structures". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 342–353. URL: http://www.vldb.org/conf/1994/P342.PDF.

Litwin, W., M.-A. Neimat, and D. A. Schneider. (1996). "LH* - A Scalable, Distributed Data Structure". *ACM Transactions on Database Systems (TODS)*. 21(4): 480–525. DOI: 10.1145/236711.236713.

Lively, T., L. Schroeder, and C. Mendizábal. (2018). "Splaying Log-Structured Merge-Trees". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1839–1841. DOI: 10.1145/3183713.3183723.

Lomet, D. B. and B. Salzberg. (1990). "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance". *ACM Transactions on Database Systems (TODS)*. 15(4): 625–658. DOI: 10.1145/99935.99949.

Lomet, D. B. and B. Salzberg. (1992). "Access Method Concurrency with Recovery". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 351–360. DOI: 10.1145/130283.130336.

Lu, H. and B. C. Ooi. (1993). "Spatial Indexing: Past and Future". *IEEE Data Engineering Bulletin*. 16(3): 16–21. URL: http://sites.computer.org/debull/93SEP-CD.pdf.

Lu, L., T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. (2016). "WiscKey: Separating Keys from Values in SSD-conscious Storage". In: *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 133–148. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu.

Luo, C. and M. J. Carey. (2022). "DynaHash: Efficient Data Rebalancing in Apache AsterixDB". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 485–497. DOI: 10.1109/ICDE53745.2022.00041.

Luo, C. and M. J. Carey. (2020). "LSM-based Storage Techniques: A Survey". *The VLDB Journal*. 29(1): 393–418. DOI: 10.1007/s00778-019-00555-y.

Ma, S., T. Ma, K. Chen, and Y. Wu. (2022). "A Survey of Storage Systems in the RDMA Era". *IEEE Transactions on Parallel and Distributed Systems*. 33(12): 4395–4409. DOI: 10.1109/TPDS.2022.3188656.

MacNicol, R. and B. French. (2004). "Sybase IQ Multiplex - Designed For Analytics". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 1227–1230. URL: http://www.vldb.org/conf/2004/IND8P3.PDF.

Al-Mamun, A., H. Wu, and W. G. Aref. (2020). "A Tutorial on Learned Multi-dimensional Indexes". In: *Proceedings of the International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*. 1–4. DOI: 10.1145/3397536.3426358.

Manegold, S. (2009). "Memory Hierarchy". In: *Encyclopedia of Database Systems*. Ed. by L. Liu and T. Özsu. Boston, MA: Springer US. 1707–1713. DOI: 10.1007/978-0-387-39940-9{_}657.

Manegold, S., P. A. Boncz, and M. L. Kersten. (2002a). "Generic Database Cost Models for Hierarchical Memory Systems". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 191–202. URL: http://dl.acm.org/citation.cfm?id=1287369.1287387.

Manegold, S., P. A. Boncz, and M. L. Kersten. (2002b). "Optimizing Main-Memory Join on Modern Hardware". *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. 14(4): 709–730. DOI: 10.1109/TKDE.2002.1019210.

Manegold, S., P. A. Boncz, and N. Nes. (2004). "Cache-Conscious Radix-Decluster Projections". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 684–695. URL: http://www.vldb.org/conf/2004/RS18P3.PDF.

Manolopoulos, Y., A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. (2006). *R-Trees: Theory and Applications. Advanced Information and Knowledge Processing*. Springer. DOI: 10.1007/978-1-84628-293-5.

Mao, Y., E. Kohler, and R. T. Morris. (2012). "Cache Craftiness for Fast Multicore Key-value Storage". In: *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 183–196. DOI: 10.1145/2168836.2168855.

Margaritov, A., D. Ustiugov, E. Bugnion, and B. Grot. (2018). "Virtual Address Translation via Learned Page Table Indexes". In: *Proceedings of the Workshop on ML for Systems at NeurIPS*. URL: http://mlforsystems.org/assets/papers/neurips2018/virtual_margaritov_2018.pdf.

Mediano, M. R., M. A. Casanova, and M. Dreux. (1994). "V-Trees - A Storage Method for Long Vector Data". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 321–330. URL: http://www.vldb.org/conf/1994/P321.PDF.

Megiddo, N. and D. S. Modha. (2004). "Outperforming LRU with an Adaptive Replacement Cache Algorithm". *Computer*. 37(4): 58–65. DOI: 10.1109/MC.2004.1297303.

Mehta, M., V. Soloviev, and D. J. DeWitt. (1993). "Batch Scheduling in Parallel Database Systems". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 400–410. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=344041.

Menon, J., D. A. Pease, R. M. Rees, L. Duyanovich, and B. L. Hillsberg. (2003). “IBM Storage Tank - A heterogeneous scalable SAN file system”. *IBM Systems Journal*. 42(2): 250–267. DOI: 10.1147/sj.422.0250.

Milojicic, D. S. and T. Roscoe. (2016). “Outlook on Operating Systems”. *IEEE Computer*. 49(1): 43–51. DOI: 10.1109/MC.2016.19.

Moerkotte, G. (1998). “Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing”. In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 476–487. URL: http://dl.acm.org/citation.cfm?id=645924.671173.

Mohan, C. (2014). “Tutorial: An In-Depth Look at Modern Database Systems”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 674. DOI: 10.5441/002/edbt.2014.72.

Morrison, D. R. (1968). “PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric”. *Journal of the ACM*. 15(4): 514–534. DOI: 10.1145/321479.321481.

Muth, P., P. E. O’Neil, A. Pick, and G. Weikum. (2000). “The LHAM Log-structured History Data Access Method”. *The VLDB Journal*. 8(3-4): 199–221.

Na, G.-J., B. Moon, and S.-W. Lee. (2011). “IPLB+-tree for Flash Memory Database Systems”. *Journal of Information Science and Engineering (JISE)*. 27(1): 111–127. URL: http://www.iis.sinica.edu.tw/page/jise/2011/201101_08.html.

Narayanan, D., A. Phanishayee, K. Shi, X. Chen, and M. Zaharia. (2021). “Memory-Efficient Pipeline-Parallel DNN Training”. In: *Proceedings of the International Conference on Machine Learning (ICML)*. Vol. 139. *Proceedings of Machine Learning Research*. 7937–7947. URL: http://proceedings.mlr.press/v139/narayanan21a.html.

Nath, S. and A. Kansal. (2007). “FlashDB: dynamic self-tuning database for NAND flash”. *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN)*.

O’Neil, E. J., P. E. O’Neil, and G. Weikum. (1993). “The LRU-K page replacement algorithm for database disk buffering”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 297–306. DOI: 10.1145/170035.170081.

O'Neil, P. E., E. Cheng, D. Gawlick, and E. J. O'Neil. (1996). "The log-structured merge-tree (LSM-tree)". *Acta Informatica*. 33(4): 351–385. URL: http://dl.acm.org/citation.cfm?id=230823.230826.

Olma, M., M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. (2020). "Adaptive partitioning and indexing for in situ query processing". *The VLDB Journal*. 29(1): 569–591. URL: https://doi.org/10.1007/s00778-019-00580-x.

Ooi, B. C., R. Sacks-Davis, and J. Han. (1993). "Indexing in Spatial Databases". *Tech. rep.*

Overmars, M. H. and J. van Leeuwen. (1981). "Worst-Case Optimal Insertion and Deletion Methods for Decomposable Searching Problems". *Information Processing Letters*. 12(4): 168–173. DOI: 10.1016/0020-0190(81)90093-4.

Pandey, P., A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. Johnson. (2021). "Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1386–1399. DOI: 10.1145/3448016.3452841.

Papon, T. I. and M. Athanassoulis. (2021a). "A Parametric I/O Model for Modern Storage Devices". In: *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*.

Papon, T. I. and M. Athanassoulis. (2021b). "The Need for a New I/O Model". In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

Papon, T. I. and M. Athanassoulis. (2023). "ACEing the Bufferpool Management Paradigm for Modern Storage Devices". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

Paul, J., S. Lu, and B. He. (2021). "Database Systems on GPUs". *Foundations and Trends in Databases*. 11(1): 1–108. DOI: 10.1561/1900000076.

Perl, Y., A. Itai, and H. Avni. (1978). "Interpolation Search—A log logN Search". *Communications of the ACM*. 21(7): 550–553. URL: http://dl.acm.org/citation.cfm?id=359545.359557.

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

Pohl, C., K.-U. Sattler, and G. Graefe. (2020). "Joins on high-bandwidth memory: a new level in the memory hierarchy". *The VLDB Journal*. 29(2-3): 797–817. DOI: 10.1007/s00778-019-00546-z.

Polychroniou, O., A. Raghavan, and K. A. Ross. (2015). "Rethinking SIMD Vectorization for In-Memory Databases". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1493–1508. DOI: 10.1145/2723372.2747645.

Porobic, D., E. Liarou, P. Tözün, and A. Ailamaki. (2014). "ATraPos: Adaptive transaction processing on hardware Islands". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 688–699. DOI: 10.1109/ICDE.2014.6816692.

Psaroudakis, I., M. Athanassoulis, and A. Ailamaki. (2013). "Sharing Data and Work Across Concurrent Analytical Queries". *Proceedings of the VLDB Endowment*. 6(9): 637–648. URL: http://dl.acm.org/citation.cfm?id=2536360.2536364.

Pugh, W. (1990). "Skip Lists: A Probabilistic Alternative to Balanced Trees". *Communications of the ACM*. 33(6): 668–676. URL: http://dl.acm.org/citation.cfm?id=78977.

Qiao, L., V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. (2008). "Main-memory Scan Sharing for Multi-core CPUs". *Proceedings of the VLDB Endowment*. 1(1): 610–621. URL: http://dl.acm.org/citation.cfm?id=1453856.1453924.

Qin, W. and S. Idreos. (2016). "Adaptive Data Skipping in Main-Memory Systems". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2255–2256. DOI: 10.1145/2882903.2914836.

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

Ramamurthy, R., D. J. DeWitt, and Q. Su. (2002). "A Case for Fractured Mirrors". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 430–441. DOI: 10.1007/s00778-003-0093-1.

Ramamurthy, R., D. J. DeWitt, and Q. Su. (2003). "A Case for Fractured Mirrors". *The VLDB Journal*. 12(2): 89–101. DOI: 10.1007/s00778-003-0093-1.

Raman, A., S. Sarkar, M. Olma, and M. Athanassoulis. (2023). "Indexing for Near-Sorted Data". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

Raman, V., 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. (2013). "DB2 with BLU Acceleration: So Much More Than Just a Column Store". *Proceedings of the VLDB Endowment*. 6(11): 1080–1091. DOI: 10.14778/2536222.2536233.

Raman, V., G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. (2008). "Constant-Time Query Processing". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 60–69. DOI: 10.1109/ICDE.2008.4497414.

Rao, J. and K. A. Ross. (2000). "Making B+-trees Cache Conscious in Main Memory". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 475–486. DOI: 10.1145/342009.335449.

Riegger, C., T. Vinçon, R. Gottstein, and I. Petrov. (2020). "MV-PBT: Multi-Version Indexing for Large Datasets and HTAP Workloads". In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 217–228. DOI: 10.5441/002/edbt.2020.20.

Riegger, C., T. Vinçon, and I. Petrov. (2017). "Multi-version indexing and modern hardware technologies: a survey of present indexing approaches". In: *Proceedings of the International Conference on Information Integration and Web-based Applications & Services (iiWAS)*. 266–275. DOI: 10.1145/3151759.3151779.

Riegger, C., T. Vinçon, and I. Petrov. (2019). "Indexing large updatable datasets in multi-version database management systems". In: *Proceedings of the Symposium on International Database Engineering & Applications (IDEAS)*. 36:1–36:5. DOI: 10.1145/3331076.3331118.

Robinson, J. T. (1981). "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*. 10–18. DOI: 10.1145/582318.582321.

Robinson, J. T. (1986). "Order Preserving Linear Hashing Using Dynamic Key Statistics". In: *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*. 91–99. DOI: 10.1145/6012.6014.

Rodeh, O. (2008). "B-trees, shadowing, and clones". *ACM Transactions on Storage*. 3(4): 2:1–2:27. DOI: 10.1145/1326542.1326544.

Roh, H., S. Park, S. Kim, M. Shin, and S.-W. Lee. (2011). "B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives". *Proceedings of the VLDB Endowment*. 5(4): 286–297. URL: http://dl.acm.org/citation.cfm?id=2095686.2095688.

Ross, K. A. (2021). "Utilizing (and Designing) Modern Hardware for Data-Intensive Computations: The Role of Abstraction". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1. DOI: 10.1145/3448016.3460535.

Ross, K. A. (2004). "Selection Conditions in Main Memory". *ACM Transactions on Database Systems (TODS)*. 29: 132–161. DOI: 10.1145/974750.974755.

Sabek, I., K. Vaidya, D. Horn, A. Kipf, M. Mitzenmacher, and T. Kraska. (2022). "Can Learned Models Replace Hash Functions?" *Proceedings of the VLDB Endowment*. 16(3): 532–545. URL: https://www.vldb.org/pvldb/vol16/p532-sabek.pdf.

Sacco, G. M. (1987). "Index Access with a Finite Buffer". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 301–309. URL: http://www.vldb.org/conf/1987/P301.PDF.

Sagiv, Y. (1986). "Concurrent Operations on B*-Trees with Overtaking". *Journal of Computer and System Sciences (JCSS)*. 33(2): 275–296. DOI: 10.1016/0022-0000(86)90021-8.

Sarkar, S. and M. Athanassoulis. (2022). "Dissecting, Designing, and Optimizing LSM-based Data Stores". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2489–2497. DOI: 10.1145/3514221.3522563.

Sarkar, S., K. Chen, Z. Zhu, and M. Athanassoulis. (2022). "Compactionary: A Dictionary for LSM Compactions". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2429–2432. DOI: 10.1145/3514221.3520169.

Sarkar, S., N. Dayan, and M. Athanassoulis. (2023). "The LSM Design Space and its Read Optimizations". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

Sarkar, S., T. I. Papon, D. Staratzis, and M. Athanassoulis. (2020). "Lethe: A Tunable Delete-Aware LSM Engine". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908. DOI: 10.1145/3318464.3389757.

Sarkar, S., D. Staratzis, Z. Zhu, and M. Athanassoulis. (2021). "Constructing and Analyzing the LSM Compaction Design Space". *Proceedings of the VLDB Endowment*. 14(11): 2216–2229. URL: http://vldb.org/pvldb/vol14/p2216-sarkar.pdf.

Schlegel, B., R. Gemulla, and W. Lehner. (2009). "k-ary search on modern processors". In: *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 52–60. DOI: 10.1145/1565694.1565705.

Scholten, H. W. and M. H. Overmars. (1989). "General Methods for Adding Range Restrictions to Decomposable Searching Problems". *Journal of Symbolic Computation*. 7(1): 1–10. DOI: 10.1016/S0747-7171(89)80002-1.

Schuhknecht, F. M., J. Dittrich, and L. Linden. (2018). "Adaptive Adaptive Indexing". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 665–676. DOI: 10.1109/ICDE.2018.00066.

Sears, R. and R. Ramakrishnan. (2012). "bLSM: A General Purpose Log Structured Merge Tree". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 217–228. DOI: 10.1145/2213836.2213862.

Sedgewick, R. (1983). "Balanced Trees". In: *Algorithms*. Addison-Wesley.

Selinger, P. G., M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. (1979). "Access Path Selection in a Relational Database Management System". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 23–34. DOI: 10.1145/582095.582099.

Severance, D. G. and G. M. Lohman. (1976). "Differential files: their application to the maintenance of large databases". *ACM Transactions on Database Systems (TODS)*. 1(3): 256–267. URL: http://dl.acm.org/citation.cfm?id=320473.320484.

Shasha, D. E. and N. Goodman. (1988). "Concurrent Search Structure Algorithms". *ACM Transactions on Database Systems (TODS)*. 13(1): 53–90. DOI: 10.1145/42201.42204.

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

Shehabi, A., S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner. (2016). "United States Data Center Energy Usage Report". *Ernest Orlando Lawrence Berkeley National Laboratory*. LBNL-10057. URL: https://eta.lbl.gov/publications/united-states-data-center-energy.

Sidirourgos, L. and M. L. Kersten. (2013). "Column Imprints: A Secondary Index Structure". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–904. URL: http://dl.acm.org/citation.cfm?id=2463676.2465306.

Sleator, D. D. and R. E. Tarjan. (1985). "Self-Adjusting Binary Search Trees". *Journal of the ACM*. 32(3): 652–686. DOI: 10.1145/3828.3835.

Smith, A. J. (1978). "Sequentiality and Prefetching in Database Systems". *ACM Trans. Database Syst.* 3(3): 223–247. DOI: 10.1145/320263.320276.

Spectra. (2017). "Digital Data Storage Outlook". *Spectra White Paper*.

Stonebraker, M. (1981). "Operating System Support for Database Management". *Communications of the ACM*. 24(7): 412–418. DOI: 10.1145/358699.358703.

Stonebraker, M., 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. (2005). "C-Store: A Column-oriented DBMS". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 553–564. URL: http://dl.acm.org/citation.cfm?id=1083592.1083658.

Sweeney, A., D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. (1996). "Scalability in the XFS File System". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. URL: http://www.usenix.org/publications/library/proceedings/sd96/sweeney.html.

Tarjan, R. E. (1978). "Complexity of Combinatorial Algorithms". *SIAM Review*. 20(3): 457–491. DOI: 10.1137/1020067.

Teixeira, E. M., P. R. P. Amora, and J. C. Machado. (2018). "MetisIDX - From Adaptive to Predictive Data Indexing". In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 485–488. DOI: 10.5441/002/edbt.2018.53.

Thonangi, R., S. Babu, and J. Yang. (2012). "A Practical Concurrent Index for Solid-State Drives". In: *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*. 1332–1341. DOI: 10.1145/2396761.2398437.

Unterbrunner, P., G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. (2009). "Predictable Performance for Unpredictable Workloads". *Proceedings of the VLDB Endowment*. 2(1): 706–717. URL: http://dl.acm.org/citation.cfm?id=1687627.1687707.

Van Sandt, P., Y. Chronis, and J. M. Patel. (2019). "Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?" In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 36–53. DOI: 10.1145/3299869.3300075.

Varman, P. J. and R. M. Verma. (1997). "An Efficient Multiversion Access STructure". *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. 9(3): 391–409. DOI: 10.1109/69.599929.

Viglas, S. D. (2015). "Data Management in Non-Volatile Memory". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1707–1711. DOI: 10.1145/2723372.2731082.

Vitter, J. S. (2001). "External Memory Algorithms and Data Structures". *ACM Computing Surveys*. 33(2): 209–271. DOI: 10.1145/384192.384193.

Wasay, A., S. Chatterjee, and S. Idreos. (2021). "Deep Learning: Systems and Responsibility". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2867–2875. DOI: 10.1145/3448016.3457541.

Wasay, A., B. Hentschel, Y. Liao, S. Chen, and S. Idreos. (2020). "MotherNets: Rapid Deep Ensemble Learning". In: *Proceedings of Machine Learning and Systems (MLSys)*. URL: https://proceedings.mlsys.org/book/301.pdf.

Wasay, A., X. Wei, N. Dayan, and S. Idreos. (2017). "Data Canopy: Accelerating Exploratory Statistical Analysis". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 557–572. DOI: 10.1145/3035918.3064051.

Wei, Z., K. Yi, and Q. Zhang. (2009). "Dynamic external hashing: the limit of buffering". In: *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 253–259. DOI: 10.1145/1583991.1584055.

Weiner, P. (1973). "Linear Pattern Matching Algorithms". In: *Proceedings of the Annual Symposium on Switching and Automata Theory (SWAT)*. 1–11. DOI: 10.1109/SWAT.1973.13.

Willhalm, T., N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. (2009). "SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units". *Proceedings of the VLDB Endowment*. 2(1): 385–394. URL: http://www.vldb.org/pvldb/2/vldb09-327.pdf.

Wilson, P. R. (1991). "Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware". *SIGARCH Computer Architecture News*. 19(4): 6–13. DOI: 10.1145/122576.122577.

Wong, C. K. and M. C. Easton. (1980). "An Efficient Method for Weighted Sampling Without Replacement". *SIAM Journal on Computing (SICOMP)*. 9(1): 111–113. DOI: 10.1137/0209009.

Wu, K., E. J. Otoo, and A. Shoshani. (2006). "Optimizing Bitmap Indices with Efficient Compression". *ACM Transactions on Database Systems (TODS)*. 31(1): 1–38. DOI: 10.1145/1132863.1132864.

Wu, X., Y. Xu, Z. Shao, and S. Jiang. (2015). "LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 71–82. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/wu.

Wulf, W. A. and S. A. McKee. (1995). "Hitting the Memory Wall: Implications of the Obvious". *ACM SIGARCH Computer Architecture News*. 23(1): 20–24. DOI: 10.1145/216585.216588.

Yao, A. C.-C. and F. F. Yao. (1976). "The Complexity of Searching an Ordered Random Table (Extended Abstract)". In: *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 173–177. DOI: 10.1109/SFCS.1976.32.

Yi, K. (2009). "Dynamic indexability and lower bounds for dynamic one-dimensional range query indexes". In: *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*. 187–196. DOI: 10.1145/1559795.1559825.

Yi, K. (2012). "Dynamic Indexability and the Optimality of B-Trees". *Journal of the ACM*. 59(4): 21:1–21:19. DOI: 10.1145/2339123.2339129.

Yuan, Z., Y. Sun, and D. Shasha. (2023). "Forgetful Forests: Data Structures for Machine Learning on Streaming Data under Concept Drift". *Algorithms*. 16(6). DOI: 10.3390/a16060278.

Zhang, H., G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. (2015). "In-Memory Big Data Management and Processing: A Survey". *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. 27(7): 1920–1948. DOI: 10.1109/TKDE.2015.2427795.

Zhang, H., D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. (2016). "Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1567–1581. DOI: 10.1145/2882903.2915222.

Zhou, J. and K. A. Ross. (2004). "Buffering Database Operations for Enhanced Instruction Cache Performance". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 191–202. DOI: 10.1145/1007568.1007592.

Zoumpatianos, K. and T. Palpanas. (2018). "Data Series Management: Fulfilling the Need for Big Sequence Analytics". In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1677–1678. DOI: 10.1109/ICDE.2018.00211.

Zukowski, M. and P. A. Boncz. (2012). "Vectorwise: Beyond Column Stores". *IEEE Data Engineering Bulletin*. 35(1): 21–27. URL: http://sites.computer.org/debull/A12mar/vectorwise.pdf.

Zukowski, M., S. Héman, N. J. Nes, and P. A. Boncz. (2007). "Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 723–734. URL: http://dl.acm.org/citation.cfm?id=1325851.1325934.