Fluid Data Structures

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

Functional (aka immutable) data structures are used extensively in data management systems. From distributed systems to data persistence, immutability makes complex programs significantly easier to reason about and implement. However, immutability also makes many runtime optimizations like tree rebalancing, or adaptive organizations, unreasonably expensive. In this paper, we propose Fluid data structures, an approach to data structure design that allows limited physical changes that preserve logical equivalence. As we will show, this approach retains many of the desirable properties of functional data structures, while also allowing runtime adaptation. To illustrate Fluid data structures, we work through the design of a lazy-loading map that we call a Fluid COG. A Fluid COG is a lock-free data structure that incrementally organizes itself in the background by applying equivalence-preserving structural transformations. Our experimental analysis shows that the resulting map structure is flexible enough to adapt to a variety of performance goals, while remaining competitive with existing structures like the C++ standard template library map.

ACM Reference Format:
Darshana Balakrishnan, Lukasz Ziarek, and Oliver Kennedy. 2019. Fluid Data Structures. In *Proceedings of DBPL (DBPL'19).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Functional (immutable) data structures [15] have several very nice properties that make them easy to use and reason about. Because their content does not change, compared to mutable data structures, there are fewer cache consistency issues to deal with, components are re-usable across structure instances, and compiler optimizations are easier to reason about. Unfortunately, functional data structures are also harder to employ in practice than mutable structures. Functional data structures are often carefully hand-tuned to avoid unnecessary data copies. Among other things, this encourages data structure designers to front-load organizational costs into write operations, since even minor organizational modifications can require rewriting significant portions of the data structure.

In *lazy* data structures, blocks of code can be used as placeholders for incomplete fragments of the structure. When a lazy data structure is traversed, any placeholder code encountered must be evaluated, or materialized, before the traversal can continue. The data structure is otherwise immutable. Lazy evaluation allows organizational costs to be time-shifted forward, away from writes, but also retain most of the benefits of classical immutable data structures.

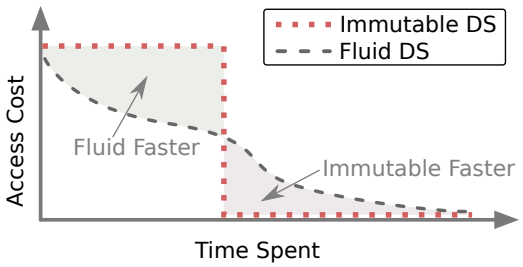


Figure 1. Immutable data structures block during updates. Fluid data structures allow for continuous, incremental performance improvements after updates.

In this paper, we argue that lazy data structures do not go far enough — specifically in the context of database indexing. First, an un-materialized node is completely useless: the data contained within is inaccessible until the corresponding code finishes executing. Second, the choice of how to materialize is selected when the node is created, and can not be adapted to changing workload parameters. Finally, once materialized, a node is completely immutable, making runtime adaptation in response to changing workloads (e.g., re-balancing or adaptive indexing) extremely expensive.

We propose a new family of data structures called *fluid data structures* that permit a limited form of mutation where the *logical* consistency of the structure is preserved. In an immutable data structure, a pointer is guaranteed to reference exactly the same byte string forever. Fluid data structures also allow an additional class of object reference called a *handle*, which only guarantees the logical stability of the referenced object; The handle may be updated at any time, as long as the newly referenced object is logically equivalent to the original. As illustrated in Figure 1, this flexibility allows for small and continuous performance improvements, along with adaptation to changing workloads, and no blocking of accessor threads.

We introduce a concrete fluid data structure, called Fluid COG. Fluid COG is based on our past work on Just-in-Time Index Adaptation [12]. Just-in-Time indexes allow graceful transitions between different physical data layouts through

DBPL'19, June 2019, Phoenix, AZ, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

small, incremental units of work. At each step along the transition, the index is in an intermediate, but still completely usable state. For example, an index transitioning from a linked list to a binary tree may be in a state that combines elements of both data structures, since reorganization is done in increments. Client threads are not blocked from reading data out, regardless of state.

To realize Fluid COG, we propose a mechanism for reasoning about fluid data structures by modeling them with a grammar. We specifically model Just-in-Time indexes with a Composable Organizational Grammar (COG), along with an algebra of transformation rules over sentences in COG. We then show that these transforms can be applied locally to alter components of the data structure without affecting the correctness or consistency of the structure as a whole. Finally, we implement COG in two forms, first in a Functional representation, and then as a more efficient *Fluid COG Data Structure* that is an immutable, thread-safe form of a Just-in-Time index.

The remainder of the paper is organized as follows: In Section 2 we define fluid data structures formally, including an abstract framework for reasoning about their correctness. Section 3 formally defines the composable organizational grammar (COG). In Section 4 we define transforms, syntactic rewrite rules over COG. Section 5 maps COG to both functional and fluid data structure implementations. Next, in Section 6 we address the practical challenges of implementing COG as a fluid data structure. Finally, Section 7 evaluates our COG-based fluid data structure against several comparable commodity data structures.

2 Fluid Data Structures

Before formalizing fluid data structures, we first introduce a key enabling concept: *handles*. References in a functional data structure preserve physical equivalence. All references to an element are guaranteed to point to exactly the same sequence of bytes forever. A fluid data structure, by comparison, distinguishes between two types of references that we call pointers and handles, respectively. Pointers behave like references in a normal functional data structure, preserving physical equivalence. Conversely, handles preserve only a form of *logical* equivalence. The element referenced by a handle may change, but the replacement must preserve some meaningful properties of the data structure as a whole. Thus, handles can be used in longer-lived components of the program (e.g., the data structure itself), while still allowing limited mutability for organization in the background. When physical stability is required (e.g., while the data structure is being accessed), the program can first dereference the handle into a (short-lived) pointer.

As illustrated in Figure 2, handles provide a layer of indirection that allows individual nodes of a data structure to be replaced with minimal impact. A fluid data structure uses

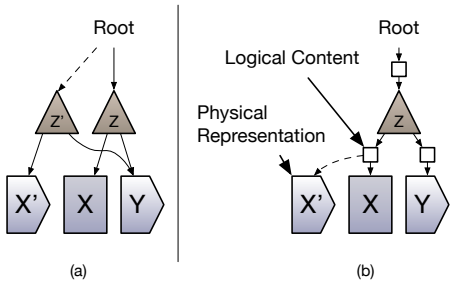


Figure 2. Classical immutable data structures (a) vs with handles (b).

handles to allow such replacements under the condition that the replacements not affect the correctness of the structure. As a result, handle updates do not need to trigger expensive cache invalidations, as any attempt to concurrently dereference the handle can safely use the older version. In other words, updates to handles are not subject to any ordering constraints, and only require a single atomic integer write.

Definition 1 (Fluid Data Structure)**.** *A fluid data structure of type $\mathcal{F}$ is defined by a 2-tuple : $\langle\ \tau,\ \triangleright\ \rangle$ where τ denotes the type of elements referenced by handles in the structure, and $\triangleright$: $\tau \times \tau \rightarrow \mathbb{B}$ is an intransitive relation over τ. Instances of a fluid data structure are denoted $f \in \mathcal{F}$. We write* $\texttt{REFS}(f)$ *to denote the set of elements (of type τ) referenced by handles in the instance f.*

Informally, the relation $\triangleright$ defines valid replacements for elements referenced by a handle. That is, if $e \in \texttt{REFS}(f)$ and there exists an e' such that $e \triangleright e'$, then e may be replaced by e'. We denote such a replacement (of e with e') by $f[e \backslash e']$. Next, we consider whether a replacement is also "safe."

Definition 2 (Validator)**.** *A validator for $\mathcal{F}$ is the 2-tuple $\mathcal{V} := \langle\ \texttt{correct}, \equiv\ \rangle$ of an indicator function* $\texttt{correct} : \mathcal{F} \rightarrow \mathbb{B}$ *and an equivalence relation $\equiv$: $\mathcal{F} \times \mathcal{F} \rightarrow \mathbb{B}$.*

A validator captures logical equivalence in a fluid data structure ($\equiv$), as well as any static correctness properties that the structure must enforce (`correct`). A fluid data structure is correct if handle replacements preserve both equivalence and correctness.

Definition 3 ($\mathcal{V}$-Correctness)**.** *Let $\mathcal{F}$ be a fluid data structure defined by $\langle\ \tau,\ \triangleright\ \rangle$ and let $\mathcal{V} = \langle\ \texttt{correct}, \equiv\ \rangle$ be a validator for $\mathcal{F}$. We say that $\mathcal{F}$ is $\mathcal{V}$-correct iff*

$$\forall f \in \mathcal{F}, e \in \texttt{REFS}(f), e' \in \tau :$$
$$(e \triangleright e') \wedge \texttt{correct}(f) \implies f \equiv f[e \backslash e'] \wedge \texttt{correct}(f[e \backslash e'])$$

Intuitively, a correct fluid data structure is one where the structure's local consistency properties (i.e., $\triangleright$) guarantee logical consistency over the entire structure as handles are updated. In the balance of the paper, we will illustrate one

technique ideally suited for such reasoning: modeling data structure instances by a grammar.

3 Data Structures As A Grammar

We adapt our prior work on just-in-time data structure compilation [12] into a compositional organizational grammar (COG). COG models in-memory map-style data structures (also called dictionaries or key-value stores). Each sentence in COG corresponds to an instance of a map-like data structure. After defining COG itself, we will define a set of correctness-preserving transformation rules over COG in Section 4. Then in Section 5, we prove that COG and its transformation rules define a fluid data structure.

3.1 Notation and Definitions

Let $r \in \mathcal{R}$ denote a record identified by a (potentially non-unique) identifier $\texttt{id}(r) \in \mathcal{I}$. We assume a total order $\preceq$ is defined over elements of $\mathcal{I}$. We abuse syntax and use records and keys interchangeably with respect to the order, writing $r \preceq k$ to mean $\texttt{id}(r) \preceq k$. 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 $r_1, \ldots, r_N$.

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

$$\begin{aligned}\mathbb{C} \;&=\; \textbf{Array}([\mathcal{R}]) \;\mid\; \textbf{Sorted}([\mathcal{R}]) \\ &\mid\; \textbf{Concat}(\mathbb{C},\mathbb{C}) \;\mid\; \textbf{BinTree}(\mathcal{I},\mathbb{C},\mathbb{C})\end{aligned}$$

Terms in COG map directly to the physical building blocks of a data structure, while full sentences 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 pair of pointers referencing two other instances. We write **typeof**(C) to denote the atom symbol at the root of an instance $C \in \mathbb{C}$.

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

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

A linked list is either a concatenation of an array, and a pointer to the next element, or a terminal array[1]*. An example linked list in this grammar is illustrated in Figure 3a.*

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

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

*A binary tree is a hierarchy of **BinTree** inner nodes, with **Array** leaf nodes. An example of a classical binary tree in this restricted grammar is illustrated in Figure 3b, while Figure 3c shows a binary tree with multiple records per leaf.*

[1]Our examples are slightly overgeneralized for simplicity. While a textbook linked list has only one record per node, the example permits more.

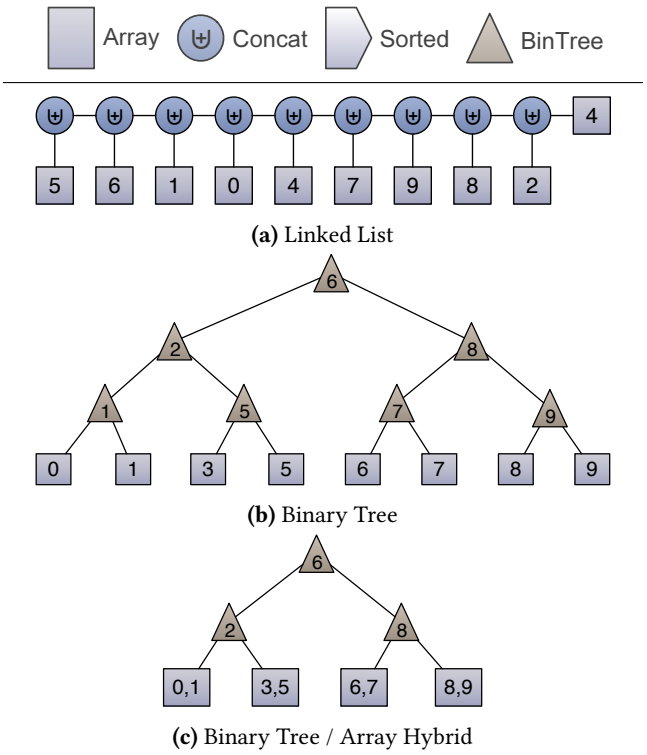


Figure 3. Example abstract syntax trees for three logically equivalent, correct sentences in COG.

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 4 (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 $C_1 \approx C_2$.*

Example 3. *The two sentences illustrated in Figure 3b and Figure 3c have the same logical contents and so are* logically equivalent. *By comparison, Figure 3a and Figure 3b are not logically equivalent; the former has extra copies of record 4.*

3.2 COG Semantics

Array and **Concat** represent the physical layout of elements of a data structure. The remaining two atoms provide semantic constraints (using the identifier order $\preceq$) over the

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

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

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

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

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

Figure 4. Examples of *correct* transforms. **Sort** and **UnSort** convert between **Array** and **Sorted** and visa versa. **Crack** and **Divide** both fragment **Array**s, 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 [10]. **PivotLeft** rotates tree structures counterclockwise and a symmetric **PivotRight** may also be defined. The function $\texttt{sort} : [\mathcal{R}] \rightarrow [\mathcal{R}]$ returns a transposition of its input sorted according to $\preceq$.

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

Case 1*. A term* ***Array*** *is always structurally correct.*

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

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

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

In short, **Sorted** is structurally correct if its records are in sorted order. 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.

4 Transforms over COG

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

Definition 6 (Transform)**.** *We define a transform T as any (endo)morphism $T : \mathbb{C} \rightarrow \mathbb{C}$ mapping between terms of COG. We denote by $\mathcal{T}$ the set of all transforms $T \in \mathcal{T}$.*

Figure 4 illustrates a range of 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 7 (Equivalence Preserving Transforms)**.** *A transform T is defined to be* equivalence preserving *if and only if $\forall C : C \approx T(C)$ (Definition 4).*

Definition 8 (Structure Preserving Transforms)**.** *A transform T is defined to be* structure preserving *if and only if $\forall C : \textsc{StrCor}(C) \implies \textsc{StrCor}(T(C))$ (Definition 5).*

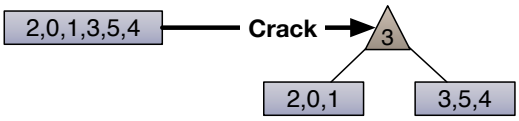
A transform is equivalence preserving if it preserves the logical content of the instance. It is structure preserving if it preserves the structure's semantic constraints (e.g., the record ordering constraint on instances of the **Sorted** atom). If it is both, we say that the transform is correct.

Definition 9 (Correct Transform)**.** *We define a transform T to be* correct *(denoted $\textsc{Correct}(T)$) if T is both structure and equivalence preserving.*

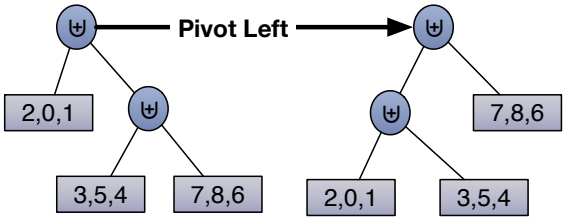
In Appendix A.1 we give proofs of correctness for each of the transforms in Figure 4.

Example 4. *Figures 5a, 5b, and 5c illustrate the* ***Sort****,* ***Crack****, and* ***PivotLeft*** *transforms respectively.* ***Sort*** *and* ***Crack*** *both*

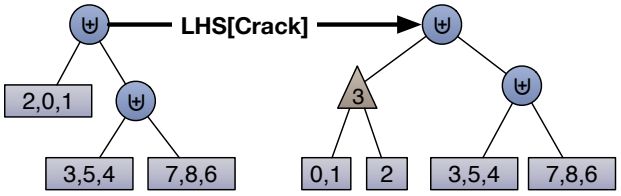
(a) Sort Transform



(b) Crack Transform



(c) PivotLeft Transform



(d) LHS Meta Transform applied to **Crack**

Figure 5. Example of transformations over COG

*operate on leaves. As a result, the input term is entirely replaced with a new, logically equivalent term. Observe that since **PivotLeft** operates on an inner node of the grammar, several subterms appear unchanged in the resulting tree.*

4.1 Meta Transforms

Transforms such as those illustrated in Figure 4 form the atomic building blocks of a policy for re-organizing data 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 10 (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 11 (Meta Transform)**.** *A meta transform M is any correctness-preserving functor $M : \mathcal{T} \to \mathcal{T}$. That is M is a meta transform if and only if $\forall T \in \mathcal{T} : \textsc{Correct}(T) \implies \textsc{Correct}(M[T])$ (Definition 9).*

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 functors over $\mathcal{T}$.*

The proof, given in Appendix A.2, 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.*

Example 5. *Continuing Example 4, Figure 5d illustrates a meta-transform: **LHS**[**Crack**]. The meta transform navigates to the left-hand-side of the root **Concat** term and simulates applying its parameter (**Crack**) there.*

5 COG-Based Data Structures

We next outline how to realize data structures based on COG. We start by defining a purely functional implementation of this data structure, and then illustrate how the resulting data structure can be re-implemented as a fluid data structure.

5.1 Functional COG

To implement COG as a functional data structure, we map the grammar directly to a union type. Each term expansion of the grammar maps to one node of the structure, as in the following ML-like data type definition:

```
type Node = Array   of Record list
          | Sorted  of Record list
          | Concat  of Node * Node
          | BinTree of Key * Node * Node
```

Similarly, transforms over COG map directly to equivalent functions with signature `Node -> Node`. For example, the **Sort** and **PivotLeft** transforms are illustrated in Algorithm 1 and Algorithm 2 respectively.

Algorithm 1 `Sort( n: Node ) -> Node`

```
match n with
  | Array(data) -> Sorted( sort_list(data) )
  | c -> c
```

Algorithm 2 `PivotLeft( n: Node ) -> Node`

```
match n with
  | Concat(A, Concat(B, C)) ->
              Concat(Concat(A, B), C)
  | BinTree(X, A, BinTree(Y, B, C)) ->
              BinTree(Y, BinTree(X, A, B), C)
  | c -> c
```

Meta-Transforms are implemented similarly as functions with signature `(Node->Node)->Node->Node`. The transform rebuilds the data structure from the ground up, for example as with **LHS** in Algorithm 3. Observe that each valid application of **LHS** (or **RHS**) performs one node allocation. Thus, any primitive transform requires a number of allocations linear in the depth at which it is applied.

Algorithm 3 `LHS( t: (Node->Node) )( n: Node ) -> Node`

```
match n with
  | Concat(A, B)      -> Concat( t(A), B )
  | BinTree(X, A, B) -> BinTree( X, t(A), B )
  | c -> c
```

Example 6. *Consider the following sentence in* COG*:*

$$\textbf{\textit{Concat}}(\quad \textbf{\textit{Concat}}(\ \textbf{\textit{Array}}([2,0,1]), \textbf{\textit{Array}}([5,2,3])\ ),\\ \textbf{\textit{Concat}}(\ \textbf{\textit{Array}}([3,5,4]), \textbf{\textit{Array}}([7,8,6])\ )\quad)$$

This sentence defines a data structure with of seven Node *elements. Applying the transform* ***RHS***[***LHS***[***Sort***]] *replaces the* Node *corresponding to* $\textbf{\textit{Array}}([3,5,4])$ *with a new* Node *corresponding to* $\textbf{\textit{Sorted}}([3,5,4])$*. Figure 6a illustrates this transform applied to a functional* COG *data structure. While four of the seven nodes can be re-used in the transformed structure, new* Node *instances must be allocated for the new node and all of its ancestors (red-dashed box).*

5.2 Fluid COG

Fluid data structures allow limited runtime modification of subtrees in the data structure. To implement COG as a fluid data structure, we track all Node instances in the structure as references.

```
type Node =
     | Array   of Record list
     | Sorted  of Record list
     | Concat  of Node ref * Node ref
     | BinTree of Key * Node ref * Node ref
```

Changes to the primitive transforms are largely cosmetic. References can be updated directly, so the function's signature changes to `Node->Unit`. Likewise, constructors are updated to use `ref`erences as needed. The key difference

Algorithm 4 `FluidLHS( t: (Node->Unit) )( n: Node )`

```
match n with
  | Concat(A, B)      -> t(!A)
  | BinTree(X, A, B) -> t(!A)
  | c -> c
```

appears in the Meta transforms, as in **LHS** as implemented in Algorithm 4.

The fluid version of **LHS** simply recurs, without needing to allocate a replacement node. However, as long as `t` is correct, the operation's side effects do not need to immediately become visible to other threads. We formalize this principle in the following theorem.

Theorem 2 (Fluid COG is a Fluid Data Structure)**.** *Define the preference relation* $\rhd_{\mathcal{A}}$ *as follows*

$$(C_1 \rhd_{\mathcal{A}} C_2) \stackrel{def}{=} \exists T \in \mathcal{A} : T(C_1) = C_2$$

Define the validator $\mathcal{V}_{\mathcal{A}}$ *by Structural Correctness (Definition 5) and Logical Equivalence (Definition 4) of* COG*.*

$$\mathcal{V}_{\mathcal{A}} := \langle\ \textit{StrCor}(\cdot), \approx\ \rangle$$

The fluid data structure defined by $\langle$ Node, $\rhd_{\mathcal{A}}$ $\rangle$ *is* $\mathcal{V}_{\mathcal{A}}$*-Correct.*

Proof. Every handle in a fluid COG instance can be reached by tree traversal. Furthermore, by definition, the valid replacements for the handle C are any $T(C)$ where $T \in \mathcal{A}$. Thus, every handle replacement corresponds to a transform in Δ. By Corrolary 1 (Section 4.1), all transforms in Δ preserve both structural correctness and logical equivalence. □

Example 7. *Continuing Example 6, Figure 6b illustrates the same transform applied to a Fluid* COG *for the same sentence. Meta transforms serve solely to navigate to a specific node of the tree and do not trigger new allocations. Only the actual leaf transformation needs to allocate new* Node*s.*

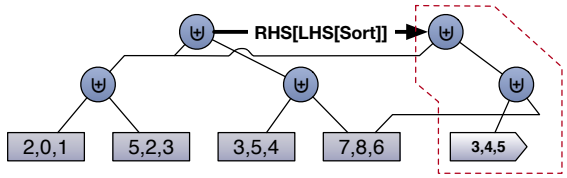
6 Fluid COG in Practice

We implemented a prototype of Fluid COG as a concurrent data structure in C++[2]. In addition to client threads accessing the data structure, a background thread acts as a *just-in-time optimizer*, incrementally improving the layout as time and resources permit.

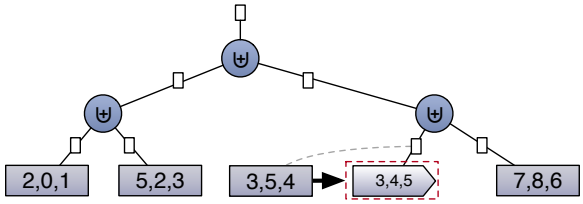
6.1 Access Paths

A Fluid COG 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 ordered iterator over records with keys greater than or equal to `lower`. As an example, Algorithm 5 implements the first of these access paths by recursively descending through the index. Semantic constraints on the layout

[2] https://github.com/UBOdin/jitd-cpp



(a) Functional COG



(b) Fluid COG

Figure 6. Applying the transform **RHS**[**LHS**[**Sort**]] to instances of both Functional COG and Fluid COG.

provided by **Sorted** and **BinTree** are exploited where they are available.

Algorithm 5 `Get( C : Node, k : Key ) -> Record option`

```
match C with
  | Array(data)  -> linearScan(data, k)
  | Sorted(data) -> binarySearch(data, k)
  | Concat(A, B) -> (
     match Get(!A) with
       | Some(result) -> result
       | None         -> Get(!B) )
  | BinTree(k', A, B) ->
     if k < k' then Get(!A) else Get(!B)
```

The iterator similarly exploits semantic constraints where available. Nodes are lazily dereferenced from Handles into Pointers as they are visited. The iterator supports concurrent updates to the structure, since the structure's logical content is immutable and updates create new versions.

6.2 Updates

Organizational effort in a Fluid COG 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:

$$\textbf{Concat}(\texttt{root}, C)$$

This subtree becomes a new version of the root. Observe that while this transformation does not preserve logical equivalence, it preserves structural correctness. Although only one thread may update the index at a time, updates do not block the background worker thread.

6.3 Background Organization

The Fluid COG relies on a just-in-time optimizer: a background thread that incrementally organizes the structure. In our prototype implementation, the optimizer considers two organizational strategies for newly loaded data: (1) database cracking [7] (i.e., **Crack**) and (2) directly sorting the data (i.e., **Sort**). 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 (creating two smaller arrays that now need to be sorted).

Our prototype Fluid COG allows users to trade off between these long- and short-term benefits via a threshold parameter. **Array** nodes larger than the threshold size are cracked, while smaller arrays are sorted. Larger arrays are transformed before smaller arrays. Once all nodes have either been sorted or cracked (i.e., the type of all nodes in the structure is either **Sorted** or **BinTree**), the organizer applies **Merge** to coalesce the entire tree into a single **Sorted**.

6.4 Concurrent Access

Unused nodes are garbage collected by reference counting using a C++ `shared_ptr`. Pointers to the nodes are themselves wrapped in a handle, also implemented as a `shared_ptr`. Access to the contents of a handle is performed by `atomic_load` and `atomic_store` to avoid split writes.

Content updates to the Fluid COG are protected by an atomic test and swap (`atomic_compare_exchange_strong`). The writer thread creates a new root node as described above, copying the old root's handle. The atomic test and swap replaces the old root handle[3] if and only if the root pointer was unchanged in the interim.

7 Evaluation

We next evaluate the performance of Fluid COG in comparison to other commonly used data structures. Our results show that: (1) In the longer term, Fluid COG has minimal overheads relative to standard in-memory data structures; (2) In the short term, Fluid COG can out-perform standard in-memory data structures; and (3) Concurrency introduces minimal overheads.

7.1 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 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

[3]Note that the handle itself is replaced, not the pointer referenced by the handle.

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[4]. 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[5]. For range scans, we selected a start value uniformly at random and the end value so as to visit approximately 1000 records. Except where noted, access times are the average of 1000 point lookups or 50 range scans.

For Fluid COG we used the **Crack**/**Sort** policy defined in section Section 6.3 and varied the threshold to either 10^6, 10^7, 10^8, or 10^9 records. When the threshold is 10^9 records (or more), the entire input is sorted in one step, modeling the behavior of a classical or immutable data structure. The initial data is encoded as a single Unsorted Array COG. For point lookups we use the `get()` access path, and for range scans we use the `iterator()` access path. By default, we measure Fluid COG read performance through a synchronous (i.e., with the worker thread paused) microbenchmark. We contrast synchronous and asynchronous performance in Section 7.3.

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 Fluid COG. 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.

7.2 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 10^3 cracked partitions) takes approximately twice as long to reach convergence as a threshold of 10^9 (sort everything upfront)

Unsurprisingly, the Hash Table has the best overall performance curve for point lookups. However, even it needs upwards of 6 minutes worth of data loading before it is ready. By comparison, a Fluid COG 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 Threshold 10^8 policy Fluid COG has already been serving point lookups with a comparable latency (after its sort phase) for nearly 5 minutes.

7.3 Synchronous vs Concurrent

Figures 8a, 8b, and 8c contrast the synchronous performance of Fluid COG 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 unsorted array nodes.

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.

7.4 Short-Term Benefits for interactive workloads

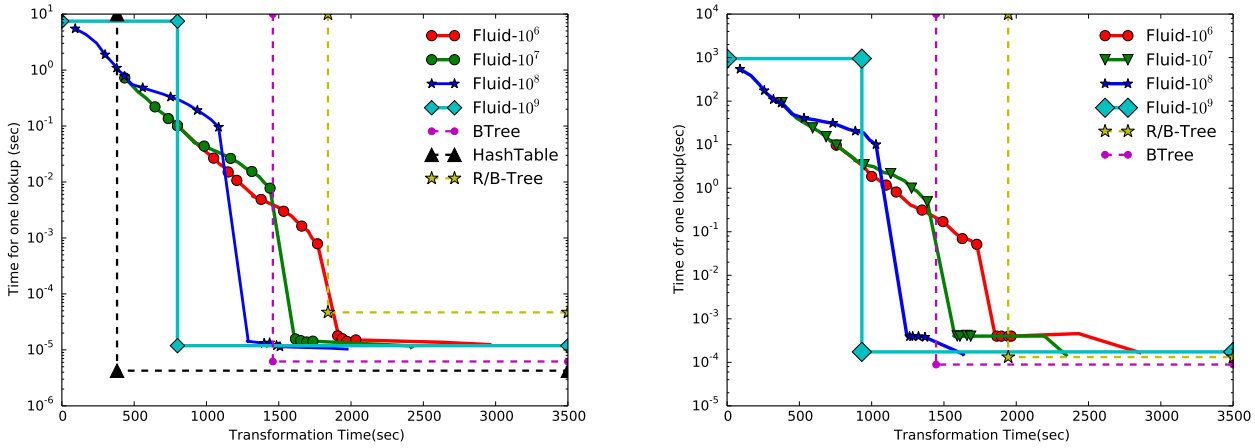
One of the primary benefits of Fluid COG data structures is that they can provide significantly better performance during the transition period. This is particularly useful in interactive settings where users pose tasks comparatively slowly. We next consider such a hypothetical scenario where a data 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 Fluid COG, or the time required to finish loading and query the structure. The baseline data structures are accessible only 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 Fluid COG is able to produce a response orders of magnitude faster.

8 Related Work

Fluid COG specifically extends our prior work on Just-in-Time Data Structures [12] with a framework for defining policies, tools for optimizing across families of policies, and a runtime

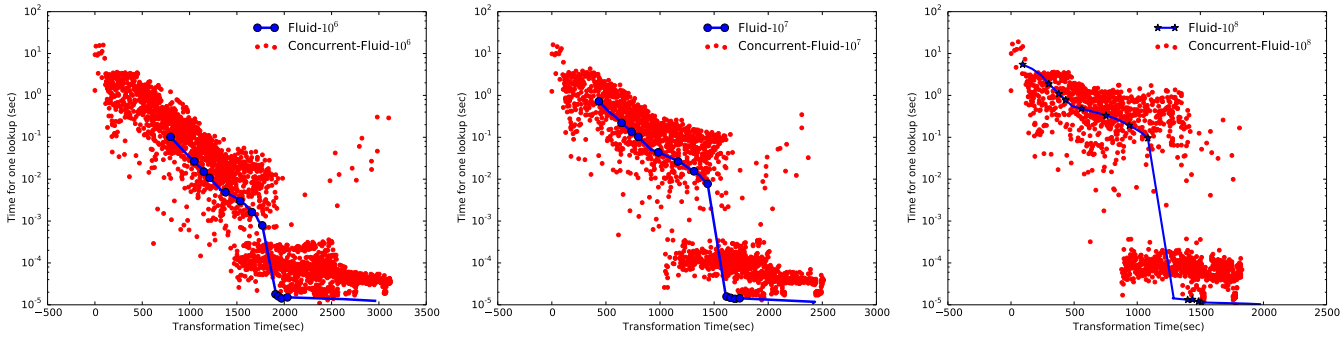
[4] https://github.com/JGRennison/cpp-btree

[5] 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 Fluid COG is organized



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

Figure 8. Synchronous vs Concurrent performance of the Fluid COG on point lookups.

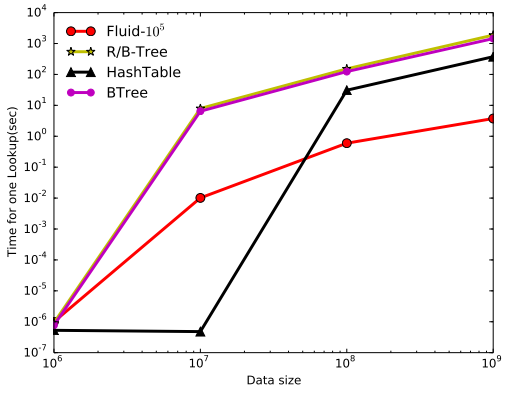


Figure 9. Point lookup latency relative to data size.

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 [11]. 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 Fluid COG, this search 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 Fluid COG to migrate between structures as the workload changes.

Also related is a recently proposed form of "Resumable" Index Construction [1]. 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. Fluid COG is a form of adaptive indexing [5, 9], an approach to indexing that re-uses work done to answer queries to improve index organization. Examples of adaptive indexes include Cracker Indexes [7, 8], Adaptive Merge Trees [6], SMIX [23], and assorted hybrids thereof [10, 12]. Notably, a study by Schuhknecht et. al. [19] 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, Fluid COG avoids introducing latency into the query itself as part 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 for background performance optimization. Active databases [24] can defer reactions to database updates until CPU cycles are available. More recently, bLSM trees [20] 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 [18], OnlinePT [2], and Peloton [17] 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 [3, 4], 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 [22] 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 [11], 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 [13] as a form of universal index structure by fitting a regression on the CDF of record keys in a sorted array. Work done in [14] is specialized for monotonically increasing data. Fluid COG are not limited to being monotonic in nature. But for substructures of Fluid COG that are generated by monotonic transforms like update we can extend Fluid COG to represent the substructures as a lattice.

Functional data structures in practice. Immutable data structures are used extensively in functional languages like Scala and Clojure. Considerable effort led to structures optimized for different data access patterns. For example, Scala's RRB Vectors [21] are a sequence structure that allows efficient concatenation, inserts and splits using "transient" nodes that lazily encode incomplete updates. Like Fluid COG, transient nodes allow for a limited form of mutability. However, while Fluid COG uses mutability to allow for dynamic adaptability, RRB Vectors use mutability to dynamically coalesce sequences of updates. Fluid COG could benefit from such optimizations to improve update efficiency.

9 Conclusions and Future Work

In this paper, we introduced *fluid data structures*. Fluid data structures rely on handles, a new form of object reference that does not guarantee physical immutability, but rather guarantees *logical* immutability. We illustrated fluid data structures in practice through a data structure called Fluid COG. The physical layout of any Fluid COG instance corresponds to a sentence in a *composable organizational grammar* (COG). We proposed an algebra of rewrite rules called transforms that correspond to small modifications to the physical layout of a Fluid COG instance. Specifically, we demonstrated a set of atomic transforms that are guaranteed to preserve the correctness and logical contents of the instance. Using a set of hierarchical meta-transforms, we also demonstrated that correctness and equivalence are preserved when an atomic transform is applied to *any* node of a Fluid COG instance. Finally, we showed how Fluid COG instances could be optimized in the background, allowing them to compete with standard off-the-shelf map data structures.

In Appendix B, we explore the generality of COG through a recently proposed index data structure taxonomy. We specifically identify three key areas where extensions to COG can make the grammar itself more general. First, more expressiveness can be achieved through new grammar terms that capture additional organizational semantics (prefix matching, hash partitioning) or that represent suspended computation (filter, join). Second, more efficient data layouts can be achieved by creating new data structure nodes by fusing existing terms together. For example, an inner node of a B+Tree can conceptually be expressed as a hierarchy of **BinTree** terms fused together. Finally, COG can only express Tree-shaped data structures. A similar conceptual model can be developed for DAG-shaped data structures. We also observe several additional areas where further 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, implementing handles as indirect references is an extremely conservative realization of fluid data structures. As a result, Fluid COG is a factor of 2 slower at convergence than mutable tree-based indexes. In some cases, it may be feasible to inline handles as direct, rather than indirect references. 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 [17]), as well as identifying statistics-driven policies that naturally converge to optimal behaviors for dynamic workloads.

10 Acknowledgements

This work is supported by NSF awards IIS-1617586 and IIS-1750460. The conclusions and opinions in this work are solely those of the authors and do not represent the views of the National Science Foundation.

References

[1] 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.

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

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

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

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] L. Kuper and R. R. Newton. Lvars:lattice-based data structures for deterministic parallelism. In *FHPC*, 2013.

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

[16] 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.

[17] 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.

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

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

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

[21] N. Stucki, T. Rompf, V. Ureche, and P. Bagwell. Rrb vector: a practical general purpose immutable sequence. In *ICFP*, 2015.

[22] 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.

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

[24] 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.

A Proofs

A.1 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 1 (Identity is correct)**.** *Let* ***id*** *denote the identity transform* $\textbf{\textit{id}}(C) = C$. ***id*** *is both equivalence preserving and structure preserving.*

Lemma 1 (**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 1.

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

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

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

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

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

Lemma 2 (**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 1.

Otherwise $C = \textbf{Sorted}([r_1 \ldots r_N])$ and we need to show first that $\mathbb{D}(\textbf{Sorted}([r_1 \ldots r_N])) = \mathbb{D}(\textbf{Array}([r_1 \ldots r_N]))$. 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 3 (**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 1.

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

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

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 4 (**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 1.

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

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

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

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

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

Lemma 5 (**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 1. 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 3 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 3 applied in reverse. For both cases, structural preservation is given by the fact that **Array** is always structurally correct. Thus **Merge** is correct. □

Lemma 6 (**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 1. 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.

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

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 [6] □

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

A.2 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 C, $\textbf{LHS}[\textbf{id}](C) = C$. In the case where $C = \textbf{Concat}(C_1, C_2)$, then

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

The case where $\textbf{typeof}(C) = \textbf{BinTree}$ is identical, and $\textbf{LHS}[T]$ 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

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

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

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

The case where $\textbf{typeof}(C) = \textbf{BinTree}$ is similar, and the remaining cases follow from $\textbf{LHS}[T] = \textbf{id}$ 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, $\textbf{LHS}[T]$ is also correct. We first consider the case where

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

$C = \textbf{Concat}(C_1, C_2)$ 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 $\textsc{StrCor}(C_1) \implies \textsc{StrCor}(T(C_1))$.

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

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

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

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 $\textbf{LHS}[T]$ is structure preserving for this C. The proof for the case where $C = \textbf{BinTree}(k, C_1, C_2)$ 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 1. Thus, **LHS** is a meta transform. □

A.3 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, **typeof**(C) must be **Array** or **Sorted** respectively. By the definition of each transform, **typeof**(C') will be **Sorted** or **Array** respectively By **??**, 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 $C = \textbf{Concat}(\textbf{Concat}(C_1, C_2), C_3)$, and consequently $C' = \textbf{PivotLeft}(C) = \textbf{Concat}(C_1, \textbf{Concat}(C_2, C_3))$. For each transform of the form $\textbf{LHS}[\textbf{LHS}[T]]$ in the active domain of C, there will be a corresponding $\textbf{LHS}[T]$, 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 $\textbf{LHS}[T]$ for $T \in \mathcal{A}$, and gain T and $\textbf{RHS}[T]$ 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. □

B On the Generality of COG

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 COG.

As a point of reference we use a taxonomy of data structures proposed as part of the Data Calculator [11]. 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 10 classifies each of the design primitives as (1) Fully supported by COG if it generalizes the entire domain, (2) Partially supported by COG 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 COG can not generalize are related to mutability, since COG targets functional and fluid data structures.

COG completely generalizes 7 of the remaining 22 primitives. We first explain these primitives and how COG 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 [16]). Although sentences in COG have no concept of updates, the right set of transforms can converge to a comparable partitioning result.

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. The specific choice of a policy for applying transformations to a Fluid COG is beyond the scope of this paper, but appropriate sequences are possible.

#	Data Calculator Primitive	COG	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 10. Support in COG for the DC Taxonomy [11]

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. All three are valid sentences in COG.

B.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.

B.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).

B.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.