Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn

Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, Zichen Zhu

Harvard University

ABSTRACT

We introduce the concept of design continuums for the data layout of key-value stores. 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 first 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 near instantly predict 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.

1. A VAST DESIGN SPACE

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 [10, 17], to event log processing in cybersecurity [18], application data caching [65], NoSQL stores [72], flash translation layer design [24], and online transaction processing [28]. In addition, increasingly key-value stores become an attractive solution as embedded systems in complex data-intensive and machine learning pipelines as well as larger systems that support more complex models. For example, key-value stores are utilized in SQL systems, e.g., FoundationDB [8] is a core part of Snowflake [22], while MyRocks integrates RockDB in MySQL as its backend storage.

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

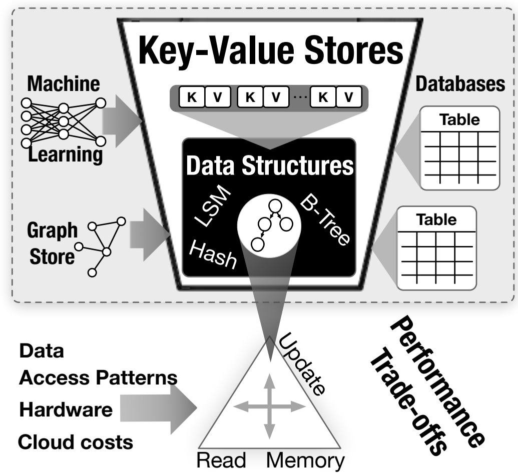


Figure 1: From performance trade-offs to data structures, key-value stores and rich applications.

There is no Perfect Design. As shown in Figure 1, 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 [12]. 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". There exists no perfect data structure that minimizes all three performance trade-offs [12, 41]. 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 desired maximum costs on the cloud? And do we have enough designs and systems to cover the needs of emerging and ever-changing data-driven applications? This is the problem we study in this paper and envision a research path that makes it easier to create custom data structure designs that match the needs of new applications, hardware, and cloud pricing schemes.

The Big Three. As of 2018, 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** [14]. 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 cascading fence pointers with a large fanout. For example, B$^+$tree is the backbone design of the BerkeleyDB key-value store [67], now owned by Oracle, and the backbone of the WiredTiger key-value store [88], now used as the primary storage engine in MongoDB [66]. 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) avoid maintaining global order. This class of designs was pioneered by the **Log-Structured Tree** (LSM-tree) [68] which partitions data temporally in a series of increasingly larger levels. Each key-value entry enters at the very top level (the in-memory 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 [23, 92]. This design has been adopted in numerous industrial settings including LevelDB [32] and BigTable [20] at Google, RocksDB [29] at Facebook, Cassandra [55], HBase [35] and Accumulo [7] at Apache, Voldemort [59] at LinkedIn, Dynamo [26] at Amazon, WiredTiger [88] at MongoDB, and bLSM [78] and cLSM [31] at Yahoo, and more designs in research such as SlimDB [73], WiscKey [62], and Monkey [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 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 force an upper bound on the number of obsolete entries. This **Log-Structured Hash-table** (LSH-table) is employed by BitCask [80] at Riak, Sparkey [82] at Spotify, FASTER [19] at Microsoft, and many more systems in research [74, 58, 2]. Overall, LSH-table 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.

The Practical Problem. While key-value stores continue to be adopted by an ever-growing set of applications, each application has to choose among the existing designs which may or may not be close to the ideal performance that could be achieved for the specific characteristics of this application. 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. Across all those cases, achieving maximum performance requires low-level storage design changes. This boils down to the one size does not fit all problem, which holds for overall system design [84] and for the storage layer [12].

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 [52], even if the performance difference is not directly felt by the end users. This implies two trends. First, getting as close to the optimal design is critical. Second, the way a data structure design translates to cost needs to be embedded in the design process as it is not necessarily about achieving maximum query throughput, but typically a more holistic view of the design is needed, including the memory footprint. Besides, the cost policy varies from one cloud provider to the next, and even for the same provider it may vary over time. For example, Amazon AWS charges based on CPU and memory for computation resources, and based on volume size, reserved throughput, and I/O performed for networked storage. Google Cloud Platform, while charging similarly for computation, only charges based on volume size for networked storage. This implies that the optimal data structure 1) is different for different cloud providers where the key-value store is expected to run, and 2) can vary over time for the same cloud provider even if the application itself and underlying hardware stay the same.

The Research Challenge. The long-term challenge is whether we can easily or even automatically find the optimal storage design for a given problem. This has been recognized as an open problem since the early days of computer science. In his seminal 1978 paper, Robert Tarjan includes this problem in his list of the five major challenges for the future (which also included P Vs NP) [85]: *"Is there a calculus of data structures by which one can choose the appropriate data representation and techniques for a given problem?"*. We propose that a significant step toward a solution includes dealing with the following **two challenges:**

1) Can we know all possible data structure designs?

2) Can we compute the performance of any design?

Toward an Answer to Challenge 1. We made a step toward the first challenge by introducing the **design space** of data structures supporting the key-value model [42]. The design space is defined by all designs that can be described as combinations and tunings of the "first principles of data layout design". A first principle is a fundamental design con-

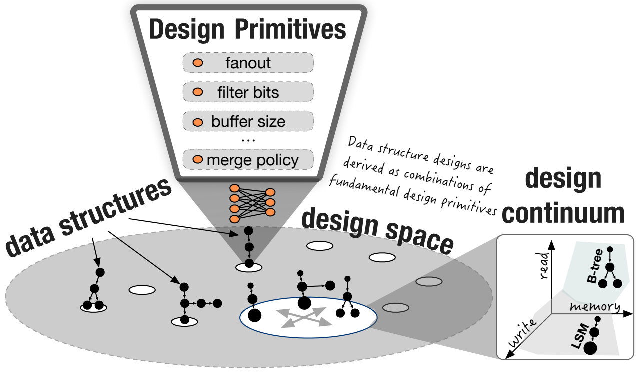


Figure 2: From data layout design principles to the design space of possible data structures, where design continuums can be observed to help navigate performance trade-offs across diverse data structure designs.

cept that cannot be broken into additional concepts, e.g., fence pointers, links, and temporal partitioning. The intuition is that, over the past several 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 [42].

As an analogy consider the periodic table of elements in chemistry; it sketched the design space of existing elements based on their fundamental components, and allowed researchers to predict the existence of unknown, at the time, elements and their properties, purely by the structure of the design space. In the same way, we created the **periodic table of data structures** [41] which describes more data structure designs than stars on the sky and can be used as a design and new data structure discovery guide.

Naturally, a design space does not necessarily describe "all possible data structures"; a new design concept may be invented and cause an exponential increase in the number of possible designs. However, after 50 years of computer science research, the chances of inventing a fundamentally new design concept have decreased exponentially; many exciting innovations, in fact, come from utilizing a design concept that, while known, it was not explored in a given context before and thus it revolutionizes how to think about a problem. Using Bloom filters as a way to filter accesses in storage and remote machines, scheduling indexing construction actions lazily [38], using ML models to guide data access [54], storage [47] and other system components [53], can all be thought of as such examples. Design spaces that cover large fundamental sets of concepts can help accelerate progress with figuring out new promising directions, and when new concepts are invented they can help with figuring out the new possible derivative designs.

Toward an Answer to Challenge 2. The next piece of the puzzle is to investigate if we can make it easy to compute the performance properties of any given data structure design. With the Data Calculator we introduced the idea of **learned cost models** [42] which allow 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 design. 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 design specification to try out next. Early results using genetic algorithms show the strong potential of such approaches [39]. However, there is still an essential missing link; given the fact that the design space size is exponential in the number of design principles (and that it will likely only expand over time), such solutions cannot find optimal designs in feasible time, at least not with any guarantee, leaving valuable performance behind [52]. This is the new problem we attack in this paper: Can we develop fast search algorithms that automatically or interactively help researchers and engineers find a close to optimal data structure design for a key-value store given a target workload and hardware environment?

Design Continuums. Like when designing any algorithm, the key ingredient is to induce domain-specific knowledge. Our insight is that there exist "design continuums" embedded in the design space of data structures. An intuitive way to think of design continuums is as a performance hyperplane that connects a specific subset of data structures designs. Design continuums are effectively a projection of the design space, a "pocket" of designs where we can identify unifying properties among its members. Figure 2 gives an abstract view of this intuition; it depicts the design space of data structures where numerous possible designs can be identified, each one being derived as a combination of a small set of fundamental design primitives and performance continuums can be identified for subsets of those structures.

1. We introduce design continuums as subspaces of the design space which connect more than one design. A design continuum has the crucial property that it creates a continuous performance tradeoff for fundamental performance metrics such as updates, inserts, point reads, long-range and short-range scans, etc.

2. We show how to construct continuums using few design knobs. For every metric it is possible to produce a closed-form formula to quickly compute the optimal design. Thus, design continuums enable us to **know** the best key-value store design for a given workload and hardware.

3. We present a design continuum that connects major classes of modern key-value stores including LSM-tree, B^{ϵ}tree, and B^{+}tree.

4. We show that for certain design decisions key-value stores should still rely on **learning** as it is hard (perhaps even impossible) to capture them in a continuum.

5. We present the vision of self-designing key-value stores, which morph across designs that are now considered as fundamentally different.

Inspiration. Our work is inspired by numerous efforts that also use first principles and clean abstractions to understand a complex design space. John Ousterhout's project Magic allows for quick verification of transistor designs so that engineers can easily test multiple designs synthesized by basic concepts [69]. Leland Wilkinson's "grammar of graphics" provides structure and formulation on the massive universe of possible graphics [87]. Timothy G. Mattson's work creates a language of design patterns for parallel algorithms [64]. Mike Franklin's Ph.D. thesis explores the possible client-server architecture designs using caching based replication as the main design primitive [30]. Joe Hellerstein's work on Generalized Search Trees makes it easy to design and test new data structures by providing templates which expose only a few options where designs need to differ [36, 5, 6, 49, 48, 50, 51]. S. Bing Yao's [91] and Stefan Manegold's [63] work on generalized hardware conscious cost models showed that it is possible to synthesize the costs of complex operations from basic access patterns. Work on data representation synthesis in programming languages enables synthesis of representations out of small sets of (3-5) existing data structures [75, 76, 21, 81, 79, 33, 34, 61, 83].

2. DESIGN CONTINUUMS

We now describe how to construct a design continuum.

2.1 From B$^+$tree to LSM-tree

We first give an example of a design continuum that connects diverse designs including Tiered LSM-tree [43, 23, 55], Lazy Leveled LSM-tree [25], Leveled LSM-tree [68, 23, 29, 32], COLA [15, 45], FD-tree [57], B$^\epsilon$tree [16, 9, 15, 44, 45, 70], and B$^+$tree [13]. The design continuum can be thought of as a **super-structure** that encapsulates all those designs. This super-structure consists of L levels where the larger Y levels are cold and the smaller $L-Y$ levels are hot. Hot levels use in-memory fence pointers and Bloom filters to facilitate lookups, whereas cold levels apply fractional cascading to connect runs in storage. Each level contains one or more runs, and each run is divided into one or more contiguous nodes. There is a buffer in memory to ingest application updates and flush to Level 1 when it fills up. This overall abstraction allows instantiating any of the data structure designs in the continuum. Figure 3 formalizes the continuum and the super-structure is shown at the bottom left.

Environmental Parameters. The upper right table in Figure 3 opens with a number of environmental parameters such as dataset size, main memory budget, etc. which are inherent to the application and context for which we want to design a key-value store.

Design Parameters. The upper right table in Figure 3 further consists of five continuous design knobs which have been chosen as the smallest set of movable design abstractions that we could find to allow differentiating among the target designs in the continuum. The first knob is the *growth factor* T between the capacities of adjacent levels of the structure (e.g., "fanout" for B$^+$tree or "size ratio" for LSM-tree). This knob allows us to control the super-structure's depth. The second knob is the *hot merge threshold* K, which is defined as the maximum number of independent sorted partitions (i.e., runs) at each of Levels 1 to $L-Y-1$ (i.e., all hot levels but the largest) before we trigger merging. The lower we set K, the more greedy merge operations become to enforce fewer sorted runs at each of these hot levels. Similarly, the third knob is the *cold merge threshold* Z and is defined as the maximum number of runs at each of Levels $L-Y$ to L (i.e., the largest hot level and all cold levels) before we trigger merging. The *node size* D is the maximal size of a contiguous data region (e.g., a "node" in a B$^+$tree or "SSTable" in an LSM-tree) within a run. Finally, the *fence and filters memory budget* M_F controls the amount of the overall memory that is allocated for in-memory fence pointers and Bloom filters.

Setting the domain of each parameter is a critical part of crafting a design continuum so we can reach the target designs and correct hybrid designs. Figure 3 describes how each design parameter in the continuum may be varied. For example, we set the maximum value for the size ratio T to be the block size B. This ensures that when fractional cascading is used at the cold levels, a parent block has enough space to store pointers to all of its children. As another example, we observe that a level can have at most $T-1$ runs before it runs out of capacity and so based on this observation we set the maximum values of K and Z to be $T-1$.

Design Rules: Forming the Super-structure. The continuum contains a set of design rules, shown on the upper right part of Figure 3. These rules enable instantiating specific designs by deterministically deriving key design aspects. Below we describe the design rules in detail.

Exponentially Increasing Level Capacities. The levels' capacities grow exponentially by a factor of T starting with the buffer's capacity. As a result, the overall number of levels L is $\lceil \log_T \frac{N \cdot E}{M_B} \rceil$, where M_B is the memory assigned to the buffer and $N \cdot E$ is the data volume.

Fence Pointers vs. Bloom Filters. Our design allocates memory for fence pointers and Bloom filters from smaller to larger levels based on the memory budget assigned by the knob M_F. Our strategy is to first assign this budget for fence pointers to as many levels as there is enough memory for. This is shown by the Equation for the fence pointers budget M_{FP} in Figure 3. The remaining portion of M_F after fence pointers is assigned to a Bloom filters memory budget M_{BF}. This can also be done in the reverse way when one designs a structure, i.e., we can define the desired buffer budget first and then give the remaining from the total memory budget to filters and fence pointers.

Optimally Allocated Bloom Filters Across Levels. The continuum assigns exponentially decreasing false positive rates (FPRs) to Bloom filters at smaller levels as this approach was shown to minimize the sum of their false positive rates and thereby minimize point read cost [23]. In Figure 3, we express the FPR assigned to Level i as p_i and give corresponding equations for how to set p_i optimally with respect to the different design knobs.

Hot vs. Cold Levels. Figure 3 further shows how to compute the number of cold levels Y for which there is no sufficient memory for fence pointers or Bloom filters (the derivation for Y is in terms of a known threshold X for when to drop a filter for a level and instead use that memory for filters at smaller levels to improve performance [25]). We derive $M_{F_{HI}}$ as the amount of memory above which all levels are hot (i.e., $Y = 0$). We also set a minimum memory requirement $M_{F_{LO}}$ on M_F to ensure that there is always enough memory for fence pointers to point to Level 1.

Fractional Cascading for Cold Levels. To be able to connect data at cold levels to the structure despite not having enough memory to point to them using in-memory fence pointers, we instead use fractional cascading. For every block within a run at a cold level, we embed a "cascading" pointer within the next younger run along with the smallest key in the target block. This allows us to traverse cold levels with one I/O for each run by following the corresponding cascading pointers to reach the target key range.

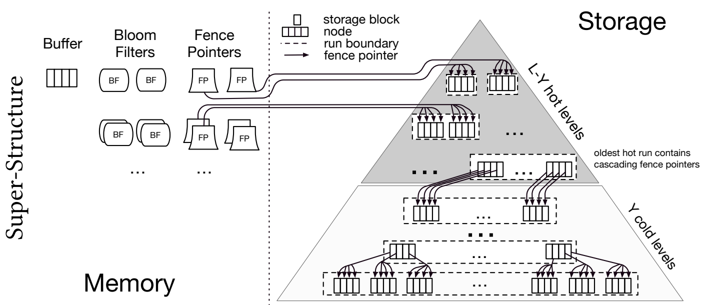
Active vs. Static Runs. Each level consists of one *active run* and a number of *static runs*. Incoming data into a level gets merged into the active run. When the active run reaches a fraction of T/K of the a levels' capacity for Levels 1 to $L-Y-1$ or T/Z for Levels $L-Y$ to L, it becomes a static run and a new empty active run is initialized.

Granular Rolling Merging. When a level reaches capacity,

Group	Term	Name	Description	Min. Value	Max. Value	Units
Environment Parameters	**B**	Block Size	# data entries that fit in a storage block.			Entries
	M	Memory	Total main memory budget.	$B \cdot E + \frac{F \cdot T \cdot M_B}{E \cdot B}$	$N \cdot E$	Bits
	N	Dataset Size	# total data entries in the dataset.			Entries
	E	Entry Size	Size of an entry.			Bits
	F	Key Size	Size of a key, also used to approximate size of a fence (fence key and pointer).			Bits
	s	Avg. Selectivity	Average selectivity of a long range query.			Entries
Design Parameters	**T**	Growth Factor	Capacity ratio between adjacent levels.	2	B	Ratio
	K	Hot Merge Threshold	Maximum # runs per hot level.	1	$T-1$	Runs
	Z	Cold Merge Threshold	Maximum # runs per cold level.	1	$T-1$	Runs
	D	Max. Node Size	Maximum size of a node; defines a contiguous data region.	1	$\frac{N}{B}$	Blocks
	M_F	Fence & Filter Memory Budget	# bits of main memory budgeted to fence pointers and filters.	$\frac{F \cdot T \cdot M_B}{E \cdot B}$	M	Bits

Derived Term	Expression	Units
L (# total levels)	$\lceil \log_T \frac{N \cdot E}{M_B} \rceil$	Levels
X (Filters Memory Threshold)	$\frac{1}{\ln 2^2} \cdot (\frac{\ln T}{T-1} + \frac{\ln K - \ln Z}{T})$	Bits per Entry
$M_{F_{HI}}$ (M_F Threshold: Hot Levels Saturation)	$N \cdot (\frac{X}{T} + \frac{F}{B})$	Bits
$M_{F_{LO}}$ (M_F Threshold: Cold Levels Saturation)	$\frac{M_B \cdot F \cdot T}{E \cdot B}$	Bits
Y (# Cold Levels)	$\begin{cases} 0 & \text{if } M_F \geq M_{F_{HI}} \\ \lfloor \log_T \frac{N}{M_F} \cdot (\frac{X}{T} + \frac{F}{B}) \rfloor & \text{if } M_{F_{LO}} < M_F < M_{F_{HI}} \\ L-1 & \text{if } M_F = M_{F_{LO}} \end{cases}$	Levels
M_{FP} (Fence Pointer Memory Budget)	$T^{L-Y+1} \cdot F \cdot \frac{M_b}{E \cdot B} \cdot \frac{T}{T-1}$	Bits
M_{BF} (Filter Memory Budget)	$M_F - M_{FP}$	Bits
M_B (Buffer Memory Budget)	$B \cdot E + (M - M_F)$	Bits
p_{sum} (Sum of BF False Positive Rates)	$e^{-\frac{M_{BF}}{N} \cdot \ln(2)^2 \cdot T^Y} \cdot Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1}$	
p_i (BF False Positive Rate at Level i)	$\begin{cases} 1 & \text{if } i > L-Y \\ \frac{p_{sum}}{Z} \cdot \frac{T-1}{T} & \text{if } i = L-Y \\ \frac{p_{sum}}{K} \cdot \frac{T-1}{T} \cdot \frac{1}{T^{L-Y-i}} & \text{if } i < L-Y \end{cases}$	Probability

Derived Design Rules



General Cost Model

Operation	Cost Expression (I/O)
Update	$O(\frac{1}{B} \cdot (\frac{T}{K} \cdot (L-Y-1) + \frac{T}{Z} \cdot (Y+1)))$
Zero Result Lookup	$O(Z \cdot e^{-\frac{M_{BF}}{N} \cdot T^Y} + Y \cdot Z)$
Single Result Lookup	$O(1 + Z \cdot e^{-\frac{M_{BF}}{N} \cdot T^Y} + Y \cdot Z)$
Short Scan	$O(K \cdot (L-Y-1) + Z \cdot (Y+1))$
Long Scan	$O(\frac{s \cdot Z}{B})$

Figure 3: An example of a design continuum: connecting complex designs with few continuous parameters.

a merge operation needs to take place to free up space. We perform a merge by first picking an *eviction key*[1]. Since each run is sorted across its constituent nodes, there is at most one node in each of the static runs at the level that intersects with the eviction key. We add these nodes into an *eviction* set and merge them into the active run in the next larger level. Hence, the *merge granularity* is controlled by the maximum node size D, and merge operations *roll* across static runs and eventually empty them out.

Fractional Cascading Maintenance. As merge operations take place at cold levels, cascading fence pointers must be maintained to keep runs connected. As an active run gradually fills up, we must embed cascading fence pointers from within the active run at the next smaller level. We must also create cascading fence pointers from a new active run into the next older static run at each level. To manage this, whenever we create a new run, we also create a *block index* in storage to correspond to the fences for this new run. Whenever we need to embed pointers into a Run i from some new Run j as Run j is being created, we include the block index for Run i in the sort-merge operation used to create Run j to embed the cascading fence pointers within.

Unified Cost Model. A design continuum includes a cost model with a closed-form equation for each one of the core performance metrics. The bottom right part of Figure 3 depicts these models for our example continuum. These cost models measure the worst-case number of I/Os issued for each of the operation types, the reason being that I/O is typically the performance bottleneck for key-value stores that store a larger amount of data than can fit in memory.[2] For example, the cost for point reads is derived by adding the expected number of I/Os due to false positives across the hot levels (given by the Equation for p_{sum}, the sum of the FPRs [25]) to the number of runs at the cold levels, since with fractional cascading we perform 1 I/O for each run. As another example, the cost for writes is derived by observing that an application update gets copied on average $O(T/K)$ times at each of the hot levels (except the largest) and $O(T/Z)$ times at the largest hot level and at each of the cold levels. We add these costs and divide by the block size B as a single write I/O copies B entries from the original runs to the resulting run.

While our models in this work are expressed in terms of asymptotic notations, we have shown in earlier work that

[1]The strategy for picking the eviction key may be as simple as round robin, though more sophisticated strategies to minimize key overlap with the active run in the next level are possible so as to minimize merge overheads [86].

[2]Future work can also try to generate in-memory design continuums where we believe learned cost models that help synthesize the cost of arbitrary data structure designs can be a good start [42].

	Terms \ Designs	Tiered LSM-Tree [55, 23, 43]	Lazy Leveled LSM-Tree [25]	Leveled LSM-Tree [32, 29, 23]	COLA [15, 45]	FD-Tree [57]	B^ϵTree [16, 15, 44, 70, 9, 45]	B+Tree [13]
Parameters	T (Growth Factor)	$[2, B]$	$[2, B]$	$[2, B]$	2	$[2, B]$	$[2, B]$	B
	K (Hot Merge Threshold)	$T-1$	$T-1$	1	1	1	1	1
	Z (Cold Merge Threshold)	$T-1$	1	1	1	1	1	1
	D (Max. Node Size)	$[1, \frac{N}{B}]$	$[1, \frac{N}{B}]$	$[1, \frac{N}{B}]$	$\frac{N}{B}$	$\frac{N}{B}$	1	1
	M_F (Fence & Filter Mem.)	$N \cdot (\frac{F}{B} + 10)$	$N \cdot (\frac{F}{B} + 10)$	$N \cdot (\frac{F}{B} + 10)$	$\frac{F \cdot T \cdot M_B}{E \cdot B}$	$\frac{F \cdot T \cdot M_B}{E \cdot B}$	$\frac{F \cdot T \cdot M_B}{E \cdot B}$	$\frac{F \cdot T \cdot M_B}{E \cdot B}$
Metrics	Update	$O(\frac{L}{B})$	$O(\frac{1}{B} \cdot (T+L))$	$O(\frac{T}{B} \cdot L)$	$O(\frac{L}{B})$	$O(\frac{T}{B} \cdot L)$	$O(\frac{T}{B} \cdot L)$	$O(L)$
	Zero Result Lookup	$O(T \cdot e^{\frac{-M_{BF}}{N}})$	$O(e^{-\frac{M_{BF}}{N}})$	$O(e^{-\frac{M_{BF}}{N}})$	$O(L)$	$O(L)$	$O(L)$	$O(L)$
	Existing Lookup	$O(1 + T \cdot e^{\frac{-M_{BF}}{N}})$	$O(1)$	$O(1)$	$O(L)$	$O(L)$	$O(L)$	$O(L)$
	Short Scan	$O(L \cdot T)$	$O(1 + T \cdot (L-1))$	$O(L)$	$O(L)$	$O(L)$	$O(L)$	$O(L)$
	Long Scan	$O(T \cdot \frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$

Figure 4: Instances of the design continuum and examples of their derived cost metrics.

such models can be captured more precisely to reliably predict worst-case performance [23, 25]. A central advantage of having a set of closed-form set of models is that they allow us to see how the different knobs interplay to impact performance, and they reveal the trade-offs that the different knobs control.

Overall, the choice of the design parameters and the derivation rules represent the infusion of expert design knowledge such that we can create a navigable design continuum. Specifically, **fewer design parameters** (for the same target designs) lead to a cleaner abstraction which in turn makes it easier to come up with algorithms that automatically find the optimal design (to be discussed later on). We minimize the number of design parameters in two ways: 1) by adding deterministic design rules which encapsulate expert knowledge about what is a good design, and 2) by collapsing more than one interconnected design decisions to a single design parameter. For example, owe used a single parameter for the memory budget of bloom filters and fence pointers as they only make sense when used together at each level.

Design Instances. Figure 4 depicts several known instances of data structure designs as they are derived from the continuum. In particular, the top part of Figure 4 shows the values for the design knobs that derive each specific design, and the bottom part shows how their costs can indeed be derived from the generalized cost model of the continuum.

For example, a B^+tree is instantiated by (1) setting the maximum node size D to be one block[3], (2) setting K and Z to 1 so that all nodes within a level are globally sorted, (3) setting M_f to the minimum amount of memory so that Levels 1 to L get traversed using fractional cascading without the utilization of Bloom filters or in-memory fence pointers, and (4) setting the growth factor to be equal to the block size. By plugging the values of these knobs into the cost expressions, the well-known write and read costs for a B^+tree of $O(L)$ I/Os immediately follow.

As a second example, a leveled LSM-tree design is instantiated by (1) setting K and Z to 1 so that there is at most one run at each level, and (2) assigning enough memory to the knob M_f to enable fence pointers and Bloom filters (with on average 10 bits per entry in the table) for all levels. We leave the knobs D and T as variables in this case as they are indeed used by modern leveled LSM-tree designs to strike different trade-offs. By plugging in the values for the design knobs into the cost models, we immediately obtain the well-known costs for a leveled LSM-tree. For example, write cost simplifies to $O(\frac{T \cdot L}{B})$ as every entry gets copied across $O(L)$ levels and on average $O(T)$ times within each level.

Construction Summary. Figure 5 summarizes the process of constructing a design continuum. We start by selecting a set of data structures. Then we select the minimum set of design knobs that can instantiate these designs and we impose design rules and domain restrictions to restrict the population of the continuum to only the best designs with respect to our target cost criteria. Finally, we derive the generalized cost models.

Definition of Continuum. We can now revisit the exact definition of the continuum. A design continuum connects previously distinct and seemingly fundamentally different data structure designs. The construction process does not necessarily result in continuous knobs in the mathematical sense (most of the design knobs have integer values).

[3]Node size can be set to whatever we want the B^+tree node size to be - we use $D = 1$ block here as an example only.

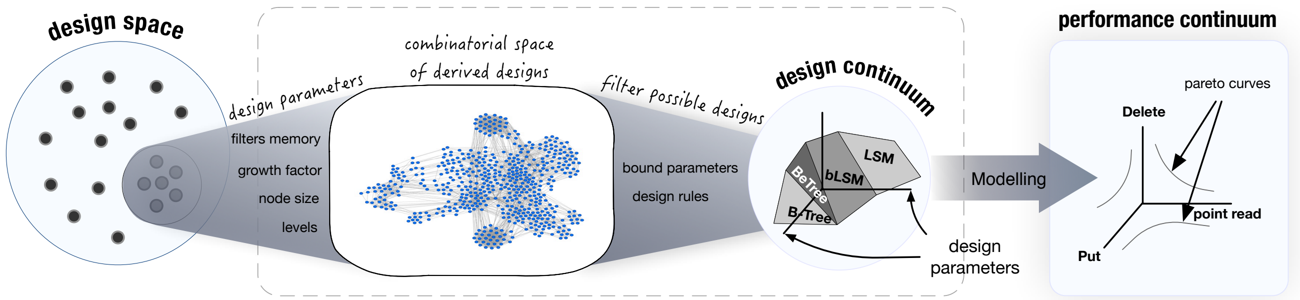


Figure 5: Constructing a design continuum: from design parameters to a performance hyperplane.

However, from a design point of view a continuum opens the subspace in between previously unconnected designs; it allows us to connect those discrete designs in fine grained steps, and this is exactly what we refer to as the "design continuum". The reason that this is critical is that it allows us to 1) "see" designs that we did not know before, derived as combinations of those fine-grained design options, and 2) build techniques that smoothly transition across discrete designs by using those intermediate states.

2.2 Interactive Design

The generalized cost models enable us to navigate the continuum, i.e., interactively design a data structure for a key-value store with the optimal configuration for a particular application as well as to react to changes in the environment, or workload. We formalize the navigation process by introducing Equation 1 to model the average operation cost θ through the costs of zero-result point lookups R, non-zero-result point lookups V, short range lookups Q, long range lookups C, and updates W (the coefficients depict the proportions of each in the workload).

$$\theta = (r \cdot R + v \cdot V + q \cdot Q + c \cdot C + w \cdot W) \tag{1}$$

To design a data structure using Equation 1, we first identify the bottleneck as the highest additive term as well as which knobs in Figure 3 can be tweaked to alleviate it. We then tweak the knob in one direction until we reach its boundary or until θ reaches the minimum with respect to that parameter. We then repeat this process with other parameters as well as with other bottlenecks that can emerge during the process. This allows us to converge to the optimal configuration without backtracking, which allows us to adjust to a variety of application scenarios reliably. For example, consider an application with a workload consisting of point lookups and updates and an initial configuration of a lazy-leveled LSM-tree with $T = 10$, $K = T-1$, $Z = 1$, $D = 64$, M_B set to 2 MB, and M_f set to $N \cdot (F/B + 10)$, meaning we have memory for all the fence pointers and in addition 10 bits per entry for Bloom filters. We can now use the cost models to react to different scenarios.

Scenario 1: Updates Increasing. Suppose that the proportion of updates increases, as is the case for many applications [78]. To handle this, we first increase Z until we reach the minimum value for θ or until we reach the maximum value of Z. If we reach the maximum value of Z, the next promising parameter to tweak is the size ratio T, which we can increase in order to decrease the number of levels across which entries get merged. Again, we increase T until we hit its maximum value or reach a minimum value for θ.

Scenario 2: Range Lookups. Suppose that the application changes such that short-range lookups appear in the workload. To optimize for them, we first decrease K to restrict the number of runs that lookups need to access across Levels 1 to $L-1$. If we reach the minimum value of K and short-range lookups remain the bottleneck, we can now increase T to decrease the overall number of levels thereby decreasing the number of runs further.

Scenario 3: Data Size Growing. Suppose that the size of the data is growing, yet most of the lookups are targeting the youngest $N_{youngest}$ entries, and we do not have the resources to continue scaling main memory in proportion to the overall data size N. In such a case, we can fix M_f to $N_{youngest} \cdot (F/B + 10)$ to ensure memory is invested to provide fast lookups for the hot working set while minimizing memory overhead of less frequently requested data by maintaining cold levels with fractional cascading.

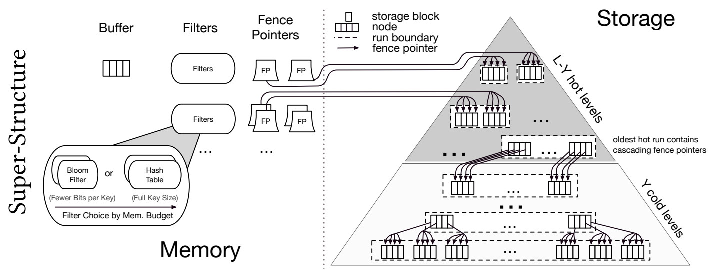
Effectively the above process shows how to quickly and reliably go from a high-level workload requirement to a low-level data structure design configuration at interactive times using the performance continuum.

Auto-Design. It is possible to take the navigation process one step further to create algorithms that iterate over the continuum and independently find the best configuration. The goal is to find the best values for T, K, Z, D, and the best possible division of a memory budget between M_F and M_B. While iterating over every single configuration would be intractable as it would require traversing every permutation of the parameters, we can leverage the manner in which we constructed the continuum to significantly prune the search space. For example, in Monkey [23], when studying a design continuum that contained only a limited set of LSM-tree variants we observed that two of the knobs have a logarithmic impact on θ, particularly the size ratio T and the memory allocation between M_b and M_f. For such knobs, it is only meaningful to examine a logarithmic number of values that are exponentially increasing, and so their multiplicative contribution to the overall search time is logarithmic in their domain. While the continuum we showed here is richer, by adding B-tree variants, this does not add significant complexity in terms of auto-design. The decision to use cascading fence pointers or in-memory fence pointers completely hinges on the allocation of memory between M_F and M_B, while the node size D adds one multiplicative logarithmic term in the size of its domain.

2.3 Success Criteria

We now outline the ideal success criteria that should guide the construction of elegant and practically useful design continuums in a principled approach.

Functionally Intact. All possible designs that can be assumed by a continuum should be able to correctly sup-



General Cost Model

Operation	Cost Expression (I/O)
Update	$O(\frac{1}{B} \cdot (\frac{T}{K} \cdot (L - Y - 1) + \frac{T}{Z} \cdot (Y + 1)))$
Zero Result Lookup	$O(Z \cdot e^{-\frac{M_{BF}}{N} \cdot T^Y} + Y \cdot (Z + \frac{T}{B}))$
Single Result Lookup	$O(1 + Z \cdot e^{-\frac{M_{BF}}{N} \cdot T^Y} + Y \cdot (Z + \frac{T}{B}))$
Short Scan	$O(K \cdot (L - Y - 1) + (Y + 1) \cdot (Z + \frac{T}{B}))$
Long Scan	$O(\frac{s \cdot Z}{B})$

Figure 6: Extending the design continuum to support Log Structured Hash table designs.

port all operation types (e.g., writes, point reads, etc.). In other words, a design continuum should only affect the performance properties of the different operations rather than the results that they return.

Pareto-Optimal. All designs that can be expressed should be Pareto-optimal with respect to the cost metrics and workloads desired. This means that there should be no two designs such that one of them is better than the other on one or more of the performance metrics while being equal on all the others. The goal of only supporting Pareto-optimal designs is to shrink the size of the design space to the minimum essential set of knobs that allow to control and navigate across only the best possible known trade-offs, while eliminating inferior designs from the space.

Bijective. A design continuum should be a bijective (one-to-one) mapping from the domain of design knobs to the co-domain of performance and memory trade-offs. As with Pareto-Optimality, the goal with bijectivity is to shrink a design continuum to the minimal set of design knobs such that no two designs that are equivalent in terms of performance can be expressed as different knob configurations. If there are multiple designs that map onto the same trade-off, it is a sign that the model is either too large and can be collapsed onto fewer knobs, or that there are core metrics that we did not yet formalize, and that we should.

Diverse. A design continuum should enable a diverse set of performance properties. For Pareto-Optimal and bijective continuums, trade-off diversity can be measured and compared across different continuums as the product of the domains of all the design knobs, as each unique configuration leads to a different unique and Pareto-optimal trade-off.

Navigable. The time complexity required for navigating the continuum to converge onto the optimal (or even near-optimal) design should be tractable. With the Monkey continuum, for example, we showed that it takes $O(log_T(N))$ iterations to find the optimal design, and for Dostoevsky, which includes more knobs and richer trade-offs, we showed that it takes $O(log_T(N)^3)$ iterations. Measuring navigability complexity in this way allows system designers from the onset to strike a balance between the diversity vs. the navigability of a continuum.

Layered. By construction, a design continuum has to strike a trade-off between diversity and navigability. The more diverse a continuum becomes through the introduction of new knobs to assume new designs and trade-offs, the longer it takes to navigate it to optimize for different workloads. With that in mind, however, we observe that design continuums may be constructed in layers, each of which builds on top of the others. Through layered design, different applications may use the same continuum but choose the most appropriate layer to navigate and optimize performance across. For example, the design continuum in Dostoevsky [23] is layered on top of Monkey [25] by adding two new knobs, K and Z, to enable intermediate designs between tiering, leveling and lazy leveling. While Dostoevsky requires $O(log_T(N)^3)$ iterations to navigate the possible designs, an alternative is to leverage layering to restrict the knobs K and Z to both always be either 1 or $T-1$ (i.e., to enable only leveling and tiering) in order to project the Monkey continuum and thereby reduce navigation time to $O(log_T(N))$. In this way, layered design enables *continuum expansion with no regret*: we can continue to include new designs in a continuum to enable new structures and trade-offs, all without imposing an ever-increasing navigation penalty on applications that need only some of the possible designs.

2.4 Expanding a Continuum: A Case-Study with LSH-table

We now demonstrate how to expand the continuum with a goal of adding a particular design to include certain performance trade-offs. The goal is to highlight the design continuum construction process and principles.

Our existing continuum does not support the LSH-table data structure used in many key-value stores such as BitCask [80], FASTER [19], and others [2, 58, 74, 82, 89]. LSH-table achieves a high write throughout by logging entries in storage, and it achieves fast point reads by using a hash table in memory to map every key to the corresponding entry in the log. In particular, LSH-table supports writes in $O(1/B)$ I/O, point reads in $O(1)$ I/O, range reads in $O(N)$ I/O, and it requires $O(F \cdot N)$ bits of main memory to store all keys in the hash table. As a result, it is suitable for write-heavy application with ample memory, and no range reads.

We outline the process of expanding our continuum in three steps: *bridging, patching, and costing.*

Bridging. Bridging entails identifying the least number of new movable design abstractions to introduce to a continuum to assume a new design. This process involves three options: 1) introducing new design rules, 2) expanding the domains of existing knobs, and 3) adding new design knobs.

Bridging increases the diversity of a design continuum, though it risks compromising the other success metrics. Designers of continuums should experiment with the three steps above in this particular order to minimize the chance of that happening. With respect to LSH-table, we need two new abstractions: one to allow assuming a log in storage, and one to allow assuming a hash table in memory.

To assume a log in storage, our insight is that with a tiered LSM-tree design, setting the size ratio to increase with respect to the number of runs at Level 1 (i.e., $T = {(N \cdot E)}/{M_B}$)

	Terms \ Designs	Log	LSH Table [80, 19, 82, 74, 58, 2, 89]	Sorted Array
Parameters	T (Growth Factor)	$\frac{N\cdot E}{M_b}$	$\frac{N\cdot E}{M_b}$	$\frac{N\cdot E}{M_b}$
	K (Hot Merge Threshold)	$T-1$	$T-1$	1
	Z (Cold Merge Threshold)	$T-1$	$T-1$	1
	D (Max. Node Size)	1	1	$\frac{N}{B}$
	M_F (Fence & Filter Mem.)	$\frac{N\cdot F}{B}$	$N\cdot F\cdot(1+\frac{1}{B})$	$\frac{N\cdot F}{B}$
Metrics	Update	$O(\frac{1}{B})$	$O(\frac{1}{B})$	$O(\frac{N\cdot E}{M_b\cdot B})$
	Zero Result Lookup	$O(\frac{N\cdot E}{M_b})$	$O(0)$	$O(1)$
	Single Result Lookup	$O(\frac{N\cdot E}{M_b})$	$O(1)$	$O(1)$
	Short Range Scan	$O(\frac{N\cdot E}{M_b})$	$O(\frac{N\cdot E}{M_b})$	$O(1)$
	Long Range Scan	$O(\frac{N\cdot E}{M_b}\cdot\frac{s}{B})$	$O(\frac{N\cdot E}{M_b}\cdot\frac{s}{B})$	$O(\frac{s}{B})$

Figure 7: Instances of the extended design continuum and examples of their derived cost metrics.

causes Level 1 to never run out of capacity. This effectively creates a log in storage as merge operations never take place. Our current design continuum, however, restricts the size ratio to be at most B. To support a log, we expand the domain of the size ratio with a new maximum value of $^{(N\cdot E)}/_{M_B}$.

To assume a hash table in memory, recall that our continuum assigns more bits per entry for Bloom filters at smaller levels. Our insight is that when the number of bits per entry assigned to given level exceeds the average key size F, it is always beneficial to replace the Bloom filters at that level with an in-memory hash table that contains all keys at the level. The reason is that a hash table takes as much memory as the Bloom filters would, yet it is more precise as it does not allow false positives at all. We therefore introduce a new design rule whereby levels with enough memory to store all keys use a hash table while levels with insufficient memory use Bloom filters[4]. With these two new additions to the continuum, we can now set the size ratio to $^{(N\cdot E)}/_{M_B}$ and K and Z to $T-1$ while procuring at least $F\cdot N$ bits of memory to our system to assume LSH-table[5]. Figure 6 shows the new super-structure of the continuum while Figure 7 shows how LSH-table can be derived.

An important point is that we managed to bridge LSH-table with our continuum without introducing new design knobs. As a rule of thumb, introducing new knobs for bridging should be a last resort as the additional degrees of freedom increase the time complexity of navigation. Our case-study here, however, demonstrates that even data structures that seem very different at the onset can be bridged by finding the right small set of movable abstractions.

Patching. Since the bridging process introduces many new intermediate designs, we follow it with a patching process to ensure that all of the new designs are functionally intact (i.e., that they can correctly support all needed types of queries). Patching involves either introducing new design rules to fix broken designs or adding constraints on the domains of some of the knobs to eliminate broken designs from the continuum. To ensure that the expanded continuum is layered (i.e., that it contains all designs from the continuum that we started out with), any new design rules or constraints introduced by patching should only affect new parts of the continuum. Let us illustrate an example of patching with the expanded continuum.

The problem that we identify arises when fractional cascading is used between two cold Levels i and $i+1$ while the size ratio T is set to be greater than B. In this case, there is not enough space inside each block at Level i to store all pointers to its children blocks (i.e., ones with an overlapping key range) at Level $i+1$. The reason is that a block contains B slots for pointers, and so a block at Level i has a greater number of children T than the number of pointer slots available. Worse, if the node size D is set to be small (in particular, when $D < {}^{T}/_{B}$), some of the blocks at Level $i+1$ will neither be pointed to from Level i nor exist within a node whereon at least one other block is pointed to from Level i. As a result, such nodes at Level $i+1$ would *leak* out of the data structure, and so the data on these blocks would be lost. To prevent leakage, we introduce a design rule that when $D < {}^{T}/_{B}$ and $B < T$, the setting at which leakage can occur, we add sibling pointers to reconnect nodes that have leaked. We introduce a rule that the parent block's pointers are spatially evenly distributed across its children (every $({}^{T}/_{(B\cdot D)})^{\text{th}}$ node at Level $i+1$ is pointed to from a block at level i) to ensure that all sibling chains of nodes within Level $i+1$ have an equal length. As these new rules only apply to new parts of our continuum (i.e., when $T > B$), they do not violate layering.

Costing. The final step is to generalize the continuum's cost model to account for all new designs. This requires either extending the cost equations and/or proving that the existing equations still hold for the new designs. Let us illustrate two examples. First, we extend the cost model with respect to the patch introduced above. In particular, the lookup costs need to account for having to traverse a chain of sibling nodes at each of the cold levels when $T > B$. As the length of each chain is ${}^{T}/_{B}$ blocks, we extend the cost equations for point lookups and short-range lookups with additional ${}^{T}/_{B}$ I/Os per each of the Y cold levels. The extended cost equations are shown in Figure 6.

In the derivation below, we start with general cost expression for point lookups in Figure 6 and show how the expected point lookup cost for LSH-table is indeed derived correctly. In Step 2, we plug in ${}^{N}/_{B}$ for T and Z to assume a log in storage. In Step 3, we set the number of cold levels to zero as Level 1 in our continuum by construction is always hot and in this case, there is only one level (i.e., $L = 1$), and thus Y must be zero. In Step 4, we plug in the key size F for the number of bits per entry for the Bloom filters, since with LSH-table there is enough space to store all keys in memory. In Step 5, we reason that the key size F must comprise on

[4]Future work toward an even more navigable continuum can attempt to generalize a Bloom filter and a hash table into one unified model with continuous knobs that allows to gradually morph between these structures based on the amount of main memory available.

[5]More precisely, $F\cdot N\cdot(1+\frac{1}{B})$ bits of memory are needed to support both the hash table and fence pointers.

average at least $log(N)$ bits to represent all unique keys. In Step 6, we simplify and omit small constants to arrive at a cost of $O(1)$ I/O per point lookup.

$$\in O(1 + Z \cdot e^{-(M_{BF}/N) \cdot T^Y} + Y \cdot (Z + {}^{T}\!/_{B}))$$

$$\in O(1 + {}^{N}\!/_{B} \cdot e^{-(M_{BF}/N) \cdot (N/B)^Y} + Y \cdot ({}^{N}\!/_{B} + {}^{N}\!/_{B^2})) \tag{2}$$

$$\in O(1 + {}^{N}\!/_{B} \cdot e^{-(M_{BF}/N)}) \tag{3}$$

$$\in O(1 + {}^{N}\!/_{B} \cdot e^{-F}) \tag{4}$$

$$\in O(1 + {}^{N}\!/_{B} \cdot e^{-\log_2(N)}) \tag{5}$$

$$\in O(1) \tag{6}$$

2.5 Elegance Vs. Performance: To Expand or Not to Expand?

As new data structures continue to get invented and optimized, the question arises of when it is desirable to expand a design continuum to include a new design. We show through an example that the answer is not always clear cut.

In an effort to make B-trees more write-optimized for flash devices, several recent B-tree designs buffer updates in memory and later flush them to a log in storage in their arrival order. They further use an in-memory indirection table to map each logical B-tree node to the locations in the log that contain entries belonging to that given node. This design can improve on update cost relative to a regular B-tree by flushing multiple updates that target potentially different nodes with a single sequential write. The trade-off is that during reads, multiple I/Os need to be issued to the log for every logical B-tree node that gets traversed in order to fetch its contents. To bound the number of I/Os to the log, a compaction process takes place once a logical node spans over C blocks in the log, where C is a tunable parameter. Overall, this design leads to a point and range read cost of $O(C \cdot \log_B(N))$ I/Os. On the other hand, update cost consists of $O(C \cdot \log_B(N))$ read I/Os to find the target leaf node and an additional amortized $O({}^{1}\!/_{C})$ write I/Os to account for the overheads of compaction. The memory footprint for the mapping table is $O({}^{(C \cdot N \cdot F)}\!/_{B})$ bits. We refer to this design as log-structured B-tree (LSB-tree). Would we benefit from including LSB-tree in our continuum?

To approach an answer to this question, we analytically compare LSB-tree against designs within our continuum to gauge the amount by which LSB-tree would allow us to achieve better trade-offs with respect to our continuum's cost metrics. We demonstrate this process in Figure 8, which plots point and range read costs against write cost for both LSB-tree and Leveled LSM-tree, a representative part of our continuum. To model write cost for LSB-tree, we computed a weighted cost of $O(C \cdot \log_B(N))$ read I/Os to traverse the tree, $O({}^{1}\!/_{C})$ write I/Os to account for compaction overheads, and we discounted the cost of a read I/O relative to a write I/O by a factor of 20 to account for read/write cost asymmetries on flash devices. We generated the curve for LSB-tree by varying the compaction factor C from 1 to 9, and the curves for the LSM-tree by varying the size ratio T from 2 to 10. To enable an apples-to-apples comparison whereby both LSB-tree and the LSM-tree have the same memory budget, we assigned however much main memory LSB-tree requires for its mapping table to the LSM-tree's fence pointers and Bloom filters. Overall, the figure serves as a first approximation for the trade-offs that LSB-tree would allow us to

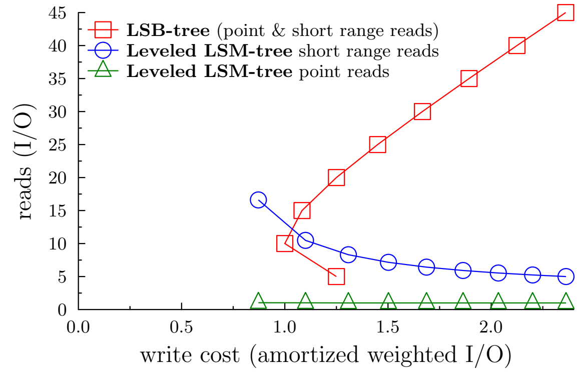


Figure 8: Leveled LSM-tree dominates LSB-tree for most of their respective continuums.

achieve relative to our continuum.

Figure 8 reveals that point read cost for the LSM-tree is much lower than for LSB-tree. The reason is that when the same amount of memory required by LSB-tree's memory budget is used for the LSM-tree's fence pointers and Bloom filters, hardly any false positives take place and so the LSM-tree can answer most point reads with just one I/O. Secondly, we observe that as we increase LSB-tree's compaction factor C, write cost initially decreases but then starts degrading rapidly. The reason is that as C grows, more reads I/Os are required by application writes to traverse the tree to identify the target leaf node for the write. On the other hand, for range reads there is a point at which LSB-tree dominates the LSM-tree as fewer blocks need to be accessed when C is small.

Elegance and Navigability versus Absolute Performance. By weighing the advantages of LSB-tree against the complexity of including it (i.e., adding movable abstractions to assume indirection and node compactions), one can decide to leave LSB-tree out of the continuum. This is because its design principles are fundamentally different than what we had included and so substantial changes would be needed that would complicate the continuum's construction and navigability. On the other hand, when we did the expansion for LSH-table, even though, it seemed initially that this was a fundamentally different design, this was not the case: LSH-table is synthesized from the same design principles we already had in the continuum, and so we could achieve the expansion in an elegant way at no extra complexity and with a net benefit of including the new performance trade-offs.

At the other extreme, one may decide to include LSB-tree because the additional performance trade-offs outweigh the complexity for a given set of desired applications. We did this analysis to make the point of elegance and navigability versus absolute performance. However, we considered a limited set of performance metrics, i.e., worst-case I/O performance for writes, point reads and range reads. Most of the work on LSB-tree-like design has been in the context of enabling better concurrency control [56] and leveraging workload skew to reduce compaction frequency and overheads [90]. Future expansion of the design space and continuums should include such metrics and these considerations described above for the specific example will be different. In this way, the decision of whether to expand or not to expand a continuum is a continual process, for which the outcome may change over time as different cost metrics change in their level of importance given target applications.

3. WHY NOT MUTUALLY EXCLUSIVE DESIGN COMPONENTS?

Many modern key-value stores are composed of mutually exclusive sets of swappable data layout designs to provide diverse performance properties. For example, WiredTiger supports separate B-tree and LSM-tree implementations to optimize more for reads or writes, respectively, while RocksDB files support either a sorted strings layout or a hash table layout to optimize more for range reads or point reads, respectively. A valid question is how does this compare to the design continuum in general? And in practice how does it compare to the vision of self-designing key-value stores?

Any exposure of data layout design knobs is similar in spirit and goals to the continuum but how it is done exactly is the key. Mutually exclusive design components can be in practice a tremendously useful tool to allow a single system to be tuned for a broader range of applications than we would have been able to do without this feature. However, it is not a general solution and leads to three fundamental problems.

1) Expensive Transitions. Predicting the optimal set of design components for a given application before deployment is hard as the workload may not be known precisely. As a result, components may need to be continually reshuffled during runtime. Changing among large components during runtime is disruptive as it often requires rewriting all data. In practice, the overheads associated with swapping components often force practitioners to commit to a suboptimal design from the onset for a given application scenario.

2) Sparse Mapping to Performance Properties. An even deeper problem is that mutually exclusive design components tend to have polar opposite performance properties (e.g., hash table vs. sorted array). Swapping between two components to optimize for one operation type (e.g. point reads) may degrade a different cost metric (e.g. range reads) by so much that it would offset the gain in the first metric and lead to poorer performance overall. In other words, optimizing by shuffling components carries a risk of overshooting the target performance properties and hitting the point of diminishing returns. A useful way of thinking about this problem is that mutually exclusive design components map sparsely onto the space of possible performance properties. The problem is that, with large components, there are no intermediate designs that allow to navigate performance properties in smaller steps.

3) Intractable Modeling. Even analytically, it quickly becomes intractable to reason about the tens to hundreds of tuning parameters in modern key-value stores and how they interact with all the different mutually exclusive design components to lead to different performance properties. An entirely new performance model is often needed for each permutation of design components, and that the number of possible permutations increases exponentially with respect to the number of components available. Creating such an extensive set of models and trying to optimize across them quickly becomes intractable. This problem gets worse as systems mature and more components get added and it boils down to manual tuning by experts.

The Design Continuum Spirit. Our work helps with this problem by formalizing this data layout design space so that educated decisions can be made easily and quickly, sometimes even automatically. Design continuums deal with

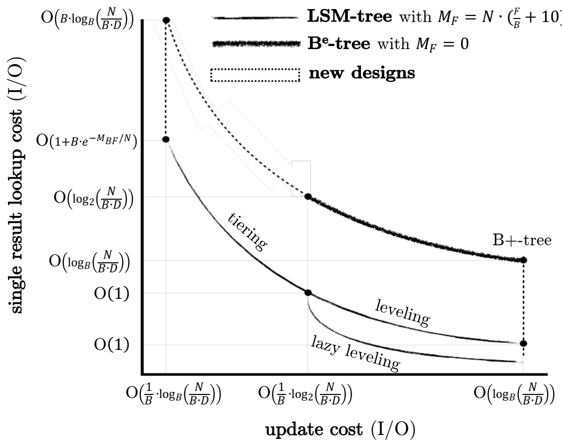


Figure 9: Visualizing the performance continuum.

even more knobs than what existing systems expose because they try to capture the fundamental design principles of design which by definition are more fine-grained concepts. For example, a sorted array is already a full data structure that can be synthesized out of many smaller decisions. However, the key is that design continuums know how to navigate those fine-grained concepts and eventually expose to designers a much smaller set of knobs and a way to argue directly at the performance tradeoff level. The key lies in constructing design continuums and key-value store systems via unified models and implementations with continuous data layout design knobs rather than swappable components.

For example, the advantage of supporting LSH-table by continuum expansion rather than as an independent swappable component is that the bridging process adds new intermediate designs into the continuum with appealing tradeoffs in-between. The new continuum allows us to gradually transition from a tiered LSM-tree into LSH-table by increasing the size ratio in small increments to optimize more for writes at the expense of range reads and avoid overshooting the optimal performance properties when tuning.

4. ENHANCING CREATIVITY

Beyond the ability to assume existing designs, a continuum can also assist in identifying new data structure designs that were unknown before, but they are naturally derived from the continuum's design parameters and rules.

For example, the design continuum we presented in this paper allows us to synthesize two new subspaces of hybrid designs, which we depict in Figure 9. The first new subspace extends the B^ϵtree curve to be more write-optimized by increasing Z and K to gather multiple linked nodes at a level before merging them. The second subspace connects B^ϵtree with LSM-tree designs, allowing first to optimize for writes and lookups at hot levels by using Bloom filters and fence pointers, and second to minimize memory investment at cold levels by using fractional cascading instead. Thus, we turn the design space into a multi-dimensional space whereon every point maps onto a unique position along a hyperplane of Pareto-optimal performance trade-offs (as opposed to having to choose between drastically different designs only).

In addition, as the knobs in a bijective continuum are dimensions that interact to yield unique designs, expanding any knob's domain or adding new knobs during the bridging process can in fact enrich a continuum with new, good designs that were not a part of the original motivation for expansion. Such examples are present in our expanded continuum where our original goal was to include LSH-table. For example, fixing K and Z to 1 and increasing the size ratio beyond B towards $N/(D \cdot P)$ allows us to gradually transition from a leveled LSM-tree into a sorted array (as eventually there is only one level). This design was not possible before, and it is beneficial for workloads with many range reads. In this way, the bridging process makes a continuum increasingly rich and powerful.

What is a new Data Structure? There are a number of open questions this work touches on. And some of these questions become even philosophical. For example, if all data structures can be described as combinations of a small set of design principles, then what constitutes a new data structure design? Given the vastness of the design space, we think that the discovery of any combination of design principles that brings new and interesting performance properties classifies as a new data structure. Historically, this has been the factor of recognizing new designs as worthy and interesting even if seemingly "small" differences separated them. For example, while an LSM-tree can simply be seen as a sequence of unmerged B-trees, the performance properties it brings are so drastically different that it has become its own category of study and whole systems are built around its basic design principles.

5. THE PATH TO SELF-DESIGN

Knowing which design is the best for a workload opens the opportunity for systems that can adapt on-the-fly. While adaptivity has been studied in several forms including adapting storage to queries [38, 4, 11, 46, 40, 27, 37, 60], the new opportunity is morphing among what is typically considered as fundamentally different designs, e.g., from an LSM-tree to a B^+tree, which can allow systems to gracefully adapt to a larger array of diverse workload patterns. Design continuums bring such a vision a small step closer because of two reasons: 1) they allow quickly computing the best data structure design (out of a set of possible designs), and 2) by knowing intermediate data structure designs that can be used as transition points in-between "distant" designs (among which it would otherwise be too expensive to transition).

There are (at least) three challenges on the way to such self-designing systems: a) designing algorithms to physically transition among any two designs, b) automatically materializing the needed code to utilize diverse designs, and c) resolving fundamental system design knobs beyond layout decisions that are hard to encapsulate in a continuum. Below we briefly touch on these research challenges, and we show hints that they are likely possible to be resolved.

Transitions. As in all adaptive studies, we need to consider the cost of a transition. The new challenge here is transitioning among fundamentally different designs. For example, assume a transition between a Leveled LSM-tree and B^+tree. Even though at a first glance these designs are vastly different, the design continuum helps us see possible efficient transitions; The difference in the specification of each structure on the design continuum indicates what we need to morph from one to the other. Specifically, between an LSM-tree and B^+tree, merging and fence pointers characterize the main design differences and so the transition policies should depend on these design principles. For example, one way to do such a transition is to wait until

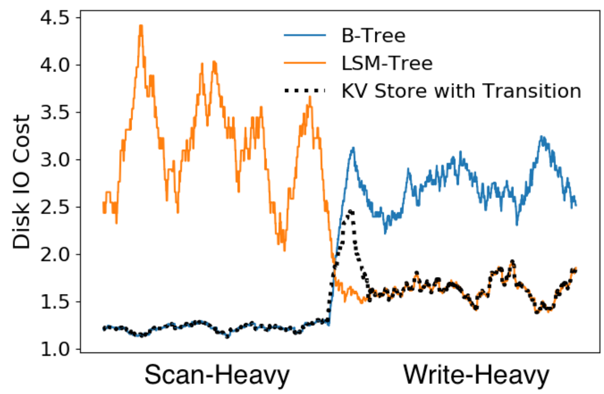


Figure 10: Potential benefit of on-the-fly transitions between B^+tree and LSM-tree.

the LSM-tree is in a state where all of the entries are at the bottom level and then build the B^+tree off of that level so that we don't have to copy the data (similar to how we build the internal index of B^+tree when we do bulk loading). Effectively waiting until merging is not a difference between the source and target design. A second option is to preemptively merge the levels of the LSM-tree into a single level so we can build the B^+tree off of that level without waiting for a natural merge to arrive. A third option is a compromise between the two: we can use the bottom level of the LSM-tree as the leaf layer of the B^+tree (avoiding copying the data) and then insert entries from the smaller levels of the B^+tree into the LSM-tree one by one.

The opposite transition, from a B^+tree to an LSM-tree, is also possible with the reverse problem that the scattered leaves of the B^+tree need to represent a contiguous run in an LSM-tree. To avoid a full write we can trick virtual memory to see these pages as contiguous [77]. The very first time the new LSM-tree does a full merge, the state goes back to physically contiguous runs.

Figure 10 depicts the potential of transitions. During the first 2000 queries, the workload is short-range scan heavy and thus favors B^+tree. During the next 2000 queries, the workload becomes write heavy, favoring LSM-Trees. While pure LSM-tree and pure B-tree designs fail to achieve globally good performance, when using transitions, we can stay close to the optimal performance across the whole workload. The figure captures the I/O behavior of these data structure designs and the transitions (in number of blocks). Overall, it is possible to do transitions at a smaller cost than reading and writing all data even if we transition among fundamentally different structures. The future path for the realization of this vision points to a **transition algebra**.

Code Generation. Tailored storage requires tailored code to get maximum performance [4]. The continuum provides the means towards such a path; since there exists a unifying model that describes the diverse designs in the continuum, this means that we can write a single generalized algorithm for each operation o that can instantiate the individual algorithm for operation o for each possible designs. For example, Algorithm 1 depicts such a generalized algorithm for the point lookup operation for the design continuum we presented in this paper.

Learning to go Beyond the Continuum. We expect that there will likely be critical design knobs that are very

```
PointLookup (searchKey)
    if M_B > E then
        entry := buffer.find(searchKey);
        if entry then
            return entry;
    // Pointer for direct block access.  Set to root.
    blockToCheck := levels[0].runs[0].nodes[0];
    for i ← 0 to L do
        // Check each level's runs from recent to oldest.
        for j ← 0 to levels[i].runs.count do
            /* Prune search using bloom filters and fences
               when available.                                  */
            if i < (L − Y) // At hot levels.
             then
                keyCouldExist :=
                  filters[i][j].checkExists(searchKey);
                if !keyCouldExist then
                    continue;
                else
                    blockToCheck :=
                      fences[i][j].find(searchKey);
            /* For oldest hot run, and all cold runs, if no
               entry is returned, then the search continues
               using a pointer into the next oldest run.    */
            entry, blockToCheck :=
              blockToCheck.find(searchKey);
            if entry then
                return entry;
    return keyDoesNotExist;
```

Algorithm 1: Lookup algorithm template for any design.

hard or even impossible to include in a well-constructed design continuum. The path forward is to combine machine learning with the design continuum. Machine learning is increasingly used to tune exposed tuning knobs in systems [3, 71]. The new opportunity here is the native combination of such techniques with the system design itself. For example, consider the critical decision of how much memory resources to allocate to the cache. What is hard about this decision is that it interacts in multiple ways with numerous other memory allocations that uniquely characterize a design (specifically the main memory buffer, the bloom filters, and the fence pointers in our design continuum) but it is also highly sensitive to the workload. However, we can use the generalized cost formulas of the continuum to derive formulas for the expected I/O savings if we increase the memory in any memory component. We can then use these estimates to implement a discrete form of stochastic gradient descent. Figure 11 shows an example of our results for a skewed workload where we tested two instances of our continuum, the Monkey design [23] which optimizes bloom filter allocation and Leveled LSM-tree design with fixed false positive ratio across all bloom filters. We evaluate all three gradients at every grid point along the simplex of simulated LSM-trees with constant total memory. We then overlay an arrow on top of the disk access contour plot pointing from the lowest gradient component to the highest gradient component (we move 8 bytes from one component to the other every time). Finally, for each grid location, the process follows the arrows until we either reach the edge of the simplex or a previously visited point. We then plot an orange dot. The yellow dot represents a global minimum found experimentally. Tests with numerous other workloads also indicate that although as expected the overall optimization problem is sometimes non-convex, we can usually reach a point close to the optimum. The net result is that design continuums can be blended with ML approaches to co-design a tailored system that both **knows** how to navigate a vast space of the design

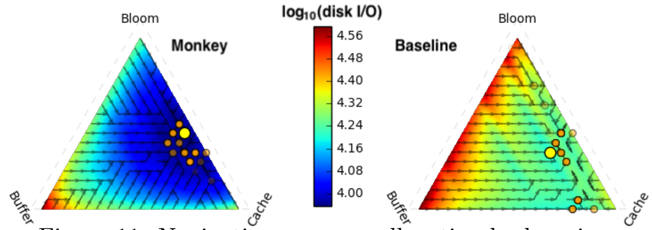


Figure 11: Navigating memory allocation by learning.

space and **learns** when needed to navigate design options that are hard to deterministically formulate how they will interact with the rest of the design.

6. NEXT STEPS

Research on data structures has focused on identifying the fundamentally best performance trade-offs. We envision a complementary line of future research to construct and improve on design continuums. The overarching goal is to flexibly harness our maturing knowledge of data structures to build more robust, diverse and navigable systems. Future steps include the construction of more and larger continuums, and especially the investigation of broader classes of data structure design, including graphs, spatial data, compression, replication as well as crucially more performance metrics such as concurrency, and adaptivity. The most challenging next step is whether the construction of design continuums itself can be (semi-) automated.

7. ACKNOWLEDGMENTS

Mark Callaghan has provided the authors with feedback and insights on key-value store design and industry practices repeatedly over the past few years and specifically for this paper. This work is supported by the National Science Foundation under grant IIS-1452595.

8. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. *ATC*, 2008.
[2] J.-S. Ahn, C. Seo, R. Mayuram, R. Yaseen, J.-S. Kim, and S. Maeng. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *TC*, 65(3):902–915, 2016.
[3] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. *SIGMOD*, 2017.
[4] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. *SIGMOD*, 2014.
[5] P. M. Aoki. Generalizing "Search" in Generalized Search Trees (Extended Abstract). *ICDE*, 1998.
[6] P. M. Aoki. How to Avoid Building DataBlades That Know the Value of Everything and the Cost of Nothing. *SSDBM*, 1999.
[7] Apache. Accumulo. *https://accumulo.apache.org/*.
[8] Apple. Foundationdb. *https://github.com/apple/foundationdb*, 2018.
[9] L. Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica*, 37(1):1–24, 2003.
[10] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: a Database Benchmark Based on the Facebook Social Graph. *SIGMOD*, 2013.
[11] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. *SIGMOD*, 2016.
[12] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. *EDBT*, 2016.
[13] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Proceedings of the ACM SIGFIDET Workshop on Data Description and Access*, 1970.
[14] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1(3):173–189, 1972.
[15] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-Oblivious Streaming B-trees. *SPAA*, 2007.

[16] G. S. Brodal and R. Fagerberg. Lower Bounds for External Memory Dictionaries. *SODA*, 2003.
[17] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. *PVLDB*, 8(2):161–172, 2014.
[18] Z. Cao, S. Chen, F. Li, M. Wang, and X. S. Wang. LogKV: Exploiting Key-Value Stores for Log Processing. *CIDR*, 2013.
[19] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. *SIGMOD*, 2018.
[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *OSDI*, 2006.
[21] D. Cohen and N. Campbell. Automating Relational Operations on Data Structures. *IEEE Software*, 10(3):53–60, 1993.
[22] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake Elastic Data Warehouse. *SIGMOD*, 2016.
[23] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. *SIGMOD*, 2017.
[24] N. Dayan, P. Bonnet, and S. Idreos. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. *SIGMOD*, 2016.
[25] N. Dayan and S. Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. *SIGMOD*, 2018.
[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Op. Sys. Rev.*, 41(6):205–220, 2007.
[27] J. Dittrich and A. Jindal. Towards a One Size Fits All Database Architecture. *CIDR*, 2011.
[28] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing Space Amplification in RocksDB. *CIDR*, 2017.
[29] Facebook. RocksDB. *https://github.com/facebook/rocksdb*.
[30] M. J. Franklin. *Caching and Memory Management in Client-Server Database Systems*. PhD thesis, University of Wisconsin-Madison, 1993.
[31] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling Concurrent Log-Structured Data Stores. *EuroSys*, 2015.
[32] Google. LevelDB. *https://github.com/google/leveldb/*.
[33] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data Representation Synthesis. *PLDI*, 2011.
[34] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Concurrent data representation synthesis. *PLDI*, 2012.
[35] HBase. Online reference. *http://hbase.apache.org/*, 2013.
[36] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. *VLDB*, 1995.
[37] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? *CIDR*, 2011.
[38] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. *CIDR*, 2007.
[39] S. Idreos, L. M. Maas, and M. S. Kester. Evolutionary Data Systems. *CoRR*, abs/1706.0, 2017.
[40] 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.
[41] S. Idreos, K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. The Periodic Table of Data Structures. *IEEE DEBULL*, 41(3):64–75, 2018.
[42] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. *SIGMOD*, 2018.
[43] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. *VLDB*, 1997.
[44] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: A Right-optimized Write-optimized File System. *FAST*, 2015.
[45] C. Jermaine, E. Omiecinski, and W. G. Yee. The Partitioned Exponential File for Database Storage Management. *VLDBJ*, 16(4):417–437, 2007.
[46] O. Kennedy and L. Ziarek. Just-In-Time Data Structures. *CIDR*, 2015.
[47] M. L. Kersten and L. Sidirourgos. A database system with amnesia. In *CIDR*, 2017.
[48] M. Kornacker. High-Performance Extensible Indexing. *VLDB*, 1999.
[49] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. *SIGMOD*, 1997.
[50] M. Kornacker, M. A. Shah, and J. M. Hellerstein. amdb: An Access Method Debugging Tool. *SIGMOD*, 1998.
[51] M. Kornacker, M. A. Shah, and J. M. Hellerstein. Amdb: A Design Tool for Access Methods. *IEEE DEBULL*, 26(2):3–11, 2003.
[52] D. Kossman. Systems Research - Fueling Future Disruptions. In *Keynote talk at the Microsoft Research Faculty Summit*, Redmond, WA, USA, aug 2018.
[53] T. Kraska, M. Alizadeh, A. Beutel, E. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *CIDR*, 2019.
[54] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. *SIGMOD*, 2018.
[55] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. *SIGOPS Op. Sys. Rev.*, 44(2):35–40, 2010.
[56] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. *ICDE*, 2013.
[57] 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.
[58] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. *NSDI*, 2014.
[59] LinkedIn. Voldemort. *http://www.project-voldemort.com*.
[60] Z. Liu and S. Idreos. Main Memory Adaptive Denormalization. *SIGMOD*, 2016.
[61] C. Loncaric, E. Torlak, and M. D. Ernst. Fast Synthesis of Fast Collections. *PLDI*, 2016.
[62] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. *FAST*, 2016.
[63] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. *VLDB*, 2002.
[64] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.
[65] Memcached. Reference. *http://memcached.org/*.
[66] MongoDB. Online reference. *http://www.mongodb.com/*.
[67] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. *ATC*, 1999.
[68] 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.
[69] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor. Magic: A VLSI Layout System. *DAC*, 1984.
[70] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. *ATC*, 2016.
[71] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-Driving Database Management Systems. *CIDR*, 2017.
[72] Redis. Online reference. *http://redis.io/*.
[73] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *PVLDB*, 10(13):2037–2048, 2017.
[74] S. M. Rumble, A. Kejriwal, and J. K. Ousterhout. Log-structured memory for DRAM-based storage. *FAST*, 2014.
[75] E. Schonberg, J. T. Schwartz, and M. Sharir. Automatic data structure selection in setl. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 197–210, 1979.
[76] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in setl programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, apr 1981.
[77] F. M. Schuhknecht, A. Jindal, and J. Dittrich. An experimental evaluation and analysis of database cracking. *VLDBJ*, 25(1):27–52, 2016.
[78] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. *SIGMOD*, 2012.
[79] O. Shacham, M. T. Vechev, and E. Yahav. Chameleon: Adaptive Selection of Collections. *PLDI*, 2009.
[80] J. Sheehy and D. Smith. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. *Basho White Paper*, 2010.
[81] Y. Smaragdakis and D. S. Batory. DiSTiL: A Transformation Library for Data Structures. *DSL*, 1997.
[82] Spotify. Sparkey. *https://github.com/spotify/sparkey*, 2014.
[83] M. J. Steindorfer and J. J. Vinju. Towards a Software Product Line of Trie-Based Collections. *GPCE*, 2016.
[84] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. *ICDE*, 2005.
[85] R. E. Tarjan. Complexity of Combinatorial Algorithms. *SIAM Review*, 20(3):457–491, 1978.
[86] R. Thonangi and J. Yang. On Log-Structured Merge for Solid-State Drives. *ICDE*, 2017.
[87] L. Wilkinson. *The Grammar of Graphics*. Springer-Verlag, 2005.
[88] WiredTiger. Source Code. *https://github.com/wiredtiger/wiredtiger*.
[89] C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein. Anna: A kvs for any scale. pages 401–412, 2018.
[90] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *TECS*, 6(3), 2007.
[91] S. B. Yao. An Attribute Based Model for Database Access Cost Analysis. *TODS*, 2(1):45–67, 1977.
[92] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. *SIGMOD*, 2018.