part§§§chapter§§§section§§§

Foundations and Trends® in Databases
Vol. XX, No. XX (2020) 1–88

DOI: 10.1561/XXXXXXXXXX

now
the essence of knowledge

Principles, Tradeoffs, and Opportunities in Data Access Method Design

Manos Athanassoulis
Boston University
mathan@bu.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

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

Contents

Abstract

Access methods to support efficient search and modification are at the core of any data-driven system. Designing *access methods* has required a continuous effort to adapt to changing workload requirements and underlying hardware. In this article we outline the shared principles and *design dimensions* of access methods that facilitate efficient access to data residing at various levels of the storage hierarchy from durable storage (spinning disks, solid state disks, other non-volatile memories) to random access memory to caches (and registers). We point out how access method designs handle the size/speed tradeoffs of these different storage types for different kinds of workload mixes among reads, inserts, deletes, and updates. We *classify* both existing and as yet uninvented access methods based on a small set of design dimensions. Finally, we discuss the *design opportunities* that are enabled because of the systematization of the *access method design space*.

now Publishers Inc.. *Principles, Tradeoffs, and Opportunities in Data Access Method Design*. Foundations and Trends® in Databases, vol. XX, no. XX, pp. 1–88, 2020.

DOI: 10.1561/XXXXXXXXXX.

1

Introduction

1.1 Access Methods Basics

Access Methods are the means by which executing programs store and obtain data. As such, access methods sit at the heart of all data-intensive computer systems. An access method consists of (1) the data, physically stored in some layout, (2) optional metadata to facilitate navigation over the data, and (3) algorithms to support storage and retrieval operations [110, 221, 123]. Other terms used in the literature for access methods include "data structures" and "data containers". This article uses the term *access method* to underscore the interplay between the design of a data storage component with the way data is accessed.

Data systems, operating systems, file systems, compilers, and network systems employ a diverse set of access methods. Our discussion in this monograph draws examples primarily from the area of large volume data systems in the sense that we consider secondary memory, but the core analysis and design dimensions apply to purely in-memory systems as well.

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

value pairs, with the intended relationship that a given key maps to one value (though the same value can be associated with many keys). The value may have semantics ranging from a pointer or a reference in the base data collection, e.g., a row id in a relational system, a reference to a large object such as an image or video in a key-value store, a record in a tabular database, or an arbitrary set of values that the application knows how to parse and use in a NoSQL system.

Access methods have enormous general utility. An access method design can, for example, be used to describe the design of (i) metadata indexes used in file, network, and operating systems, (ii) base data layouts and indexes in relational systems [110], and (iii) NoSQL and NewSQL data layout designs [181].

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

1.2 Tradeoffs in Access Method Designs

There is no perfect access method [124]. Every access method represents a particular workload-dependent performance tradeoff. For example, the more structure a data set has (either in the form of metadata or within the main data), the easier it is to search it, but the harder it is to insert or update because the structure needs to be maintained (by re-organizing data or by updating metadata).

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

As applications and hardware evolve, they require the invention of new data structures. For example, secondary storage considerations inspirted the generalization of 2-3 trees to B-trees [34] and then the B^+-Trees [66, 93, 110] with its many variants. During the four decades of relational data system [18, 65] evolution, access methods have been developed to keep pace with new applications. The same is true for NoSQL systems which heavily rely on specialized classes of data structures such as LSM-trees for data systems that support write-heavy applications [166, 189], B-trees for systems that target more read-optimized applications [186, 243], and hashing based systems that are the core of systems that support write-heavy applications with point lookups only [56].

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

Workload-Driven Designs. As an example from data systems, the rise of analytical applications as of the early 2000s in which relational tables tended to be wide (i.e., have many columns) but for which only a few columns would be accessed favored a columnar layout [67] (as had been present in vector languages like APL [78] for some time). In the research community, this led to the development of a new column-oriented data system architecture, also called column-stores [1, 47, 80, 81, 86, 87, 117, 132, 144, 231, 261]. The column-store design works better for long analytical queries on few columns, while the row-store design works well for short selective queries that require most fields of each row. Thus, using a columnar instead of a row-oriented layout in data management systems is an *access method* decision [31, 32, 141, 146, 147, 148, 149, 202, 203, 204]. Further, various hybrid approaches offer benefits from the worlds of row-stores and column-stores: (i) by

nesting columnar data organization within data pages [5, 6], and (ii) by grouping multiple columns and offering specialized code for accessing groups of columns [10, 76, 80, 81, 101, 132].

Other examples of workload driven designs come from the rich ecosystem of new non-relational data management systems, typically categorized as NoSQL or newSQL [181]. Such systems often employ a log-structured merge tree (LSM-Tree) [166, 189] design that amortizes the update cost by storing incoming data in immutable sorted files. For example, a different memory allocation strategy allows for a better read vs. update performance tradeoff [70, 71]. Lazy and eager merging of sorted runs can optimize for write-intensive or read-intensive workloads respectively [140, 166]. Another approach, called lazy leveling, performs lazy merging throughout the tree, except at the last level and uses additional memory to balance the read costs when needed [72] as well as creating levels of variable size ratios to "delay" writes [73]. Using hashing instead of sorting can efficiently support workloads with point accesses and no range queries [13, 30, 56, 74, 75, 224]. These are only a few examples of access method designs that are strongly tied to either expected or temporary workload patterns.

In addition, access methods have been invented that autonomously adapt to the workload. For example, Database Cracking [118, 119, 120] utilizes the access patterns in incoming queries to continuously and incrementally physically reorganize the core access methods such as both the base data and the index metadata can perform well for the exact workload observed. Database Cracking has been developed in the context of analytical column-store systems but the idea have been extended to traditional B-tree design as well [97, 98, 122] Similarly, E-Tree [167] adapts between read-optimized B-tree nodes and write-optimized B-tree nodes based on the read/write pattern.

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

at low price but at high access latency, and as we move higher, that is, closer to the processor, the storage is faster but smaller and more expensive per byte. In the storage/memory hierarchy there is always a level that is the bottleneck for a given application, which depends on the size of the application data relative to the sizes of the storage at the different levels of the hierarchy.

For example, in data management systems early access methods like B^+-Trees were optimized for disk accesses [93]. As the memory sizes grew, however, the bottleneck quickly moved higher to the main memory and non-volatile memory. This changed the tradeoffs dramatically. In particular, a key hardware trend has been the growing disparity between processor speed and the speed of off-chip memory, termed "the memory wall" [250]. Since the early 2000s, operating systems [178] and data management systems [128] have been carefully re-designed to account for the memory wall by optimizing for the increasing number of cache memories [5, 7, 46, 47, 62, 143, 171, 172, 173, 210, 257].

Additionally, secondary storage is already at a crossover point. Traditional hard disks have hit their physical limits [19], and storage technologies like shingled disks and flash are now addressing this performance stagnation [113]. Shingled disks increase the density of storage on the magnetic medium, changing the nature of disks because the granularity of reads and writes is now different [113]. Flash-based drives offer significantly faster read performance than traditional disks, but may suffer from relatively poor write performance. Further, flash-based drives are equipped with a complex firmware called the Flash Translation Layer (FTL) which, when updated, can lead to drastic changes in performance. Thus, flash hardware performance changes both when hardware changes and when the firmware is updated. Such changes may create a need for new access methods to optimize for the new hardware/firmware [20, 22, 23, 74, 126, 131, 137, 155, 157, 183, 184, 209, 234]. For example, because writing certain kinds of flash devices requires writing full blocks at a time, access methods append incoming data without sorting until a flash erase block is full, at which point data is re-organized in memory and then re-written as one block. This minimizes write overhead, sacrificing read performance (contents of a

block are frequently not sorted) for write efficiency [4, 21, 25, 26, 64].

1.3 Contributions: Design Space and Classification

While each access method design depends on hardware and workloads as described above, at the same time, application workloads and hardware evolve continuously. In addition, the various tradeoffs that appear in access method design and the way they are affected by the different design choices create a notoriously complex problem space [123]. Understanding this space requires a systematic classification of concepts and techniques used to design access methods. This classification, in turn, will enable us to answer frequent questions that come up when building complex data-driven systems. Which access method efficiently supports a given workload? What is the effect of adding a new design component to an access method and workload? Is it beneficial to get more or faster memory for a system? These are only a small sample of the vast set of design, research, and practical questions that researchers and practitioners face on a daily basis.

To help answer such questions we provide a classification in this survey by studying how the low level fundamental design choices in access methods lead the final designs to occupy a particular balance in the overall tradeoff space.

Design Dimensions. After studying numerous access method designs that cover a substantial part of the state of the art, we have distilled them to a small number of *independent design dimensions* that identify a design.

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

Design Space. We show how this set of design dimensions can be used to describe any access method and to explain its behavior and properties. First, we use the design dimensions to propose an *access*

method design space. Each access method is a "point" in this space and designs that correspond to *families of access methods* cover a sub-space of this design space.

We (i) describe the motivation behind each design dimension and its available options, (ii) characterize existing access methods as points in this space, and (iii) study the impact of combining various design elements to create new access methods. iv) provide a practical access method design guide.

This survey builds on our research efforts in classifying access methods [27, 123] and building new detailed abstractions [28, 116, 124]. The design space of data structures can be seen at many granularities depending on how fine grained the design choices are. Different granulaties of design help with different problems and classification/understanding. The Data Calculator work creates as fine-grained a design space as possible by identifying the first principles of design [123] which helps with in-memory data structure design where even the smallest design choices affect performance. In this survey, we utilize the prior design principles to summarize classes of design choices and characterize their impact in terms of disk performance for big data systems, leading to a practical guide for designers.

The remainder of this paper is organized as follows. Chapter 2 introduces the performance tradeoffs with respect to access patterns which we use to rank designs. Then, Chapter 3 presents the design dimensions in detail and incrementally builds a universal I/O model that helps characterize the impact of arbitrary design choices. Then, in Chapter 4 we give examples of how state of the art access method designs utilize the design principles and how they affect the performance tradeoffs (Table 4.1). In Chapter 5 we discuss open research questions with respect to both the design space and the individual designs themselves. Finally, Chapter 6 discusses related work and Chapter 7 concludes.

2

Performance Tradeoffs & Access Patterns

We now review the memory/storage hierarchy and its importance for access method design. We also define in detail the core performance tradeoffs we use to characterize design principles throughout the paper.

Memory Hierarchy. Access methods live across the memory hierarchy. That is, a sequence of memory devices that complement each other in terms of how fast we can read and write data from each one. At least one level of the memory hierarchy is typically persistent so the data is not lost when power is lost. Also one level of the memory hierarchy is large enough to hold all the data needed on a particular machine. Data is transferred back and forth across levels as requests for data arrive or new data is inserted and old data is deleted or updated.

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

pacity because it is waiting for data from disk or even random access memory.

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

Overall, this performance metric holds true for the majority of operations in the key-value-model and typical analytics. For example, this is true for file systems, network routers, NoSQL systems and SQL database systems. When designing access methods purely for an in-memory environment or for an application with very high computational costs, computational costs have to be taken into account. This is an emerging field of study that requires learned models combined with traditional cost models [124]. In addition, as we move to non-volatile memories with wider buses and lack of mechanical constraints in secondary storage, the cell probe model [252] might be necessary.

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

Read vs. Update Captures Workload. In order to compare access methods and decide which one to use under particular conditions, we first need to define the appropriate metrics. The most common metrics quantify the tradeoff of each access method design between read performance and update performance [49, 253, 254]. The first one defines how fast we can retrieve data while the latter describes how fast we can insert new data or change existing data. In this way, these metrics collectively describe the performance that an access method design provides for a given workload. Such metrics can be measured in terms of actual (expected) response time or more typically in terms of the

amount of data we need to move to complete an operation.

Read vs. Update vs. Memory Captures Storage. The common denominator of the lines of work that consider read and update performance as metrics is that they assume that disk capacity is cheap or even free in the ideal setting. This assumption comes from the time that disk was used as secondary storage and was so much cheaper than memory that the storage cost was considered insignificant and the main consideration was storage performance [100]. Since then, the storage hierarchy has been augmented with various devices including solid-state disks (SSDs), shingled magnetic recording disks (SMR), nonvolatiles memories (NVMs) and other devices. The new storage media can be either expensive per byte and fast, or cheap but slow [19]. Sometimes the higher performance comes at significant energy cost [225]. In addition, the data generation trends typically outpace the rate at which storage devices are delivered leading to a data-capacity storage gap [43, 112, 229].

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 access methods [28, 77, 255]. The wildly different cost of secondary storage among the different storage technologies makes space utilization and storage cost critical factors of access method design.

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

2.2 RUM Performance Tradeoffs

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

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