Self-Adapting Index Compilation

Darshana Balakrishnan
University at Buffalo
dbalakri@buffalo.edu

Lukasz Ziarek
University at Buffalo
lziarek@buffalo.edu

Oliver Kennedy
University at Buffalo
okennedy@buffalo.edu

ABSTRACT

Creating or modifying a primary index is a time-consuming process, as the index typically needs to be rebuilt from scratch. In this paper, we explore a more graceful "just-in-time" approach to index reorganization, where small changes are dynamically applied in the background. To enable this type of reorganization, we formalize a composable organizational grammar, expressive enough to capture instances of not only existing index structures, but arbitrary hybrids as well. We introduce an algebra of rewrite rules for such structures, and a framework for defining and optimizing policies for just-in-time rewriting. Our experimental analysis shows that the resulting index structure is flexible enough to adapt to a variety of performance goals, while also remaining competitive with existing structures like the C++ standard template library map.

1 INTRODUCTION

An in-memory index is backed by a data structure that stores and facilitates access to records. An alphabet soup of such data structures have been developed to date ([6, 7, 10, 12, 19–22, 26, 30] to list only a few). Each structure targets a specific trade-off between a range of performance metrics (e.g., read cost, write cost), resource constraints (e.g., memory, cache), and supported functionality (e.g. range scans or out-of-core storage). As a trivial example, contrast linked lists with a sorted arrays: The former provides fast writes and slow lookups, while the latter does exactly the opposite.

Creating or modifying an in-memory index is a time-consuming process, since the data structure backing the index typically needs to be rebuilt from scratch when its parameters change. During this time, the index is unusable, penalizing the performance of any database relying on it. In this paper, we propose a more graceful approach to runtime index adaptation. Self-Adapting Indexes (SAIs) continuously make small, incremental reorganizations in the background, while client threads continue to access the structure. Each reorganization brings the SAI closer to a state that mimicks a specific target data structure. As illustrated in Figure 1, the performance of a SAI continuously improves as it transitions from one state to another, while other data structures improve only after fixed investments of organizational effort.

Three core challenges must be addressed to realize SAIs. First, because each individual step is small, at any given

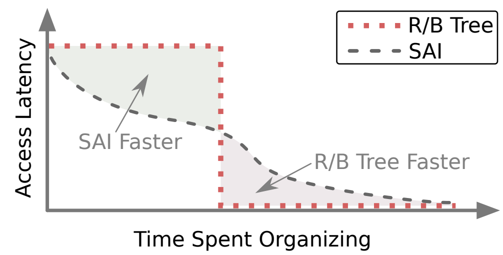


Figure 1: A classical index data structure provides no benefits until ready, while SAIs provide continuous incremental performance improvements.

point in time an SAI may need to be in some intermediate state between two classical data structures. For example, an SAI transitioning from a linked list to a binary tree may need to occupy a state that is neither linked list, nor binary tree, but some combination of the two. Second, there may be multiple pathways to transition from a given source state to the desired target state. For example, to get from an unsorted array to a sorted array, we might sort the array (faster in the long-term) or crack [12] the array (more short-term benefits). Finally, we want to avoid blocking client access to the SAI while it is being reorganized. Client threads should be able to query the structure while the background thread works.

We address the first challenge by building on just-in-time data structures [17], a form of adaptive index that dynamically assembles indexes from composable, immutable building blocks. Mimicking the behavior of a just-in-time compiler, a just-in-time data structure dynamically reorganizes building blocks to improve index performance. Our main contributions in this paper address the remaining challenges.

We first precisely characterize the space of available state transitions by formalizing the behavior of just-in-time data structures into a composable organizational grammar (COG). A sentence in COG corresponds directly to a specific physical layout. Many classical data structures like binary trees, linked lists, and arrays are simply syntactic restrictions on COG. Lifting these restrictions allows intermediate hybrid structures that combine elements of each. Thus, the grammar can precisely characterize any possible state of a SAI.

Next, we define *transforms*, syntactic rewrite rules over COG and show how these rewrite rules can be combined into a *policy* that dictate how and where transforms should be applied. This choice generally requires runtime decisions, so we identify a specific family of "local hierarchical" policies

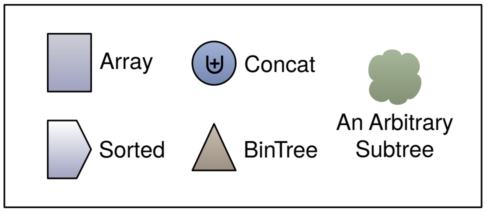


Figure 2: Node types in a SAI

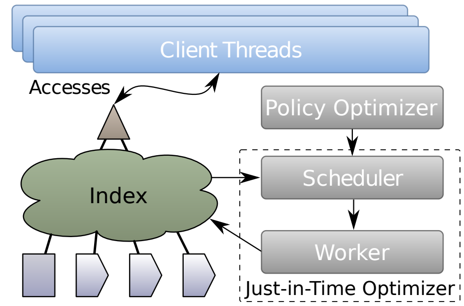


Figure 3: A SAI

in which runtime decisions can be implemented by an efficiently maintainable priority heap. As an example, we define a family of policies for transitioning between unsorted and sorted arrays (e.g., for interactive analysis on a data file that has just been loaded [1]).

To automate policy design, we provide a simulator framework that predictively models the performance of a `SAI` under a given policy. The simulator can generate performance-over-time curves for a set of potential policies. These curves can then be queried to find a policy that best satisfies user desiderata like "get to 300*ms* lookups as soon as possible" or "give me the best scan performance possible within 5*s*".

Finally, we address the issue of concurrency by proposing a new form of "semi-functional" data structure. Like a functional (immutable) data structure, elements of a semi-functional data structure are stable once created. However, using handle-style [11] pointer indirection, we draw a clear distinction between code that expects physical stability and code that merely expects logical stability. In the latter case correctness is preserved even if the element is modified, so long as the element's logical content remains unchanged.

1.1 System Overview

A Self-Adjusting Index (`SAI`) is a key-value style primary (clustered) index storing a collection of records, each (non-uniquely) identified by a key with a well defined sort order. As illustrated in Figure 3, a `SAI` consists of three parts: an index, an optimizer, and a policy simulator. The `SAI`'s index is a tree rooted at a node designated `root`. Following just-in-time data structures, `SAI`s use four types of nodes, summarized in Figure 2: (1) **Array**: A leaf node storing an unsorted array of records, (2) **Sorted**: A leaf node storing a *sorted* array of records, (3) **Concat**: An inner node pointing to two additional nodes, and (4) **BinTree**: A binary tree node that segments records in the two nodes it points to by a separator value.

The second component of `SAI` is a just-in-time optimizer, an asynchronous process that incrementally reorganizes the index, progressively rewriting its component parts to adapt it to the currently running workload. These rewrites are guided by a *policy*, a set of rules for identifying points in the index to be rewritten and for determining what rewrites to apply. To help users to select an appropriate policy, `SAI` includes a policy simulator that generates predicted performance over time curves for specific policies. This simulator can be used to quickly compare policies, helping users to select the policy that best meets the user's requirements for latency, preparation time, or throughput.

1.2 Access Paths

A `SAI` provides lock-free access to its contents through *access paths* that recursively traverse the index: (1) `get(key)` returns the first record with a target key, (2) `iterator(lower)` returns an un-ordered iterator over records with keys greater than or equal to `lower`, and (3) `ordered_iterator(lower)` returns an iterator over the same records, but in key order. As an example, Algorithm 1 implements the first of these access paths by recursively descending through the index. Semantic constraints on the layout provided by **Sorted** and **BinTree** are exploited where they are available.

Algorithm 1 `Get`(C, k)

Require: C: A `SAI` node $\quad$ k: A key
Ensure: r: A record with key k or **None** if none exist.
  if C **matches Array**($\vec{r}$) **then**
    return `linearScan`($k, \vec{r}$)
  else if C **matches Sorted**($\vec{r}$) **then**
    return `binarySearch`($k, \vec{r}$)
  else if C **matches Concat**(C_1, C_2) **then**
    r = `Get`(C_1, k)
    if $r \neq$ **None then return** r
    else return `Get`(C_2, k)
  else if C **matches BinTree**(k', C_1, C_2) **then**
    if $k' \leq k$ **then return** `Get`(C_2, k)
    else return `Get`(C_1, k)

1.3 Updates

Organizational effort in a `SAI` is entirely offloaded to the just-in-time optimizer. Client threads performing updates do the minimum work possible to register their changes. To insert, the updating thread instantiates a new **Array** node C and creates a subtree linking it and the current index root:

Concat(`root`, C)

This subtree becomes a new version of the root. Although only one thread may update the index at a time, updates can proceed concurrently with the background worker thread. This is achieved through a layer of indirection called a *handle* that we introduce and discuss further in Section 5.

1.4 Organization and Policy

The background worker thread is responsible for iteratively rewriting fragments of the index into (hopefully) more efficient forms. It needs (1) to identify fragments of the structure that need to be rewritten, (2) to decide how to rewrite those fragments, and (3) to decide how to prioritize these tasks. We address the first two challenges by defining a fixed set of transformations for `SAI`. Like rewrite rules in an optimizing compiler, transformations replace subtrees of the grammar with logically equivalent structures. Following this line of thought, we first develop a formalism that treats the state of the index at any point in time as a sentence in a grammar over the four node types. We show that transformations can be expressed as structural rewrites over this grammar, and that for any sentence (i.e., index instance) we can enumerate the sentence fragments to which a transformation can be successfully applied. A policy that balances the trade-offs between different types of transformations is then defined to prioritize which transforms should be applied and when.

1.5 Paper Outline

The remainder of this paper is organized as follows.

Encoding hybrid index structures. In Section 2, we introduce and formalize the COG grammar and show how it allows us to encode a wide range of tree-structured physical data layouts. These include restricted sub-grammars that capture, for example, singly linked lists or binary trees. The grammar can express transitional physical layouts that combine elements of multiple classes of data structure.

Data structure transitions as an algebra. We next outline an algebra over the COG grammar in Section 3. Specifically we introduce the concept of transforms, rewrites on the structure of a sentence in COG that preserve logical equivalence and syntactic constraints over the structure.

Combining transforms into a policy. Next, in Section 4, we show how sequences of transforms, guided by a policy, may be used to incrementally re-organize an index. In order to remain competitive with classical index structures, policies need to make split-second decisions on which transforms to apply. Accordingly, we identify a specific class of local hierarchical policies that can be implemented via an incrementally maintained priority queue that tracks organizational goals and efficiently selects transforms.

Implementation and runtime. After providing a theoretical basis for `SAI`s, we describe how we addressed key challenges in implementing them, outline the primary components of the `SAI` runtime, and provide an illustrative example policy: Crack-or-Sort.

Policy optimization. Section 6 introduces a `SAI` simulator. This simulator emulates the evolution of a `SAI`, allowing us to efficiently determine which of a range of alternative policies best meets user-provided performance goals for transitioning between index structures.

Assessing `SAI`'s generality. Section 7 uses a taxonomy of index data structures proposed in [16] to evaluate `SAI`'s generality. We propose three ideas for future work that could fully generalize 19 of the 22 design dimensions identified.

Evaluation. Finally, in Section 8, we assess the performance overheads `SAI`s, relative to both commonly used and state-of-the art in-memory indexes.

2 A GRAMMAR OF DATA STRUCTURES

Each record $r \in \mathcal{R}$ is accessed exclusively by a (potentially non-unique) identifier $\mathtt{id}(r) \in \mathcal{I}$. We assume a total order $\leq$ is defined over elements of $\mathcal{I}$. We abuse syntax and use records and keys interchangeably with respect to the order, writing $r \leq k$ to mean $\mathtt{id}(r) \leq k$. We write $[\tau]$, $\{\tau\}$, and $\{\!|\, \tau \,|\!\}$ to denote the type of arrays, sets, and bags (respectively) with elements of type τ. We write $[r_1, \ldots, r_N]$ (resp., $\{\ldots\}$, $\{\!|\ldots|\!\}$) to denote an array (or set or bag) with elements $r_1, \ldots, r_N$.

To support incremental index transitions, we need a way to represent intermediate states of an index, part way between one physical layout and another. In this section we propose a compositional organizational grammar (COG) that will allow us to reason about the state of a `SAI`, and the correctness of its state transitions.

2.1 Notation and Definitions

The atoms of COG are defined by four symbols **Array**, **Sorted**, **Concat**, **BinTree**. A COG *instance* is a sentence in COG, defined by the grammar $\mathcal{C}$ as follows:

$$\begin{array}{rcl} \mathcal{C} & = & \mathbf{Array}([\mathcal{R}]) \;\mid\; \mathbf{Sorted}([\mathcal{R}]) \\ & \mid & \mathbf{Concat}(\mathcal{C}, \mathcal{C}) \;\mid\; \mathbf{BinTree}(\mathcal{I}, \mathcal{C}, \mathcal{C}) \end{array}$$

Atoms in COG map directly to physical building blocks of a data structure, while atom instances correspond to instances of a data structure or one of its sub-structures. For example an instance of **Array** represents an array of records laid out contiguously in memory, while **Concat** represents a tuple of pointers referencing other instances. We write $\mathbf{typeof}(C)$ to denote the atom symbol at the root of an instance $C \in \mathcal{C}$.

Example 1 (Linked List). *A linked list may be defined as a syntactic restriction over* cog *as follows*

$$\mathcal{LL} \quad = \quad \textbf{\textit{Concat}}(\textbf{\textit{Array}}([\mathcal{R}]), \mathcal{LL}) \;\mid\; \textbf{\textit{Array}}([\mathcal{R}])$$

A linked list is either a concatenation of an array (with one element by convention), and a pointer to the next element, or a terminal array (with no elements by convention).

Two different instances, corresponding to different representations may still encode the same data. We describe the logical contents of an instance C as a bag, denoted by $\mathbb{D}(C)$, and use this term to define logical equivalence between two instances.

$$\mathbb{D}(C) = \begin{cases} \{\!|\, r_1, \ldots, r_N \,|\!\} & \textbf{if } C = \textbf{Array}([r_1, \ldots, r_N]) \\ \{\!|\, r_1, \ldots, r_N \,|\!\} & \textbf{if } C = \textbf{Sorted}([r_1, \ldots, r_N]) \\ \mathbb{D}(C_1) \uplus \mathbb{D}(C_2) & \textbf{if } C = \textbf{Concat}(C_1, C_2) \\ \mathbb{D}(C_1) \uplus \mathbb{D}(C_2) & \textbf{if } C = \textbf{BinTree}(\_, C_1, C_2) \end{cases}$$

Definition 1 (Logical Equivalence). *Two instances C_1 and C_2 are* logically equivalent *if and only if $\mathbb{D}(C_1) = \mathbb{D}(C_2)$. To denote logical equivalence we write $C_1 \approx C_2$.*

We write C^* to denote the bag consisting of the instance C and its descendants.

$$C^* = \begin{cases} C_1^* \uplus C_2^* \uplus \{\!|\, C \,|\!\} & \textbf{if } C = \textbf{Concat}(C_1, C_2) \\ C_1^* \uplus C_2^* \uplus \{\!|\, C \,|\!\} & \textbf{if } C = \textbf{BinTree}(\_, C_1, C_2) \\ \{\!|\, C \,|\!\} & \textbf{otherwise} \end{cases}$$

Proposition 1. *The set C^* is finite for any C.*

2.2 cog Semantics

Array and **Concat** represent the physical layout of elements of a data structure. The remaining two atoms provide provide semantic constraints (using the identifier order $\preceq$) over the physical layout that can be exploited to make the structure more efficient to query. We say that instances satisfying these constraints are *structurally correct.*

Definition 2 (Structural Correctness). *We define the structural correctness of an instance $C \in \mathcal{C}$ (denoted by the unary relation* StrCor(C)*) for each atom individually:*

Case 1. ***Array*** *instance is structurally correct.*

Case 2. *The instance* ***Concat***(C_1, C_2) *is structurally correct if and only if C_1 and C_2 are both structurally correct.*

Case 3. *The instance* ***Sorted***$([r_1, \ldots, r_N])$ *is structurally correct if and only if $\forall 0 \leq i < j \leq N \;:\; r_i \leq r_j$*

Case 4. *The instance* ***BinTree***(k, C_1, C_2) *is structurally correct if and only if both C_1 and C_2 are structurally correct, and if $\forall r_1 \in \mathbb{D}(C_1) : r_1 \prec k$ and $r_2 \in \mathbb{D}(C_2) \;:\; k \preceq r_2$.*

In short, **Sorted** is structurally correct if it represents a sorted array. Similarly, **BinTree** is structurally correct if it corresponds to a binary tree node, with its children partitioned by its identifier. Both **Concat** and **BinTree** additionally require that their children be structurally correct.

Example 2 (Binary Tree). *A binary tree may be defined as a syntactic restriction over* cog *as follows*

$$\mathcal{B} \quad = \quad \textbf{\textit{BinTree}}(\mathcal{I}, \mathcal{B}, \mathcal{B}) \;\mid\; \textbf{\textit{Array}}([\mathcal{R}])$$

A binary tree is a hierarchy of ***BinTree*** *inner nodes, over* ***Array*** *leaf nodes (containing one element by convention).*

3 TRANSFORMS OVER COG

We next formalize state transitions in a SAI through pattern-matching rewrite rules over cog called transforms.

Definition 3 (Transform). *We define a transform T as any member of the family $\mathcal{T}$ of endomorphisms over* cog *instances. Equivalently, any transform $T \in \mathcal{T}$ is a morphism $T : \mathcal{C} \to \mathcal{C}$ from instance to instance.*

Figure 4 illustrates a range of common transforms that correspond to common operations on index structures. For consistency, we define transforms over all instances and not just instances where the operation "makes sense." On other instances, transforms behave as the identity ($\textbf{id}(C) = C$).

Clearly not all possible transforms are useful for organizing data. For example, the well defined, but rather unhelpful transform $\textbf{Empty}(C) = \textbf{Array}([])$ transforms any cog instance into an empty array. To capture this notion of a "useful" transform, we define two correctness properties: structure preservation and equivalence preservation.

Definition 4 (Equivalence Preserving Transforms). *A transform T is defined to be* equivalence preserving *if and only if $\forall C : C \approx T(C)$ (Definition 1).*

Definition 5 (Structure Preserving Transforms). *A transform T is defined to be* structure preserving *if and only if $\forall C :$ StrCor$(C) \implies$ StrCor$(T(C))$ (Definition 2).*

A transform is equivalence preserving if it preserves the logical content of the instance. It is structure preserving if it preserves the structure's semantic constraints (e.g., the record ordering constraint on instances of the **Sorted** atom). If it is both, we say that the transform is correct.

Definition 6 (Correct Transform). *We define a transform T to be* correct *(denoted* Correct(T)*) if T is both structure and equivalence preserving.*

In Appendix A we give proofs of correctness for each of the transforms in Figure 4.

3.1 Meta Transforms

Transforms such as those illustrated in Figure 4 form the atomic building blocks of a policy for re-organizing data

$$\textbf{Sort}(C) \;=\; \begin{cases}\textbf{Sorted}(\texttt{sort}(\vec{r}\,)) & \textbf{if } C = \textbf{Array}(\vec{r}\,)\\ C & \textbf{otherwise}\end{cases} \qquad \textbf{UnSort}(C) \;=\; \begin{cases}\textbf{Array}(\vec{r}\,) & \textbf{if } C = \textbf{Sorted}(\vec{r}\,)\\ C & \textbf{otherwise}\end{cases}$$

$$\textbf{Divide}(C) \;=\; \begin{cases}\textbf{Concat}(\textbf{Array}(\left[r_1 \ldots r_{\lfloor \frac{N}{2} \rfloor}\right]), \textbf{Array}(\left[r_{\lfloor \frac{N}{2} \rfloor+1} \ldots r_N\right])) & \textbf{if } C = \textbf{Array}([r_1 \ldots r_N])\\ C & \textbf{otherwise}\end{cases}$$

$$\textbf{Crack}(C) \;=\; \begin{cases}\textbf{BinTree}(\texttt{id}\left(r_{\lfloor \frac{N}{2} \rfloor}\right), \textbf{Array}(\left[r_i \;\middle|\; r_i \prec r_{\lfloor \frac{N}{2} \rfloor}\right]), \textbf{Array}(\left[r_i \;\middle|\; r_{\lfloor \frac{N}{2} \rfloor} \preceq r_i\right]) & \textbf{if } C = \textbf{Array}([r_1 \ldots r_N]))\\ C & \textbf{otherwise}\end{cases}$$

$$\textbf{Merge}(C) \;=\; \begin{cases}\textbf{Array}([r_1 \ldots r_N, r_{N+1} \ldots r_M]) & \textbf{if } C = \textbf{Concat}(\textbf{Array}([r_1 \ldots r_N]), \textbf{Array}([r_{N+1} \ldots r_M]))\\ \textbf{Array}([r_1 \ldots r_N, r_{N+1} \ldots r_M]) & \textbf{if } C = \textbf{BinTree}(\_\,, \textbf{Array}([r_1 \ldots r_N]), \textbf{Array}([r_{N+1} \ldots r_M]))\\ C & \textbf{otherwise}\end{cases}$$

$$\textbf{PivotLeft}(C) \;=\; \begin{cases}\textbf{Concat}(\textbf{Concat}(C_1, C_2), C_3) & \textbf{if } C = \textbf{Concat}(C_1, \textbf{Concat}(C_2, C_3))\\ \textbf{BinTree}(k_2, \textbf{BinTree}(k_1, C_1, C_2), C_3) & \textbf{if } C = k_1 \prec k_2 \textbf{ and } \textbf{BinTree}(k_1, C_1, \textbf{BinTree}(k_2, C_2, C_3))\\ C & \textbf{otherwise}\end{cases}$$

Figure 4: Examples of *correct* transforms. Sort and UnSort convert between Array and Sorted and visa versa. Crack and Divide both fragment Arrays, and both are reverted by Merge. Crack in particular uses an arbitrary array element to partition its input value (the $\frac{N}{2}$th element in this example), analogous to the RadixCrack operation of [15]. PivotLeft rotates tree structures counterclockwise and a symmetric PivotRight may also be defined. The function `sort` : $[\mathcal{R}] \to [\mathcal{R}]$ returns a transposition of its input sorted according to $\preceq$.

structures. For the purposes of this paper, we refer to these six transforms, together with **PivotRight** and the identity transform **id**, collectively as the *atomic transforms*, denoted $\mathcal{A}$. We next introduce a framework for constructing more complex transforms from these building blocks.

Definition 7 (Composition). *For any two transforms $T_1, T_2 \in \mathcal{T}$, we denote by $T_1 \circ T_2$ the composition of T_1 and T_2:*

$$(T_1 \circ T_2)(C) \stackrel{def}{=} T_2(T_1(C))$$

Transform composition allows us to build more complex transforms from the set of atomic transforms. We also consider *meta transforms* that manipulate transform behavior.

Definition 8 (Meta Transform). *A meta transform M is any correctness-preserving endofunctor over the set of transforms. That is, any functor $M : \mathcal{T} \to \mathcal{T}$ is a meta transform if and only if $\forall T \in \mathcal{T} : \textsc{Correct}(T) \implies \textsc{Correct}(M[T])$ (Definition 6).*

We are specifically interested in two meta transforms that will allow us to apply transforms not just to the root of an instance, but to any of its descendants as well.

$$\textbf{LHS}[T](C) = \begin{cases}\textbf{Concat}(T(C_1), C_2) & \textbf{if } C = \textbf{Concat}(C_1, C_2)\\ \textbf{BinTree}(k, T(C_1), C_2) & \textbf{if } C = \textbf{BinTree}(k, C_1, C_2)\\ C & \textbf{otherwise}\end{cases}$$

$$\textbf{RHS}[T](C) = \begin{cases}\textbf{Concat}(C_1, T(C_2)) & \textbf{if } C = \textbf{Concat}(C_1, C_2)\\ \textbf{BinTree}(k, C_1, T(C_2)) & \textbf{if } C = \textbf{BinTree}(k, C_1, C_2)\\ C & \textbf{otherwise}\end{cases}$$

Theorem 1 (**LHS** and **RHS** are meta transforms). ***LHS** and **RHS** are correctness-preserving endofunctors over $\mathcal{T}$.*

The proof, given in Appendix B, is a simple structural recursion over cases.

We refer to the closure of **LHS** and **RHS** over the atomic transforms as the set of *hierarchical transforms*, denoted Δ.

$$\Delta = \mathcal{A} \cup \{\, \textbf{LHS}[T] \mid T \in \Delta \,\} \cup \{\, \textbf{RHS}[T] \mid T \in \Delta \,\}$$

Corrolary 1. *Any hierarchical transform is correct.*

4 POLICIES FOR TRANSFORMS

Transforms give us a means of manipulating instances, but to actually allow an index to transition from one form to another we need a set of rules, called a *policy*, to dictate which transform to apply and when. We begin by defining policies broadly, before refining them into an efficiently implementable family of *enumerable score-based policies*.

Definition 9 (Policy). *A policy $\mathcal{P}$ is defined by the 2-tuple $\mathcal{P} = \langle\, \mathcal{D}, \mathcal{H} \,\rangle$, where the policy's* domain *$\mathcal{D} \subseteq \mathcal{T}$ is a set of transforms and $\mathcal{H} : C \to \mathcal{D}$ is a heuristic function that selects one of these transforms to apply to a given instance.*

A policy guides the transition of an index from an instance representing its initial state to a final state achieved by repeatedly applying transforms selected by the heuristic $\mathcal{H}$. We call the sequence of instances reached in this way a *trace*.

Definition 10 (Trace). *The trace of a policy $\mathcal{P} = \langle\ \mathcal{D}, \mathcal{H}\ \rangle$ on instance C_0, denoted $\textbf{Trace}(\mathcal{P}, C_0)$, is defined as the infinite sequence of instances $[C_0, C_1, \ldots]$ starting with C_0, and with subsequent instances C_i obtained as:*

$$C_i \stackrel{def}{=} T_i(C_{i-1}) \quad \textbf{\textit{where}} \quad T_i = \mathcal{H}(C_{i-1})$$

Although traces are infinite, we are specifically interested in policies with traces that reach a steady (fixed point) state. We say that such a trace (resp., any policy guaranteed to produce such a trace) is *terminating*.

Definition 11 (Terminating Trace, Policy). *A trace* $[C_1, C_2, \ldots]$ terminates *after N steps if and only if $\forall i, j > N$: $C_i = C_j$,. A policy $\mathcal{P}$ is* terminating *when*

$$\forall C \exists N : \textbf{\textit{Trace}}(\mathcal{P}, C) \textbf{\textit{ terminates after }} N \textbf{\textit{ steps}}$$

A policy's domain may be large, or even infinite as in the case of the hierarchical transforms. However, only a much smaller fragment will typically be useful for any specific instance. We call this fragment the *active domain*.

Definition 12. *The* active domain *of a policy $\langle\ \mathcal{D}, \mathcal{H}\ \rangle$, relative to an instance C (denoted $\mathcal{D}_C$) is the subset of the policy's domain that does not behave as the identity on C.*

$$\mathcal{D}_C \stackrel{def}{=} \{\ T \mid T \in \mathcal{D} \wedge T(C) \neq C\ \}$$

4.1 Bounding the Active Domain

A policy's heuristic function will be called numerous times in the course of an index transition, making it a prime candidate for performance optimization. We next explore one particular family of policies that admit a stateful, incremental implementation of their heuristic function. This approach treats the heuristic function as a ranking query over the active domain, selecting the most appropriate (highest scoring) transform at any given time. However, rather than recomputing scores at every step, we incrementally maintain a priority queue over the active domain. For this incremental approach to be feasible, we need to ensure that only a finite (and ideally small) number of scores change with each step.

Definition 13 (Enumerable Policy). *A policy $\langle\ \mathcal{D}, \mathcal{H}\ \rangle$ is enumerable if and only if its active domain is finite for every finite instance C, or equivalently when $\forall C : |\mathcal{D}_C| \in \mathbb{N}$*

We are particularly interested in policies that use the hierarchical transforms as their domain. We also refer to such policies as *hierarchical*. In order to show that hierarchical policies are enumerable, we first define a utility *target* function that "unrolls" an arbitrarily deep stack of **LHS** and **RHS** meta transforms. The target function returns (1) The atomic transform at the base of the stack of meta transforms and (2) the descendant that this atomic transform would be applied to.

Definition 14. *Given a hierarchical policy $\langle\ \Delta, \mathcal{H}\ \rangle$ and an instance C, let the* target function $f^*_C : \mathcal{D}_C \to (C^* \times \mathcal{A})$ *of the policy on C is defined as follows*

$$f^*_C(T) \stackrel{def}{=} \begin{cases} \langle\ C, T\ \rangle & \textbf{\textit{if typeof}}(C) \in \{\textbf{\textit{Array}}, \textbf{\textit{Sorted}}\} \\ \langle\ C, T\ \rangle & \textbf{\textit{else if}}\ T \in \mathcal{A} \\ f^*_{C_1}(T') & \textbf{\textit{else if}}\ T = \textbf{\textit{LHS}}[T'] \\ f^*_{C_2}(T') & \textbf{\textit{else if}}\ T = \textbf{\textit{RHS}}[T'] \end{cases}$$

Lemma 1 (Injectivity of f^*_C). *The target function f^*_C of any hierarchical policy $\langle\ \Delta, \mathcal{H}\ \rangle$ for any instance C is injective.*

Proof. By recursion over C. The base case occurs when $\textbf{typeof}(C) \in \{\textbf{Array}, \textbf{Sorted}\}$. In this case $C^* = \{\!|\, C \,|\!\}$. Furthermore, $\forall T$: $\textbf{LHS}[T] = \textbf{RHS}[T] = \textbf{id}$ and so $\mathcal{D}_C \subseteq \mathcal{A}$. The target function always follows its first case and is trivially injective. The first recursive case occurs for $C = \textbf{Concat}(C_1, C_2)$. By definition, a hierarchical transform can be (1) An atomic transform, (2) $\textbf{LHS}[T]$, or (3) $\textbf{RHS}[T]$. Each of the latter three cases covers one of each of the three parts of the definition of a hierarchical transform. Assuming that $f^*_{C_1}$ and $f^*_{C_2}$ are injective, f^*_C will also be injective because each case maps to a disjoint partition of $C^* = C^*_1 \uplus C^*_2 \uplus \{\!|\, C \,|\!\}$. The proof for the second recursive case, where $\textbf{typeof}(C) = \textbf{BinTree}$ is identical. Thus $\forall C : f^*_C$ is injective □

Using injectivity of the target function, we can show that any hierarchical policy is enumerable.

Theorem 2 (Hierarchical policies are enumerable). *Any hierarchical policy $\langle\ \Delta, \mathcal{H}\ \rangle$ is enumerable.*

Proof. Recall the definition of hierarchical transforms

$$\Delta = \mathcal{A} \cup \{\ \textbf{LHS}[T] \mid T \in \Delta\ \} \cup \{\ \textbf{RHS}[T] \mid T \in \Delta\ \}$$

By Lemma 1, $|\mathcal{D}_C| \leq |C^* \times \mathcal{A}| \leq |C^*| \times |\mathcal{A}|$. By Proposition 1, C^* is finite and the set of atomic transforms $\mathcal{A}$ is finite by definition Thus, $\mathcal{D}_C$ must also be finite. □

Intuitively, there is a finite number of atomic transforms ($|\mathcal{A}$), that can be applied at a finite set of positions within C ($|C^*$). Any other hierarchical transform must be idempotent, so we can (very loosely) bound the active domain of a hierarchical policy on instance C by $|C^*| \times |\mathcal{A}|$

4.2 Scoring Heuristics

As previously noted, we are particularly interested in policies that work by scoring the set of available transforms with respect to their utility.

Definition 15 (Scoring Policy). *Let* $\texttt{score} : (\mathcal{D} \times C) \to \mathbb{N}_0$ *be a* scoring function *for every transform, instance pair* (T, C) *that satisfies the constraint:* $(T(C) = C) \Rightarrow (\texttt{score}(T, C) = 0)$ *A* scoring policy $\langle \mathcal{D}, \mathcal{H}_{\texttt{score}} \rangle$ *is a policy with a heuristic function defined as* $\mathcal{H}_{\texttt{score}}(C) \stackrel{def}{=} \boldsymbol{argmax}_{T \in \mathcal{D}}(\texttt{score}(T, C))$

In short, a scoring heuristic policy one that selects the next transform to apply based on a scoring function score, breaking ties arbitrarily. Additionally, we require that transforms not in the active domain (i.e., that leave their inputs unchanged) must be assigned the lowest score (0).

As we have already established, the number of scores that need to be computed is finite and enumerable. However, it is also linear in the number of atoms in the instance. Ideally, we would like to avoid recomputing all of the scores at each iteration by precomputing the scores once and then incrementally maintaining them as the instance is updated. For this to be feasible, we also need to bound the number of scores that change with each step of the policy. We do this by first defining two properties of policies: independence, which requires that the score of a (hierarchical) transform be exclusively dependent on its target atom (Definition 14); and locality, which further requires that the score of a transform be independent of the node's descendants past a bounded depth. We then show that with any scoring function that satisfies these properties, only a finite number of scores change with any transform, and consequently that the output of the scoring function on every element of the active domain can be efficiently incrementally maintained.

Definition 16 (Independent Policy). *Let* $C^{<}$ *be the set of instances with* C *as a left child.*

$$\begin{aligned} C^{<} = \{\ & \boldsymbol{Concat}(C, C') \mid C' \in \mathbb{C}\ \} \\ & \cup \{\ \boldsymbol{BinTree}(k, C, C') \mid k \in \mathbb{I} \wedge C' \in \mathbb{C}\ \} \end{aligned}$$

and define $C^{>}$ *symmetrically as the set of instances with* C *as a right child. We say that a hierarchical scoring policy* $\langle \Delta, \mathcal{H}_{\texttt{score}} \rangle$ *is* independent *if and only if for any* T, C

$$\begin{aligned} \forall C' \in C^{<} : \texttt{score}(T, C) &= \texttt{score}(\boldsymbol{LHS}[T], C') \\ \forall C' \in C^{>} : \texttt{score}(T, C) &= \texttt{score}(\boldsymbol{RHS}[T], C') \end{aligned}$$

Definition 17 (Local Policy). *An independent hierarchical scoring policy* $\langle \Delta, \mathcal{H}_{\texttt{score}} \rangle$ *is local if and only if:*

$$\begin{aligned} \forall T \forall C_1 \forall C_2\ \boldsymbol{s.t.}\ (C_1^* - \{\!|\, C_1 \,|\!\}) = (C_2^* - \{\!|\, C_2 \,|\!\}) : \\ \texttt{score}(T, C_1) = \texttt{score}(T, C_2) \end{aligned}$$

The following definition uses the policy's target function (Definition 14) to define a weighted list of all of the policy's targets.

Definition 18 (Weighted Targets). *Let* $\langle \Delta, \mathcal{H}_{\texttt{score}} \rangle$ *be a hierarchical scoring policy. The weighted targets of instance* C*, denoted* $\mathcal{W}_C : \{\!|\, \mathcal{A} \times \mathbb{N}_0 \,|\!\}$ *is bag of 2-tuples defined as*

$$\mathcal{W}_C = \left\{\!\!\left|\ \langle\ T', \texttt{score}(T, C')\ \rangle\ \middle|\ T \in \mathcal{D}_C \wedge (C', T') = f_C^*(T)\ \right|\!\!\right\}$$

Theorem 3 (Bounded Target Updates). *Let* $\langle \Delta, \mathcal{H}_{\texttt{score}} \rangle$ *be a local hierarchical scoring policy,* C *be an instance,* $T \in \mathcal{D}_C$ *be a transform, and* $C' = T(C)$*. The weighted targets of* C *and* C' *differ by at most* $4 \times |\mathcal{A}|$ *elements.*

$$\left|(\mathcal{W}_C \uplus \mathcal{W}_{C'}) - (\mathcal{W}_C \cap \mathcal{W}_{C'})\right| \leq 4 \times |\mathcal{A}|$$

The proof, given in Appendix C, is based on the observation that the independence and locality properties restrict changes to the target function's outputs to exactly the set of nodes added, removed, or modified by the applied transform, excluding ancestors or descendants. In the worst cases (**Divide**, **Crack**, or **Merge**) this is 4 nodes.

Theorem 3 shows that we can incrementally maintain the weighted target list incrementally, as only a finite number of its elements change at any policy step. This allows us to materialize the weighted target list as a priority queue, who's first element is always the policy's next transform.

5 IMPLEMENTING THE SAI RUNTIME

So far, we have introduced cog and shown how policies can be used to gradually reorganize a cog instance by repeatedly applying incremental transforms to the structure.

In this section, we discuss the challenges in translating SAIs from the theory we have defined so far into practice. As already noted, cog instances describe the physical layout of a SAI. We implemented each atom as a C++ class using the reference-counted `shared_ptrs` for garbage collection. To implement the **Array** and **Sorted** atoms, we used the C++ Standard Template Library `vector` class.

5.1 Concurrency and Handles

Because SAIs rely on background optimization, efficient concurrency is critical. This motivated our choice to base the SAI index on functional data structures. In a functional data structure, objects are immutable once instantiated. Only the root may be updated to a new version, typically through an atomic pointer swap. Explicit versioning makes it possible for the background worker thread to construct a new version of the structure without taking out any locks in the process. Only a short lock is required to swap in the new version.

Immutability does come with a cost: any mutations must also copy un-modified data into a new object. However, careful use of pointers can minimize the impact of such copies.

Example 3. *Figure 5.a shows the effects of applying* ***LHS*[*Sort*]** *to an immutable* cog *instance, replacing an unsorted* ***Array*** `X` *with a* ***Sorted*** *equivalent* `X'`*. Note that in addition to replacing* `X`*, each of its ancestors must also be replaced.*

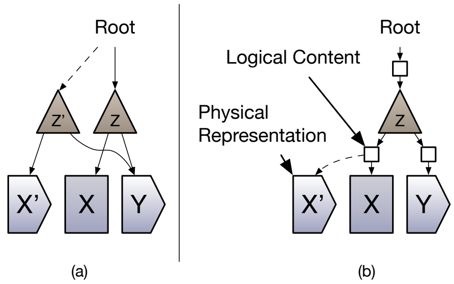


Figure 5: Classical immutable data structures (a) vs with handles (b).

Replacing a logarithmic number of ancestors is better than replacing the entire structure. However, even a logarithmic number of new objects for every update can be a substantial expense when individual transforms can take on the order of micro-seconds. We avoid this overhead by using indirection to allow limited mutability under controlled circumstances. Inspired by early forms of memory management [11], we define a new object called a *handle*.

Handles store a pointer to a COG atom, and all COG atoms (i.e., **BinTree** and **Concat**), as well as the root, use handles as indirect references to their children. Handles provide clear semantics for a programmer expectations: A pointer to an atom guarantees *physical* immutability, while a pointer to a handle guarantees only *logical* immutability. Thus, any thread can safely replace the pointer stored in a handle with a pointer to any other logically equivalent atom. Accordingly, we refer to such structures as *semi-functional* data structures.

EXAMPLE 4. *Continuing the example, Figure 5.b shows the same operation performed on a structure that uses handles. Ancestors of the modified node are unchanged: Only the handle pointer is modified.*

We observe that COG atoms can safely be implemented using handles. The only correctness property we need to enforce is structural correctness, which depends only on the node itself and the *logical* contents ($\mathbb{D}(\cdot)$) of its descendants. Thus only logical consistency is needed and handles suffice.

Similarly, the **LHS** and **RHS** and meta transform creates an exact copy of the root, modulo the affected pointer. Furthermore the only node modified is the one reached by unrolling the stack of meta transforms, and by definition correct transforms must produce a new structure that is logically equivalent. Thus, any hierarchical transform can be safely, efficiently applied to a `SAI` by a single modification to the handle of the target atom (Definition 14).

5.2 Concurrent Access Paths

We have already described the get method in Section 1.1. The remaining access paths instantiate iterators that traverse the tree, lazily dereferencing handles as necessary. Un-ordered iterators provide two methods:

➤ $r \leftarrow$ `Get()` returns the iterator's current record
➤ `Step()` advances the iterator to the next record

Additionally, ordered iterators provide the method:

➤ `Seek(`k`)` advances to the first record r where $r \geq k$

For iterators over **Sorted** and **Array** atoms, we directly use the C++ vector class's iterator. Generating an ordered iterator over an **Array** atom forces a **Sort** first. Iterators for the remaining atom types lazily create a replica of the root instance using only physical references to ensure consistency. Unordered iterators traverse trees left to right. Ordered iterators over **Concat** atoms are implemented using merge-sort. We implement a special-case iterator for **BinTree**s that iterates over contiguous **BinTree**s, lazily loading nodes from their handles as needed.

5.3 Handles and Updates

Handles also make possible concurrency between a `SAI`'s worker thread and threads updating the `SAI`. In keeping with the convention that structures referenced by a handle pointers can only be swapped with logically equivalent structures, a thread updating a `SAI` must replace the root handle with an entirely new handle. Because the worker thread will only ever swap pointers referenced by a handle, it will never undo the effects of an update. Better still, if the old root handle is re-used as part of the new structure (as discussed in Section 1.1), optimizations applied to the old root or any of its descendants will seamlessly be applied to the new version of the index as well.

5.4 Transforms and the Policy Scheduler

Our policy scheduler is optimized for local hierarchical policies. Policies are implemented by defining a scoring function

$$\mathbb{N}_0 \leftarrow \texttt{score}(T,\ C) \textbf{ where } T \in \mathcal{A}$$

Based on this function, the policy scheduler builds a priority queue of 3-tuples $\langle\ \texttt{handle}, \mathcal{A}, \mathbb{N}_0\ \rangle$, including a `handle` to a descendant of the root, an atomic transform to apply to the descendant instance, and the policy's score for the transform applied to the instance referenced by the handle. As an optimization, only the highest-scoring transform for each handle is maintained in the queue. The scheduler iteratively selects the highest scoring transform and applies it to the structure. Handles destroyed (resp., created) by applied transforms are removed from (resp., added to) the priority queue. The iterator continues until no transforms remain in the queue or all remaining transforms have a score of zero, at which point we say the policy has *converged*.

5.5 Example Policy: Crack-Sort-Merge

As an example of policies being used to manage cost/benefit tradeoffs in index structures, we compare two approaches to data loading: database cracking [12] and upfront organization. In a study comparing cracking to upfront indexing, Schuhknecht et. al. [25] observe that for workloads consisting of more than a few scans, it is faster to build an index upfront. Here, we take a more subtle approach to the same problem. The **Crack** transform has lower upfront cost than the **Sort** transform (scaling as $O(N)$ vs $O(N \log N)$), but provides a smaller benefit. Given a fixed time budget or fixed latency goal, is it better to repeatedly crack, sort, or mix the two approaches together. We address this question with a family of scoring functions $\texttt{score}_\theta$, parameterized by a threshold value θ as follows:

$$\texttt{score}_\theta(T, \textbf{Array}([r_1 \dots r_N])) = \begin{cases} N & \textbf{if } T = \textbf{Sort and } N < \theta \\ N & \textbf{if } T = \textbf{Crack and } N < \theta \\ 0 & \textbf{otherwise} \end{cases}$$

Arrays smaller than the threshold are sorted, while those larger are cracked. Larger instances are preferred over smaller. All other instances are ignored. Once all **Array**s are sorted, the resulting **Sorted**s are iteratively **Merge**ed, ultimately leaving behind a single **Sorted**.

This is one example of a parameterized policy, a reorganizational strategy that uses thresholds to guide its behavior. Once such a policy is defined, the next challenge is to select appropriate values for its parameters.

6 POLICY OPTIMIZATION

A SAI's performance curve depends on its policy. As we may have a range of policies to choose from — for example by varying policy parameters as mentioned above — we want a way to evaluate the utility of a policy for a given workload. Naively, we might do this by repeatedly evaluating the structure under each policy, but doing so can be expensive. Instead, we next propose a performance model for SAIs, policies, and a lightweight simulator that approximates the performance of a policy over time. Our approach is to see each transformation as an overhead performed in exchange for improved query performance. Hence, our model is based on two measured characteristics of the SAI: The costs of accessing an instance, and the cost of applying a transform. A separate driver program measures (1) the cost of each access path on each instance atom type, varying every parameter available, and (2) the cost of each case of every transform.

Example 5. *As an illustrative example, we will use the Crack-or-Sort policy described above. This policy makes use of the **Array**, **Sorted**, and **BinTree** atoms, as well as the **Crack** and **Sort** transforms. For this policy we need to measure 5 factors.*

Operation	***Symbol***	***Scaling***
`Get`(**Array**($[r_1 \dots r_N]$))	$\alpha(N)$	$O(N)$
`Get`(**Sorted**($[r_1 \dots r_N]$)))	$\beta(N)$	$O(\log_2(N))$
`Get`(**BinTree**(k, C_1, C_2)))	γ	$O(1)$
Crack(***Array***($[r_1 \dots r_N]$))	$\delta(N)$	$O(N)$
Crack(***Array***($[r_1 \dots r_N]$))	$\nu(N)$	$O(n \log_2(n))$

The driver program fits each of the five functions by conducting multiple timing experiments, varying the size of N where applicable.

The simulator mirrors the behavior of the full SAI, but uses a lighter-weight version of the Cog grammar that does not store actual data:

$$\begin{aligned} C^\ell \quad = \quad & \textbf{Array}(\mathbb{N}_0) \ | \ \textbf{Sorted}(\mathbb{N}_0) \\ | \quad & \textbf{Concat}(C^\ell, C^\ell) \ | \ \textbf{BinTree}(C^\ell, C^\ell) \end{aligned}$$

The simulator iteratively simulates applying transforms to instances expressed in C^ℓ according to the policy being simulated. After each transform, the simulator uses the measured cost of the transform to estimate the cumulative time spent reorganizing the index. The simulator captures multiple performance metrics $\texttt{metric} : C \to \mathbb{R}$.

Example 6. *Continuing the example, one useful metric is the read latency for a uniformly distributed read workload on a Crack-or-Sort index.*

$$\texttt{latency}(C) = \begin{cases} \alpha(N) & \textbf{\textit{if}}\ C = \textbf{\textit{Array}}(N) \\ \beta(N) & \textbf{\textit{if}}\ C = \textbf{\textit{Array}}(N) \\ \gamma + \frac{|C_1|}{|C|}\texttt{latency}(C_1) & \\ \quad + \frac{|C_2|}{|C|}\texttt{latency}(C_2) & \textbf{\textit{if}}\ C = \textbf{\textit{BinTree}}(C_1, C_2) \end{cases}$$

*where $|C|$ is the sum of sizes of **Array**s and **Sorted**s in C^*.*

The simulator produces a sequence of status intervals: periods during which index performance is fixed, prior to the pointer swap after the next transform is computed. A user-provided utility function aggregates these intervals to provide a final utility score for the entire policy. Given a finite set of policies, the optimizer tries each in turn and selects the one that best optimizes the utility function. Given a parameterized policy, the optimizer instead uses gradient descent.

Example 7. *Examples of utility functions for Crack-or-Sort include: (1) Minimize time spent with more than θ `Get()` latency, (2) Maximize throughput for N seconds, (3) Minimize runtime of N queries.*

7 ON THE GENERALITY OF SAIS

Ideally, we would like Cog to be expressive enough to encode the instantaneous state of any data structure. Infinite generality is obviously out of scope for this paper. However we now take a moment to assess exactly what index data structure design patterns are supported in a SAI.

#	Data Calculator Primitive	SAI	Note
1	Key retention	◗	No partial keys
2	Value retention	❍	
3	Key order	◗	K-ary orders unsupported
4	Key-Value layout	❍	No columnar layouts
5	Intra-node access	●	
6	Utilization	✗	
7	Bloom filters	❍	
8	Zone map filters	◗	Implicit via **BinTree**
9	Filter memory layout	❍	Requires filters (7,8)
10	Fanout/Radix	❍	Limited to 2-way fanout
11	Key Partitioning	●	
12	Sub-block capacity	✗	
13	Immediate node links	❍	Simulated by iterator impl.
14	Skip-node links	❍	
15	Area links	❍	Simulated by iterator impl.
16	Sub-block physical location.	❍	Only pointed supported
17	Sub-block physical layout.	◗/ ✗	Realized by merge rewrite
18	Sub-block homogeneous	●	
19	Sub-block consolidation	●	Depends on policy
20	Sub-block instantiation	●	Depends on policy
21	Sub-block link layout	❍	Requires links (13,14,15)
22	Recursion allowed	●	

●: Full Support ◗: Partial Support ❍: Support Possible
✗: Not applicable to immutable data structures

Figure 6: SAI support for the DC Taxonomy [16]

As a point of reference we use a taxonomy of data structures proposed as part of the Data Calculator [16]. The data calculator taxonomy identifies 22 design primitives, each with a domain of between 2 and 7 possible values. Each of the roughly 10^{18} valid points in this 22-dimensional space describes one possible index structure. To the best of our knowledge, this represents the most comprehensive a survey of the space of possible index structures developed to date.

The data calculator taxonomy views index structures through the general abstraction of a tree with inner nodes and leaf nodes. This abstraction is sometimes used loosely: A hash table of size N, for example, is realized as as a tree with precisely one inner-node and N leaf nodes. Each of the taxonomy's design primitives captures one set of mutually exclusive characteristics of the nodes of this tree and how they are translated to a physical layout.

Figure 6 classifies each of the design primitives as (1) Fully supported by SAI if it generalizes the entire domain, (2) Partially supported by SAI if it supports more than one element of the domain, or (3) Not supported otherwise. We further subdivide this latter category in terms of whether support is feasible or not. In general, the only design primitives that SAI can not generalize are related to mutability, since SAI's (semi-)immutability is crucial for concurrency, which is in turn required for optimization in the background.

SAI completely generalizes 7 of the remaining 22 primitives. We first explain these primitives and how SAIs generalize them. Then, we propose three extensions that, although beyond the scope of this paper, would fully generalize the final 14 primitives. For each, we briefly discuss the extension and summarize the challenges of realizing it.

Key retention (1). This primitive expresses whether inner nodes store keys (in whole or in part), mirroring the choice between **Concat** and **BinTree**.

Intra-node access (5). This primitive expresses whether nodes (inner or child) allow direct access to specific children or whether they require a full scan, mirroring the distinction between COG nodes with and without semantic constraints.

Key partitioning (9). This primitive expresses how newly added values are partitioned. Examples include by key range (as in a B+Tree) or temporally (as in a log structured merge tree [22]). Although a SAI only allows one form of insertion, policies can converge to the full range of states permitted for this primitive.

Sub-block homogeneous (18). This primitive expresses whether all inner nodes are homogeneous or not.

Sub-block consolidation/instantiation (19/20). These primitives express how and when organization happens, as would be determined by a SAI's policy.

Recursion allowed (22). This primitive expresses whether inner nodes form a bounded depth tree, a general tree, or a "tree" with a single node at the root. SAIs support all three.

7.1 Supporting New COG Atoms

Five of the remaining primitives can be generalized by the addition of three new atoms to COG. First, we would need a generalization of **BinTree** atoms capable of using partial keys as in a Trie (primitive 1), or hash values (primitive 3) Second, a unary **Filter** atom that imposes a constraint on the records below it could implement both boom filters (primitives 7,9) and zone maps (primitives 8,9). These two atoms are conceptually straightforward, but introduce new transforms and increase the complexity of the search for effective policies.

The remaining challenge is support for columnar/hybrid layouts (primitive 4). Columnar layouts increase the complexity of the formalism by requiring multiple record types and support for joining records. Accordingly, we posit that a binary **Join** atom, representing the collection of records obtained by joining its two children could efficiently capture the semantics of columnar (and hybrid) layouts.

7.2 Atom Synthesis

Five of the remaining primitives express various tactics for removing pointers by inlining groups of nodes into contiguous regions of memory. These primitives can be generalized by the addition of a form of atom synthesis, where new atoms are formed by merging existing atoms. Consider the Linked List of Example 1. Despite the syntactic restriction over COG, a single linked list element must consist of two nodes (a **Concat** and a (single-record) **Array**), and an unnecessary pointer de-reference is incurred on every lookup. Assume that we could define a new node type: A linked list element (**Link**($\mathcal{R}, C$)) consisting of a record and a forward pointer. Because this node type is defined in terms of existing node types, it would be possible to automatically synthesize new transformations for it from existing transformations, and existing performance models could likewise be adapted.

Atom synthesis could be used to create inner nodes that store values (primitive 2), increase the fanout of **Concat** and **BinTree** nodes (primitive 10), inline nodes (primitive 16), and provide finer-grained control over physical layout of data (primitive 17).

7.3 Links / DAG support

The final four remaining properties (13, 14, 15, and 20) express a variety of forms of link between inner and leaf nodes. Including such links turns the resulting structure into a directed acyclic graph (DAG). In principle, it should be possible to generalize transforms for arbitrary DAGs rather than just trees as we discuss in this paper. Such a generalization would require additional transforms that create/maintain the non-local links and more robust garbage collection.

8 EVALUATION

We next evaluate the performance of `SAI`s in comparison to other commonly used data structures. Our results show that: (1) In the longer term, `SAI`s have minimal overheads relative to standard in-memory data structures; (2) The `SAI` policy simulator reliably models the behavior of a `SAI`; (3) In the short term, `SAI`s can out-perform standard in-memory data structures; (4) Concurrency introduces minimal overheads; and (5) `SAI`s scale well with data, both in their access costs and their organizational costs.

8.1 Experimental setup

All experiments were run on a 2×6-core 2.5 GHz Intel Xeon server with 198 GB of RAM and running Ubuntu 16.04 LTS. Experimental code was written in C++ and compiled with GNU C++ 5.4.0. Each element in the data set is a pair of key and value, each an 8-Byte integer. Unless otherwise noted, we use a data size of up to a maximum of 10^9 records (16GB) with keys generated uniformly at random. To mitigate experimental noise, we use srand() with an arbitrary but consistent value for all data generation. To put our performance numbers into context, we compare against (1) **R/B Tree**: the C++ standard-template library (STL) `map` implementation (a classical red-black tree), (2) **HashTable** the C++ standard-template library (STL) `unordered-map` implementation (a hash table), and (3) **BTree** a publicly available implementation of b-trees[1]. For all three, we used the `find()` method for point lookups and `lower_bound()/++` (where available) for range-scans. For point lookups, we selected the target key uniformly at random[2]. For range scans, we selected a start value uniformly at random and the end value to visit approximately 1000 records. Except where noted, access times are the average of 1000 point lookups or 50 range scans.

We specifically evaluated `SAI`s using the Crack-Sort-Merge family of policies described in Section 5.5, varying the crack threshold over 10^6, 10^7, 10^8, and 10^9 records. When there are exactly 10^9 records, this last policy simply sorts the entire input in one step. For point lookups we use the `get()` access path, and for range scans we use the `ordered_iterator()` access path. By default, we measure `SAI` read performance through a synchronous (i.e., with the worker thread paused) microbenchmark. We contrast synchronous and asynchronous performance in Section 8.4.

Synchronous read performance was measured through a sequence of trials, each with a progressively larger number of transforms (i.e., a progressively larger fragment of the policy's trace) applied to the `SAI`. We measured total time to apply the trace fragment (including the cost of selecting which transforms to apply) before measuring access latencies. For concurrent read performance a client thread measured access latency approximately once per second.

8.2 Cost vs Benefit Over Time

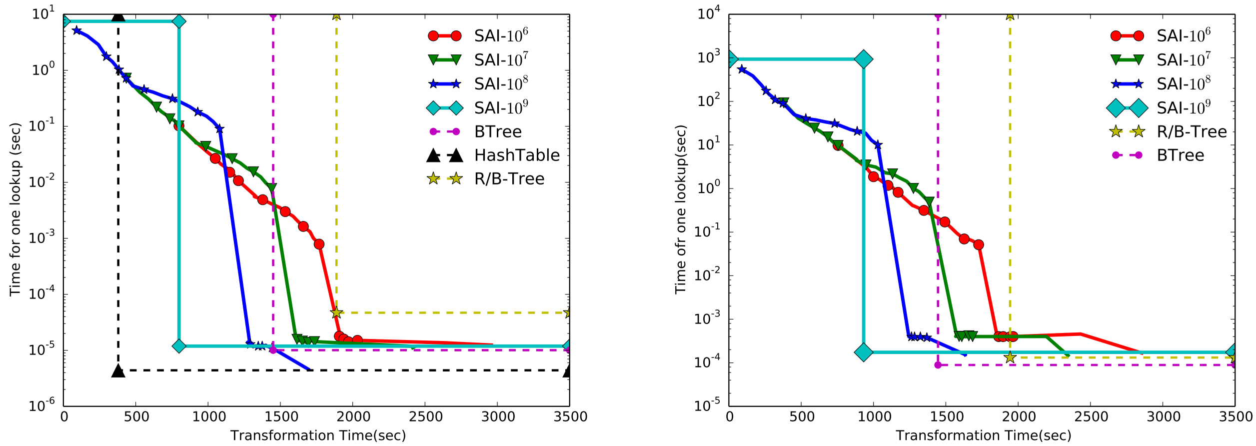
Our first set of experiments mirrors Figure 1, tracking the synchronous performance of point lookups and range scans over time. The results are shown in Figure 7a and Figure 7b The x-axis shows time elapsed, while the y-axis shows index access latency at that point in time. In both sets of experiments, we include access latencies and setup time for the R/B-Tree (yellow star), the HashTable (black triangle), and the BTree (pink circles) We treat the cost of accessing an incomplete data structure as infinite, stepping down to the structure's normal access costs once it is complete.

In general, lower crack thresholds achieve faster upfront performance by sacrificing long-term performance. A crack threshold of 10^6 (approximately $\frac{1}{10^5}$ cracked partitions) takes approximately twice as long to reach convergence as a threshold of 10^9 (sort everything upfront)

Unsurprisingly, for point lookups the Hash Table has the best overall performance curve. However, even it needs upwards of 6 minutes worth of data loading before it is ready.

[1]https://github.com/JGRennison/cpp-btree

[2]We also tested a heavy-hitter workload that queried for 30% of the keyspace 80% of the time, but found no significant differences between the workloads.



(a) Point Lookups **(b) Range Scans**

Figure 7: Performance improvement over time as each SAI is organized

By comparison, a SAI starts off with a 10 second response time, and has dropped to under 3 seconds by the 3 minute mark. The BTree significantly outperforms the R/B-Tree on both loading and point lookup cost, but still takes nearly 25 minutes to fully load. By that point the Threshold10^8 policy SAI has already been serving point lookups with a comparable latency (after its sort phase) for nearly 5 minutes. Note that lower crack thresholds have a slightly slower peak performance than higher ones before their merge phase This is a consequence of deeper tree structures and the indirection resulting from handles.The performance at convergence of the 10^8 threshold point scan trial is surprising, as it suggests binary search is as fast as a hash lookup. We suspect this due to lucky cache hits, but have not yet been able to confirm it.

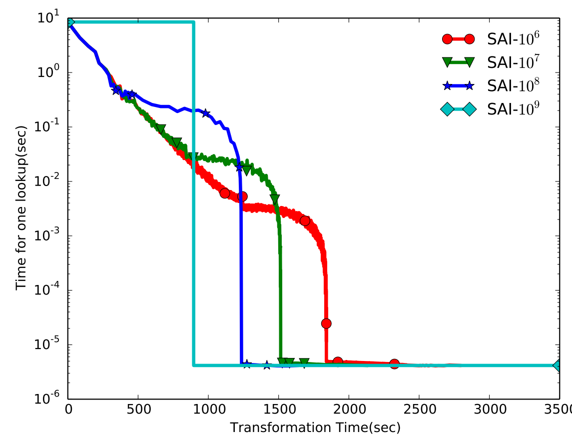


Figure 8: Predicted performance using the simulator.

8.3 Simulated vs Actual Performance

Figure 8 shows the result of using our simulator to predict the performance curves of Figure 7a. As can be seen, performance is comparable. Policy runtimes are replicated reliably, features like time to convergence and crossover are replicated virtually identically.

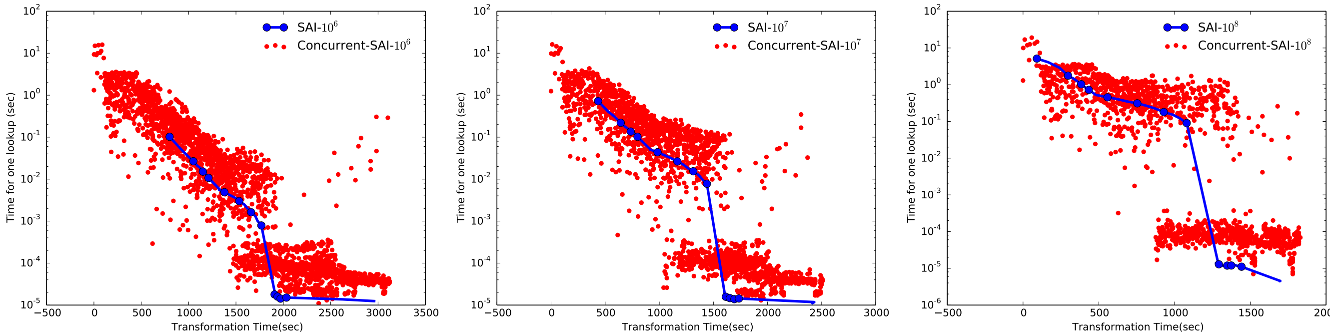
8.4 Synchronous vs Concurrent

Figures 9a, 9b, and 9c contrast the synchronous performance of the SAI with a more realistic concurrent workload. Performance during the crack phase is comparable, though admittedly with a higher variance. As expected, during the sort phase performance begins to bifurcate into fast-path accesses to already sorted arrays and slow-path scans over array nodes at the leaves.

The time it takes the worker to converge is largely unaffected by the introduction of concurrency. However, as the structure begins to converge, we see a constant $100\mu s$ overhead compared to synchronous access. We also note periodic $100ms$ bursts of latency during the sort phases of all trials. We believe these are caused when the worker thread pointer-swaps in a new array during the merge phase, as the entire newly created array is cold for the client thread.

8.5 Short-Term Benefits for interactive workloads

One of the primary benefits of SAIs is that they can provide significantly better performance during the transition period. This is particularly useful in interactive settings where users pose tasks comparatively slowly. We next consider such a hypothetical scenario where a data file is loaded and each data structure is given a short period of time (5 seconds) to prepare. In these experiments, we use a cracking threshold of 10^5 (our worst case), and vary the size of the data set from 10^6 records (16MB) to 10^9 records (16GB). The lookup time is the time until an answer is produced: the cost of a point lookup for the SAI. The baseline data structures are accessible only



(a) Crack Threshold = 10^6 **(b) Crack Threshold = 10^7** **(c) Crack Threshold = 10^8**

Figure 9: Synchronous vs Concurrent performance of the SAI on point lookups.

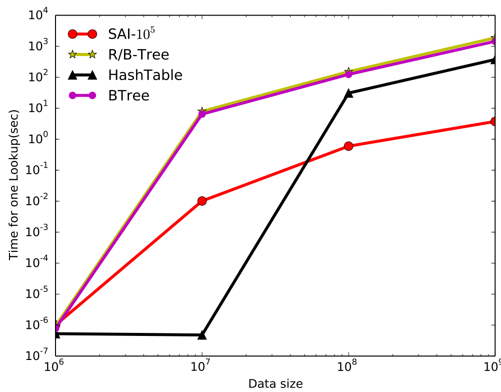


Figure 10: Point lookup latency relative to data size.

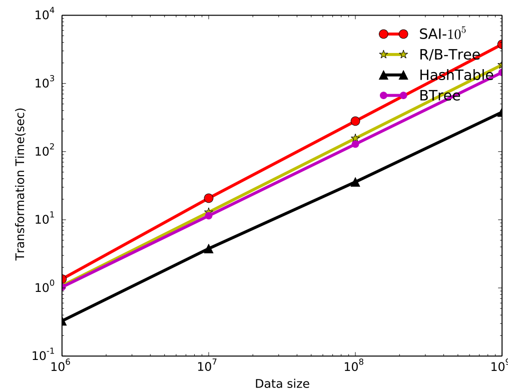


Figure 11: The time required to load and fully organize a data set relative to data size

once fully loaded, so we model the user waiting until the structure is ready before doing a point lookup. Up through 10^7 records, the `unordered_map` completes loading within 5 seconds. In every other case, the SAI is able to produce a response orders of magnitude faster.

8.6 DataSize Vs TransformTime

Figure 11 illustrates the scalability of SAI from the perspective of data loading. As before, we vary the size of the data set and use the time taken to load a comparable amount of data into the base data structures. Note that data is accessible virtually immediately after being loaded into a SAI. We measure the cost for the SAI to reach convergence. The performance of the SAI and the other data structures both scale linearly with the data size (note the log scale).

8.7 CrackThreshold Vs ScanTime

Figure 12 explores the effects of the crack threshold on performance at convergence of the Crack-Sort policy. The Merge Policy was excluded from testing as at convergence it would lead to one huge sorted array of size 10^9 irrespective of the crack threshold. In these set of experiments the crack threshold for cracking an array in the JITD structure was varied from 10^6 to 10^9. For each, we performed one thousand point scans, measuring the total time and computing the average cost per scan. This figure shows the overhead from handles — at a crack threshold of 10^9, the entire array is sorted in a single step. As the crack threshold grows by a factor of 10, the depth of the tree increases by roughly a factor of three, necessitating approximately 3 additional random accesses via handles rather than directly on a sorted array, and as shown in the graph, increasing access time by roughly 1 μs.

9 RELATED WORK

SAIs specifically extend work by Kennedy and Ziarek on Just-in-Time Data Structures [17] with a framework for defining policies, tools for optimizing across families of policies, and a runtime that supports optimization in the background rather than as part of queries. Most notably, this enables efficient dynamic data reorganization as an ongoing process rather than as an inline, blocking part of query execution.

Our goal is also spiritually similar to The Data Calculator [16]. Like our policy optimizer, it searches through a large space of index design choices for one suitable for a target workload. However, in contrast to SAIs, this search

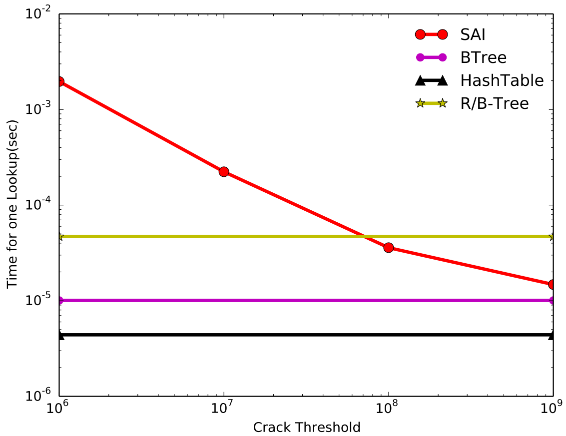


Figure 12: For the Uniform workload each point on the graph indicates the change in scan time for different crack thresholds.

happens once at compile time and explores mostly homogeneous structures. In principle, the two approaches could be combined, using the Data Calculator to identify optimal structures for each workload and using SAIs to migrate between structures as the workload changes.

Also related is a recently proposed form of "Resumable" Index Construction [2]. The primary challenge addressed by this work is ensuring that updates arriving after index construction begins are properly reflected in the index. While we solve this problem (semi-)functional data structures, the authors propose the use of temporary buffers.

Adaptive Indexing. SAIs are a form of adaptive indexing [8, 14], an approach to indexing that re-uses work done to answer queries to improve index organization. Examples of adaptive indexes include Cracker Indexes [12, 13], Adaptive Merge Trees [9], SMIX [28], and assorted hybrids thereof [15, 17]. Notably, a study by Schuhknecht et. al. [25] compares (among other things) the overheads of cracking to the costs of upfront indexing. Aiming to optimize overall runtime, upfront indexing begins to outperform cracker indexes after thousands to tens of thousands of queries. By optimizing the index in the background, SAIs avoid the overheads of data reorganization *as part of the query itself.*

Organization in the Background. Unlike adaptive indexes, which inline organizational effort into normal database operations, several index structures are designed with background performance optimization in mind. These begin with work in active databases [29], where reactions to database updates may be deferred until CPU cycles are available. More recently, bLSM trees [26] were proposed as a form of log-structured merge tree that coalesces partial indexes together in the background. A wide range of systems including COLT [24], OnlinePT [3], and Peloton [23] use workload modeling to dynamically select, create, and destroy indexes, also in the background.

Self-Tuning Databases. Database tuning advisors have existed for over two decades [4, 5], automatically selecting indexes to match specific workloads. However, with recent advances in machine learning technology, the area has seen significant recent activity, particularly in the context of index selection and design. OtterTune [27] uses fine-grained workload modeling to predict opportunities for setting database tuning parameters, an approach complimentary to our own.

Generic Data Structure Models. More spiritually similar to our work is The Data Calculator [16], which designs custom tree structures by searching through a space of dozens of parameters describing both tree and leaf nodes. A similarly related effort uses small neural networks [18] as a form of universal index structure by fitting a regression on the CDF of record keys in a sorted array.

10 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced SAIs a type of in-memory index that can incrementally morph its performance characteristics to adapt to changing workloads. To accomplish this, we formalized a composable organizational grammar (COG) and a simple algebra over it. We introduced a range of equivalence- and structure-preserving rewrite rules called transforms that serve as the basis of organizational policies that guide the transition from one performance envelope to another. We described a simulation framework that enables efficient optimization of policy parameters. Finally, we demonstrated that a SAI can be implemented with minimal overhead relative to classical in-memory index structures.

Our work leaves open several challenges. We have already identified three specific challenges in Section 7: New atoms, Atom synthesis, and DAG support. Addressing each of these challenges would allow COG to capture a wide range of data structure semantics. There are also several key areas where performance tuning is possible: First, our use of reference-counted pointers also presents a performance bottleneck for high-contention workloads — we plan to explore more active garbage-collection strategies. Second, Handles are an extremely conservative realization of semi-functional data structures. As a result, SAIs are a factor of 2 slower at convergence than other tree-based indexes. We expect that this performance gap can be reduced or eliminated by identifying situations where Handles are unnecessary (e.g., at convergence). A final open challenge is the use of statistics to guide rewrite rules, both detecting workload shifts to trigger policy shifts (e.g., as in Peloton), as well as identifying statistics-driven policies that naturally converge to optimal behaviors for dynamic workloads.

REFERENCES

[1] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. In *SIGMOD Conference*, pages 241–252. ACM, 2012.

[2] P. Antonopoulos, H. Kodavalla, A. Tran, N. preti, C. Shah, and M. Sztajno. Resumable online index rebuild in SQL server. *PVLDB*, 10(12):1742–1753, 2017.

[3] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835. IEEE Computer Society, 2007.

[4] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD*, 1998.

[5] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, pages 3–14, 2007.

[6] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[7] G. Graefe. B-tree indexes for high update rates. In *Data Always and Everywhere - Management of Mobile, Ubiquitous, Pervasive, and Sensor Data*, volume 05421 of *Dagstuhl Seminar Proceedings*, 2005.

[8] G. Graefe, S. Idreos, H. A. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *TPCTC*, volume 6417 of *Lecture Notes in Computer Science*, pages 169–184. Springer, 2010.

[9] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, volume 426 of *ACM International Conference Proceeding Series*, pages 371–381. ACM, 2010.

[10] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21. IEEE Computer Society, 1978.

[11] A. Hertzfeld. Hungarian. http://www.folklore.org/StoryView.py?project=Macintosh&story=Hungarian.txt.

[12] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[13] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD Conference*, pages 413–424. ACM, 2007.

[14] S. Idreos, S. Manegold, and G. Graefe. Adaptive indexing in modern database kernels. In *EDBT*, pages 566–569. ACM, 2012.

[15] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.

[16] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *SIGMOD Conference*, pages 535–550. ACM, 2018.

[17] O. Kennedy and L. Ziarek. Just-in-time data structures. In *CIDR*, 2015.

[18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD Conference*, pages 489–504. ACM, 2018.

[19] P. Larson. Dynamic hash tables. *Commun. ACM*, 31(4):446–457, 1988.

[20] D. Lee, S. Baek, and H. Bae. acn-rb-tree: Update method for spatio-temporal aggregation of moving object trajectory in ubiquitous environment. In *ICCSA Workshops*, pages 177–182. IEEE Computer Society, 2009.

[21] C. Okasaki. *Purely Functional data structures*. Cambridge University Press, 1999.

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

[23] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR*, 2017.

[24] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: continuous on-line tuning. In *SIGMOD Conference*, pages 793–795. ACM, 2006.

[25] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *PVLDB*, 7(2):97–108, 2013.

[26] R. Sears and R. Ramakrishnan. blsm: a general purpose log structured merge tree. In *SIGMOD Conference*, pages 217–228. ACM, 2012.

[27] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machinelearning. In *SIGMOD Conference*, pages 1009–1024. ACM, 2017.

[28] H. Voigt, T. Kissinger, and W. Lehner. SMIX: self-managing indexes for dynamic workloads. In *SSDBM*, pages 24:1–24:12. ACM, 2013.

[29] J. Widom and S. Ceri. Introduction to active database systems. In *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pages 1–41. Morgan Kaufmann, 1996.

[30] Z. Zhou, J. Tang, L. Zhang, K. Ning, and Q. Wang. Egf-tree: an energy-efficient index tree for facilitating multi-region query aggregation in the internet of things. *Personal and Ubiquitous Computing*, 18(4):951–966, 2014.

A CORRECTNESS OF EXAMPLE TRANSFORMS

As a warm-up and an example of transform correctness, we next review each of the transforms given in Figure 4 and prove the correctness of each.

Proposition 2 (Identity is correct). *Let* $\textbf{id}$ *denote the identity transform* $\textbf{id}(C) = C$. $\textbf{id}$ *is both equivalence preserving and structure preserving.*

Lemma 2 (**Sort** is correct). ***Sort*** *is both equivalence preserving and structure preserving.*

Proof. For any instance C where $\textbf{typeof}(C) \neq \textbf{Array}$, correctness follows from Proposition 2.

Otherwise $C = \textbf{Array}([r_1, \ldots, r_N])$, and consequently $\textbf{Sort}(C) = \textbf{Sorted}(\texttt{sort}([r_1, \ldots, r_N]))$. To show correctness we first need to prove that

$$\mathbb{D}\left(\textbf{Array}([r_1, \ldots, r_N])\right) = \mathbb{D}\left(\textbf{Sorted}(\texttt{sort}([r_1, \ldots, r_N]))\right)$$

Let the one-to-one (hence invertable) function $f : [1, N] \rightarrow [1, N]$ denote the transposition applied by `sort`.

$$\begin{aligned}
\mathbb{D}\left(\textbf{Sorted}(\texttt{sort}([r_1, \ldots, r_N]))\right) &= \mathbb{D}\left(\textbf{Sorted}(\left[r_{f^{-1}(1)}, \ldots, r_{f^{-1}(N)}\right])\right) \\
&= \left\{\!\!\left| r_{f^{-1}(1)}, \ldots, r_{f^{-1}(N)} \right|\!\!\right\} \\
&= \{\!| r_1, \ldots, r_N |\!\} \\
&= \mathbb{D}\left(\textbf{Array}([r_1, \ldots, r_N])\right)
\end{aligned}$$

giving us equivalence preservation. Structure preservation requires that $\left[r_{f^{-1}(1)}, \ldots, r_{f^{-1}(N)}\right]$ be in sorted order, which it is by construction. Thus, **Sort** is a correct transform. □

Lemma 3 (**UnSort** is correct). ***UnSort*** *is both equivalence preserving and structure preserving.*

Proof. For any instance C where $\textbf{typeof}(C) \neq \textbf{Sorted}$, correctness follows from Proposition 2.

Otherwise $C = \textbf{Sorted}([r_1 \ldots r_N])$ and we need to show first that $\mathbb{D}\left(\textbf{Sorted}([r_1 \ldots r_N])\right) = \mathbb{D}\left(\textbf{Array}([r_1 \ldots r_N])\right)$. The logical contents of both are $\{\!| r_1 \ldots r_N |\!\}$, so we have equivalence. Structure preservation is a given since any **Array** instance is structurally correct. ◻

Lemma 4 (**Divide** is correct). ***Divide*** *is both equivalence preserving and structure preserving.*

Proof. For any instance C where $\textbf{typeof}(C) \neq \textbf{Array}$, correctness follows from Proposition 2.

Otherwise $C = \textbf{Array}([r_1 \ldots r_N])$ and we need to show first that

$$\mathbb{D}(\textbf{Array}([r_1 \ldots r_N])) =$$
$$\mathbb{D}\left(\textbf{Concat}\left(\textbf{Array}(\left[r_1 \ldots r_{\lfloor \frac{N}{2} \rfloor}\right]), \textbf{Array}(\left[r_{\lfloor \frac{N}{2} \rfloor + 1} \ldots r_N\right])\right)\right)$$

Evaluating the right hand side of the equation recursively and simplifying, we have

$$\begin{aligned} &= \left\{\!\!\left| r_1 \ldots r_{\lfloor \frac{N}{2} \rfloor} \right|\!\!\right\} \uplus \left\{\!\!\left| r_{\lfloor \frac{N}{2} \rfloor + 1} \ldots r_N \right|\!\!\right\} \\ &= \left\{\!\!\left| r_1 \ldots r_{\lfloor \frac{N}{2} \rfloor}, r_{\lfloor \frac{N}{2} \rfloor + 1} \ldots r_N \right|\!\!\right\} \\ &= \{\!| r_1 \ldots r_N |\!\} \quad = \quad \mathbb{D}(\textbf{Array}([r_1 \ldots r_N])) \end{aligned}$$

Hence we have equivalence preservation. The **Array** instances are always structurally correct and **Concat** instances are structurally correct if their children are, so we have structural preservation as well. Hence, **Divide** is correct. ◻

Lemma 5 (**Crack** is correct). ***Crack*** *is both equivalence preserving and structure preserving.*

Proof. For any instance C where $\textbf{typeof}(C) \neq \textbf{Array}$, correctness follows from Proposition 2.

Otherwise $C = \textbf{Array}([r_1 \ldots r_N])$ and we need to show first that

$$\mathbb{D}(\textbf{Array}([r_1 \ldots r_N])) =$$
$$\mathbb{D}\left(\textbf{BinTree}\left(k, \textbf{Array}([\ r_i \mid r_i \prec k\ ]), \textbf{Array}([\ r_i \mid k \preceq r_i\ ])\right)\right)$$

Here $k = \texttt{id}(r_i)$ for an arbitrary i. Evaluating the right hand side of the equation recursively and simplifying, we have

$$\begin{aligned} &= \{\!\!|\ r_i \mid r_i \prec k\ |\!\!\} \uplus \{\!\!|\ r_i \mid k \preceq r_i\ |\!\!\} \\ &= \{\!\!|\ r_i \mid (r_i \prec k) \vee (k \preceq r_i)\ |\!\!\} \\ &= \{\!| r_1 \ldots r_N |\!\} \quad = \quad \mathbb{D}(\textbf{Array}([r_1 \ldots r_N])) \end{aligned}$$

Instances of **Array** are always structurally correct. The newly created **BinTree** instance is structurally correct by construction. Thus **Crack** is correct. ◻

Lemma 6 (**Merge** is correct). ***Merge*** *is both equivalence preserving and structure preserving.*

Proof. For any instance C that matches neither of **Merge**'s cases, correctness follows from Proposition 2. Of the remaining two cases, we first consider

$$C = \textbf{Concat}(\textbf{Array}([r_1 \ldots r_N]), \textbf{Array}([r_{N+1} \ldots r_M]))$$

The proof of equivalence preservation is identical to that of Theorem 4 applied in reverse. In the second case

$$C = \textbf{BinTree}(\ \_\ , \textbf{Array}([r_1 \ldots r_N]), \textbf{Array}([r_{N+1} \ldots r_M]))$$

Noting that $\textbf{BinTree}(\ \_\ , C_1, C_2) \approx \textbf{Concat}(C_1, C_2)$ by the definition of logical contents, the proof of equivalence preservation is again identical to that of Theorem 4 applied in reverse. For both cases, structural preservation is given by the fact that **Array** is always structurally correct. Thus **Merge** is correct. ◻

Lemma 7 (**PivotLeft** is correct). ***PivotLeft*** *is both equivalence preserving and structure preserving.*

Proof. For any instance C that matches neither of **PivotLeft**'s cases, correctness follows from Proposition 2. Of the remaining two cases, we first consider

$$C = \textbf{Concat}(C_1, \textbf{Concat}(C_2, C_3))$$

Equivalence follows from from associativity of bag union.

$$\begin{aligned} \mathbb{D}(\textbf{Concat}(C_1, \textbf{Concat}(C_2, C_3))) &= \mathbb{D}(C_1) \uplus \mathbb{D}(C_2) \uplus \mathbb{D}(C_3) \\ &= \mathbb{D}(\textbf{Concat}(\textbf{Concat}(C_1, C_2), C_3)) \end{aligned}$$

Concat instances are structurally correct if their children are, so the transformed instance is structurally correct if $\alpha(C_1)$, $\alpha(C_2)$, and $\alpha(C_3)$. Hence, if the input is structurally correct, then so is the output and the transform is structurally preserving in this case. The proof of equivalence preservation is identical for the case where

$$C = \textbf{BinTree}(k_1, C_1, \textbf{BinTree}(k_2, C_2, C_3)) \textbf{ and } k_1 \prec k_2$$

For structural preservation, we additionally need to show: **(1)** $\forall r \in \mathbb{D}(C_1) : r \prec k_1$, **(2)** $\forall r \in \mathbb{D}(C_2) : k_1 \preceq r$, **(3)** $\forall r \in \mathbb{D}(\textbf{BinTree}(k_1, C_1, C_2)) : r \prec k_2$, and **(4)** $\forall r \in \mathbb{D}(C_3) : k_2 \preceq r$ given that C is structurally correct.

Properties (1) and (4) follow trivially from the structural correctness of C. Property (2) follows from structural correctness of C requiring that $\forall r \in (\mathbb{D}(C_2) \uplus \mathbb{D}(C_3)) : k_1 \preceq r$ To show property (3), we first use transitivity to show that $\forall r \in \mathbb{D}(C_1) : r \prec k_1 \prec k_2$. For the remaining records, $\forall r \in \mathbb{D}(C_2) : r \prec k_2$ follows trivially from the structural correctness of C. Thus **PivotLeft** is correct [3] ◻

Corrolary 2. ***PivotRight*** *is correct.*

B LHS / RHS ARE META TRANSFORMS

Proof. We show only the proof for **LHS**; The proof for **RHS** is symmetric. We first show that **LHS** is an endofunctor. The kind of **LHS** is appropriate, so we only need to show that it satisfies the properties of a functor. First, we show that **LHS** commutes the identity (**id**). In other words, for any instance

[3]Note the limit on $k_1 \prec k_2$, which could be violated with an empty C_2.

C, $\mathbf{LHS}[\mathbf{id}](C) = C$. In the case where $C = \mathbf{Concat}(C_1, C_2)$, then

$$\mathbf{LHS}[\mathbf{id}](C) = \mathbf{Concat}(\mathbf{id}(C_1), C_2) = \mathbf{Concat}(C_1, C_2)$$

The case where $\mathbf{typeof}(C) = \mathbf{BinTree}$ is identical, and $\mathbf{LHS}[T]$ is already the identity in all other cases. Next, we need to show that **LHS** distributes over composition. That is, for any instance C and transforms T_1 and T_2 we need that

$$\mathbf{LHS}[T_1 \circ T_2](C) = (\mathbf{LHS}[T_1] \circ \mathbf{LHS}[T_2])(C)$$

If $C = \mathbf{Concat}(C_1, C_2)$, $\mathbf{LHS}[T_1 \circ T_2](C) = \mathbf{Concat}(C_1', C_2)$, where $C_1' = T_2(T_1(C_1))$. For the other side of the equation:

$$\begin{aligned}(\mathbf{LHS}[T_1] \circ \mathbf{LHS}[T_2])(C) &= \mathbf{LHS}[T_2](\mathbf{LHS}[T_1](C)) \\ &= \mathbf{LHS}[T_2](\mathbf{Concat}(T_1(C_1), C_2) \\ &= \mathbf{Concat}(T_2(T_1(C_1)), C_2)\end{aligned}$$

The case where $\mathbf{typeof}(C) = \mathbf{BinTree}$ is similar, and the remaining cases follow from $\mathbf{LHS}[T] = \mathbf{id}$ for all other cases. Thus **LHS** is an functor. For **LHS** to be a meta transform, it remains to show that for any correct transform T, $\mathbf{LHS}[T]$ is also correct. We first consider the case where $C = \mathbf{Concat}(C_1, C_2)$ and assume that $T(C_1)$ is both equivalence and structure preserving, or equivalently that $\mathbb{D}(C_1) = \mathbb{D}(T(C_1))$ and $\textsc{StrCor}(C_1) \implies \textsc{StrCor}(T(C_1))$.

$$\begin{aligned}\mathbb{D}(\mathbf{LHS}[T](C)) &= \mathbb{D}(\mathbf{Concat}(T(C_1), C_2)) \\ &= \mathbb{D}(\mathbf{Concat}(C_1, C_2)) = \mathbb{D}(C)\end{aligned}$$

Thus, $\mathbf{LHS}[T]$ is equivalence preserving for this case. The proof of structure preservation follows a similar pattern

$$\begin{aligned}\textsc{StrCor}(\mathbf{LHS}[T](C)) &= \textsc{StrCor}(\mathbf{Concat}(T(C_1), C_2)) \\ &= \textsc{StrCor}(T(C_1)) \wedge \textsc{StrCor}(C_2)\end{aligned}$$

Given $\textsc{StrCor}(C) = \textsc{StrCor}(C_1) \wedge \textsc{StrCor}(C_2)$ and the assumption of $\textsc{StrCor}(C_1) \implies \textsc{StrCor}(T(C_1))$, it follows that $\mathbf{LHS}[T]$ is structure preserving for this C. The proof for the case where $C = \mathbf{BinTree}(k, C_1, C_2)$ is similar, but also requires showing that $\forall r \in \mathbb{D}(T(C_1)) : r \prec k$ under the assumption that $\forall r \in \mathbb{D}(C_1) : r \prec k$. This follows from our assumption that $\mathbb{D}(T(C_1)) = \mathbb{D}(C_1)$. The remaining cases of **LHS** are covered under Proposition 2. Thus, **LHS** is a meta transform. ◻

C TARGET UPDATES ARE BOUNDED

Proof. By recursion over T. The atomic transforms are the base case. By definition **id** is not in the active domain, so we only need to consider seven possible atomic transforms. For **Sort** or **UnSort** to be in the active domain, $\mathbf{typeof}(C)$ must be **Array** or **Sorted** respectively. By the definition of each transform, $\mathbf{typeof}(C')$ will be **Sorted** or **Array** respectively By Theorem 2, the active domain of any **Array** or **Sorted** instance is bounded by $|\mathcal{A}|$ and by construction, $|\mathcal{W}_C| = |\mathcal{D}_C| \leq |\mathcal{A}|$. Hence, the total change in the weighted targets for this case is at most $2 \times |\mathcal{A}|$. Following a similar line of reasoning, the weighted targets change by at most $4 \times |\mathcal{A}|$ elements as a result of any **Divide**, **Crack**, or **Merge**. Next consider $C = \mathbf{Concat}(\mathbf{Concat}(C_1, C_2), C_3)$, and consequently $C' = \mathbf{PivotLeft}(C) = \mathbf{Concat}(C_1, \mathbf{Concat}(C_2, C_3))$. For each transform of the form $\mathbf{LHS}[\mathbf{LHS}[T]]$ in the active domain of C, there will be a corresponding $\mathbf{LHS}[T]$, as C_1 is identical in both paths. Similar reasoning holds for C_2 and C_3. Because the policy is local, the weighted targets are independent of any **LHS** or **RHS** meta transforms modifying them. Thus, at most, the active domain will lose T and $\mathbf{LHS}[T]$ for $T \in \mathcal{A}$, and gain T and $\mathbf{RHS}[T]$ for $T \in \mathcal{A}$, and the weighted targets will change by no more than $4 \times |\mathcal{A}|$ elements. Similar lines of reasoning hold for the other case of **PivotLeft** and for both cases of **PivotRight**. The recursive cases are trivial, since the weighted targets are independent of prefixes in a local policy. ◻