# Design Tradeoffs of Data Access Methods

Manos Athanassoulis
*manos@seas.harvard.edu*
Harvard University

Stratos Idreos
*stratos@seas.harvard.edu*
Harvard University

## ABSTRACT

Database researchers and practitioners have been building methods to store, access, and update data for more than five decades. Designing *access methods* has been a constant effort to adapt to the ever changing underlying hardware and workload requirements. The recent explosion in data system designs – including, in addition to traditional SQL systems, NoSQL, NewSQL, and other relational and non-relational systems – makes understanding the tradeoffs of designing access methods more important than ever. Access methods are at the core of any new data system. In this tutorial we survey recent developments in access method design and we place them in the design space where each approach focuses primarily on one or a subset of read performance, update performance, and memory utilization. We discuss how to utilize designs and lessons-learned from past research. In addition, we discuss new ideas on how to build access methods that have tunable behavior, as well as, what is the scenery of open research problems.

## 1. INTRODUCTION

**Access Methods and Data Systems.** A key aspect of all data management systems is the *access methods* they employ. An access method is a collection of *algorithms and data structures for organizing and accessing data* [36]. Finding the proper physical design (through static autotuning [19], online tuning [13], or adaptively [38]) has been a key data management research challenge for several decades. The way we physically organize data on storage devices (disk, flash, memory, caches) defines and restricts the possible ways that we can read and update it. For example, a scan access method in a modern main-memory read optimized column-store consists of an array of dense values and a cache conscious for-loop access pattern [1]. Traditional data management systems access the base data through their scan and index access methods, however, applications, use cases, and the system design itself are becoming more complex. As a result, an increasing number of new access method designs and variations of existing have been proposed.

The recent explosion of applications has led to the development of relational and non-relational data systems typically categorized under the SQL, NoSQL, and newSQL terminology [59]. Regardless how a system is categorized, its core functionality is storing and accessing data. Thus, designing the right access methods still remains a key challenge. Given the broad spectrum of workloads and applications today, it is more important than ever to both learn from past work and to keep inventing new solutions. Helping with this task is exactly the goal of this tutorial.

**Design Tradeoffs.** A close look at existing proposals on access methods reveals that each is confronted with the same fundamental challenges and design decisions again and again. In particular, there are three quantities and design parameters that researchers always try to minimize: (i) the read overhead (**R**), (ii) the update overhead (**U**), and (iii) the memory (or storage) overhead (**M**), henceforth called the *RUM overheads* [6]. Deciding which overhead(s) to optimize for and to what extent, remains a prominent part of the process of designing a new access method, especially as hardware and workloads change over time. For example, in the 1970s one of the critical aspects of every database algorithm was to minimize the number of random accesses on disk; fast-forward 40 years and a similar strategy is still used, only now we minimize the number of random accesses to main memory. Today, different hardware runs different applications but the concepts and design choices remain the same. New challenges, however, arise from the exponential growth in the amount of data generated and processed, and the wealth of emerging data-driven applications, both stressing existing access methods. The three RUM overheads form a design space with a three-way balance: optimizing for any two overheads negatively impacts the third [6].

Similar tradeoffs are typically observed for access methods designed for domain-specific use cases. We identify three broad classes: (i) high-dimensional data, (ii) time-series data, and (iii) graph data. Typically the relevant workloads are not transactional in the same way that relational and key-value data are required to be.

**Tutorial Outline and Goals.** This tutorial gives a comprehensive introduction to the design tradeoffs of access methods, discussing the basic methodologies used to optimize for each. Specifically it includes the following sections:

**1. Introduction:** We start with an introduction of access method design and how this affects data systems design and broadly data management research. We give specific



*SIGMOD '16, June 26–July 01, 2016, San Francisco, CA, USA*

DOI: http://dx.doi.org/10.1145/2882903.2912569

examples both from research and industry (including both startups and established companies).

**2. Design Tradeoffs:** We dissect the key tradeoffs of access method design: read performance, update performance, and memory. We present the intuition behind the fact that we cannot optimize, at the same time, for all three goals. Orthogonally to whether the access methods are designed for reads, updates, or memory utilization, we highlight that there are common high-level concepts across all three groups: (a) logarithmic design, (b) continuous reorganization, (c) space-efficient design.

**3. Logarithmic Design:** We start by presenting a common theme among several access methods: their logarithmic design. We cover how such access methods are designed to be optimized for reads [8, 28, 30, 47, 64, 67], for updates [17, 45, 49, 65, 72, 85], and for memory [9, 21, 48, 50, 57, 69].

**4. Continuous Reorganization:** We continue with another common theme, workload-driven continuous reorganization which can be used either when optimizing for read performance [30, 31, 32, 33, 35, 38, 39, 40, 41, 71] or, with differential updates, for update performance [2, 4, 5, 29, 37, 42, 73, 75].

**5. Space-Efficient Designs:** We conclude the discussion for the design tradeoffs presenting how space-efficient access methods are used to offer good read performance when the workload permits [11, 15, 16, 22, 63, 66, 82, 83], and how we can trade memory utilization for performance with approximate indexing and data skipping techniques [3, 27, 52, 58, 77, 86].

**6. Other Access Method Classes:** We dedicate a part of this tutorial to three additional access method classes. In particular, we present the core principles of access method designs for (i) high-dimensional data, (ii) time-series data, and (iii) graph data. We conclude this part by highlighting the RUM tradeoffs in the key solutions for these data types.

**7. Open Research Challenges:** Finally, we discuss open problems in access method design as they manifest in modern data systems, including ideas about declarative and shape-shifting access methods.

# 2. DESIGNING ACCESS METHODS

In order to understand the design space, we introduce the audience to the different design elements used in access methods. Typically, each design element offers specific behavior (e.g., read-friendly), however in multiple cases it can be multi-faceted and offer different optimizations. Table 1 shows the classification of the papers we discuss. We discuss three main design elements of access methods, logarithmic design, continuous reorganization, and space efficiency, and we see how each of them affects the read, the update, and the memory overhead. The key point of this presentation is the clustering of the presented access methods in terms of techniques, in order to argue about the fundamental tradeoffs. That way we can learn how to navigate this access method design "space" of read-optimized, update-optimized and memory-optimized designs.

## 2.1 Logarithmic Design

We first introduce the generic concept of logarithmic design used by several data structures, like B-Trees, Tries and variations of them. The key idea is to organize metadata in a way that the search cost of a specific element is logarithmic to the size of the dataset. We differentiate between access methods with logarithmic design optimized for reads and for updates.

**Optimizing Reads.** Tree-based access methods are typically optimized for read performance. Logarithmic-time structures such as B-Trees [30], and Tries [28, 47, 61] offer fast read access but increase space overhead. Tries, in particular, typically suffer from space inefficiency. Adaptive Radix Tree [47] overcomes the shortcoming of prior tries by offering update-friendly, and space-efficient trie indexing. The key insight is that instead of following a common node design throughout the tree, each node is locally adapted to what is needed based on the density of the domain for the corresponding values. Such data structures are the state-of-the-art used by widely adopted commercial or open-source data management systems. SB-Tree [64] is designed to support high-performance sequential disk access for long range retrievals. It assumes an array of disks from which it retrieves data or intermediate nodes should they not be in memory, by employing multi-page reads during sequential access to any node level below the root. The bounded disorder access method is introduced as a generic framework for access methods [52, 53]. The method aims at increasing the ratio of file size to index size (i.e., to decrease the size of the index) by building a shallow tree to create range partitioning in wide ranges and using hashing for distributing the keys in a multi-bucket node, corresponding to each range. The bounded disorder access method does not decrease the index size too aggressively for good reason; since, by doing that it would cause more random accesses on the storage medium, which is assumed to be traditional hard disks. On the contrary, the goal is to guarantee that point queries would get read overhead similar to the what a hash index offers, and range queries would get read overhead similar to what a $B^+$-Tree index offers.

**Optimizing Updates.** We then continue to discuss how logarithmic access methods can be used to efficiently treat updates. Logarithmic access methods for updates typically follow a hierarchical log-structured approach. When the top level receives enough updates it is merged with lower levels and these updates gradually propagate in the tree. A prime example which has been used as inspiration for multiple follow-up approaches is the Log-Structure Merge Tree (LSM-Tree) [65]. An LSM-Tree is comprised of a number of exponentially growing levels; each level is $k$ times larger than the previous. Incoming data are buffered and when the buffer is filled they are sorted and written on the first level of the tree. When $k$ such buffers accumulate, the sorted buffers are merged and written out in the next level, a process that keeps repeating. That way, the number of physical writes per insert is kept low and the typically efficient sequential write on the device can be exploited. More recently, a number of access methods use the logarithmic method to offer a tradeoff between reads and updates [9, 37, 49, 72, 75, 85]. In addition to LSM [65], the Partitioned B-tree (PBT) [29], and the Positional Delta Tree (PDT) [37] offer good performance under updates but increase the read costs and the space overhead. Fractal-Tree and FD-Tree [9, 49] use fractional cascading [20] to restrict the number of reads to one per level. LSM-Trie [85] merges less greedily by using $T$ substructures in each level. That way it reduces write-amplification by a factor of $T$, at the expense of losing the ability to do range-queries. BigTable uses LSM-Trees

| | *Logarithmic Design* | *Continuous Reorganization* | *Space Efficient* |
|---|---|---|---|
| *Read* | Trees/Tries/Skiplists [8, 28, 30, 47, 64, 67] | Cracking/Adaptive Indexing [30, 31, 32, 33, 35, 38, 39, 40, 41, 71] | Bitmaps/Hash Index [11, 15, 16, 22, 63, 66, 82, 83] |
| *Update* | Log-Structured Trees [17, 45, 49, 65, 72, 85] | Differential Updates [2, 4, 5, 29, 37, 42, 73, 75] | |
| *Memory* | Cache Optimizations [9, 21, 48, 50, 57, 69] | | Approximate/Data Skipping [3, 27, 52, 58, 77, 86] |

**Table 1: Classification of access method designs. Each row corresponds to the main optimization goal and each column corresponds to the underlying design element. Some approaches may fit in multiple cells; we discuss this in detail in the tutorial content but for simplicity we list them in the cell that fits the motivation of each design.**

with a fixed size ratio equal to ten between adjacent levels [17]. Bigtable also uses Bloom filters to avoid reading unnecessary levels. These Bloom filters are tuned statically and uniformly across levels, a decision which can be further optimized with careful tuning. In addition, Cassandra [46], LOCS [81], VT-Tree [75], and bLSM [72] build on top of BigTable, LevelDB, and LSM-Trees to provide logarithmic access method design. LOCS [81] is a version of LevelDB that utilizes SSD parallelism. VT-Tree [75] avoids repeatidly writing sorted data by stitching non-overlapping runs. bLSM [72] restricts the number of levels, restricting read and write performance irrespectively of the workload.

**Optimizing for Memory.** Several approaches leverage knowledge about the memory hierarchy. Fractal Prefetching $B^+$-Trees [21] use different node sizes for disk-based and in-memory processing in order to have the optimal for both cases. Cache-sensitive $B^+$-Trees [69] physically cluster sibling nodes together to reduce the number of cache misses, and decrease the node size using offsets rather than pointers. SB-Trees [64] operate in an analogous way when the index is disk-based, while BW-Tree [48] and Masstree [57] present a number of optimizations related to cache memory, main memory and flash-based secondary storage. SILT [50] combines write-optimized logging, read-optimized immutable hashing, and, a sorted store, carefully designed to balance the tradeoffs of the various memory levels.

## 2.2 Continuous Reorganization

In the next part we present the notion of continuous reorganization, which we see in many workload-driven approaches. This approach is taken by query-driven adaptive indexing and by differential updates. While the goal of the adaptive indexing is classically optimizing for read performance, the latter aims at offering efficient updates.

**Optimizing Reads.** Adaptive indexes are flexible data structures designed to actively balance the tradeoffs. Most existing data structures provide tunable parameters that can be used to balance the tradeoffs offline, however, adaptive data structures balance the tradeoffs online across a larger area of the design space.

Notable proposals are Database Cracking [38], which reorganizes the data in memory to match how queries access data, and adaptive indexing [31, 33] that follows the same principle as database cracking by focusing the index optimization on key ranges used in actual queries. The hybrid, Adaptive Merging [41] balances the read performance versus the overhead of creating an index. Although much more flexible than traditional data structures, existing adaptive data structures cannot cover the whole tradeoff spectrum as they are typically designed for a particular type of hardware and application.

**Optimizing Updates.** Differential files is the first access method that proposed to store only differential updates instead of in-place updates [73]. The fundamental idea is to consolidate updates and apply them in bulk.

Differential updates have been further studied in the context of updating analytical datasets with the stepped-merge algorithm [42] and the Materialized Sort-Merge (MaSM) [4, 5] algorithm. The stepped-merge algorithm stores updates lazily by maintaining updates in memory in sorted runs, and eventually forming a $B^+$-Tree of these updates using an external merge-sort. MaSM operates at a similar level, where updates are kept in sorted runs on flash devices and merged only once more before being migrated to the main data, forming large immutable sorted runs. The stepped-merge approach aims at minimizing random I/O requests. On the other hand, the MaSM algorithms focus on minimizing memory consumption and unnecessary writes on flash at the expense of more, yet efficient, random read I/O requests.

## 2.3 Space-Efficient Designs

The third category to discuss is the space-efficient design elements. We distinguish between designs that (i) have small size but they support fewer types of queries, (ii) have small size because they exploit underlying data organization, and (iii) offer approximate indexing.

**Optimizing Reads.** Hash Indexing is a typical example of an access method that has small size because it trades functionality (i.e., trade how large is the variety of types of queries it efficiently supports, for size). Next, we discuss bitmap indexing. It is typically used for large data sets. It leverages fast bitwise operations [15, 16, 63, 66, 82], and, today, is commonly used for a number of applications ranging from scientific data management [83] to analytics and data warehousing [18, 55, 70, 78, 79, 84]. Bitmap indexes are utilized by several popular database systems, including open-source systems like PostgreSQL and commercial systems like Oracle [74], SybaseIQ [55, 66], and DB2 [14]. While bitmap indexes typically are designed for read performance, UpBit offers efficient updates at the expense of additional metadata and a small penalty in read performance [7].

**Optimizing for Memory.** We discuss two main categories for access methods that optimize for memory size by exploiting existing structure of data (due to data generation or data preparation): data skipping and approximate indexing.

Existing structure in data can be utilized by light-weight access methods that can skip touching non-qualifying data via simple checks. At its core, data skipping is a form of scan enhancement. Zonemaps [27, 68, 80, 86] and Column Imprints [77] are prime examples where we maintain metadata for zones of a column, such as min/max information, and a scan uses that metadata to decide whether to scan a

zone or not. Such approaches work quite well when data is clustered or fully sorted.

Approximate indexing capitalizes on existing structure of the data in a similar way, but it can be equally efficiently used both for point reads, and for range queries. BF-Tree [3] implements approximate indexing using probabilistic data structures like Bloom filters [11] in order to be able to tune accuracy of the index as a function of the size of the index structure. In addition to Bloom filters, approximate indexing can use other probabilistic data structures like quotient filters [10] and cuckoo filters [26].

# 3. OTHER ACCESS METHOD CLASSES

In this tutorial we primarily focus on the design elements and tradeoffs of access methods in one-dimensional, key-value or relational data. However, there is a wealth of use cases that face similar tradeoffs and design decisions and require different, typically specialized solutions. We present a representative subset.

First, spatial and other high-dimensional data require access methods that can navigate efficiently a high-dimensional space addressing the "curse of dimensionality" in the context of access methods [56]. While this dimension is not present for one-dimensional data, most of the other observations and design decisions are shared. A second class is time-series data. While time-series can be represented and stored by relational systems, typically specialized time and data series systems offer better performance by taking into account the fundamental characteristics of time series: there is always a sorted dimension (time or simply measurement id), and data are appended either in the form of new – more recent – data items, or new data series. A third class that we identify is access methods for graph based data. We see similar tradeoffs between read, update, and memory performance for all three classes, having however, a varying degree of similarity to the design elements we outline in this tutorial.

## 3.1 High-Dimensional Access Methods

Spatial and other high-dimensional data (two or more dimensions) have long been attracting specialized solutions. A key insight which has been used as the baseline to build upon is R-Trees [34] which use a hierarchy of minimum bounding rectangles (MBR). An R-Tree shares the main properties of a $B^+$-Tree: it is a balanced search tree and it is designed for storage on disk. As the number of dimensions increases the efficiency of an R-Tree is dramatically reduced facing the "curse of dimensionality". Similarly to the case of access methods for uni-dimensional data, access methods for high-dimension data face the RUM tradeoffs. Typically, we cannot quantify the read cost with accuracy and we rely on experimental estimations [12]. Due to the aforementioned "curse of dimensionality", however, it often makes sense to break the space into small MBRs in order to have less overlap, leading to more metadata needed for the indexing structure, facing one of the RUM tradeoffs. On the other hand, in this case updating the index is more expensive. A high-dimensional update-friendly access method typically allows for higher overlap between MBRs, making, as a result, reads more expensive, facing yet again, a RUM tradeoff. After the initial proposal of R-Trees, a number of follow up approaches have been proposed, offering domain specific optimizations and a different designs in terms of the balance of RUM tradeoffs, surveyed in detail [12, 56, 60, 62].

## 3.2 Time-Series Access Methods

The key insight for efficient time series data processing is the representation of the series through segmented means [44, 87], which allowed for space-efficient representation at the expense of accuracy. Using this representation allowed Lin et al. [51] to introduce the Symbolic Aggregate approXimation (SAX), which in turn was later extended to an indexable representation called iSAX [76]. The main challenge iSAX faces is that of building time. In order to address the long index building time ADS follows the approach of query-driven *continuous reorganization* to offer fast build time with small memory footprint at the expense of gradually decreasing query latency [88], an approach that manages to offer overall better workload latency as it cleverly takes into account past queries to add structure gradually. The updates in time-series workloads are significantly different than in relational systems. Typically, the aforementioned approaches treat updates that are in effect new time-series in a large collection of time series leaving old data unaffected and requiring append-only style of updates. Irrespectively of the update scheme, both iSAX and ADS add indexing metadata in order to facilitate read queries making the update more expensive; in essence optimizing for read performance at the expense of both index size and update overhead.

## 3.3 Graph Access Methods

In graph storage systems we do not typically have a clear distinction between the access methods and the system. The reason is twofold; first, the graph representation plays a big role in performance and each representation may require a very different way to access the data; second, there has been a much smaller effort in standardization when compared with relational systems, hence allowing more native solutions which do not adhere to a global standard. However, the increased interest in graph-based data analysis has lead to the development of a number of approaches that face similar tradeoffs than the ones described in this tutorial.

Similar to the time-series data, graph data are heavily impacted by the data representation. A space efficient graph representation is the compressed spared row (CSR) representation [25] which typically offers immutable data, a key difference from relational data. LLAMA, a recently proposed graph analytics tool, however, allows operating on multi-version CSR data, in essence allowing operating on graphs in the presence of incremental data ingest and mutation [54]. LLAMA in essence exchanges additional storage needs (multiple versions) for efficient support of updates balancing off the RUM tradeoffs in a similar way that RUM Conjecture anticipates them to balance for any access method.

**Summary.** In this section, we briefly summarize the core design decisions for each access method class, demonstrating that similar tradeoffs as with one-dimensional data do exist. However, covering extensively the design choices and the tradeoffs of access methods of these three classes requires a separate detailed study.

# 4. OPEN PROBLEMS

The final part of the tutorial presents open problems. A recurring theme of this tutorial is that by studying and classifying all the access method designs, we identify different strategies to achieve a specific optimization goal, e.g., optimize for read performance. In fact there are two fun-

damentally different ways to reach a static point in the read/update/memory design space: either by using a static design, or by using an adaptive design which eventually leads to the desired design point. A number of new research directions build upon this concept. For example, *how can we build access methods that support a dynamic optimization goal, which is reached either by a static or an adaptive to the workload design [6, 43]?* In addition, *can we build an "access method design optimizer" to help us choose the right access method given specific hardware and workload properties [6]?* Similar directions are taken at the system level. For example, *can we build a declarative storage engine that does not have to chose between being either row-oriented or column-oriented [23, 24]?* In the final part of the tutorial, we discuss these open problems and we highlight opportunities for innovation.

# 5. BIOGRAPHIES

**Manos Athanassoulis** is a postdoctoral researcher fellow at DASlab, the Data Systems Laboratory, at Harvard SEAS. Manos works on designing data systems and access methods for dynamic environments that workload and hardware constantly change. Manos received his undergraduate and Masters' degree from the University of Athens, Greece, and his Ph.D. from Ecole Polytechnique Fédérale de Lausanne, Switzerland. In 2011 he interned at IBM Research, New York. Manos is the recipient of a Postdoc Mobility Fellowship from the Swiss National Science Foundation, a "Best of VLDB" selection for 2010, and an IBM Ph.D. Fellowship.

**Stratos Idreos** is an assistant professor of Computer Science at Harvard University where he leads DASlab, the Data Systems Laboratory, at Harvard SEAS. Stratos works on data systems architectures with emphasis on designing systems for big data exploration. For his doctoral work on Database Cracking, Stratos won the 2011 ACM SIGMOD Jim Gray Doctoral Dissertation award and the 2011 ERCIM Cor Baayen award as "most promising European young researcher in computer science and applied mathematics" from the European Research Council on Informatics and Mathematics. In 2010 he was awarded the IBM zEnterpise System Recognition Award by IBM Research, and in 2011 he won the VLDB Challenges and Visions best paper award. In 2015 he received an NSF CAREER award and the 2015 IEEE TCDE Early Career Award from the IEEE Technical Committee on Data Engineering.

**Acknowledgments.** We thank the reviewers and the members of DASlab for their valuable feedback. This work is supported by SNSF Grant No. P2ELP2_158936 and by NSF Grant No. IIS-1452595.

# 6. REFERENCES

[1] D. J. Abadi et al. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases*, 5(3):197–280, 2013.

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

[3] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. *PVLDB*, 7(14):1881–1892, 2014.

[4] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *SIGMOD*, 2011.

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

[6] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *EDBT*, 2016.

[7] M. Athanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *SIGMOD*, 2016.

[8] R. Bayer and K. Unterauer. Prefix B-trees. *TODS*, 2(1):11–26, 1977.

[9] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-Oblivious Streaming B-trees. In *SPAA*, 2007.

[10] M. A. Bender et al. Don't Thrash: How to Cache Your Hash on Flash. *PVLDB*, 5(11):1627–1637, 2012.

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

[12] C. Böhm, S. Berchtold, amd D. A. Keim. Searching in High-dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *Comp. Surv.*, 33(3):322–373, 2001.

[13] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, 2007.

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

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

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

[17] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

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

[19] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.

[20] B. Chazelle and L. Guibas. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica*, 1(2):133–162, 1986.

[21] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-Trees. In *SIGMOD*, 2002.

[22] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[23] P. Cudré-Mauroux, E. Wu, and S. Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.

[24] J. Dittrich and A. Jindal. Towards a One Size Fits All Database Architecture. In *CIDR*, 2011.

[25] J. Dongarra, P. Koev, X. Li, J. Demmel, and H. van der Vorst. 10. Common Issues. In *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, page 22. 2000.

[26] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *CoNEXT*, 2014.

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

[28] E. Fredkin. Trie memory. *CACM*, 3(9):490–499, 1960.

[29] G. Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR*, 2003.

[30] G. Graefe. Modern B-Tree Techniques. *Found. Trends Databases*, 3(4):203–402, 2011.

[31] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.

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

[33] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[34] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

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

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

[37] S. Héman, M. Zukowski, and N. J. Nes. Positional update handling in column stores. In *SIGMOD*, 2010.

[38] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[39] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[40] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

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

[42] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *VLDB*, 1997.

[43] O. Kennedy and L. Ziarek. Just-In-Time Data Structures. In *CIDR*, 2015.

[44] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *KAIS*, 3(3):263–286, 2001.

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

[46] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. *SIGOPS Op. Sys. Rev.*, 44(2):35–40, 2010.

[47] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, 2013.

[48] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*, 2013.

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

[50] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[51] J. Lin, E. J. Keogh, S. Lonardi, and B. Y.-c. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *DMKD*, 2003.

[52] W. Litwin and D. B. Lomet. The Bounded Disorder Access Method. In *ICDE*, 1986.

[53] D. B. Lomet. A simple bounded disorder file organization with good performance. *TODS*, 13(4):525–551, 1988.

[54] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *ICDE*, 2015.

[55] R. MacNicol and B. French. Sybase IQ Multiplex - Designed For Analytics. In *VLDB*, 2004.

[56] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Springer, 2006.

[57] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[58] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*, 1998.

[59] C. Mohan. Tutorial: An In-Depth Look at Modern Database Systems. In *EDBT*, 2014.

[60] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE DEBULL*, 26(2):40–49, 2003.

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

[62] L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). *IEEE DEBULL*, 33(2):46–55, 2010.

[63] P. E. O'Neil. Model 204 Architecture and Performance. In *HPTS*, 1987.

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

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

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

[67] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *CACM*, 33(6):668–676, 1990.

[68] V. Raman et al. DB2 with BLU acceleration: so much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[69] J. Rao and K. A. Ross. Making B+- trees cache conscious in main memory. In *SIGMOD*, 2000.

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

[71] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.

[72] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *SIGMOD*, 2012.

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

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

[75] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-trees. In *FAST*, 2013.

[76] J. Shieh and E. J. Keogh. \emph{i}SAX: indexing and mining terabyte sized time series. In *SIGKDD*, 2008.

[77] L. Sidirourgos and M. L. Kersten. Column Imprints: A Secondary Index Structure. In *SIGMOD*, 2013.

[78] K. Stockinger. Bitmap Indices for Speeding Up High-Dimensional Data Analysis. In *DEXA*, 2002.

[79] M. Stonebraker et al. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[80] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained Partitioning for Aggressive Data Skipping. In *SIGMOD*, 2014.

[81] P. Wang et al. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *EuroSys*, 2014.

[82] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit Transposed Files. In *VLDB*, 1985.

[83] K. Wu et al. FastBit: interactively searching massive data. *J. of Physics: Conference Series*, 180(1):012053, 2009.

[84] M.-C. Wu and A. P. Buchmann. Encoded Bitmap Indexing for Data Warehouses. In *ICDE*, 1998.

[85] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *ATC*, 2015.

[86] R. S. Xin et al. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.

[87] B.-K. Yi and C. Faloutsos. Fast Time Sequence Indexing for Arbitrary Lp Norms. In *VLDB*, 2000.

[88] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *SIGMOD*, 2014.