The Periodic Table of Data Structures

Stratos Idreos Kostas Zoumpatianos Manos Athanassoulis Niv Dayan Brian Hentschel
Michael S. Kester Demi Guo Lukas Maas Wilson Qin Abdul Wasay Yiyou Sun

Harvard University

Abstract

We describe the vision of being able to reason about the design space of data structures. We break this down into two questions: 1) Can we know all data structures that is possible to design? 2) Can we compute the performance of arbitrary designs on a given hardware and workload without having to implement the design or even access the target hardware? If those challenges are possible, then an array of exciting opportunities would become feasible such as interactive what-if design to improve the productivity of data systems researchers and engineers, and informed decision making in industrial settings with regards to critical hardware/workload/data structure design issues. Then, even fully automated discovery of new data structure designs becomes possible. Furthermore, the structure of the design space itself provides numerous insights and opportunities such as the existence of design continuums that can lead to data systems with deep adaptivity, and a new understanding of the possible performance tradeoffs. Given the universal presence of data structures at the very core of any data-driven field across all sciences and industries, reasoning about their design can have significant benefits, making it more feasible (easier, faster and cheaper) to adopt tailored state-of-the-art storage solutions. And this effect is going to become increasingly more critical as data keeps growing, hardware keeps changing and more applications/fields realize the transformative power and potential of data analytics. This paper presents this vision and surveys first steps that demonstrate its feasibility.

1 Automating Data Structure Design

Data structures is how we store and access data. A data structure design consists of 1) the data organization, 2) an optional index, and 3) the algorithms that support basic operations (e.g., put, get, update). All **algorithms** deal with data and most often their design starts by defining a data structure that will minimize computation and data movement. Effectively, the possible ways to design an algorithm (and whole data systems) are restricted by the data structure choices [31, 76, 77, 47, 24, 8, 1, 46]. For example, we can only utilize an optimized sorted search algorithm if the data is sorted and if we can maintain the data efficiently in such a state. In turn, a **data analytics** pipeline is effectively a complex collection of such algorithms that deal with increasingly growing amounts of data. Overall, data structures are one of the most fundamental components of computer science and broadly data science; they are at the core of all subfields, including data analytics and the tools and infrastructure needed to perform analytics, ranging from data systems (relational, NoSQL, graph), compilers, and networks, to storage for machine learning models, and practically any ad-hoc program that deals with data.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

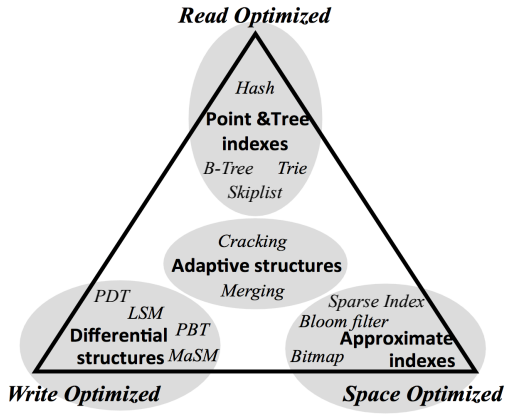

Figure 1: Each design compromises between read, update and memory amplification.

There is no perfect data structure design; each design is a compromise among the fundamental performance trade-offs [8]: read, update and memory amplification. This is depicted visually in Figure 1. In this way, to get good performance, the design of a data structure has to be tailored to the specific data, and query workload of the target application in addition to the underlying hardware where the data structure is expected to live.

A continuous challenge. Since the early days of computer science, the research community publishes 40-80 data structures per year. The pace has increased over the last decade because of the growth of data, the ever-increasing numbers of new data-driven applications, more fields moving to a computational paradigm where data analysis becomes critical, and hardware changing faster with characteristics that require a complete redesign of data structures and algorithms. Overall, with workloads and hardware changing frequently and rapidly, designing new data structures becomes a continuous challenge. Furthermore, for the several past decades, the trend in terms of hardware evolution is that computation becomes relatively faster than data movement; this makes data structure design even more critical as the way we store data dictates how much data an algorithm has to move.

The vast design space slows progress. There are so many different ways to design a data structure and so many moving targets that it has become a notoriously hard problem; it takes several months even for experts to design and optimize a new structure. Most often, the initial data structure decisions for a data system remain intact; it is too complex to redesign a data structure, predict what its impact would be as workload and hardware keep changing, implement it, and integrate it. For example, it is extremely hard to decide whether to acquire new hardware and how to adjust the core data structure design to it. Similarly, given the quick pace of adding new features in modern applications, the workload also changes rapidly; it is extremely hard to predict the potential performance impact and what possible design changes are best to allow for graceful adaptation by investing and timing engineering efforts accordingly. In the same lines, it is very hard to know when and why a data system would "break", i.e., when performance would drop below an acceptable threshold, so that we can invest in protecting against those scenarios and avoid prolonged degradation in performance or even down-time.

For fields without computer science/engineering expertise that move towards a data-driven paradigm, these low-level choices are impossible to make, and the only viable solution is using suboptimal off-the-shelf designs or spending more money in hiring experts. Modern data analytics scenarios and data science pipelines have many of those characteristics, i.e., domain experts with limited expertise on data structure design are confronted with dynamic, ever changing scenarios to explore and understand data. The likelihood that a single off-the-self data structure fits such arbitrary and sometimes unpredictable data access patterns is close to zero, yet data movement is the primary bottleneck as data grows and we do need more data to extract value out of data analytics.

The source of the problem is that, **there is no good way to predict the performance impact** of new workload, new hardware and new data structure design choices before fully implementing and testing them, and **there is no good way to be aware of all the possible designs**. We make three critical observations.

1. Each data structure design can be described as a set of design concepts. These are all low-level decisions that go into a given design such as using partitioning, pointers or direct addressing.

2. Each new data structure design can be classified in one of three ways; it is a set that includes: a) a new combination of existing design concepts, b) a new tuning of existing concepts or c) a new design concept.

3. By now we have invented so many fundamental design concepts such that most new designs fall within the first two categories, i.e., they are combinations or tunings of existing design ideas.

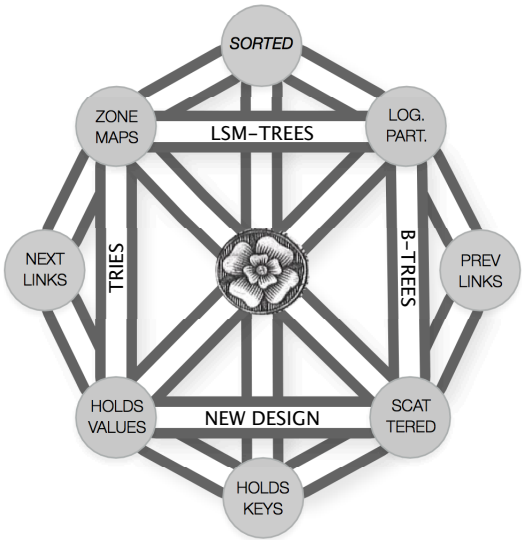

Figure 2: Reasoning about the design space of data structures by understanding its first principles and how they combine.

The design space structure is key. With most designs being combinations of existing design ideas, we argue that it is a great time to study in depth the design space itself to discover its structure and properties. Let us make an analogy to make it more clear why this is an important goal long term. Consider the periodic table of elements in chemistry. It organized existing elements into a universal structure, a design space formed based on their properties and their components, i.e., the atomic number and electron configuration. The crucial observation is that this design space not only classifies and explains connections among existing elements, but its structure (i.e., the gaps) provide hints about possible elements that had not been discovered yet, and it even helped argue about their expected properties. Even a hundred years after the original periodic table was conceived, scientists kept doing research driven by the gaps in the table, and kept discovering elements whose existence and properties had been predicted purely by the structure of the design space. Another example comes from computer science research on cache consistency algorithms in client-server environments. Mike Franklin's Ph.D. work led to a design space that mapped existing algorithms based on the common fundamental design decisions under which we could describe all these algorithms [21]. Similarly to the periodic table, once the structure was in place, we could classify existing algorithms, but also we could see additional algorithms that had not been invented yet; these new algorithms are formed as combinations of the design principles and the design space allows us to even argue about their properties.

Art vs. science. Just because a specific design is a combination of existing design concepts, it does not mean that it is easy to be conceived manually. In other words, when the number of options and possible design combinations is so big, it is hard even for experts to "see" the optimal solutions even when these solutions consist of existing concepts only. This is why many argue that research on algorithms and data structures is a form of art, i.e., driven by inspiration. Our goal is to accelerate this process and bring more structure to it so that inspiration is complemented by intuitive and interactive tools.

We set out to discover the first principles of data structure design and map the design space that they form. Our goal is to be able to **reason about their combinations**, tunings, and performance properties they bring. The high-level concept is depicted in Figure 2 using a small sample of principles for brevity. Effectively, the principles and their structure form a "grammar" with which we can describe any data structure in a systematic way, including designs that have not been discovered yet but they can be "calculated" from the design space given constraints such as the expected workload, and hardware environment. Figure 2 is inspired by a similar figure in the Ph.D. thesis of Gottfried Leibniz who in the 1600s envisioned an engine that calculates physical laws from a small set of primitives [48]. In the rest of this paper, we describe the process of discovering the **first principles of design**, we provide examples, and discuss the process of synthesizing arbitrary data structure designs. Similarly to the periodic table of elements in chemistry, the final overall structure, which we call **the periodic table of data structures**, provides insights and explanations on existing designs but it also points to exciting research opportunities. We then discuss how to automatically **synthesize the optimal algorithms** of the basic operations of a data structure based on the target workload and hardware using **an expert system and learned cost models**. Learned cost models represent fundamental access patterns (sorted search, random access, etc.) that can be combined into arbitrary algorithms. In addition, they allow computing the performance (response time) of a data structure design on a target workload and hardware without implementing and testing it. Finally, we discuss new opportunities that arise from the design space, i.e., discovering **design continuums** and making steps toward the ultimate goal of **automatic data structure design**.

2 The Quest for the First Principles of Data Structure Design

We define as a first principle a design concept that is not possible to break into more fine-grained concepts. For example, consider the design choice of linking two nodes of a data structure with a pointer. While there are potential tuning options (e.g., the size of the pointer), it is not possible to break this decision further; we either introduce a pointer or not.

Layout vs. access. We separate design principles that have to do with the layout of a data structure from those that have to do with how we access the data; layout drives the design of algorithms. Our intuition is that we can describe data structures solely by their layout, while the **access algorithms can be automatically derived** for a given set of layout decisions.

Thus, we focus on discovering, structuring and understanding the first principles of layout design. This is a trial and error process, i.e., there is no explicit algorithm. There is a **high-level recipe**, though. The recipe is about iterating between two processes: 1) decomposing existing data structures into first principles, and 2) stress testing the design space by attacking open problems through its structure. The second step is critical as many principles that were thought to be essential for specific subspaces of the design space can be optimized out, while new principles are being discovered for design decisions that remained fixed until now. The end goal is to develop a universal model that describes all structures as compositions of design principles. As more data structure instances are being considered, the set of first principles and the universal model change. This happens at a decreasing pace as by definition there are much fewer principles than data structure instances.

Arrays, B-Trees, and LSM-trees have been our primary playground for applying the above recipe as they cover a massive part of the overall design space. Arrays are practically part of all data structures (e.g., the nodes of a B-tree, the buckets of a hash-table). Arrays are also the fundamental storage model in modern analytical systems and large-scale cloud infrastructures where columnar storage is the leading paradigm. B-trees represent the evolution of tree-based structures, and they are the primary form of indexing on data systems for the past several decades to support read-intensive workloads. Finally, LSM-trees form the backbone of NoSQL data systems being a highly write-intensive design and a rather unexplored design space. Decomposing these three major classes of data structures and their variations into their first principles has led us to several surprising results that arise purely by the design space and the study of its structure.

For example, a study of access path selection in relational databases (i.e., dynamically choosing between a scan over an array and accessing a B-tree secondary index) shows that following the fixed selectivity threshold paradigm as was invented in the 1970s is suboptimal. After considering all design principles such as multi-core and SIMD parallelism, read and write patterns and costs, and shared reads, we showed that there should not be a fixed selectivity threshold for access path selection but rather a dynamic one that depends on the number of concurrent requests [39]. Similarly, the work on Smooth Scan shows that it is possible to combine these two fundamental access paths into a new hybrid one that combines their properties and requires no access path selection [12]. Finally, the work on Column Sketches showed that by considering the design principle of lossy compression, we can enable new performance properties for scans over an array that were not possible before, i.e., robust performance regardless of selectivity, data, and query properties [32].

Decomposing LSM-trees has revealed that all modern designs in industry are suboptimal due to fixing fundamental design principles to suboptimal decisions. For example, the work on Monkey shows that for a given memory budget the number of bits assigned to the bloom filter of each level (or run) should be exponentially smaller for bigger levels [18]. This allows for smaller false positive ratios for smaller levels and since each level contributes in the same way to the overall LSM-tree worst case false positive ratio, this tuning provides a drastic improvement that leads to orders of magnitude better performance. Similarly, the work on Dostoevsky shows that the merge policy should not be fixed across levels and that we can tune it in a different way for each level. This helps lower the bounds for writes without hurting reads and it also allows to tune the merge policy according to the desired low bounds for point reads, range reads and writes [19]. We have performed a similar process in other areas including bit vector structures [9] and structures for statistics computation [73].

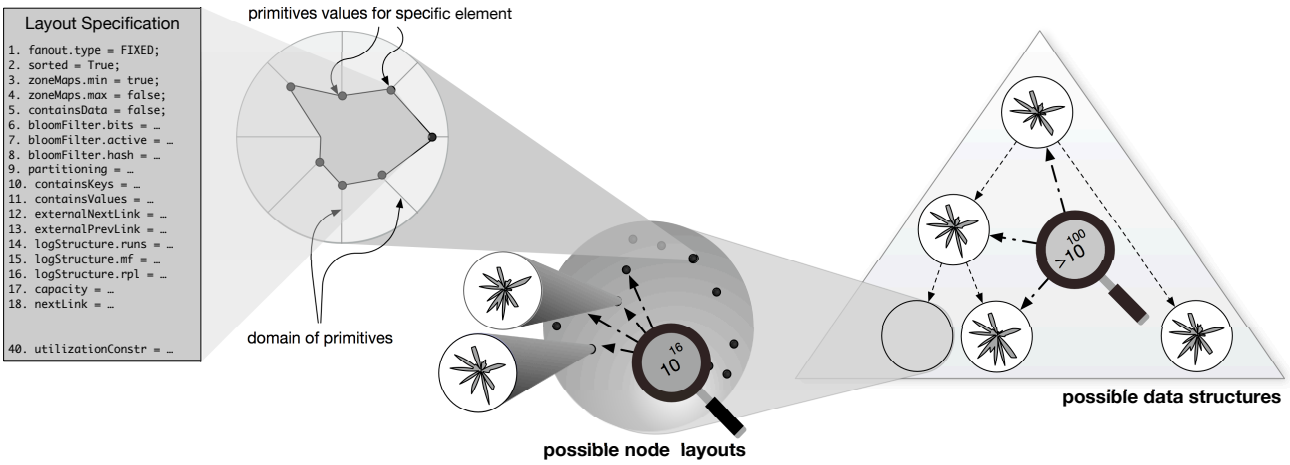

Figure 3: Synthesizing a vast number of data structure designs from first principles.

Examples of first principles include fundamental design concepts such as whether a given node of a structure contains keys or values. For example, an internal node of a B+tree does not contain any keys or values while a leaf node does. Furthermore, there are decisions that have to do with the physical layout. As an example, say that a node does contain keys and values; are these physically stored as key-value pairs (e.g., a row-based format) or as separate arrays (e.g., a columnar format)? A critical design principle is partitioning; does a node help break the domain into smaller pieces and, if yes, how exactly and what is the fanout? For example, we may use range partitioning, radix partitioning, temporal partitioning or any other function. If partitioning is enabled, then this means that each node generates sub-blocks; how do we access these sub-blocks? Through direct addressing or pointers? In addition, are the sub-blocks connected to each other with immediate links like a linked list or skip links like in a skip list? Further design principles have to do with filters. Do we have filters that help us access the sub-blocks of a node, e.g., bloom filters? And if we do, how many bits and how many hash functions should be used? Other types of filters are also possible such as zone maps. Furthermore, how are filters stored? Are they scattered across the sub-blocks or stored in a contiguous area? Then, there are design principles that have to do with how the design evolves outside the current node. For example, if the sub-blocks do not live inside the current node, then they form its children. Are these children stored scattered in memory, e.g., as the children of a B+tree node or they are stored in a contiguous block as the children of a cache-conscious B+tree node?

The overall model is based on iteratively describing one node of a data structure at a time. Figure 3 depicts a high-level view of this process (from left to right). Through the principles, we define data nodes as sets of design decisions. Their combinations craft the space of possible node descriptions. Then a data structure is described by connecting one or more node types and describing how they connect and potentially recurse. In practice, it is slightly more complex. With each set of principles, we describe a "logical block" of data that may be directly mapped onto an actual data structure node type, but it may also be a function we apply to the data. Most data structures in the literature consist of two logical blocks. For example, a B+Tree consists of two logical blocks, describing the internal and the leaf nodes respectively, while a hash table also consists of two logical blocks, describing the hash function and the hash buckets respectively. The initial set of layout principles covers the key-value model for read-only operations and appeared in the Data Calculator work [36]. The model allows the description of numerous well-known structures and their hybrids, including arrays, linked-lists, skip-lists, queues, hash-tables, binary trees and (cache-conscious) B-trees, tries, MassTree, and FAST.

What comes after crafting the design space is understanding and utilizing its structure. Given the sheer size of the design space, though, before we can do that effectively we need to be able to argue about any point in the space so we can quickly and automatically compare designs. Section 3 discusses cost synthesis which helps with this problem and then Section 4 describes existing results and opportunities in utilizing the design space.

3 Algorithm and Cost Synthesis from Learned Cost Models

To fully utilize the knowledge of the vast design space we need to be able to compare different designs in terms of the expected performance. Complexity analysis explains how the properties of a design scale with data but does not give a precise performance computation given a workload and hardware. On the other hand, full implementation and testing on the target workload and hardware provide the exact performance while the combination of both complexity analysis and implementation gives the complete picture. The problem is that with so many designs possible in the design space, it is not feasible to analyze, implement and test the numerous valid candidates. This is exactly the complex and time consuming process we are trying to avoid in the first place. A less explored option is building generalized hardware-conscious cost models [55, 76]. Similar to a full implementation, such models provide results very close to ground truth, and similar to complexity analysis they provide hints on how performance scales. However, building generalized models is nearly equally hard to a full implementation given the level of detail and hardware-conscious properties needed [39]. In addition, any change on hardware properties requires retuning the model. Overall, arguing formally about the performance of diverse data structures designs, especially as workload and hardware properties change, is a notoriously hard problem [55, 78, 72, 13, 76, 79, 39] and in our case we want to perform this task for numerous designs concurrently and at interactive response times.

In fact, before being able to argue about performance, we need to "complete" the design. The design space we crafted in the previous section is about the layout only, while a full design includes the access algorithms as well. We show in the Data Calculator [36] that algorithms can be built automatically by **synthesizing operations from fundamental access primitives** on blocks of data through a rule-based expert system. Rules are statements that map layouts to access patterns, e.g., if a data block is sorted we should use some kind of sorted search. However, the performance that we would get out of sorted search depends on 1) the exact algorithm (e.g., binary search, interpolation search), 2) the exact hardware, and 3) the exact engineering (e.g., using a new set of SIMD instructions). The rule base system gives us only a high-level sketch of the desired algorithm. Then, we synthesize the detailed algorithm and compute the cost via a hybrid of analytical models, implementation, and machine learning as combinations of a small set of fundamental access principles. We call these **learned cost models**. For each generic class of access pattern on a block of data (e.g., sorted search) there exist many instances of algorithms/implementations. These are short implementations that capture the exact access pattern. For example, for a binary search the implementation consists of an array of sorted integers where we perform a series of searches. During a training phase this code runs on the desired hardware and we learn a model as data scales (across the memory hierarchy). Each model captures the subtle performance details of diverse hardware settings and the exact engineering used to build the code for each access pattern. To make training easier, our models start out as analytical models since we know how these access patterns will likely behave.

Overall, after taking as input the layout of a design, a workload and a hardware, the Data Calculator goes step by step through the operations in the workload and for each operation it computes the expected state of the data structure at this point in the workload. This includes the number of nodes that exist and the number of elements in each node. For each operation it then computes the path that it should follow over the data structure by using the specification of its layout. This results in a set of nodes that should be visited. For each node, the expert system then provides the high level access pattern that should be used, e.g., probe a bloom filter, scan, sorted search. Once the high-level access pattern is known, the system then scans all possible models for particular implementations that exist in its library and chooses the one that is expected to behave the best on the target hardware and data size. Putting everything together, by repeating this process for all nodes in the path of an operation we can compute the total cost of this operation and by repeating this for all operations in a workload we get the overall cost. The exact process includes further low-level details to handle random access, caching and skew effects which are all handled by a combination of additional models and rules in the expert system for cost synthesis. As a first step, the Data Calculator work [36] shows how the above process takes place for bulk-loading, point, and range queries.

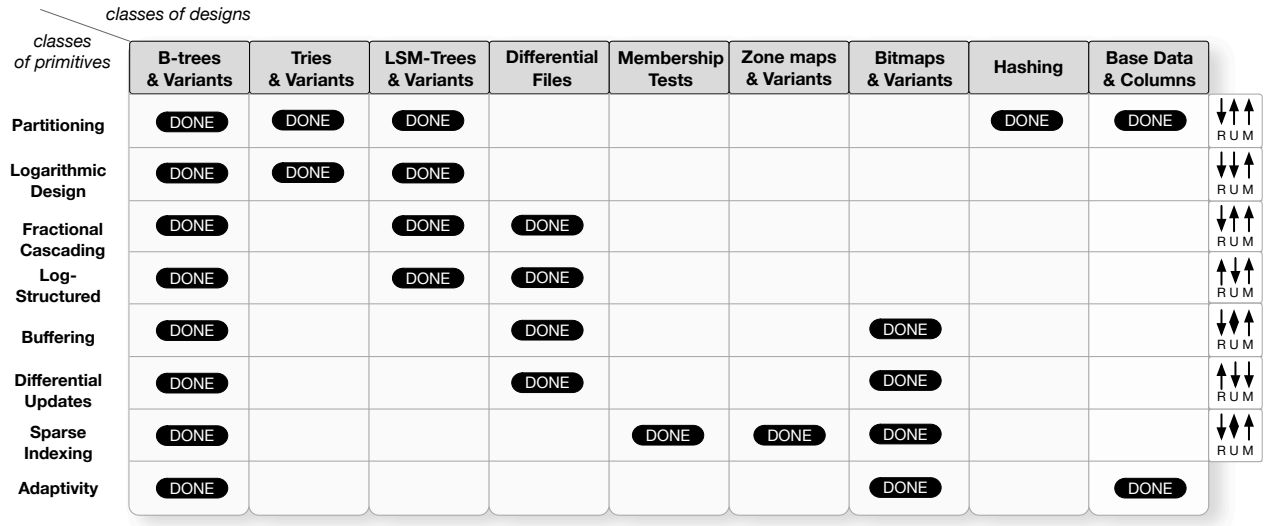

Figure 4: The periodic table of data structures: a vast unexplored design space.

4 The Periodic Table of Data Structures

In the same way that the arithmetic calculator can be used in any application where numbers are necessary (e.g., from complex scientific problems to restaurant bills), we expect that the Data Calculator will be useful in any application where data is essential. Here we describe a sample of exciting results, applications, and opportunities that have to do both with the structure of the design space, and with the ability to understand and navigate it.

4.1 Structure, design guides, gaps, and facts

Figure 4 depicts our first effort to unify known classes of data structures under a unified model. The table groups data structures and design primitives into classes. Every cell indicates whether any such principle has been applied to a category of data structures (i.e., a relevant paper has been published). The observation is that most of the cells are in fact empty. While this does not necessarily mean that these combinations have not been studied before, and it does not even suggest that they will all be useful for modern applications and hardware, it is still surprising. Also since this is a "zoomed out" table it loses some detail for purposes of visualization and thus even if a cell is marked as done, this does not mean that further work is not possible. By counting the exact possible designs from the detailed design space presented in the Data Calculator work [36], we find that there are at least 10^{32} two-node structures (and 10^{48} three-node ones). This is more possible designs than stars on the sky (10^{24}). We have barely published a couple of thousand of designs in the last sixty years, and at a rate of a hundred designs a year (as indicated by data from DBLP) we are moving very slowly.

Furthermore, the structure of the design space helps to predict how specific choices may affect the properties of a design with respect to read, update, and memory amplification. This is shown on the right-hand side of the table on Figure 4. Such hints can be used as a design guide by engineers and researchers (and even auto design algorithms) to accelerate design space exploration during the design process.

4.2 Interactive Design to Accelerate Engineering and Research

When combining the design space knowledge with the ability to do algorithm and cost synthesis we are presented with a unique and new opportunity for interactive design. That is, enabling researchers and engineers to quickly iterate over different designs without having to implement, test and even access the target hardware. For example, one can give as input a high-level specification of a data structure design, a sample workload, and the target hardware. The Data Calculator can then compute the performance we would get if we were to

implement this design and test it on this workload and hardware [36]. This computation takes a few seconds to complete, allowing researchers and engineers to **quickly iterate over: 1) multiple designs** to decide which one to spend time implementing or how specific changes may affect performance in an existing design; **2) multiple workloads** to inspect how envisioned changes in the workload (e.g., due to new application features) may affect the performance; **3) multiple hardware environments** to decide which is the best hardware for placement in a large cluster or to drive equipment purchase decisions by knowing the expected outcome on performance given the current design and workload. In our most recent results, such computations take in the order of 20 seconds for 1 Million data entries and 100 queries for a diverse class of structures including B-trees, Tries, and Hash-tables [36]. At the same time, training for typical modern hardware takes in the order of 50-100 seconds, and it is a one time process; once it is done, one can try out any workload and design on this hardware.

When it comes to interactive design a crucial question is that of **extensibility**. What if one wants to test a new access pattern that is not included in the Data Calculator? For example, say a researcher comes up with a new way to perform sorted search and wants to know the expected performance impact on their B-tree design. The new idea may be an algorithmic one or even a new hardware-conscious way of implementing an existing high-level algorithm, e.g., using a new set of SIMD instructions. All the researcher has to do is add a new benchmark under the sorted search family of access patterns and train it for the desired hardware. Then this becomes permanently part of the library of access patterns which has two positive side-effects: 1) the researcher can now test and refine their design with and without the new access pattern in a variety of workloads and hardware settings before initiating a complex implementation phase, and 2) the Data Calculator can now consider the new access pattern for any data structure design that includes sorted data (e.g., the buckets of a hash-table); in this way, future queries on the Data Calculator by this or other researchers in the same team can benefit by an expanded design space. On the other hand, extending the design space of layout principles requires expansion of the universal model and the associated rules for algorithm and cost synthesis. This is not a task that can be done interactively by users; it is the ongoing research effort to develop the theory behind such tools.

4.3 Design Continuums and Self-designing Systems

We define as design continuum a part of the design space of data structures that can be seen as a unified space because it is defined by the same set of first principles. Thus, there is a path between any two designs in this space by tuning the same set of principles. While tuning has always been possible and critical for data structures (e.g., tuning the fanout of a B-tree), the new opportunity here is that by structuring and understanding the design space, we can see such continuums across classes of data structures that are traditionally considered fundamentally different. For example, in our recent key-value store work we showed that there is a design continuum between logs, sorted arrays and LSM-trees [18, 19]. At the one end of the continuum exists a log, i.e., a write optimized structure where we can simply append new pages without having to maintain any order. Reads of course are slow as they require a full scan. On the other end of the spectrum exists a sorted array where we can quickly search with a binary search but writes require moving on average half the data. In between these two extremes exist LSM-trees and variations which give a balance between read and write costs. We showed that we can tune the shape of the structure **anywhere between a log and a sorted array** by tuning the merge policy of an LSM-tree. Specifically, an infinite size ratio and a tiering policy turn an LSM-tree into a log, while when using leveling and again infinite size ratio we get a sorted array. As we give different values to the size ratio and vary the merge policy for different levels, we get data structure designs which occupy points within the design continuum, can be seen as LSM-tree hybrids, and allow for custom read/write ratios and resource management [18, 19].

The discovery of design continuums opens the door for a new form of "deep adaptivity" by being able to "transition" between different designs. We term such systems **self-designing systems**, i.e., systems that can drastically morph their shape at run-time. Contrary to existing work on topics such as adaptive indexing which allows on-the-fly data adaptation, self-designing systems can transition across designs that are perceived as

fundamentally different (e.g., from log to LSM-tree to sorted array [18, 19]) and thus enabling a system to assume a much larger set of performance properties.

4.4 Automated Design: Computing Instead of Inventing

The ultimate dream has always been fully automated design [71, 76, 13]. We redefine this challenge here to include hardware; given a hardware and a workload, can we compute the best data structure design? The design space gives us a clear set of possible designs. In turn, algorithm and cost synthesis give us the ability to rank designs for a given workload and hardware. Together these two features allow the design of algorithms that traverse the design space, turning data structure design into a search problem. Early results have demonstrated that dynamic programming algorithms, searching part of the design space only, can automatically design custom data structures that fit the workload exactly [36]. What is even more intriguing is that these designs are new, i.e., they have not been published in the research literature, yet they stem from the design space as combinations of existing design principles and they match workloads better than existing state-of-the-art human-made designs. In another example, we showed that the search space can also be traversed via genetic algorithms, automatically designing column-store/row-store hybrids [35]. Further usages of the search capability include **completing partial designs**, and **inspecting existing designs** in terms of how far away they are from the optimal [36]. In addition, it is not necessary that the ultimate design should always be the target. For example, returning the top ten designs allows engineering teams to make decisions based on performance as well as implementation complexity, maintainability and other desirable properties.

The grant challenge is building algorithms that can traverse the whole design space. Given the exponential nature of the problem, we have started experimenting with reinforcement learning and Bayesian optimization with early results indicating that they can search a larger part of the design space within the same time budget.

5 Inspiration and Related Work

Our work is inspired by several lines of work across many fields of computer science. John Ousterhout's project Magic in the area of computer architecture allows for quick verification of transistor designs so that engineers can easily test multiple designs [58]. Leland Wilkinson's "grammar of graphics" provides structure and formulation on the massive universe of possible graphics one can design [74]. Mike Franklin's Ph.D. thesis explores the possible client-server architecture designs using caching based replication as the main design primitive and proposes a taxonomy that produced both published and unpublished (at the time) cache consistency algorithms [21]. Joe Hellerstein's work on Generalized Search Indexes [31, 6, 43, 42, 44, 45] makes it easy to design and test new data structures by providing templates that significantly minimize implementation time. S. Bing Yao's work on generalized cost models [76] for database organizations, and Stefan Manegold's work on generalized cost models tailored for the memory hierarchy [54] showed that it is possible to synthesize the costs of database operations from basic access patterns and based on hardware performance properties. Work on data representation synthesis in programming languages [63, 64, 17, 68, 66, 28, 29, 52, 69, 53] enables selection and synthesis of representations out of small sets of (3-5) existing data structures.

Work on tuning [37, 15] and adaptive systems is also relevant as conceptually any adaptive technique tunes along part of the design space. For example, work on hybrid data layouts and adaptive indexing automates selection of the right layout [7, 3, 27, 33, 20, 65, 4, 51, 18, 62, 25, 60, 80, 34, 38, 67]. Similarly works on tuning via experiments [10], learning [5], and tuning via machine learning [2, 30] can adapt parts of a design using feedback from tests. Further, the Data Calculator shares concepts with the stream of work on modular systems: in databases for easily adding data types [22, 23, 56, 57, 70] with minimal implementation effort, or plug and play features and whole system components with clean interfaces [50, 49, 16, 11, 14, 40, 61, 75], as well as in software engineering [59], computer architecture [58], and networks [41].

Research on new data structures keeps pushing the possible design space further. For example, an exciting idea is that of Learned Indexes which drastically expands the possible design space by proposing mixing machine learning models and traditional data structures; among other benefits, it enables new opportunities for succinct, yet performant, data structures [46]. Our work is complementary to such efforts as it is not toward expanding the design space, but rather toward mapping, understanding and navigating it. The Data Calculator can be seen as a step toward the Automatic Programmer challenge set by Jim Gray in his Turing award lecture [26], and as a step toward the "calculus of data structures" challenge set by Turing award winner Robert Tarjan [71]: *"What makes one data structure better than another for a certain application? The known results cry out for an underlying theory to explain them."*

6 Summary and Future Steps

We show that it is possible to argue about the design space of data structures. By discovering the first principles of the design of data structures and putting them in a universal model, we study their combinations and their impact on performance. We show that it is possible to accelerate research and decision making concerning data structure design, hardware, and workload by being able to quickly compute the performance impact of a vast number of designs; several orders of magnitude more designs than what has been published during the last six decades. Critical future steps include 1) expanding the supported design space (e.g., updates, concurrency, graphs, spatial), 2) supporting scalable and fast cost synthesis to allow for self-designing systems that change their shape at run-time, and 3) navigating the data structure design space by modeling it as a search, optimization and learning problem.

References

[1] D. J. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 2013.

[2] D. V. Aken, A. Pavlo, G. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. *SIGMOD*, 2017.

[3] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. *SIGMOD*, 2014.

[4] V. Alvarez, F. Schuhknecht, J. Dittrich, and S. Richter. Main Memory Adaptive Indexing for Multi-Core Systems. *DAMON*, 2014.

[5] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A Data System for Feature Engineering. *CIDR*, 2013.

[6] P. Aoki. Generalizing "Search" in Generalized Search Trees (Extended Abstract). *ICDE*, 1998.

[7] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. *SIGMOD*, 2016.

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

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

[10] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated Experiment-Driven Management of (Database) Systems. *HotOS*, 2009.

[11] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An Extensible Database Management System. *Transactions on Software Engineering*, 1988.

[12] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser. Smooth Scan: Statistics-Oblivious Access Paths. *ICDE*, 2015.

[13] A. Cardenas. Evaluation and selection of file organization - A model and system. *Commun. ACM*, 1973.

[14] M. Carey and D. DeWitt. An Overview of the EXODUS Project. *Data Eng. Bul.*, 1987.

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

[16] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. *VLDB*, 2000.

[17] D. Cohen and N. Campbell. Automating relational operations on data structures. *Software*, 1993.

[18] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. *SIGMOD*, 2017.

[19] 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.

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

[21] M. J. Franklin. Caching and Memory Management in Client-Server Database Systems. *Ph.D. Univ. of Wisconsin-Madison*, 1993.

[22] D. Goldhirsch and J. Orenstein. Extensibility in the PROBE Database System. *Data Eng. Bul.*, 1987.

[23] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE*, 1994.

[24] G. Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4), 2011.

[25] G. Graefe, F. Halim, S. Idreos, H. Kuno, S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 2012.

[26] J. Gray. What Next? A Few Remaining Problems in Information Technology. *SIGMOD Dig. Symp. Coll.*, 2000.

[27] R. Hankins and J. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. *VLDB*, 2003.

[28] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data Representation Synthesis. *PLDI*, 2011.

[29] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent data representation synthesis. *PLDI*, 2012.

[30] M. Heimel, M. Kiefer, and V. Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. *SIGMOD*, 2015.

[31] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. *VLDB*, 1995.

[32] B. Hentschel, M. Kester, and S. Idreos. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. *SIGMOD*, 2018.

[33] S. Idreos, M. Kersten, and S. Manegold. Database Cracking. *CIDR*, 2007.

[34] S. Idreos, M. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-Stores. *SIGMOD*, 2009.

[35] S. Idreos, L. Maas, and M. Kester. Evolutionary data systems. *CoRR*, abs/1706.05714, 2017.

[36] S. Idreos, K. Zoumpatianos, B. Hentschel, M. Kester, and D. Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. *SIGMOD*, 2018.

[37] Y. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. *SIGMOD*, 1987.

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

[39] M. Kester, M. Athanassoulis, and S. Idreos. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? *SIGMOD*, 2017.

[40] Y. Klonatos, C. Koch, T. Rompf, H. Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB'14*

[41] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click Modular Router. *TOCS*, 2000.

[42] M. Kornacker. High-Performance Extensible Indexing. *VLDB*, 1999.

[43] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and Recovery in Generalized Search Trees. *SIGMOD'97*

[44] M. Kornacker, M. Shah, and J. Hellerstein. amdb: An Access Method Debugging Tool. *SIGMOD*, 1998.

[45] M. Kornacker, M. Shah, and J. Hellerstein. Amdb: A Design Tool for Access Methods. *Data Eng. Bul.*, 2003.

[46] T. Kraska, A. Beutel, E. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. *SIGMOD*, 2018.

[47] T. Lehman, M. Carey. A Study of Index Structures for Main Memory Database Management Systems. *VLDB*, 1986.

[48] G. Leibniz. Dissertation on the art of combinations. *PhD Thesis, Leipzig University*, 1666.

[49] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 2013.

[50] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. *ICDE*, 2013.

[51] Z. Liu and S. Idreos. Main Memory Adaptive Denormalization. *SIGMOD*, 2016.

[52] C. Loncaric, E. Torlak, and M. Ernst. Fast Synthesis of Fast Collections. *PLDI*, 2016.

[53] C. Loncaric, E. Torlak, and M. Ernst. Generalized Data Structure Synthesis. *ICSE*, 2018.

[54] S. Manegold. Understanding, modeling, and improving main-memory database performance. *Ph.D. UVA*, 2002.

[55] S. Manegold, P. Boncz, M. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. *VLDB*, 2002.

[56] J. McPherson and H. Pirahesh. An Overview of Extensibility in Starburst. *Data Eng. Bul.*, 1987.

[57] S. Orborn. Extensible Databases and RAD. *Data Eng. Bul.*, 10(2), 1987.

[58] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G. Taylor. Magic: A VLSI Layout System. *DAC*, 1984.

[59] D. Parnas. Designing Software for Ease of Extension and Contraction. *TSE*, 1979.

[60] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. *SIGMOD*, 2015.

[61] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - A vector algebra for portable database performance on modern hardware. *PVLDB*, 2016.

[62] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. Kersten. Database cracking: fancy scan, not poor man's sort! *DAMON*, 2014.

[63] E. Schonberg, J. Schwartz, and Sharir. Automatic data structure selection in SETL. *POPL*, 1979.

[64] E. Schonberg, J. Schwartz, and Sharir. An automatic technique for selection of data representations in SETL programs. *Transactions on Programming Languages and Systems*, 3(2), 1981.

[65] F. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 2013.

[66] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive Selection of Collections. *PLDI*, 2009.

[67] D. Sleator and R. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 1985.

[68] Y. Smaragdakis and D. Batory. Distil: A transformation library for data structures. *DSL*, 1997.

[69] M. Steindorfer and J. Vinju. Towards a Software Product Line of Trie-Based Collections. *GPCE*, 2016.

[70] M. Stonebraker, J. Anton, and M. Hirohama. Extendability in POSTGRES. *Data Eng. Bul.*, 1987.

[71] R. Tarjan. Complexity of combinatorial algorithms. *SIAM Rev*, 1978.

[72] T. Teorey and K. Das. Application of an analytical model to evaluate storage structures. *SIGMOD*, 1976.

[73] A. Wasay, X. Wei, N. Dayan, S. Idreos. Data Canopy: Accelerating Exploratory Statistical Analysis. *SIGMOD'17*.

[74] L. Wilkinson. The grammar of graphics. *Springer*, 2005.

[75] H. Xing, S. Floratos, S. Blanas, S. Byna, Prabhat, K. Wu, and P. Brown. ArrayBridge: Interweaving declarative array processing in SciDB with imperative HDF5-based programs. *ICDE*, 2018.

[76] S. Yao. An Attribute Based Model for Database Access Cost Analysis. *TODS*, 1977.

[77] S. Yao and D. DeJong. Evaluation of Database Access Paths. *SIGMOD*, 1978.

[78] S. Yao and A. Merten. Selection of file organization using an analytic model. *VLDB*, 1975.

[79] M. Zhou. Generalizing database access methods. *Ph.D. Thesis. University of Waterloo*, 1999.

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