# The Data Calculator*: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models

Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, Demi Guo
Harvard University

## ABSTRACT

Data structures are critical in any data-driven scenario, but they are notoriously hard to design due to a massive design space and the dependence of performance on workload and hardware which evolve continuously. We present a design engine, the Data Calculator, which enables interactive and semi-automated design of data structures. It brings two innovations. First, it offers a set of fine-grained design primitives that capture the first principles of data layout design: how data structure nodes lay data out, and how they are positioned relative to each other. This allows for a structured description of the universe of possible data structure designs that can be synthesized as combinations of those primitives. The second innovation is computation of performance using learned cost models. These models are trained on diverse hardware and data profiles and capture the cost properties of fundamental data access primitives (e.g., random access). With these models, we synthesize the performance cost of complex operations on arbitrary data structure designs without having to: 1) implement the data structure, 2) run the workload, or even 3) access the target hardware. We demonstrate that the Data Calculator can assist data structure designers and researchers by accurately answering rich what-if design questions on the order of a few seconds or minutes, i.e., computing how the performance (response time) of a given data structure design is impacted by variations in the: 1) design, 2) hardware, 3) data, and 4) query workloads. This makes it effortless to test numerous designs and ideas before embarking on lengthy implementation, deployment, and hardware acquisition steps. We also demonstrate that the Data Calculator can synthesize entirely new designs, auto-complete partial designs, and detect suboptimal design choices.

*Let us calculate.* —Gottfried Leibniz

**ACM Reference Format:**
Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18)*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3199671


*SIGMOD'18, June 10–15, 2018, Houston, TX, USA*

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3199671

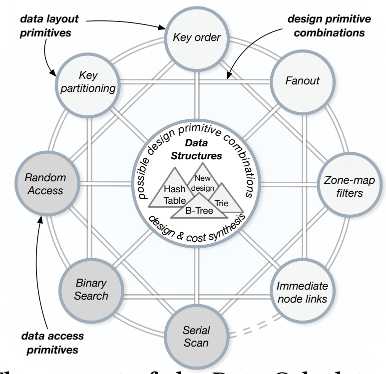


**Figure 1: The concept of the Data Calculator: computing data access method designs as combinations of a small set of primitives. (Drawing inspired by a figure in the Ph.D. thesis of Gottfried Leibniz who envisioned an engine that calculates physical laws from a small set of primitives [52].)**

## 1 FROM MANUAL TO INTERACTIVE DESIGN

**The Importance of Data Structures.** Data structures are at the core of any data-driven software, from relational database systems, NoSQL key-value stores, operating systems, compilers, HCI systems, and scientific data management to any ad-hoc program that deals with increasingly growing data. Any operation in any data-driven system/program goes through a data structure whenever it touches data. Any effort to rethink the design of a specific system or to add new functionality typically includes (or even begins by) rethinking how data should be stored and accessed [1, 9, 33, 38, 51, 75, 76]. In this way, the design of data structures has been an active area of research since the onset of computer science and there is an ever-growing need for alternative designs. This is fueled by 1) the continuous advent of new applications that require tailored storage and access patterns in both industry and science, and 2) new hardware that requires specific storage and access patterns to ensure longevity and maximum utilization. Every year dozens of new data structure designs are published, e.g., more than fifty new designs appeared at ACM SIGMOD, PVLDB, EDBT and IEEE ICDE in 2017 according to data from DBLP.

**A Vast and Complex Design Space.** A data structure design consists of 1) a data layout to describe how data is stored, and 2) algorithms that describe how its basic functionality (search, insert, etc.) is achieved over the specific data layout. A data structure can be as simple as an array or arbitrarily complex using sophisticated combinations of hashing, range and radix partitioning, careful data placement, compression and encodings. Data structures may also be referred to as "data containers" or "access methods" (in which case the term "structure" applies only to the layout). The data layout

*The name "Calculator" pays tribute to the early works that experimented with the concept of calculating complex objects from a small set of primitives [52].

design itself may be further broken down into the base data layout and the indexing information which helps navigate the data, i.e., the leaves of a B+tree and its inner nodes, or buckets of a hash table and the hash-map. We use the term data structure design throughout the paper to refer to the overall design of the data layout, indexing, and the algorithms together as a whole.

We define “design” as the set of all decisions that characterize the layout and algorithms of a data structure, e.g., “Should data nodes be sorted?”, “Should they use pointers?”, and “How should we scan them exactly?”. The number of possible valid data structure designs explodes to $\gg 10^{32}$ even if we limit the overall design to only two different kinds of nodes (e.g., as is the case for B+trees). If we allow every node to adopt different design decisions (e.g., based on access patterns), then the number of designs grows to $\gg 10^{100}$.[1] We explain how we derive these numbers in Section 2.

**The Problem: Human-Driven Design Only.** The design of data structures is a slow process, relying on the expertise and intuition of researchers and engineers who need to mentally navigate the vast design space. For example, consider the following design questions.

(1) We need a data structure for a specific workload: Should we strip down an existing complex data structure? Should we build off a simpler one? Or should we design and build a new one from scratch?
(2) We expect that the workload might shift (e.g., due to new application features): How will performance change? Should we redesign our core data structures?
(3) We add flash drives with more bandwidth and also add more system memory: Should we change the layout of our B-tree nodes? Should we change the size ratio in our LSM-tree?
(4) We want to improve throughput: How beneficial would it be to buy faster disks? more memory? or should we invest the same budget in redesigning our core data structure?

This complexity leads to a slow design process and has severe cost side-effects [12, 22]. Time to market is of extreme importance, so new data structure design effectively stops when a design “is due” and only rarely when it “is ready”. Thus, the process of design extends beyond the initial design phase to periods of reconsidering the design given bugs or changes in the scenarios it should support. Furthermore, this complexity makes it difficult to predict the impact of design choices, workloads, and hardware on performance. We include two quotes from a systems architect with more than two decades of experience with relational systems and key-value stores.

*(1) “I know from experience that getting a new data structure into production takes years. Over several years, assumptions made about the workload and hardware are likely to change, and these changes threaten to reduce the benefit of a data structure. This risk of change makes it hard to commit to multi-year development efforts. We need to reduce the time it takes to get new data structures into production.”*

*(2) “Another problem is the limited ability we have to iterate. While some changes only require an online schema change, many require a dump and reload for a data service that might be running 24x7. The budget for such changes is limited. We can overcome the limited budget with tools that help us determine the changes most likely to be useful. Decisions today are frequently based on expert opinions, and these experts are in short supply.”*

[1]For comparison, the estimated number of stars in the universe is $10^{24}$.

**Vision Step 1: Design Synthesis from First Principles.** We propose a move toward the new design paradigm captured in Figure 1. Our intuition is that most designs (and even inventions) are about combining a small set of fundamental concepts in different ways or tunings. If we can describe the set of the first principles of data structure design, i.e., the core design principles out of which all data structures can be drawn, then we will have a structured way to express all possible designs we may invent, study, and employ as combinations of those principles. An analogy is the periodic table of elements in chemistry. It classifies elements based on their atomic number, electron configuration, and recurring chemical properties. The structure of the table allows one to understand the elements and how they relate to each other but crucially it also enables arguing about the possible design space; more than one hundred years since the inception of the periodic table in the 18th century, we keep discovering elements that are predicted (synthesized) by the “gaps” in the table, accelerating science.

**Our vision is to build the periodic table of data structures** so we can express their massive design space. We take the first step in this paper, presenting a set of first principles that can synthesize orders of magnitude more data structure designs than what has been published in the literature. It captures basic hardware conscious layouts and read operations; future work includes extending the table for additional parts of the design space, such as updates, concurrency, compression, adaptivity, and security.

**Vision Step 2: Cost Synthesis from Learned Models.** The second step in our vision is to accelerate and automate the design process. Key here, is being able to argue about the performance behavior of the massive number of designs so we can rank them. Even with an intuition that a given design is an excellent choice, one has to implement the design, and test it on a given data and query workload and onto specific hardware. This process can take weeks at a time and has to be repeated when any part of the environment changes. Can we accelerate this process so we can quickly test alternative designs (or different combinations of hardware, data, and queries) on the order of a few seconds? If this is possible, then we can 1) accelerate design and research of new data structures, and 2) enable new kinds of adaptive systems that can decide core parts of their design, and the right hardware.

Arguing formally about the performance of diverse designs is a notoriously hard problem [13, 58, 72, 75, 77, 78] especially as workload and hardware properties change; even if we can come up with a robust analytical model it may soon be obsolete [43]. We take a hybrid route using a combination of analytical models, benchmarks, and machine learning for a small set of fundamental access primitives. For example, all pointer based data structures need to perform random accesses as operations traverse their nodes. All data structures need to perform a write during an update operation, regardless of the exact update strategy. We synthesize the cost of complex operations out of models that describe those simpler more fundamental operations inspired by past work on generalized models [58, 75]. In addition, our models start out as analytical models since we know how these primitives will likely behave. However, they are also trained across diverse hardware profiles by running benchmarks that isolate the behavior of those primitives. This way, we learn a set of coefficients for each model that capture the subtle performance details of diverse hardware settings.

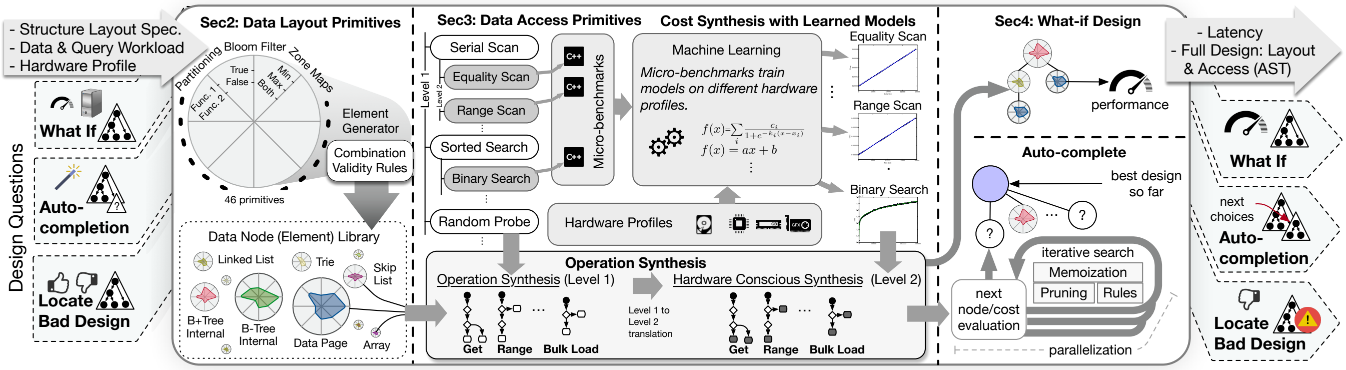


**Figure 2: The architecture of the Data Calculator: From high-level layout specifications to performance cost calculation.**

**The Data Calculator: Automated What-if Design.** We present a "design engine" – the Data Calculator – that can compute the performance of arbitrary data structure designs as combinations of fundamental design primitives. It is an interactive tool that accelerates the process of design by turning it into an exploration process, improving the productivity of researchers and engineers; it is able to answer what-if data structure design questions to understand how the introduction of new design choices, workloads, and hardware affect the performance (latency) of an existing design. It currently supports read queries for basic hardware conscious layouts. It allows users to give as input a high-level specification of the layout of a data structure (as a combination of primitives), in addition to workload, and hardware specifications. The Data Calculator gives as output a calculation of the latency to run the input workload on the input hardware. The architecture and components of the Data Calculator are captured in Figure 2 (from left to right): (1) a library of fine-grained data layout primitives that can be combined in arbitrary ways to describe data structure layouts; (2) a library of data access primitives that can be combined to generate designs of operations; (3) an operation and cost synthesizer that computes the design of operations and their latency for a given data structure layout specification, a workload and a hardware profile, and (4) a search component that can traverse part of the design space to supplement a partial data structure specification or inspect an existing one with respect to both the layout and the access design choices.

**Inspiration.** 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 [62]. 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. Joe Hellerstein's work on Generalized Search Indexes [6, 7, 38, 47–50] 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 [75] for database organizations, and Stefan Manegold's work on generalized cost models tailored for the memory hierarchy [57] 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 [15, 18–21, 24–27] enables selection and synthesis of representations out of small sets of (3-5) existing data structures. The Data Calculator can be seen as a step toward the Automatic Programmer challenge set by Jim Gray in his Turing award lecture [35], 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."*

**Contributions.** Our contributions are as follows:

(1) We introduce a set of data layout design primitives that capture the first principles of data layouts including hardware conscious designs that dictate the relative positioning of data structure nodes (§2).
(2) We show how combinations of the design primitives can describe known data structure designs, including arrays, linked-lists, skip-lists, queues, hash-tables, binary trees and (Cache-conscious) b-trees, tries, MassTree, and FAST (§2).
(3) We show that in addition to known designs, the design primitives form a massive space of possible designs that has only been minimally explored in the literature (§2).
(4) We show how to synthesize the latency cost of basic operations (point and range queries, and bulk loading) of arbitrary data structure designs from a small set of access primitives. Access primitives represent fundamental ways to access data and come with learned cost models which are trained on diverse hardware to capture hardware properties (§3).
(5) We show how to use cost synthesis to interactively answer complex what-if design questions, i.e., the impact of changes to design, workload, and hardware (§4).
(6) We introduce a design synthesis algorithm that completes partial layout specifications given a workload and hardware input; it utilizes cost synthesis to rank designs (§4).
(7) We demonstrate that the Data Calculator can accurately compute the performance impact of design choices for state-of-the-art designs and diverse hardware (§5).
(8) We demonstrate that the Data Calculator can accelerate the design process by answering rich design questions in a matter of seconds or minutes (§5).

## 2 DATA LAYOUT PRIMITIVES AND STRUCTURE SPECIFICATIONS

In this section, we discuss the library of data layout design primitives and how it enables the description of a massive number of both known and unknown data structures.

**Data Layout Primitives.** The Data Calculator contains a small set of design primitives that represent fundamental design choices when constructing a data structure layout. Each primitive belongs to a class of primitives depending on the high-level design concept it refers to such as node data organization, partitioning, node physical placement, and node metadata management. Within each class, individual primitives define design choices and allow for alternative tunings. The complete set of primitives we introduce in this paper is shown in Figure 11 in the appendix; they describe basic data layouts and cache conscious optimizations for reads. For example, "Key Order (none|sorted|k-ary)" defines how data is laid out in a node. Similarly, "Key Retention (none|full|func)" defines whether and how keys are included in a node. In this way, in a B+tree all nodes use "sorted" for order maintenance, while internal nodes use "none" for key retention as they only store fences and pointers, and leaf nodes use "full" for key retention.

The logic we use to generate primitives is that each one should represent a fundamental design concept that does not break down into more useful design choices (otherwise, there will be parts of the design space we cannot express). Coming up with the set of primitives is a trial and error task to map the known space of design concepts to an as clean and elegant set of primitives as possible.

Naturally, not all layout primitives can be combined. Most invalid relationships stem from the structure of the primitives, i.e., each primitive combines with every other standalone primitive. Only a few pairs of primitive tunings do not combine which generates a small set of invalidation rules. These are mentioned in Figure 11.

**From Layout Primitives to Data Structures.** To describe complete data structures, we introduce the concept of *elements*. An element is a full specification of a single data structure node; it defines the data and access methods used to access the node's data. An element may be "terminal" or "non-terminal". That is, an element may be describing a node that further partitions data to more nodes or not. This is done with the "fanout" primitive whose value represents the maximum number of children that would be generated when a node partitions data. Or it can be set to "terminal" in which case its value represents the capacity of a terminal node. A data structure specification contains one or more elements. It needs to have at least one terminal element, and it may have zero or more non-terminal elements. Each element has a destination element (except terminal ones) and a source element (except the root). Recursive connections are allowed to the same element.

**Examples.** A visualization of the primitives can be seen at the left side of Figure 3. It is a flat representation of the primitives shown in Figure 11 which creates an entry for every primitive signature. The radius depicts the domain of each primitive but different primitives may have different domains, visually depicted via the multiple inner circles in the radar plots of Figure 3. The small radar plots on the right side of Figure 3 depict descriptions of nodes of known data structures as combinations of the base primitives. Even visually it starts to become apparent that state-of-the-art designs which are meant to handle different scenarios are "synthesized from the same pool of design concepts". For example, using the non-terminal B+tree element and the terminal sorted data page element we can construct a full B+tree specification; data is recursively broken down into internal nodes using the B+tree element until we reach the leaf level, i.e., when partitions reach the terminal node size. Figure 3 also depicts Trie and Skip-list specifications. Figure 11 provides complete specifications of Hash-table, Linked-list, B+tree, Cache-conscious B-tree, and FAST.

**Elements "Without Data".** For flat data structures without an indexing layer, e.g., linked-lists and skip-lists, there need to be elements in the specification that describe the algorithm used to navigate the terminal nodes. Given that this algorithm is effectively a model, it does not rely on any data, and so such elements do not translate to actual nodes; they only affect algorithms that navigate across the terminal nodes. For example, a linked-list element in Figure 11 describes that data is divided into nodes that can only be accessed via following the links that connect terminal nodes. Similarly, one can create complex hierarchies of non-terminal elements that do not store any data but instead their job is to synthesize a collective model of how the keys should be distributed in the data structure, e.g., based on their value or other properties of the workload. These elements may lead to multiple hierarchies of both non-terminal nodes with data and terminal ones, synthesizing data structure designs that treat parts of the data differently. We will see such examples in the experimental analysis.

**Recursive Design Through Blocks.** A block is a logical portion of the data that we divide into smaller blocks to construct an instance of a data structure specification. The elements in a specification are the "atoms" with which we construct data structure instances by applying them recursively onto blocks. Initially, there is a single block of data, all data. Once all elements have been applied, the original block is broken down into a set of smaller blocks that correspond to the internal nodes (if any) and the terminal nodes of the data structure. Elements without data can be thought of as if they apply on a logical data block that represents part of the data with a set of specific properties (i.e., all data if this is the first element) and partitions the data with a particular logic into further logical blocks or physical nodes. This recursive construction is used when we test, cost, and search through multiple possible designs concurrently over the same data for a given workload and hardware as we will discuss in the next two sections, but it is also helpful to visualize designs as if "data is pushed through the design" based on the elements and logical blocks.

**Cache-Conscious Designs.** One critical aspect of data structure design is the relative positioning of its nodes, i.e., how "far" each node is positioned with respect to its predecessors and successors in a query path. This aspect is critical to the overall cost of traversing a data structure. The Data Calculator design space allows to dictate how nodes should be positioned explicitly: each non-terminal element defines how its children are positioned physically with respect to each other and with respect to the current node. For example, setting the layout primitive "Sub-block physical layout" to BFS tells the current node that its children are laid out sequentially. In addition, setting the layout primitive "Sub-blocks homogeneous" to true implies that all its children have the same layout (and therefore fixed width), and allows a parent node to access any of its children

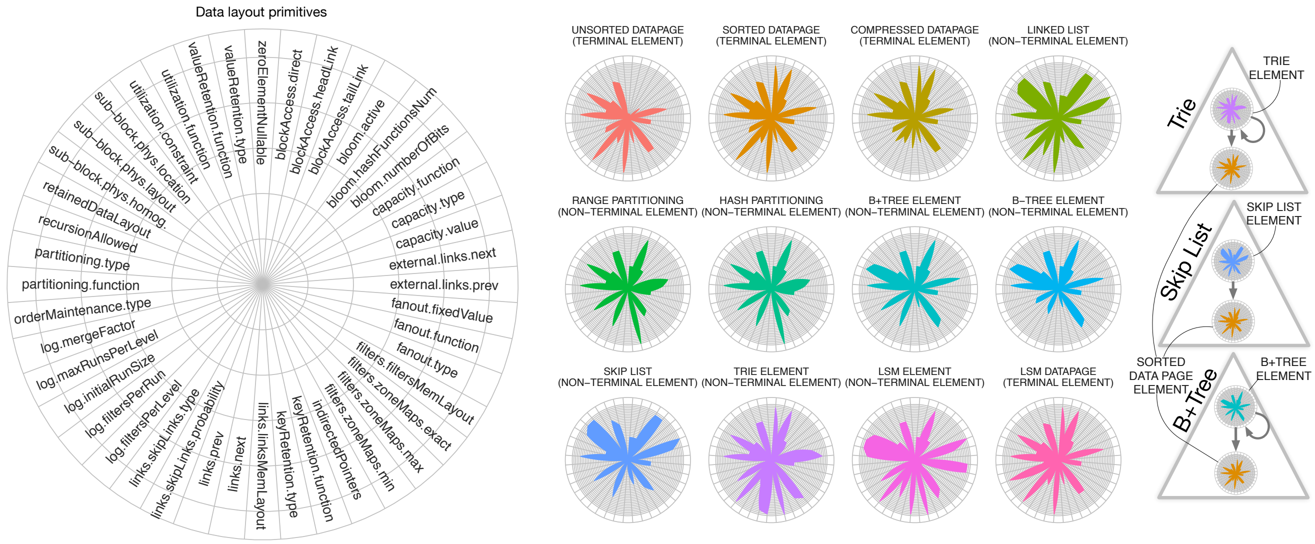


**Figure 3: The data layout primitives and examples of synthesizing node layouts of state-of-the-art data structures.**

nodes directly with a single pointer and reference number. This, in turn, makes it possible to fit more data in internal nodes because only one pointer is needed and thus more fences can be stored within the same storage budget. Such primitives allow specifying designs such as Cache Conscious B+tree [67] (Figure 11 provides the complete specification), but also the possibility of generalizing the optimizations made there to arbitrary structures.

Similarly, we can describe FAST [44]. First, we set "Sub-block physical location" to inline, specifying that the children nodes are directly after the parent node physically. Second, we set that the children nodes are homogeneous, and finally, we set that the children have a sub-block layout of "BFS Layer List (Page Size / Cache Line Size)". Here, the BFS layer list specifies that on a higher level, we should have a BFS layout of sub-trees containing Page Size/Cache Line Size layers; however, inside of those sub-trees pages are laid out in BFS manner by a single level. The combination matches the combined Page Level blocking and Cache Line level blocking of FAST. Additionally, the Data Calculator realizes that all child node physical locations can be calculated via offsets, and so eliminates all pointers. Figure 11 provides the complete specification.

**Size of the Design Space.** To help with arguing about the possible design space we provide formal definitions of the various constructs.

*Definition 2.1 (Data Layout Primitive).* A primitive $p_i$ belongs to a domain of values $\mathcal{P}_i$ and describes a layout aspect of a data structure node.

*Definition 2.2 (Data Structure Element).* A Data Structure Element $E$ is defined as a set of data layout primitives: $E = \{p_1, ..., p_n\} \in \mathcal{P}_i \times ... \times \mathcal{P}_n$, that uniquely identify it.

Given a set of $Inv(\mathcal{P})$ invalid combinations, the set of all possible elements $\mathcal{E}$, (i.e., node layouts) that can be designed as distinct combinations of data layout primitives has the following cardinality.

$$|\mathcal{E}| = \mathcal{P}_i \times ... \times \mathcal{P}_n - Inv(\mathcal{P}) = \prod_{\forall P_i \in E} |\mathcal{P}_i| - Inv(\mathcal{P}) \qquad (1)$$

*Definition 2.3 (Blocks).* Each non-terminal element $E \in \mathcal{E}$, applied on a set of data entries $D \in \mathcal{D}$, uses function $B_E(D) = \{D_1, ..., D_f\}$ to divide $D$ into $f$ blocks such that $D_1 \cup ... \cup D_f = D$.

A polymorphic design where every block may be described by a different element leads to the following recursive formula for the cardinality of all possible designs.

$$c_{poly}(D) = |\mathcal{E}| + \sum_{\forall E \in \mathcal{E}} \sum_{\forall D_i \in B_E(D)} c_{poly}(D_i) \qquad (2)$$

**Example: A Vast Space of Design Opportunities.** To get insight into the possible total designs we make a few simplifying assumptions. Assume the same fanout $f$ across all nodes and terminal node size equal to page size $p_{size}$. Then $N = \lceil \frac{|D|}{p_{size}} \rceil$ is the total number of pages in which we can divide the data and $h = \lceil log_f(N) \rceil$ is the height of the hierarchy. We can then approximate the result of Equation 2 by considering that we have $|E|$ possibilities for the root element, and $f * |E|$ possibilities for its resulting partitions which in turn have $f * |E|$ possibilities each up to the maximum level of recursion $h = log_f(N)$. This leads to the following result.

$$c_{poly}(D) \approx |\mathcal{E}| * (f * |\mathcal{E}|)^{\lceil log_f(N) \rceil} \qquad (3)$$

Most sophisticated data structure designs use only two distinct elements, each one describing all nodes across groups of levels of the structure, e.g., B-tree designs use one element for all internal nodes and one for all leaves. This gives the following design space for most standard designs.

$$c_{stan}(D) \approx |\mathcal{E}|^2 \qquad (4)$$

Using Equations 1, 3 and 4 we can get estimations of the possible design space for different kinds of data structure designs. For example, given the existing library of data layout primitives, and by limiting the domain of each primitive as shown in Figure 11 in appendix, then from Equation 1 we get $|\mathcal{E}| = 10^{16}$, meaning we can describe data structure layouts from a design space of $10^{16}$ possible node elements and their combinations. This number includes only valid combinations of layout primitives, i.e., all invalid combinations as defined by the rules in Figure 11 are excluded. Thus, we have a design space of $10^{32}$ for standard two-element structures (e.g., where B-tree and Trie belong) and $10^{48}$ for three-element structures (e.g., where MassTree [59] and Bounded-Disorder [55] belong). For polymorphic structures, the number of possible designs grows more quickly, and it also depends on the size of the training data used to find a specification, e.g., it is $> 10^{100}$ for $10^{15}$ keys.

The numbers in the above example highlight that data structure design is still a wide-open space with numerous opportunities for innovative designs as data keeps growing, application workloads keep changing, and hardware keeps evolving. Even with hundreds of new data structures manually designed and published each year, this is a slow pace to test all possible designs and to be able to argue about how the numerous designs compare. The Data Calculator is a tool that accelerates this process by 1) providing guidance about what is the possible design space, and 2) allowing to quickly test how a given design fits a workload and hardware setting. A technical report includes a more detailed description of its primitives [23].

## 3 DATA ACCESS PRIMITIVES AND COST SYNTHESIS

We now discuss how the Data Calculator computes the cost (latency) of running a given workload on a given hardware for a particular data structure specification. Traditional cost analysis in systems and data structures happens through experiments and the development of analytical cost models. Both options are not scalable when we want to quickly test multiple different parts of the massive design space we define in this paper. They require significant expertise and time, while they are also sensitive to hardware and workload properties. Our intuition is that we can instead synthesize complex operations from their fundamental components as we do for data layouts in the previous section, and then develop a hybrid way (through both benchmarks and models but without significant human effort needed) to assign costs to each component individually; The main idea is that we learn a small set of cost models for fine-grained data access patterns out of which we can synthesize the cost of complex dictionary operations for arbitrary designs in the possible design space of data structures.

The middle part of Figure 2 depicts the components of the Data Calculator that make cost synthesis possible: 1) the library of data access primitives, 2) the cost learning module which trains cost models for each access primitive depending on hardware and data properties, and 3) the operation and cost synthesis module which synthesizes dictionary operations and their costs from the access primitives and the learned models. Next, we describe the process and components in detail.

**Cost Synthesis from Data Access Primitives.** Each access primitive characterizes one aspect of how data is accessed. For example, a binary search, a scan, a random read, a sequential read, a random write, are access primitives. The goal is that these primitives should be fundamental enough so that we can use them to synthesize operations over arbitrary designs as sequences of such primitives. There exist two levels of access primitives. Level 1 access primitives are marked with white color in Figure 2 and Level 2 access primitives are nested under Level 1 primitives and marked with gray color. For example, a scan is a Level 1 access primitive used any time an operation needs to search a block of data where there is no order. At the same time, a scan may be designed and implemented in more than one way; this is exactly what Level 2 access primitives represent. For example, a scan may use SIMD instructions for parallelization if keys are nicely packed in vectors, and predication to minimize branch mispredictions with certain selectivity ranges. In the same way, a sorted search may use interpolation search if keys are arranged with uniform distribution. In this way, each Level 1 primitive is a conceptual access pattern, while each Level 2 primitive is an actual implementation that signifies a specific set of design choices. Every Level 1 access primitive has at least one Level 2 primitive and may be extended with any number of additional ones. The complete list of access primitives currently supported by the Data Calculator is shown in Table 1 in appendix.

**Learned Cost Models.** For every Level 2 primitive, the Data Calculator contains one or more models that describe its performance (latency) behavior. These are not static models; they are trained and fitted for combinations of data and hardware profiles as both those factors drastically affect performance. To train a model, each Level 2 primitive includes a minimal implementation that captures the behavior of the primitive, i.e., it isolates the performance effects of performing the specific action. For example, an implementation for a scan primitive simply scans an array, while an implementation for a random access primitive simply tries to access random locations in memory. These implementations are used to run a sequence of benchmarks to collect data for learning a model for the behavior of each primitive. Implementations should be in the target language/environment.

The models are simple parametric models; given the design decision to keep primitives simple (so they can be easily reused), we have domain expertise to expect how their performance behavior will look like. For example, for scans, we have a strong intuition they will be linear, for binary searches that they will be logarithmic, and that for random memory accesses that they will be smoothed out step functions (based on the probability of caching). These simple models have many advantages: they are interpretable, they train quickly, and they don't need a lot of data to converge. Through the training process, the Data Calculator learns coefficients of those models that capture hardware properties such as CPU and data movement costs.

Hardware and data profiles hold descriptive information about data and hardware respectively (e.g., data distribution for data, and CPU, Bandwidth, etc. for hardware). When an access primitive is trained on a data profile, it runs on a sample of such data, and when it is trained for a hardware profile, it runs on this exact hardware. Afterward, though, design questions can get accurate cost estimations on arbitrary access method designs without going over the data or having to have access to the specific machine. Overall, this is an offline process that is done once, and it can be repeated to include new hardware and data profiles or to include new access primitives.

**Example: Binary Search Model.** To give more intuition about how models are constructed let us consider the case of a Level 2 primitive of binary searching a sorted array as shown on the upper right part of Figure 4. The primitive contains a code snippet that implements the bare minimum behavior (Step 1 in Figure 4). We

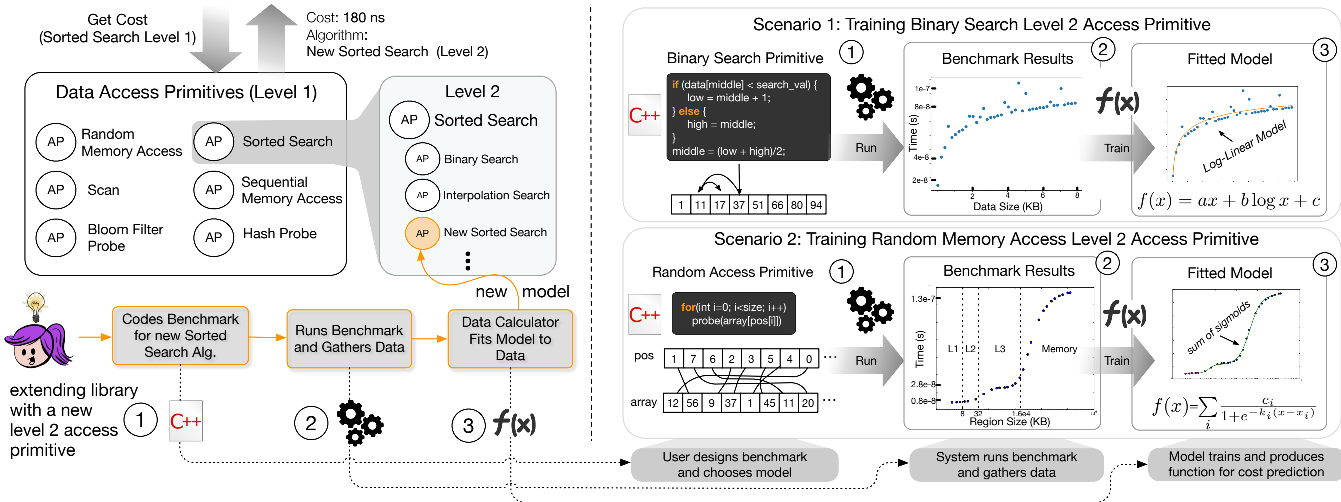


**Figure 4: Training and fitting models for Level 2 access primitives and extending the Data Calculator.**

observe that the benchmark results (Step 2 in Figure 4) indicate that performance is related to the size of the array by a logarithmic component. As expected there is also bias as the relationship for small array sizes (such as just 4 or 8 elements) might not fit exactly a logarithmic function. We additionally add a linear term to capture some small linear dependency on the data size. Thus, the cost of binary searching an array of $n$ elements can be approximated as $f(n) = c_1 n + c_2 \log n + y_0$ where $c_1, c_2$, and $y_0$ are coefficients learned through linear regression. The values of these coefficients help us translate the abstract model, $f(n) = O(\log n)$, into a realized predictive model which has taken into account factors such as CPU speed and the cost of memory accesses across the sorted array for the specific hardware. The resulting fitted model can be seen in Step 3 on the upper right part of Figure 4. The Data Calculator can then use this learned model to query for the performance of binary search within the trained range of data sizes. For example, this would be used when querying a large sorted array as well as a small node of a complex data structure that is sorted.

Certain critical aspects of the training process can be automated as part of future research. For example, the data range for which we should train a primitive depends on the memory hierarchy (e.g., size of caches, memory, etc.) on the target machine and what is the target setting in the application (i.e., memory only, or also disk/flash, etc.). In turn, this also affects the length of the training process. Overall, such parameters can eventually be handled through high-level knobs, letting the system make the lower level tuning choices. Furthermore, identification of convergence can also be automated. There exist primitives that require more training than others (e.g., due to more complex code, random access or sensitivity to outliers), and so the number of benchmarks and data points we collect should not be a fixed decision.

**Synthesizing Latency Costs.** Given a data layout specification and a workload, the Data Calculator uses Level 1 access primitives to synthesize operations and subsequently each Level 1 primitive is translated to the appropriate Level 2 primitive to compute the cost of the overall operation. Figure 5 depicts this process and an example specifically for the `Get` operation. This is an expert system, i.e., a sequence of rules that based on a given data structure specification defines how to traverse its nodes.[2] To read Figure 5 start from the top right corner. The input is a data structure specification, a test data set, and the operation we need to cost, e.g., `Get key x`. The process simulates populating the data structure with the data to figure out how many nodes exist, the height of the structure, etc. This is because to accurately estimate the cost of an operation, the Data Calculator needs to take into account the expected state of the data structure at the particular moment in the workload. It does this by recursively dividing the data into blocks given the elements used in the specification.

In the example of Figure 5 the structure contains two elements, one for internal nodes and one for leaves. For every node, the operation synthesis process takes into account the data layout primitives used. For example, if a node is sorted it uses binary search, but if the node is unsorted, it uses a full scan. The rhombuses on the left side of Figure 5 reflect the data layout primitives that operation `Get` relies on, while the rounded rectangles reflect data access primitives that may be used. For each node the per-node operation synthesis procedure (starting from the left top side of Figure 5), first checks if this node is internal or not by checking whether the node contains keys or values; if not, it proceeds to determine which node it should visit next (left side of the figure) and if yes, it continues to process the data and values (right side of the figure). A non-terminal element leads to data of this block being split into $f$ new blocks and the process follows the relevant blocks only, i.e., the blocks that this operation needs to visit to resolve.

In the end, the Data Calculator generates an abstract syntax tree with the access patterns of the path it had to go through. This is expressed in terms of Level 1 access primitives (bottom right part of Figure 5). In turn, this is translated to a more detailed abstract syntax tree where all Level 1 access primitives are translated to Level 2 access primitives along with the estimated cost for each one given the particular data size, hardware input, and any primitive specific input. The overall cost is then calculated as the sum of all those costs.

[2]Due to space restrictions Figure 5 is a subset of the expert system. The complete version can be found in a technical report [23].

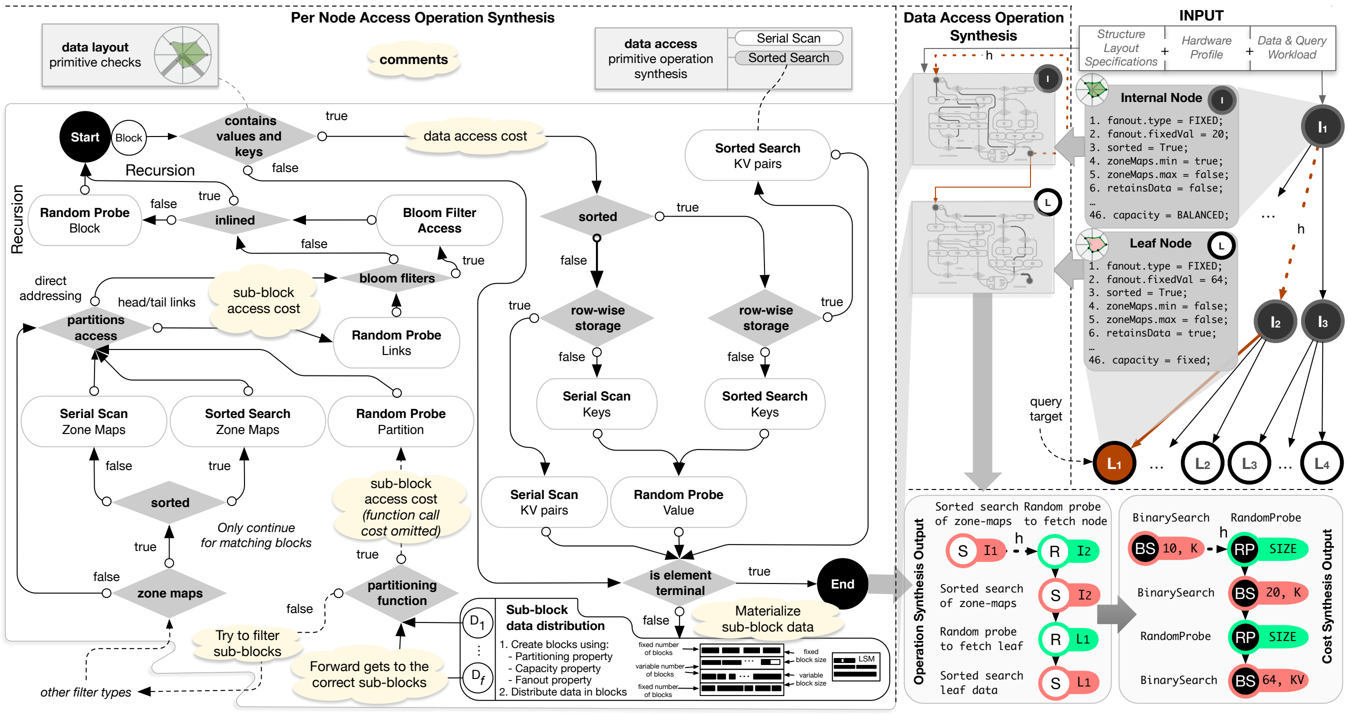


**Figure 5: Synthesizing the operation and cost for dictionary operation Get, given a data structure specification.**

**Calculating Random Accesses and Caching Effects.** A crucial part in calculating the cost of most data structures is capturing random memory access costs, e.g., the cost of fetching nodes while traversing a tree, fetching nodes linked in a hash bucket, etc. If data is expected to be cold, then this is a rather straightforward case, i.e., we may assign the maximum cost a random access is expected to incur on the target machine. If data may be hot, though, it is a more involved scenario. For example, in a tree-like structure internal nodes higher in the tree are much more likely to be at higher levels of the memory hierarchy during repeated requests. We calculate such costs using the random memory access primitive, as shown in the lower right part of Figure 4. Its input is a "region size", which is best thought of as the amount of memory we are randomly accessing into (i.e., we don't know where in this memory region our pointer points to). The primitive is trained via benchmarking access to an increasingly bigger contiguous array (Step 1 in Figure 4). The results (Step 2 in Figure 4) depict a minor jump from L1 to L2 (we can see a small bump just after $10^4$ elements). The bump from L2 to L3 is much more noticeable, with the cost of accessing memory going from $0.1 \times 10^7$ seconds to $0.3 \times 10^7$ seconds as the memory size crosses the 128 KB boundary. Similarly, we see a bump from $0.3 \times 10^7$ seconds to $1.3 \times 10^7$ seconds when we go from L3 to main memory, at the L3 cache size of 16 MB[3]. We capture this behavior as a sum of sigmoid functions, which are essentially smoothed step functions, using

$$c(x) = \sum_{i=1}^{k} f(x) = \sum_{i=1}^{k} \frac{c_i}{1 + e^{-k_i(\log x - x_i)}} + y_0.$$

This primitive is used for calculating random access to any physical or logical region (e.g., a sequence of nodes that may be cached together). For example, when traversing a tree, the cost synthesis operation, costs random accesses with respect to the amount of data that may be cached up to this point. That is, for every node we need to access at Level $x$ of a tree, we account for a region size that includes all data in all levels of the tree up to Level $x$. In this way, accessing a node higher in the tree costs less than a node at lower levels. The same is true when accessing buckets of a hash table. We give a detailed step by step example below.

**Example: Cache-aware Cost Synthesis.** Assume a B-tree -like specification as follows: two node types, one for internal nodes and one for leaf nodes. Internal nodes contain fence pointers, are sorted, balanced, have a fixed fanout of 20 and do not contain any keys or values. Leaf nodes instead are terminal; they contain both keys and values, are sorted, have a maximum page size of 250 records, and follow a full columnar format, where keys and values are stored in independent arrays. The test dataset consists of $10^5$ records where keys and values are 8 bytes each. Overall, this means that we have 400 full data pages, and thus a tree of height 2. The Data Calculator needs two of its access primitives to calculate the cost of a Get operation. Every Get query will be routed through two internal nodes and one leaf node: within each node, it needs to binary search (through fence pointers for internal nodes and through keys in leaf nodes) and thus it will make use of the Sorted Search access primitive. In addition, as a query traverses the tree it needs to perform a random access for every hop.

Now, let us look in more detail how these two primitives are used given the exact specification of this data structure. The Sorted Search primitive takes as input the size of the area over which we

[3]These numbers are in line with Intel's Vtune.

will binary search and the number of keys. The Random Access primitive takes as input the size of the path so far which allows us to takes into account caching effects. Each query starts by visiting the root node. The data calculator estimates the size of the path so far to be 312 bytes. This is because the size of the path so far is in practice equal to the size of the root node which containing 20 pointers (because the fanout is 20) and 19 values sums up at $root = internalnode = 20 * 8 + 19 * 8 = 312$ bytes. In this way, the Data Calculator logs a cost of $RandomAccess(312)$ to access the root node. Then, it calculates the cost of binary search across 19 fences, thus logging a cost of $SortedSearch(RowStore, 19 * 8)$. It uses the "RowStore" option as fences and pointers are stored as pairs within each internal node. Now, the access to the root node is fully accounted for, and the Data Calculator moves on to cost the access at the next tree level. Now the size of the path so far is given by accounting for the whole next level in addition to the root node. This is in total $level2 = root + fanout * internalnode = 312 + 20 * 312 = 6552$ bytes (due to fanout being 20 we account for 20 nodes at the next level). Thus to access the next node, the Data Calculator logs a cost of $RandomAccess(6552)$ and again a search cost of $SortedSearch(RowStore, 19 * 8)$ to search this node. The last step is to search the leaf level. Now the size of the path so far is given by accounting for the whole size of the tree which is $level2 + 400 * (250 * 16) = 1606552$ bytes since we have 400 pages at the next level (20x20) and each page has 250 records of key-value pairs (8 bytes each). In this way, the Data Calculator logs a cost of $RandomAccess(1606552)$ to access the leaf node, followed by a sorted search of $SortedSearch(ColumnStore, 250 * 8)$ to search the keys. It uses the "ColumnStore" option as keys and values are stored separately in each leaf in different arrays. Finally, a cost of $RandomAccess(2000)$ is incurred to access the target value in the values array (we have $8 * 250 = 2000$ in each leaf).

**Sets of Operations.** The description above considers a single operation. The Data Calculator can also compute the latency for a set of operations concurrently in a single pass. This is effectively the same process as shown in Figure 5 only that in every recursion we may follow more than one path and in every step we are computing the latency for all queries that would visit a given node.

**Workload Skew and Caching Effects.** Another parameter that can influence caching effects is workload skew. For example, repeatedly accessing the same path of a data structure results in all nodes in this path being cached with higher probability than others. The Data Calculator first generates counts of how many times every node is going to be accessed for a given workload. Using these counts and the total number of nodes accessed we get a factor $p = count/total$ that denotes the popularity of a node. Then to calculate the random access cost to a node for an operation $k$, a weight $w = 1/(p * sid)$ is used, where $sid$ is the sequence number of this operation in the workload (refreshed periodically). Frequently accessed nodes see smaller access costs and vice versa.

**Training Primitives.** All access primitives are trained on warm caches. This is because they are used to calculate the cost on a node that is already fetched. The only special case is the Random Access primitive which is used to calculate the cost of fetching a node. This is also trained on warm data, though, since the cost synthesis infrastructure takes care at a higher level to pass the right region size as discussed; in the case this region is big, this can still result in costing a page fault as large data will not fit in the cache which is reflected in the Random Access primitive model.

**Limitations.** For individual queries certain access primitives are hard to estimate precisely without running the actual code on an exact data instance. For example, a scan for a point Get may abort after checking just a few values, or it may need to go all the way to the end of an array. In this way, while lower or upper performance bounds can be computed with absolute confidence for both individual queries and sets of queries, actual performance estimation works best for sets.

**More Operations.** The cost of range queries, and bulk loading is synthesized as shown in Figure 10 in appendix.

**Extensibility and Cross-pollination.** The rationale of having two Levels of access primitives is threefold. First, it brings a level of abstraction allowing higher level cost synthesis algorithms to operate at Level 1 only. Second, it brings extensibility, i.e., we can add new Level 2 primitives without affecting the overall architecture. Third, it enhances "cross-pollination" of design concepts captured by Level 2 primitives across designs. Consider the following example. An engineer comes up with a new algorithm to perform search over a sorted array, e.g., exploiting new hardware instructions. To test if this can improve performance in her B-tree design, where she regularly searches over sorted arrays, she codes up a benchmark for a new sorted search Level 2 primitive and plugs it in the Calculator as shown in Figure 4. Then the original B-tree design can be easily tested with and without the new sorted search across several workloads and hardware profiles without having to undergo a lengthy implementation phase. At the same time, the new primitive can now be considered by any data structure design that contains a sorted array such as an LSM-tree with sorted runs, a Hash-table with sorted buckets and so on. This allows easy transfer of ideas and optimizations across designs, a process that usually requires a full study for each optimization and target design.

## 4 WHAT-IF DESIGN AND AUTO-COMPLETION

The ability to synthesize the performance cost of arbitrary designs allows for the development of algorithms that search the possible design space. We expect there will be numerous opportunities in this space for techniques that can use this ability to: 1) improve the productivity of engineers by quickly iterating over designs and scenarios before committing to an implementation (or hardware), 2) accelerate research by allowing researchers to easily and quickly test completely new ideas, 3) develop educational tools that allow for rapid testing of concepts, and 4) develop algorithms for offline auto-tuning and online adaptive systems that transition between designs. In this section, we provide two such opportunities for what-if design and auto-completion of partial designs.

**What-if Design.** One can form design questions by varying any one of the input parameters of the Data Calculator: 1) data structure (layout) specification, 2) hardware profile, and 3) workload (data and queries). For example, assume one already uses a B-tree-like design for a given workload and hardware scenario. The Data Calculator can answer design questions such as "What would be the performance impact if I change my B-tree design by adding a bloom

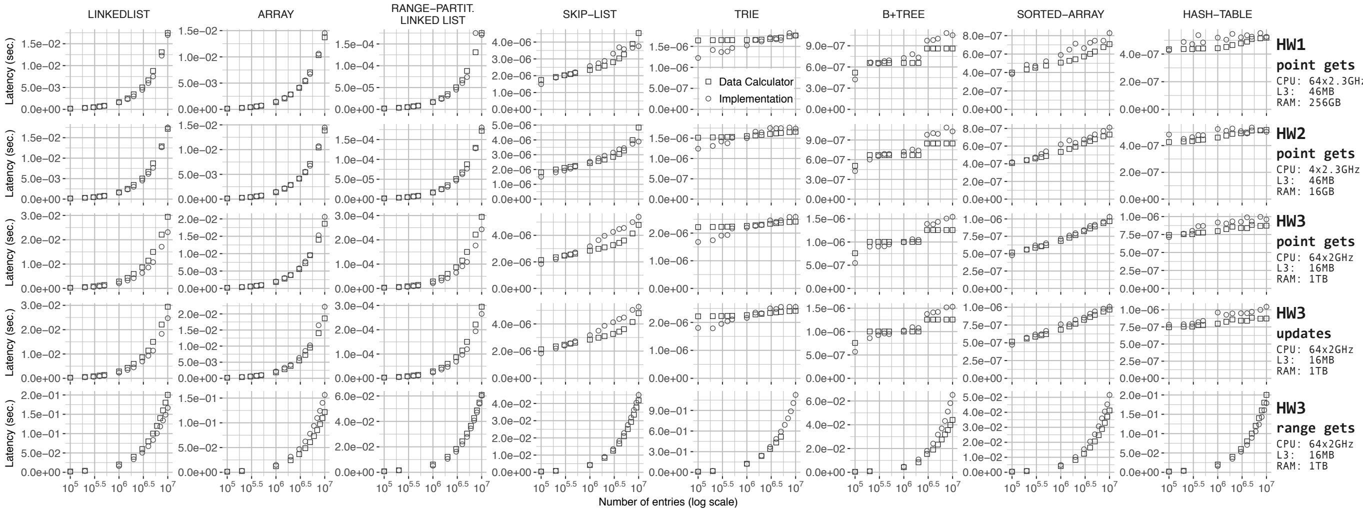


**Figure 6: The Data Calculator can accurately compute the latency of arbitrary data structure designs across a diverse set of hardware and for diverse dictionary operations.**

filter in each leaf?" The user simply needs to give as input the high-level specification of the existing design and cost it twice: once with the original design and once with the bloom filter variation. In both cases, costing should be done with the original data, queries, and hardware profile so the results are comparable. In other words, users can quickly test variations of data structure designs simply by altering a high level specification, without having to implement, debug, and test a new design. Similarly, by altering the hardware or workload inputs, a given specification can be tested quickly on alternative environments without having to actually deploy code to this new environment. For example, in order to test the impact of new hardware the Calculator only needs to train its Level 2 primitives on this hardware, a process that takes a few minutes. Then, one can test the impact this new hardware would have on arbitrary designs by running what-if questions without having implementations of those designs and without accessing the new hardware.

**Auto-completion.** The Data Calculator can also complete partial layout specifications given a workload, and a hardware profile. The process is shown in Algorithm 1 in the appendix: The input is a partial layout specification, data, queries, hardware, and the set of the design space that should be considered as part of the solution, i.e., a list of candidate elements. Starting from the last known point of the partial specification, the Data Calculator computes the rest of the missing subtree of the hierarchy of elements. At each step the algorithm considers a new element as candidate for one of the nodes of the missing subtree and computes the cost for the different kinds of dictionary operations present in the workload. This design is kept only if it is better than all previous ones, otherwise it is dropped before the next iteration. The algorithm uses a cache to remember specifications and their costs to avoid recomputation. This process can also be used to tell if an existing design can be improved by marking a portion of its specification as "to be tested". Solving the search problem completely is an open challenge as the design space drawn by the Calculator is massive. Here we show a first step which allows search algorithms to select from a restricted set of elements which are also given as input as opposed to searching the whole set of possible primitive combinations.

## 5 EXPERIMENTAL ANALYSIS

We now demonstrate the ability of the Data Calculator to help with rich design questions by accurately synthesizing performance costs.

**Implementation.** The core implementation of the Data Calculator is in C++. This includes the expert systems that handle layout primitives and cost synthesis. A separate module implemented in Python is responsible for analyzing benchmark results of Level 2 access primitives and generating the learned models. The benchmarks of Level 2 access primitives are also implemented in C++ such that the learned models can capture performance and hardware characteristics that would affect a full C++ implementation of a data structure. The learning process for each Level 2 access primitive is done each time we need to include a new hardware profile; then, the learned coefficients for each model are passed to the C++ back-end to be used for cost synthesis during design questions. For learning we use a standard loss function, i.e., least square errors, and the actual process is done via standard optimization libraries, e.g., SciPy's curve fit. For models which have non-convex loss functions such as the *sum of sigmoids* model, we algorithmically set up good initial parameters.

**Accurate Cost Synthesis.** In our first experiment we test the ability to accurately cost arbitrary data structure specifications across different machines. To do this we compare the cost generated automatically by the Data Calculator with the cost observed when testing a full implementation of a data structure. We set-up the experiment as follows. To test with the Data Calculator, we manually wrote data structure specifications for eight well known access methods 1) Array, 2) Sorted Array, 3) Linked-list, 4) Partitioned Linked-list, 5) Skip-list, 6) Trie, 7) Hash-table, and 8) B+tree. The Data Calculator was then responsible for generating the design of operations for each data structure and computing their latency given a workload. To verify the results against an actual implementation, we implemented all data structures above. We also implemented algorithms for each of their basic operations: `Get`, `Range Get`, `Bulk Load` and `Update`. The first experiment then starts with a data workload of $10^5$ uniformly distributed integers and a

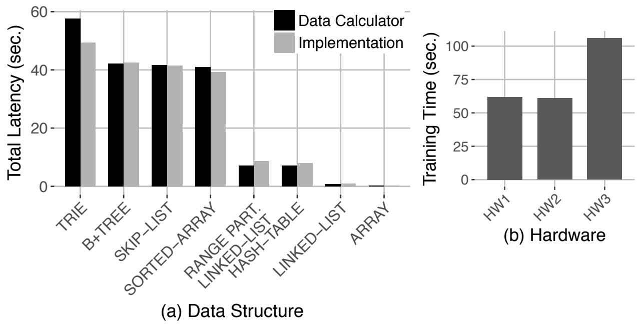


**Figure 7: Computing Bulk-loading cost (left) and Training cost across diverse hardware (right).**

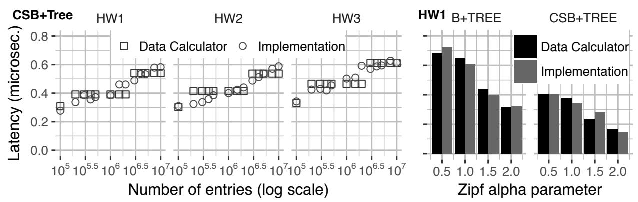


**Figure 8: Accurately computing the latency of cache conscious designs in diverse hardware and workloads.**

sequence of $10^2$ `Get` requests, also uniformly distributed. We incrementally insert more data up to a total of $10^7$ entries and at each step we repeat the query workload.

The top row of Figure 6 depicts results using a machine with 64 cores and 264 GB of RAM. It shows the average latency per query as data grows as computed by the Data Calculator and as observed when running the actual implementation on this machine. For ease of presentation results are ranked horizontally from slower to faster (left to right). The Data Calculator gives an accurate estimation of the cost across the whole range of data sizes and regardless of the complexity of the designs both in terms of the data structure. It can accurately compute the latency of both simple traversals in a plain array and the latency of more complex access patterns such as descending a tree and performing random hops in memory.

**Diverse Machines and Operations.** The rest of the rows in Figure 6 repeat the same experiment as above using different hardware in terms of both CPU and memory properties (Rows 2 and 3) and different operations (Rows 4 and 5). The details of the hardware are shown on the right side of each row in Figure 6. Regardless of the machine or operation, the Data Calculator can accurately cost any design. By training its Level 2 primitives on individual machines and maintaining a profile for each one of them, it can quickly test arbitrary designs over arbitrary hardware and operations. Updates here are simple updates that change the value of a key-value pair and so they are effectively the same as a point query with an additional write access. More complex updates that involve restructures are left for future work both in terms of the design space and cost synthesis. Finally, Figure 7a) depicts that the Data Calculator can accurately synthesize the bulk loading costs for all data structures.

**Training Access Primitives.** Figure 7b) depicts the time needed to train all Level 2 primitives on a diverse set of machines. Overall, this is an inexpensive process. It takes merely a few minutes to train multiple different combinations of data and hardware profiles.

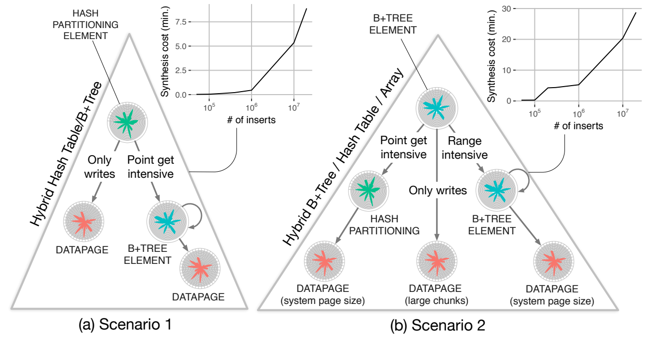


**Figure 9: The Data Calculator designs new hybrids of known data structures to match a given workload.**

**Cache Conscious Designs and Skew.** In addition, Figure 8 repeats our base fitting experiment using a cache-conscious design, Cache Conscious B+tree (CSB). Figure 8a) depicts that the Data Calculator accurately predicts the performance behavior across a diverse set of machines, capturing caching effects of growing data sizes and design patterns where the relative position of nodes affects tree traversal costs. We use the "Full" design from Cache Conscious B+tree [67]. Furthermore, Figure 8b) tests the fitting when the workload exhibits skew. For this experiment Get queries were generated with a Zipfian distribution: $\alpha = \{0.5, 1.0, 1.5, 2.0\}$. Figure 8b) shows that for the implementation results, workload skew improves performance and in fact it improves more for the standard B+tree. This is because the same paths are more likely to be taken by queries resulting in these nodes being cached more often. Cache Conscious B+tree improves but at a much slower rate as it is already optimized for the cache hierarchy. The Data Calculator is able to synthesize these costs accurately, capturing skew and the related caching effects.

**Rich Design Questions.** In our next, experiment we provide insights about the kinds of design questions possible and how long they may take, working over a B-tree design and a workload of uniform data and queries: 1 million inserts and 100 point Gets. The hardware profile used is HW1 (defined in Figure 6). The user asks "What if we change our hardware to HW3?". It takes the Data Calculator only 20 seconds (all runs are done on HW3) to compute that the performance would drop. The user then asks: "Is there a better design for this new hardware and workload if we restrict search on a specific set of five possible elements?" (from the pool of element on right side of Figure 3). It takes only 47 seconds for the Data Calculator to compute the best choice. The user then asks "Would it be beneficial to add a bloom filter in all B-tree leaves?" The Data Calculator computes in merely 20 seconds that such a design change would be beneficial for the current workload and hardware. The next design question is: "What if the query workload changes to have skew targeting just 0.01% of the key space?" The Data Calculator computes in 24 seconds that this new workload would hurt the original design and it computes a better design in another 47 seconds.

In two of the design phases the user asked "give me a better design if possible". We now provide more intuition for this kind of design questions regarding the cost and scalability of computing such designs as well as the kinds of designs the Data Calculator may produce to fit a workload. We test two scenarios for a workload of mixed reads and writes (uniformly distributed inserts and point reads) and hardware profile HW3. In the first scenario, all reads are point queries in 20% of the domain. In the second scenario, 50% of the reads are point reads and touch 10% of the domain, while the other half are range queries and touch a different (non intersecting with the point reads) 10% of the domain. We do not provide the Data Calculator with an initial specification. Given the composition of the workload our intuition is that a mix of hashing, B-tree like indexing (e.g., with quantile nodes and sorted pages), and a simple log (unsorted pages) might lead to a good design, and so we instruct the Data Calculator to use those four elements to construct a design (this is done using Algorithm 1 but starting with an empty specification. Figure 9 depicts the specifications of the resulting data structures For the first scenario (left side of Figure 9) the Data Calculator computed a design where a hashing element at the upper levels of the hierarchy allows to quickly access data but then data is split between the write and read intensive parts of the domain to simple unsorted pages (like a log) and B+tree -style indexing for the read intensive part. For the second scenario (right side of Figure 9), the Data Calculator produces a design which similarly to the previous one takes care of read and writes separately but this time also distinguishes between range and point gets by allowing the part of the domain that receives point queries to be accessed with hashing and the rest via B+tree style indexing. The time needed for each design question was in the order of a few seconds up to 30 minutes depending on the size of the sample workload (the synthesis costs are embedded in Figure 9 for both scenarios). Thus, the Data Calculator quickly answers complex questions that would normally take humans days or even weeks to test fully.

## 6 RELATED WORK

To the best of our knowledge this is the first work to discuss the problem of interactive data structure design and to compute the impact on performance. However, there are numerous areas from where we draw inspiration and with which we share concepts.

**Interactive Design.** Conceptually, the work on Magic for layout on integrated circuits [62] comes closest to our work. Magic uses a set of design rules to quickly verify transistor designs so they can be simulated by designers. In other words, a designer may propose a transistor design and Magic will determine if this is correct or not. Naturally, this is a huge step especially for hardware design where actual implementation is extremely costly. The Data Calculator pushes interactive design one step further to incorporate cost estimation as part of the design phase by being able to estimate the cost of adding or removing individual design options which in turn also allows us to build design algorithms for automatic discovery of good and bad designs instead of having to build and test the complete design manually.

**Generalized Indexes.** One of the stronger connections is the work on Generalized Search Tree Indexes (GiST) [6, 7, 38, 47–50]. GiST aims to make it easy to extend data structures and tailor them to specific problems and data with minimal effort. It is a template, an abstract index definition that allows designers and developers to implement a large class of indexes. The original proposal focused on record retrieval only but later work added support for concurrency [48], a more general API [6], improved performance [47], selectivity estimation on generated indexes [7] and even visual tools that help with debugging [49, 50]. While the Data Calculator and GiST share motivation, they are fundamentally different: GiST is a template to implement tailored indexes while the Data Calculator is an engine that computes the performance of a design enabling rich design questions that compute the impact of design choices before we start coding, making these two lines of work complementary.

**Modular/Extensible Systems and System Synthesizers.** A key part of the Data Calculator is its design library, breaking down a design space in components and then being able to use any set of those components as a solution. As such the Data Calculator shares concepts with the stream of work on modular systems, an idea that has been explored in many areas of computer science: in databases for easily adding data types [31, 32, 60, 61, 70] with minimal implementation effort, or plug and play features and whole system components with clean interfaces [11, 14, 17, 45, 53, 54], as well as in software engineering [63], computer architecture [62], and networks [46]. Since for every area the output and the components are different, there are particular challenges that have to do with defining the proper components, interfaces and algorithms. The concept of modularity is similar in the context of the Data Calculator. The goal and application of the concept is different though.

**Additional Topics.** Appendix B discusses additional related topics such as auto-tuning systems and data representation synthesis in programming languages.

## 7 SUMMARY AND NEXT STEPS

Through a new paradigm of first principles of data layouts and learned cost models, the Data Calculator allows researchers and engineers to interactively and semi-automatically navigate complex design decisions when designing or re-designing data structures, considering new workloads, and hardware. The design space we presented here includes basic layout primitives and primitives that enable cache conscious designs by dictating the relative positioning of nodes, focusing on read only queries. The quest for the first principles of data structures needs to continue to find the primitives for additional significant classes of designs including updates, compression, concurrency, adaptivity, graphs, spatial data, version control management, and replication. Such steps will also require innovations for cost synthesis. For every design class added (or even for every single primitive added), the knowledge gained in terms of the possible data structures designs grows exponentially. Additional opportunities include full DSLs for data structures, compilers for code generation and eventually certified code [66, 73], new classes of adaptive systems that can change their core design on-the-fly, and machine learning algorithms that can search the whole design space.

## 8 ACKNOWLEDGMENTS

We thank the reviewers for valuable feedback and direction. Mark Callaghan provided the quotes on the importance of data structure design. Harvard DASlab members Yiyou Sun, Mali Akmanalp and Mo Sun helped with parts of the implementation and the graphics. This work is partially funded by the USA National Science Foundation project IIS-1452595.

## REFERENCES


[1] Daniel J. Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280.

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 1009–1024.

[3] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 1103–1114.

[4] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. 2014. Main Memory Adaptive Indexing for Multi-Core Systems. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON).* 3:1—-3:10.

[5] Michael R. Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael J. Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. 2013. Brainwash: A Data System for Feature Engineering. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR).*

[6] Paul M Aoki. 1998. Generalizing "Search" in Generalized Search Trees (Extended Abstract). In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* 380–389.

[7] Paul M Aoki. 1999. How to Avoid Building DataBlades That Know the Value of Everything and the Cost of Nothing. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM).* 122–133.

[8] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.*

[9] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT).* 461–466.

[10] Shivnath Babu, Nedyalko Borisov, Songyun Duan, Herodotos Herodotou, and Vamsidhar Thummala. 2009. Automated Experiment-Driven Management of (Database) Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS).*

[11] Don S Batory, J R Barnett, J F Garza, K P Smith, K Tsukuda, B C Twichell, and T E Wise. 1988. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering (TSE)* 14, 11 (1988), 1711–1730.

[12] Philip A. Bernstein and David B. Lomet. 1987. CASE Requirements for Extensible Database Systems. *IEEE Data Engineering Bulletin* 10, 2 (1987), 2–9.

[13] Alfonso F. Cardenas. 1973. Evaluation and Selection of File Organization - A Model and System. *Commun. ACM* 16, 9 (1973), 540–548.

[14] Michael J Carey and David J DeWitt. 1987. An Overview of the EXODUS Project. *IEEE Data Engineering Bulletin* 10, 2 (1987), 47–54.

[15] M.J. Steindorfer, and J.J. Vinju. 2016. Towards a software product line of trie-based collections. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.*

[16] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB).* 146–155.

[17] Surajit Chaudhuri and Gerhard Weikum. 2000. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB).* 1–10.

[18] C. Loncaric, E.Torlak, and M.D Ernst. 2016. Fast synthesis of fast collections. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[19] D. Cohen and N. Campbell. 1993. Automating Relational Operations on Data Structures. *IEEE Software* 10, 3 (1993), 53–60.

[20] E. Schonberg, J.T. Schwartz and Sharir. 1979. Automatic Data Structure Selection in SETL. In *Proceedings of the ACM Symposium on Principles of Programming Languages.*

[21] E. Schonberg, J.T. Schwartz and Sharir. 1981. An Automatic Technique for Selection of Data Representations in SETL Programs. *ACM Transactions on Programming Languages and Systems* 3, 2 (1981), 126–143.

[22] Alvin Cheung. 2015. Towards Generating Application-Specific Data Management Systems. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR).*

[23] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Mike Kester, Demi Guo. 2018. The Internals of the Data Calculator. *Harvard Data Systems Laboratory, Technical Report* (2018).

[24] O. Shacham, M. Vechev, and E. Yahav. 2009. Chameleon: Adaptive Selection of Collections. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[25] P. Hawkins, A. Aiken, K. Fisher, M.C. Rinard and M. Sagiv. 2011. Data representation synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[26] P. Hawkins, A. Aiken, K. Fisher, M.C. Rinard, and M. Sagiv. 2012. Concurrent data representation synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[27] Y. Smaragdakis, and D. Batory. 1997. DiSTiL: A Transformation Library for Data Structures. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL).*

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

[29] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A Reliable Randomized Algorithm for the Closest-Pair Problem. *J. Algorithms* (1997).

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

[31] David Goldhirsch and Jack A Orenstein. 1987. Extensibility in the PROBE Database System. *IEEE Data Engineering Bulletin* 10, 2 (1987), 24–31.

[32] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 6, 1 (feb 1994), 120–135.

[33] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

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

[35] Jim Gray. 2000. What Next? A Few Remaining Problems in Information Technlogy. *ACM SIGMOD Digital Symposium Collection* 2, 2 (2000).

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

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

[38] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB).* 562–573.

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

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

[41] Yannis E Ioannidis and Eugene Wong. 1987. Query Optimization by Simulated Annealing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 9–22.

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

[43] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 715–730.

[44] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 339–350.

[45] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-Level Language. *Proceedings of the VLDB Endowment* 7, 10 (2014), 853–864.

[46] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[47] Marcel Kornacker. 1999. High-Performance Extensible Indexing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB).* 699–708.

[48] Marcel Kornacker, C Mohan, and Joseph M. Hellerstein. 1997. Concurrency and Recovery in Generalized Search Trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 62–72.

[49] Marcel Kornacker, Mehul A. Shah, and Joseph M. Hellerstein. 1998. amdb: An Access Method Debugging Tool. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 570–571.

[50] Marcel Kornacker, Mehul A. Shah, and Joseph M. Hellerstein. 2003. Amdb: A Design Tool for Access Methods. *IEEE Data Engineering Bulletin* 26, 2 (2003), 3–11.

[51] Tobin J Lehman and Michael J Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB).* 294–303.

[52] Gottfried Wilhelm Leibniz. 1666. Dissertation on the Art of Combinations. *PhD Thesis, Leipzig University* (1666).

[53] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proceedings of the VLDB Endowment* 6, 10 (2013), 877–888.

[54] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* 302–313.

[55] Witold Litwin and David B. Lomet. 1986. The Bounded Disorder Access Method. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* 38–48.

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

[57] Stefan Manegold. 2002. Understanding, modeling, and improving main-memory database performance. *Ph.D. Thesis. University of Amsterdam* (2002).

[58] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Generic Database Cost Models for Hierarchical Memory Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB).* 191–202.

[59] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys).* 183–196.

[60] John McPherson and Hamid Pirahesh. 1987. An Overview of Extensibility in Starburst. *IEEE Data Engineering Bulletin* 10, 2 (1987), 32–39.

[61] Sylvia L Orborn. 1987. Extensible Databases and RAD. *IEEE Data Engineering Bulletin* 10, 2 (1987), 10–15.

[62] John K Ousterhout, Gordon T Hamachi, Robert N Mayo, Walter S Scott, and George S Taylor. 1984. Magic: A VLSI Layout System. In *Proceedings of the Design Automation Conference (DAC).* 152–159.

[63] David Lorge Parnas. 1979. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)* 5, 2 (1979), 128–138.

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

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

[66] Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural synthesis of provably-correct data-structure manipulations. *PACMPL* 1 (2017), 65:1–65:28.

[67] Jun Rao and Kenneth A. Ross. 2000. Making B+-trees Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 475–486.

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

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

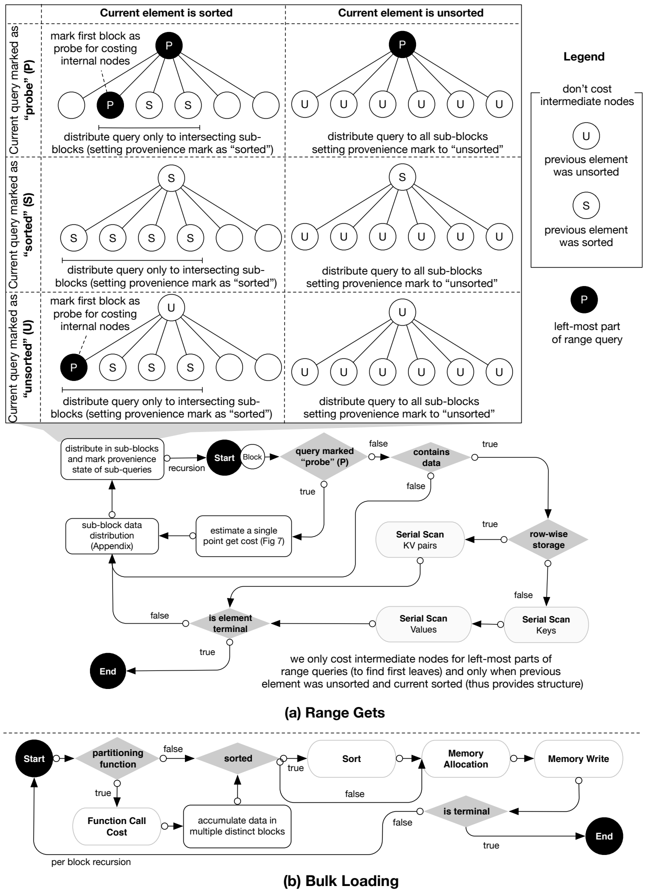


**Figure 10: Cost synthesis Range Gets and Bulk Loading.**


[70] Michael Stonebraker, Jeff Anton, and Michael Hirohama. 1987. Extendability in POSTGRES. *IEEE Data Engineering Bulletin* 10, 2 (1987), 16–23.

[71] R.E. Tarjan. 1978. Complexity of combinatorial algorithms. *SIAM Rev* (1978).

[72] Toby J. Teorey and K. Sundar Das. 1976. Application of an Analytical Model to Evaluate Storage Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[73] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *PACMPL* 1 (2017), 79:1–79:26.

[74] Leland Wilkinson. 2005. The Grammar of Graphics. *Springer* (2005).

[75] S Bing Yao. 1977. An Attribute Based Model for Database Access Cost Analysis. *ACM Transactions on Database Systems (TODS)* 2, 1 (1977), 45–67.

[76] S. Bing Yao and D. DeJong. 1978. Evaluation of Database Access Paths. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[77] S. Bing Yao and Alan G. Merten. 1975. Selection of File Organization Using an Analytic Model. In *Proceedings of the International Conference on Very Large Data Bases*.

[78] Ming Zhou. 1999. Generalizing Database Access Methods. *Ph.D. Thesis. University of Waterloo* (1999).

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


# A ADDITIONAL RELATED AREAS

**Auto-tuning and Adaptive Systems.** Work on tuning [16, 41] and adaptive systems is also relevant as conceptually any adaptive technique tunes along a part of the design space. For example, work on hybrid data layouts and adaptive indexing automates selection of the right layout [3, 4, 8, 28, 30, 34, 36, 39, 40, 42, 56, 64, 65, 68, 69, 79]. Typically, in these lines of work the layout adapts to incoming requests. Similarly works on tuning via experiments [10], learning [5], and tuning via machine learning [2, 37] can adapt parts of a design using feedback from tests. While there are shared concepts with

```
Function CompleteDesign (Q, ℰ, l, currentPath = [], H)
    if blockReachedMinimumSize(H.page_size) then
        return END_SEARCH;
    if !meaningfulPath(currentPath, Q, l) then
        return END_SEARCH;
    if (cacheHit = cachedSolution(Q, l, H)) != null then
        return cacheHit;
    bestSolution = initializeSolution(cost=∞);
    for E ∈ ℰ do
        tmpSolution = initializeSolution();
        tmpSolution.cost = synthesizeGroupCost(E, Q);
        updateCost(E, Q, tmpSolution.cost);
        if createsSubBlocks(E) then
            Q′ = createQueryBlocks(Q);
            currentPath.append(E);
            subSolution = CompleteDesign(Q′, ℰ, l + 1, currentPath);
            if subSolution.cost != END_SEARCH then
                tmpSolution.append(subSolution);
        if tmpSolution.cost ≤ bestSolution.cost then
            bestSolution = tmpSolution ;
    cacheSolution(Q, l, bestSolution);
    return bestSolution;
```

Algorithm 1: Complete a partial data structure layout specification.

these lines of work, they are all restricted to much smaller design spaces, typically to solve a very specific systems bottleneck, e.g., incrementally building a specific index or smoothly transitioning among specific layouts. The Data Calculator, on the other hand, provides a generic framework to argue about the whole design space of data layouts. Its capability to quickly test the potential performance of a design can potentially lead to new adaptive techniques that will also leverage experience in existing adaptive systems literature to adapt across the massive space drawn by the Data Calculator.

**Data Representation Synthesis.** Data representation synthesis aims for programming languages that automate data structure selection. SETL [20, 21] was the first language to generate structures in the 70s as combinations of existing data structures: array, and linked hash table. A series of works kept providing further functionality, and expanding on the components used [18, 19, 24–27]. Cozy [18] is the latest system; it supports complex retrieval operations such as disjunctions, negations, and inequalities and by uses a library of five data structures: array (sorted and plain), linked list, binary tree, and hash map. These works compose data structure designs out of a small set of existing data structures. This is parallel to the tuning and access path selection problem in databases. The Data Calculator introduces a new vision for what-if design and focuses on two new dimensions: 1) design out of fine-grained primitives, and 2) calculation of the performance cost given a hardware profile and a workload. The focus on fine-grained primitives enables exploration of a massive design space. For example, using the equations of Section 2 for homomorphic two-node designs, a fixed design space of 5 possible elements can generate 25 designs, while the Data Calculator can generate $10^{32}$ designs. The gap grows for polymorphic designs, i.e, $2 * 10^9$ for a 5 element library, while the Data Calculator can generate up to $1.6 * 10^{55}$ valid designs (for a 10M dataset and 4K pages). In addition, the focus on cost synthesis through learned models of fine-grained access primitives means that we can capture hardware and data properties for arbitrary designs. Array Mapped Tries [15] use fine-grained primitives, but the focus is only on trie-based collections and without cost synthesis.

| | | | *Unless otherwise specified, we use a reduced default values domain of 100 values for integers, 10 values for doubles, and 1 value for functions.* | | Hash Table | | | B+Tree/CSB+Tree/FAST | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | LPL | | | | | |
| | | **Primitive** | **Domain** | **size** | **H** | **LL** | **UDP** | **B+** | **CSB+** | **FAST** | **ODP** |
| **Node organization** | 1 | **Key retention.** *No:* node contains no real key data, e.g., intermediate nodes of b+trees and linked lists. *Yes:* contains complete key data, e.g., nodes of b-trees, and arrays. *Function:* contains only a subset of the key, i.e., as in tries. | yes \| no \| function(func) | 3 | no | no | yes | no | no | no | yes |
| | 2 | **Value retention.** *No:* node contains no real value data, e.g., intermediate nodes of b+trees, and linked lists. *Yes:* contains complete value data, e.g., nodes of b-trees, and arrays. *Function:* contains only a subset of the values. | yes \| no \| function(func) | 3 | no | no | yes | no | no | no | yes |
| | 3 | **Key order.** Determines the order of keys in a node or the order of fences if real keys are not retained. | none \| sorted \| k-ary (k: int) | 12 | none | none | none | sorted | sorted | 4-ary | sorted |
| | 4 | **Key-value layout.** Determines the physical layout of key-value pairs.<br>Rules: requires key retention != no or value retention != no. | row-wise \| columnar \| col-row-groups(size: int) | 12 | | | col. | | | | col. |
| | 5 | **Intra-node access.** Determines how sub-blocks (one or more keys of this node) can be addressed and retrieved within a node, e.g., with direct links, a link only to the first or last block, etc. | direct \| head_link \| tail_link \| link_function(func) | 4 | direct | head | direct | direct | direct | direct | direct |
| | 6 | **Utilization.** Utilization constraints in regards to capacity. For example, >= 50% denotes that utilization has to be greater than or equal to half the capacity. | = (X%) \| function(func) \| none *(we currently only consider X=50)* | 3 | none | none | none | >= 50% | >= 50% | >= 50% | none |
| **Node filters** | 7 | **Bloom filters.** A node's sub-block can be filtered using bloom filters. Bloom filters get as parameters the number of hash functions and number of bits. | off \| on(num_hashes: int, num_bits: int) *(up to 10 num_hashes considered)* | 1001 | off | off | off | off | off | off | off |
| | 8 | **Zone map filters.** A node's sub-block can be filitered using zone maps, e.g., they can filter based on mix/max keys in each sub-block. | min \| max \| both \| exact \| off | 5 | off | off | off | min | min | min | off |
| | 9 | **Filters memory layout.** Filters are stored contiguously in a single area of the node or scattered across the sub-blocks.<br>Rules: requires bloom filter != off or zone map filters != off. | consolidate \| scatter | 2 | | | | scatter | scatter | scatter | |
| **Partitioning** | 10 | **Fanout/Radix.** Fanout of current node in terms of sub-blocks. This can either be unlimited (i.e., no restriction on the number of sub-blocks), fixed to a number, decided by a function or the node is terminal and thus has a fixed capacity. | fixed(value: int) \| function(func) \| nited \| terminal(cap: int) *(up to 10 different capacities and up to 10 fixed fanout values are considered)* | 22 | fixed(100) | unlimited | term(256) | fixed(20) | fixed(20) | fixed(16) | term(256) |
| | 11 | **Key partitioning.** Set if there is a pre-defined key partitioning imposed. e.g. the sub-block where a key is located can be dictated by a radix or range partitioning function. Alternatively, keys can be temporaly partitioned. If partitioning is set to none, then keys can be forward or backwards appended. | none(fw-append \| bw-append) \| range() \| radix() \| function (func) \| temporal(size_ratio: int, merge_policy: [tier\| level]) | 205 | range(100) | none(fw) | none(fw) | none(fw) | none(fw) | none(fw) | none(fw) |
| | 12 | **Sub-block capacity.** Capacity of each sub-block. It can either be fixed to a value, or balanced (i.e., all sub-blocks have the same size), unrestricted or functional.<br>Rules: requires key partitioning != none. | fixed(value: int) \| balanced \| stricted \| function(func) *(up to 10 different fixed capacity values are considered)* | 13 | unrestrict. | fixed(256) | | balanced | balanced | balanced | |
| | 13 | **Immediate node links.** Whether and how sub-blocks are connected. | next \| previous \| both \| none | 4 | none | next | none | none | none | none | none |
| | 14 | **Skip node links.** Each sub-block can be connected to another sub-block (not only the next or previous) with skip-links. They can be perfect, randomized or custom. | perfect \| randomized(prob: double) \| function(func) \| none | 13 | none | none | none | none | none | none | none |
| | 15 | **Area-links.** Each sub-tree can be connected with another sub-tree at the leaf level throu area links. Examples include the linked leaves of a B+Tree. | forward \| backward \| both \| none | 4 | none | none | forw. | none | none | none | none |
| **Children layout** | 16 | **Sub-block physical location.** This represents the physical location of the sub-blocks. Pointed: in heap, Inline: block physically contained in parent. Double-pointed: in heap but with pointers back to the parent.<br>Rules: requires fanout/radix != terminal. | inline \| pointed \| double-pointed | 3 | pointed | inline | | pointed | pointed | pointed | |
| | 17 | **Sub-block physical layout.** This represents the physical layout of sub-blocks. Scatter: random placement in memory. BFS: laid out in a breadth-first layout. BFS layer list: hierarchical level nesting of BFS layouts.<br>Rules: requires fanout/radix != terminal. | BFS \| BFS layer(level-grouping: nt) \| scatter *(up to 3 different values for layer-grouping are considered)* | 5 | scatter | scatter | | scatter | BFS | BFS-LL | |
| | 18 | **Sub-blocks homogeneous.** Set to true if all sub-blocks are of the same type.<br>Rules: requires fanout/radix != terminal. | boolean | 2 | true | true | | true | true | true | |
| | 19 | **Sub-block consolidation.** Single children are merged with their parents.<br>Rules: requires fanout/radix != terminal. | boolean | 2 | false | false | | false | false | false | |
| | 20 | **Sub-block instantiation.** If it is set to eager, all sub-blocks are initialized, otherwise they are initialized only when data are available (lazy).<br>Rules: requires fanout/radix != terminal. | lazy \| eager | 2 | lazy | lazy | | lazy | lazy | lazy | |
| | 21 | **Sub-block links layout.** If there exist links, are they all stored in a single array (consolidate) or spread at a per partition level (scatter).<br>Rules: requires immediate node links != none or skip links != none. | consolidate \| scatter | 2 | | scatter | | | | | |
| **Recursion** | 22 | **Recursion allowed.** If set to yes, sub-blocks will be subsequently inserted into a node of the same type until a maximum depth (expressed as a function) is reached. Then the terminal node type of this data structure will be used.<br>Rules: requires fanout/radix != terminal. | yes(func) \| no | 3 | no | no | | yes(logn) | yes(logn) | yes(logn) | |

Total number of property combinations > 10^18 / 60 invalid combinations ≈ 10^16

**Node descriptions: H** : Hash, **LL**: Linked List, **LPL**: Linked Page-List, **UDP**: Unordered Data Page, **B+**: B+Tree Internal Node
**CSB+**: CSB+Tree Internal Node, **FAST**: FAST Internal node, **ODP**: Ordered Data Page (Nodes highlighted with gray are terminal leaf nodes)

**Figure 11: Data layout primitives and synthesis examples of data structures.**

## Data Access Primitives and Fitted Models

| | Data Access Primitives Level 1 (required parameters ; optional parameters) | Model Parameters | Data Access Primitives Layer 2 | Fitted Models |
|---|---|---|---|---|
| 1 | **Scan** | Data Size | Scalar Scan (RowStore, Equal) | Linear Model (1) |
| 2 | (Element Size, Comparison, | | Scalar Scan (RowStore, Range) | Linear Model (1) |
| 3 | Data Layout; None) | | Scalar Scan (ColumnStore, Equal) | Linear Model (1) |
| 4 | | | Scalar Scan (ColumnStore, Range) | Linear Model (1) |
| 5 | | | SIMD-AVX Scan (ColumnStore, Equal) | Linear Model (1) |
| 6 | | | SIMD-AVX Scan (ColumnStore, Range) | Linear Model (1) |
| 7 | **Sorted Search** | Data Size | Binary Search (RowStore) | Log-Linear Model (2) |
| 8 | (Element Size, Data Layout; ) | | Binary Search (ColumnStore) | Log-Linear Model (2) |
| 9 | | | Interpolation Search (RowStore) | Log + LogLog Model (3) |
| 10 | | | Interpolation Search (ColumnStore) | Log + LogLog Model (3) |
| 11 | **Hash Probe** (; Hash Family) | Structure Size | Linear Probing (Multiply-shift [29]) | Sum of Sigmoids (5), Weighted Nearest Neighbors (7) |
| 12 | | | Linear Probing (k-wise independent, k=2,3,4,5) | Sum of Sigmoids (5), Weighted Nearest Neighbors (7) |
| 13 | **Bloom Filter Probe** (; Hash Family) | Structure Size, Number of Hash Functions | Bloom Filter Probe (Multiply-shift [29]) | Sum of Sum of Sigmoids (6), Weighted Nearest Neighbors (7) |
| 14 | | | Bloom Filter Probe (k-wise independent, k=2,3,4,5) | Sum of Sum of Sigmoids (6), Weighted Nearest Neighbors (7) |
| 15 | **Sort** | Data Size | QuickSort | NLogN Model (4) |
| 16 | (Element Size; Algorithm) | | MergeSort | NLogN Model (4) |
| 17 | | | ExternalMergeSort | NLogN Model (4) |
| 18 | **Random Memory Access** | Region Size | Random Memory Access | Sum of Sigmoids (5), Weighted Nearest Neighbors (7) |
| 19 | **Batched Random Memory Access** | Region Size | Batched Random Memory Access | Sum of Sigmoids (5), Weighted Nearest Neighbors (7) |
| 20 | **Unordered Batch Write** | Write Data Size | Contiguous Write (RowStore) | Linear Model (1) |
| 21 | (Layout; ) | | Contiguous Write (ColumnStore) | Linear Model (1) |
| 22 | **Ordered Batch Write** | Write Data Size, | Batch Ordered Write (RowStore) | Linear Model (1) |
| 23 | (Layout; ) | Data Size | Batch Ordered Write (ColumnStore) | Linear Model (1) |
| 24 | **Scattered Batch Write** | Number of Elements, Region Size | ScatteredBatchWrite | Sum of Sum of Sigmoids (6), Weighted Nearest Neighbors (7) |

## Models used for fitting data access primitives

| | Model | Description | Formula |
|---|---|---|---|
| 1 | Linear | Fits a simple line through data | $f(\mathbf{v}) = \mathbf{w}^\top \phi(\mathbf{v}) + y_0; \phi(v) = (v)$ |
| 2 | Log-Linear | Fits a linear model with a basis composed of the identity and logarithmic functions plus a bias | $f(\mathbf{v}) = \mathbf{w}^\top \phi(\mathbf{v}) + y_0; \phi(v) = \begin{pmatrix} v \\ \log v \end{pmatrix}$ |
| 3 | Log + LogLog | Fits a model with log, log log, and linear components | $f(\mathbf{v}) = \mathbf{w}^\top \phi(\mathbf{v}) + y_0; \phi(v) = \begin{pmatrix} v \\ \log v \\ \log\log v \end{pmatrix}$ |
| 4 | NLogN | Fits a model with primarily an NLogN and linear component | $f(\mathbf{v}) = \mathbf{w}^\top \phi(\mathbf{v}) + y_0; \phi(v) = \begin{pmatrix} v \log v \\ v \end{pmatrix}$ |
| 5 | Sum of Sigmoids | Fits a model that has $k$ approximate steps. Seen as sigmoids in $\log x$ as this fits various cache behaviors nicely | $f(x) = \sum_{i=1}^{k} \frac{c_i}{1+e^{-k_i(\log x - x_i)}} + y_0$ |
| 6 | Sum of Sum of Sigmoids | Fits a model which has two cost components, both of which have $k$ approximate steps occuring at the same locations. | $f(x, m) = \sum_{i=1}^{k} \frac{c_{i1}}{1+e^{-k_i(\log x - x_i)}} +$ $(m-1)(\sum_{i=1}^{k} \frac{c_{i2}}{1+e^{-k_i(\log x - x_i)}} + y_1) + y_0$ |
| 7 | Weighted Nearest Neighbors | Takes the $k$ nearest neighbors under the $l_2$ norm and computes a weighted average of their outputs. The input $x$ is allowed to be a vector of any size. | Let $x_1, \ldots x_k$ be the nearest neighbors of $x$ with costs $y_1, \ldots, y_k$. Then $f(x) = \frac{1}{\sum_{i=1}^{k} \frac{1}{d(x, x_i)}} \sum_{i=1}^{k} \frac{1}{d(x, x_k)} y_k$ |

Notation: $f$ is a function, $\mathbf{v}$ is a vector, and x, m are scalars. $\log(\mathbf{v})$ returns a vector with log applied on an element by element basis to $\mathbf{v}$; i.e. if $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$, then $\log v = \begin{pmatrix} \log v_1 \\ \log v_2 \end{pmatrix}$. Finally, if we have vectors $v^{(1)}, v^{(2)}$ of lengths $n, m$ stacked on each other as $\begin{pmatrix} v^{(1)} \\ v^{(2)} \end{pmatrix}$, then this signifies the $n + m$ length vector produced by stacking the entries of $v^{(1)}$ on top of the entries of $v^{(2)}$; i.e. $\begin{pmatrix} v^{(1)} \\ v^{(2)} \end{pmatrix} = \left(v_1^{(1)}, \ldots, v_n^{(1)}, v_1^{(2)}, \ldots, v_m^{(2)}\right)^\top$.

**Table 1: Data access primitives and models used for operation cost synthesis.**