Key-Value Storage Engines

Stratos Idreos
Harvard University

Mark Callaghan
MongoDB

ABSTRACT

Key-value stores are everywhere. They power a diverse set of data-driven applications across both industry and science. Key-value stores are used as stand-alone NoSQL systems but they are also used as a part of more complex pipelines and systems such as machine learning and relational systems. In this tutorial, we survey the state-of-the-art approaches on how the core storage engine of a key-value store system is designed. We focus on several critical components of the engine, starting with the core data structures to lay out data across the memory hierarchy. We also discuss design issues related to caching, timestamps, concurrency control, updates, shifting workloads, as well as mixed workloads with both analytical and transactional characteristics. We cover designs that are read-optimized, write-optimized as well as hybrids. We draw examples from several state-of-the-art systems but we also put everything together in a general framework which allows us to model storage engine designs under a single unified model and reason about the expected behavior of diverse designs. In addition, we show that given the vast number of possible storage engine designs and their complexity, there is a need to be able to describe and communicate design decisions at a high level descriptive language and we present a first version of such a language. We then use that framework to present several open challenges in the field especially in terms of supporting increasingly more diverse and dynamic applications in the era of data science and AI, including neural networks, graphs, and data versioning.

ACM Reference Format:
Stratos Idreos and Mark Callaghan. 2020. Key-Value Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3318464.3383133

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3383133

1 KEY-VALUE STORAGE ENGINES, AND APPLICATIONS

Key-Value systems support a key-value API and store and manage key-value pairs. Such **key-value stores are everywhere**, providing the storage backbone for an ever-growing number of diverse applications. The scenarios range from graph processing in social media [9, 14], to event log processing in cybersecurity [15], application data caching [51], NoSQL stores [57], flash translation layer design [21], time-series management [42, 43], and online transaction processing [26]. In addition, key-value stores increasingly have become an attractive solution as embedded systems in complex data-intensive applications, machine learning pipelines, and larger systems that support more complex data models. For example, key-value stores are utilized in SQL systems, e.g., FoundationDB [8] is a core part of Snowflake [18], while MyRocks integrates RockDB in MySQL as its back-end storage.

Storage Engine Design. At its core a key-value store implements a data structure that stores key-value pairs. Each data structure design achieves a specific balance regarding the fundamental trade-offs of read, update, and memory amplification [11]. For example, read amplification is defined as "how much more data do we have to read for every key we are looking for on top of accessing this particular key". In fact read, update, and memory amplification further break down to more fine grained metrics such as point reads, range reads, updates, deletes, inserts, and memory needed for caching. The design of the core data structure affects each one of those performance properties as well as every feature and property of the system. For example, to support time-travel queries we need to decide how to store timestamps. To accelerate queries on recently accessed data we need to balance the available memory between caching and other structures needed to accelerate processing on base data, e.g., in-memory bloom filters, fence pointers that help skip I/O. In order to support efficient concurrent read and write requests, a storage engine might have to change the data layout from an in-place to an out-of-place paradigm.

Rapidly Changing System Requirements. Today more than ever, we want to build, or change and adapt a data system quickly such that we can keep up with the needs of ever changing applications and hardware. New applications or new features in existing applications, with new workload patterns, appear frequently. A single system is not capable of efficiently supporting diverse workloads. This is a problem for

several increasingly pressing reasons. First, new applications appear many of which introduce new workload patterns that were not typical before. Second, existing applications keep redefining their services and features which affects their workload patterns directly and in many cases renders the existing underlying storage decisions sub-optimal or even bad. Third, hardware keeps changing which affects the CPU/bandwidth/latency balance; maximum performance requires low-level storage design changes. These problems boil down to the one size does not fit all problem which holds for overall system design [65] and for the storage layer [11]. Especially, in today's cloud-based world even slightly sub-optimal designs by 1% translate to a massive loss in energy utilization and thus costs [44].

There is no Perfect Design. There exists no perfect data structure that minimizes all performance trade-offs [11, 39]. For example, if we add a log to support efficient out of place writes, we sacrifice memory/space cost as we may have duplicate entries, and read cost as future queries have to search both the core data structure and the log. In turn, this means that there exists no perfect key-value store that covers diverse performance requirements. Every design is a compromise. But then how do we know which design is best for an application, e.g., for specific data, access patterns, hardware used, or even a maximum financial budget on the cloud? And do we have enough designs and systems to cover the needs of emerging and ever-changing data-driven applications? Especially today with an increasingly diverse and large array of applications driven by new opportunities in data science, and machine learning, it is very critical to have systems properly designed at their core to match the requirements of the target application.

Tutorial Structure. The tutorial consists of three parts. The first part goes over the overall problem setting as described in this section, 1) defining key-value store systems, 2) describing traditional as well as emerging applications across data science fields, and 3) giving a view of critical open challenges especially towards diverse workloads and dynamic application scenarios. The second part discusses major research and industry trends from the past several years that have formed the state-of-the-art in key-value storage engines. In particular, we cover 1) read-optimized storage engines, 2) write-optimized storage engines, 3) design options and considerations on allocating memory across system components, 4) how to store time-stamps, and 5) how to store large values. In the third part, we present a unified framework that encapsulates all state-of-the-art designs and allows us to reason about the possible design space of storage engines. We also discuss how to describe and communicate storage engine designs and why this is a critical issue. Finally, we present the open challenge of self-designed key-value storage engines. We explain 1) the new opportunities they bring compared to past solutions, 2) how they can be applied to solve practical problems across many classes of data-intensive applications, and 3) the new research opportunities that arise from their fusion with past work. Self-designed systems [35, 40] know the possible design choices and their combinations for critical system design components such as data storage, and can choose the most appropriate design among drastically different choices

2 STATE-OF-THE-ART ENGINE DESIGN

The Big Three. There are three predominant data structure designs for key-value stores to organize data. To give an idea of the diverse design goals and performance balances they provide, we go briefly through their core design characteristics. The first one is the **B$^+$tree** [13]. The prototypical B$^+$tree design consists of a leaf level of independent nodes with sorted key-value pairs (typically multiple storage blocks each) and an index (logarithmic at the number of leaf nodes) which consists of nodes of fractional cascading fence pointers with a large fanout. For example, B$^+$tree is the backbone design of the BerkeleyDB key-value store [53], now owned by Oracle, and the backbone of the WiredTiger key-value store [66], now used as the primary storage engine in MongoDB [52]. FoundationDB [8] also relies on a B$^+$tree. Overall, B$^+$tree achieves a good balance between read and write performance with a reasonable memory overhead that is primarily due to its fill factor in each node (typically 50%) and the auxiliary internal index nodes.

In the early 2000s, a new wave of applications emerged requiring faster writes, while still giving good read performance. At the same time, the advent of flash-based SSDs has made write I/Os 1-2 orders of magnitude costlier than read I/Os [1]. These workload and hardware trends led to two data structure design decisions for key-value stores: 1) buffering new data in memory, batching writes in secondary storage, and 2) avoiding global order maintenance. This class of designs was pioneered by the **Log-Structured Merge Tree** (LSM-tree) [54] which partitions data temporally in a series of increasingly larger levels. Each key-value entry enters at the very top level (the in-memory write buffer) and is sort-merged at lower levels as more data arrives. In-memory structures such as Bloom filters, fence pointers and Tries help filter queries to avoid disk I/O [19, 67]. This design has been adopted in numerous industrial settings including LevelDB [30] and BigTable [17] at Google, RocksDB [27] at Facebook, Cassandra [45], HBase [33] and Accumulo [7] at Apache, Voldemort [47] at LinkedIn, Dynamo [24] at Amazon, WiredTiger [66] at MongoDB, and bLSM [61] and cLSM [29] at Yahoo, and more designs in research such as SlimDB [58], WiscKey [49], Monkey [19, 20], Dostoevsky [22], and LSM-bush [23]. Relational databases such as MySQL and

SQLite4 support this design too by mapping primary keys to rows as values. Overall, LSM-tree-based designs achieve better writes than B$^+$tree-based designs but they typically give up some read performance (e.g., for short-range queries) given that we have to look for data through multiple levels, and they also give up some memory amplification to hold enough in-memory filters to support efficient point queries. Crucial design knobs, such as fill factor for B$^+$tree and size ratio for LSM-tree, define the space amplification relationship among the two designs.

More recently, a third design emerged for applications that require even faster ingestion rates. The primary data structure design decision was to drop order maintenance. Data accumulates in an in-memory write buffer. Once full, it is pushed to secondary storage as yet another node of an ever-growing single level log. An in-memory index, e.g., a hash table, allows locating any key-value pair easily while the log is periodically merged to enforce an upper bound on the number of obsolete entries. This **Log and Index** design is employed by BitCask [62] at Riak, Sparkey [64] at Spotify, FASTER [16] at Microsoft, and many more systems in research [2, 46, 59]. Most systems use a hash table as the index over the log. Overall, such a design achieves excellent write performance, but it sacrifices read performance (for range queries), while the memory footprint is also typically higher since now all keys need to be indexed in-memory to minimize I/O needs per key.

Memory Management. One of the most critical decisions in key-value stores is how to distribute the available bits across the various in-memory components. For example, in an LSM-tree like design it is common to have numerous Bloom-filters in memory and other helper structures to help skip disk reads. Similarly, caching is used to help with access to recent items or data blocks. Buffers that holds recent updates/inserts also compete for memory bits. Overall each one of these components can have a positive impact in system performance but given a fixed budget and a workload and storage engine base design it is not always clear how to best assign memory.

Compactions and Splits. Depending on the exact design a NoSQL engine will need to frequently reorganize data such as to maintain certain performance invariants. For example, an LSM-tree like design needs to perform compactions as new data arrives such as to maintain order and remove past invalid values which have been updated out of place. Depending on the frequency of compactions the system can be characterized from read to write optimized. Furthermore, compactions may happen in-place or out-of-place. The latter allows queries to be served over the previous layout while the compaction is happening. This comes at the cost of temporarily duplicating the relevant data. In-place does not require any extra memory but needs to block queries. Understanding the design space of compactions and splits is critical for NoSQL storage engine design so we can navigate diverse application requirements.

Concurrency Control. Key-value storage engines need to support large numbers of concurrent queries. When reads and writes arrive at the same time then depending on the exact design of the engine there are a plethora of approaches on increasing throughput. For example, LSM-trees are inherently more able to process concurrent requests compared to a typical B-tree design because they update data out of place. Similarly, a log-structured hash table design goes a step further by performing much fewer compactions and thus creating fewer conflicts for reads and writes (at the expense of read cost). B-tree designs can also adopt an out of place approach by stacking updates in leaf nodes like BW-tree or across any node like B$^\epsilon$tree.

Time-travel Queries. A very useful property in business applications is to be able to query data based on the status of the system at a particular point in time. This means that key-value pairs should be associated with timestamps. Supporting even small number of versions, though, can be a significant overhead in terms of storage. At the same time the exact way the timestamps are stored is critical. For example, if timestamps are stored inline with the base date, then this can lead to significant overheads for all queries (since timestamps will need to be read along with the base data).

CPU vs I/O Cost. Primarily key-value engines deal with big data and as such the bulk of their performance cost comes from moving data from disk and across the memory hierarchy. However, still a significant part of the cost does come from CPU. For example, depending on the mix of access patterns in the workload an engine may be more or less intensive in moving data. Similarly, using compression leads to increased CPU costs and the exact form of compression used defines the balance of I/O saved versus CPU sacrificed. This trade-off becomes especially important on the cloud where both CPU and I/O cost contribute to the required budget while different cloud providers offer different (and often changing) cost policies.

Adaptive Indexing and Layouts. One way to “blend" performance properties for diverse workloads is through adaptivity. While the concept has not been studied in NoSQL storage there are a lot of parallels that can be drown and we will use these concepts when we touch on the open challenges for NoSQL engines. Adaptive indexing [36] is a lightweight approach in self-tuning databases. Adaptive indexing addresses the limitations of offline and online indexing for dynamic workloads; it reacts to workload changes by building or refining indices partially and incrementally as part of query processing. That is, no DBA or offline processing is needed. By reacting to every single query with

lightweight actions, adaptive indexing manages to instantly adapt to a changing work load. As more queries arrive, the more the indices are refined and the more performance improves. Recently this area has received considerable attention with numerous works that study adaptivity with regards to base storage in relational systems, NoSQL systems, updates, concurrency, and time-series data management [4, 5, 10, 19, 25, 31, 32, 36, 37, 41, 48, 55, 56, 60, 63, 68]. Typically, in these lines of work the layout adapts to incoming requests. Similarly works on tuning via experiments [12], learning [6], and tuning via machine learning [3, 34] can adapt parts of a design using feedback from tests.

3 SELF-DESIGNED NOSQL STORAGE

In the third part of the tutorial we will describe in detail new research opportunities to create custom storage engines to match tailored applications requirements.

Self-designed Systems. Self-designed systems rely on the notion of mapping the possible space of critical design decisions in a system. For example, the Data Calculator introduced the design space of key-value storage [40]. The design space is defined by all designs that can be described as combinations and tunings of the "first principles of design". A first principle is a fundamental design concept that cannot be broken into additional concepts, e.g., for data structure design: fence pointers, links, temporal partitioning, and so on. The intuition is that, over the past decades, researchers have invented numerous fundamental design concepts such that a plethora of new valid designs with interesting properties can be synthesized out of those. The design space presented in [40] is shown to cover state-of-the-art designs, but it also reveals that a massive number of additional storage designs can be derived. As an analogy consider the periodic table of elements in chemistry; it categorized existing elements, but it also predicted unknown elements and their properties. In the same way, we can create the periodic table of data structures [39] which describes more key-value store designs than stars on the sky. Similar efforts have created design spaces in cache coherency protocols for database servers [28] and the design of parallel algorithms [50].

A self-designed system uses the design space to automatically generate designs that fit best a target workload and hardware. To do that we need to know how the various points in the space differ in terms of the performance properties they give to the resulting system. For example, learned cost models [40] is a method that enables learning the costs of fundamental access patterns (random access, scan, sorted search) out of which we can synthesize the costs of complex algorithms for a given data structure specification. These costs can, in turn, be used by machine learning algorithms that iterate over machine generated data structure specifications to label designs, and to compute rewards, deciding which specification to try out next. For example, early results using genetic algorithms [38] and dynamic programming [40] show the strong potential of such approaches to automatically discover close to optimal storage designs.

In addition, **design continuums** [20, 22, 35] is another direction which allows for accurate fast search for the best design. A design continuum is a performance hyperplane that connects a specific subset of designs within the set of all possible designs. Design continuums are effectively a projection of the design space, a "pocket" of designs where we can identify unifying properties among its members. A design continuum unifies major distinct data structure designs under the same model. The critical insight and potential long-term impact is that such unifying models 1) render what we consider up to now as fundamentally different data structures to be seen as "views" of the very same overall design space, and 2) allow "seeing" new data structure designs with performance properties that are not feasible by existing designs. The core intuition behind the construction of design continuums is that all data structures arise from the very same set of fundamental design principles, i.e., a small set of data layout design concepts out of which we can synthesize any design that exists in the literature as well as new ones. We show how to construct, evaluate, and expand, design continuums and we also present the a continuum that unifies major data structure designs, i.e., B^+tree, B^ϵtree, LSM-tree, and LSH-table.

The practical benefit of a design continuum is that it creates a fast inference engine for the design of data structures. For example, we can predict near instantly how a specific design change in the underlying storage of a data system would affect performance, or reversely what would be the optimal data structure (from a given set of designs) given workload characteristics and a memory budget. In turn, these properties allow us to envision a new class of self-designing key-value stores with a substantially improved ability to adapt to workload and hardware changes by transitioning between drastically different data structure designs to assume a diverse set of performance properties at will.

Storage Engine Description. Another critical issue triggered by the vast design space of storage engine designs and its complexity is how we communicate the specifics of a particular design. This is critical for system architects and engineers to understand and maintain systems as they evolve. We argue that a high level language is needed to describe storage engine and we present ideas and open problems to achieving that.

Applications: Big Data, Data Science. In the last part of the tutorial we touch on how the new directions described can apply across numerous data-intensive areas. We discuss

broad data science applications such as statistics-heavy processing and machine learning systems. Effectively, all these areas have in common the need to process ever growing amounts of data. The data storage and exact processing algorithms supported need to vary depending on the exact access patterns desired by the high level algorithms/workloads and by the hardware. As data grows, even the slightest suboptimality in these decisions can cost anything from hours to days in processing. The new ideas presented in this tutorial showcase open research problems towards being able to generate close to optimal storage systems for such data-intensive applications.

4 AUDIENCE, AND OUTPUT

Audience. The target audience for this tutorial is students, academics, researchers and software engineers with basic knowledge on data structures, algorithms and data system design. We assume basic understanding of fundamental data structures such as B-trees, LSM-trees, and Hash-tables. In addition, we assume basic knowledge of modeling and optimization. The tutorial is self-contained in providing all necessary background, no prior knowledge is needed on auto-tuning, adaptive systems/indexing, self-designed and learned systems.

Output. The target learning output is as follows:

(1) understanding use cases of key-value systems
(2) understanding state-of-the-art storage engine design
(3) understanding the long term technical limitations of state-of-the-art systems
(4) understanding the need of building new tailored systems that match the application needs
(5) exposure to the new research challenges with self-designed NoSQL systems
(6) exposure to the new opportunities across diverse data-intensive contexts: data science, machine learning, graphs

5 PRESENTERS

Stratos Idreos is an associate professor of Computer Science at Harvard University where he leads the Data Systems Laboratory. His research focuses on making it easy and even automatic to design workload and hardware conscious data structures and data systems with applications on relational, NoSQL, and broad data science and data exploration problems. Stratos was awarded the ACM SIGMOD Jim Gray Doctoral Dissertation award for his thesis on adaptive indexing. He received 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 2015 he was awarded the IEEE TCDE Rising Star Award from the IEEE Technical Committee on Data Engineering for his work on adaptive data systems. Stratos is also a recipient of the National Science Foundation Career award, and the Department of Energy Early Career award.

Mark Callaghan is a Distinguished Engineer at MongoDB and works on database performance. Previously he lead web-scale MySQL efforts at Facebook and Google for 14 years and worked on DBMS internals at Oracle and Informix for 9 years. He has an MS in CS from UW-Madison. He is interested in using math to understand the relationship between performance and efficiency in index structures.

6 ACKNOWLEDGMENTS

This work is partially funded by the USA Department of Energy project DE-SC0020200.

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 57–70.
[2] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Transactions on Computers (TC)* 65, 3 (2016), 902–915.
[3] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1009–1024.
[4] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1103–1114.
[5] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. 2014. Main Memory Adaptive Indexing for Multi-Core Systems. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 3:1—-3:10.
[6] Michael R. Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael J. Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. 2013. Brainwash: A Data System for Feature Engineering. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.
[7] Apache. [n. d.]. Accumulo. *https://accumulo.apache.org/* ([n. d.]).
[8] Apple. 2018. FoundationDB. *https://github.com/apple/foundationdb* (2018).
[9] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1185–1196.
[10] Joy Arulraj, Andrew Pavlo, and Prashanth 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*.
[11] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 461–466.
[12] Shivnath Babu, Nedyalko Borisov, Songyun Duan, Herodotos Herodotou, and Vamsidhar Thummala. 2009. Automated Experiment-Driven Management of (Database) Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*.
[13] Rudolf Bayer and Edward M. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the ACM SIGFIDET Workshop on Data Description and Access*, Vol. 1. 107–141.
[14] Yingyi Bu, Vinayak R. Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. 2014. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. *Proceedings of the VLDB Endowment* 8, 2 (2014), 161–172.
[15] Zhao Cao, Shimin Chen, Feifei Li, Min Wang, and Xiaoyang Sean Wang. 2013. LogKV: Exploiting Key-Value Stores for Log Processing. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.
[16] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J Levandoski, James Hunter, and Mike 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.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert 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.

[18] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W Lee, Ashish Motivala, Abdul Q Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 215–226.

[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94.

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

[21] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 327–342.

[22] Niv Dayan and Stratos 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.

[23] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220.

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

[26] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[27] Facebook. [n. d.]. RocksDB. *https://github.com/facebook/rocksdb* ([n. d.]).

[28] Michael J Franklin. 1993. *Caching and Memory Management in Client-Server Database Systems*. Ph.D. Dissertation. University of Wisconsin-Madison.

[29] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 32:1–32:14.

[30] Google. [n. d.]. LevelDB. *https://github.com/google/leveldb/* ([n. d.]).

[31] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, and Stefan Manegold. 2012. Concurrency control for adaptive indexing. *Proceedings of the VLDB Endowment* 5, 7 (2012), 656–667.

[32] Richard A Hankins and Jignesh M Patel. 2003. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 417–428.

[33] HBase. 2013. Online reference. *http://hbase.apache.org/* (2013).

[34] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1477–1492.

[35] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Biennial Conference on Innovative Data Systems Research (CIDR)*.

[36] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

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

[38] Stratos Idreos, Lukas M Maas, and Mike S Kester. 2017. Evolutionary Data Systems. *CoRR* abs/1706.0 (2017). arXiv:1706.05714

[39] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Engineering Bulletin* 41, 3 (2018), 64–75.

[40] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. 2018. 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.

[41] Oliver Kennedy and Lukasz Ziarek. 2015. Just-In-Time Data Structures. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[42] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *VLDB* 11, 6 (2018), 677–690.

[43] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut Palm: Static and Streaming Data Series Exploration Now in your Palm. In *SIGMOD*.

[44] Donald Kossman. 2018. Systems Research - Fueling Future Disruptions. In *Keynote talk at the Microsoft Research Faculty Summit*. Redmond, WA, USA.

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

[46] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 429–444.

[47] LinkedIn. [n. d.]. Voldemort. *http://www.project-voldemort.com* ([n. d.]).

[48] Zezhou Liu and Stratos Idreos. 2016. Main Memory Adaptive Denormalization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2253–2254.

[49] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi 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.

[50] Tim Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*. Addison-Wesley Professional.

[51] Memcached. [n. d.]. Reference. *http://memcached.org/* ([n. d.]).

[52] MongoDB. [n. d.]. Online reference. *http://www.mongodb.com/* ([n. d.]).

[53] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 183–191.

[54] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

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

[56] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. 2014. Database cracking: fancy scan, not poor man's sort!. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 1–8.

[57] Redis. [n. d.]. Online reference. *http://redis.io/* ([n. d.]).

[58] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[59] Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 1–16.

[60] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *Proceedings of the VLDB Endowment* 7, 2 (2013), 97–108.

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

[62] Justin Sheehy and David Smith. 2010. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. *Basho White Paper* (2010).

[63] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686.

[64] Spotify. 2014. Sparkey. *https://github.com/spotify/sparkey* (2014).

[65] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 2–11.

[66] WiredTiger. [n. d.]. Source Code. *https://github.com/wiredtiger/wiredtiger* ([n. d.]).

[67] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 323–336.

[68] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1555–1566.