Just-In-Time Data Structures

Oliver Kennedy, Lukasz Ziarek
The University at Buffalo
{okennedy, lziarek}@buffalo.edu

ABSTRACT

Adaptive indexing is a promising alternative to classical offline index optimization. Under adaptive indexing, index creation and re-organization take place automatically and incrementally as a side-effect of query execution. Adaptive indexing implementations optimize the index's structure by progressively rewriting it until it converges to a single idealized form such as a sorted array or B-Tree. However, the ideal representation changes over time: An adaptive index that is initially optimal for one workload becomes suboptimal as the workload's characteristics change.

In this paper we generalize adaptive indexing, adding the ability to adjust the layout and behavior of the index to workload changes even after convergence. This radical *just-in-time data structure* approach to index construction and maintenance allows for indexes that dynamically adapt to changing workloads. Even with this generality, specialization is still possible. A just-in-time data structure emulates classical adaptive indexing schemes when appropriate, while also being able to adopt a hybrid stance tailored to a specific workload. We show that our approach is feasible and enables indexes that quickly pivot between different behaviors.

1. INTRODUCTION

The performance of a Database Management System is closely coupled to the index structures it uses, making index selection an extremely important part of any database deployment [8]. As workloads change, index structures must adapt. Standard DBMSes often take an all or nothing approach to this problem, where indexes are discarded or built up from scratch, incurring delays during the rebuilding process [24]. A recently developed class of data structures called adaptive indexes [21, 25] removes this limitation by facilitating *incremental, online* changes to the index. Examples of this class include Cracker Indexes [20, 21], Adaptive Merge Trees [18], and hybrid variants thereof [25, 32]. Adaptive indexes automatically optimize their physical representation in response to incoming queries, reusing work used to answer the query to also improve subsequent queries. Given enough time, a stable workload, and queries that touch all data objects, an adaptive index eventually converges to a data representation similar to that of a static index.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

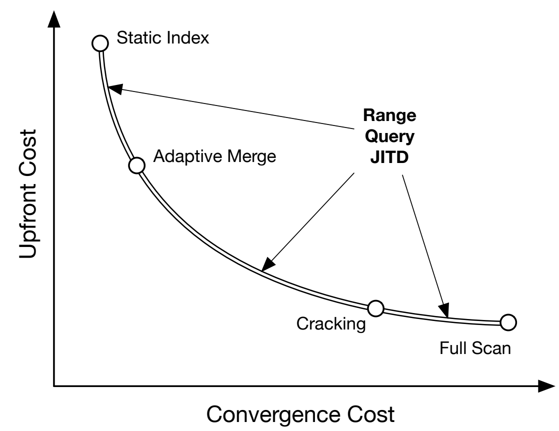


Figure 1: Adaptive indexes present a variety of static performance tradeoff options. A just-in-time data structure gracefully transitions *to any point* on the tradeoff curve *at runtime.*

In practice, adaptive indexes have been shown to outperform classical indexing strategies during periods of high load, when building new indexes is difficult or impossible. Each index has a "sweet-spot" workload, as shown in Figure 1. For example, Cracker Indexes perform well if data is frequently modified, while Adaptive Merge Trees converge to the performance of a static index far faster. However, each of these index structures occupies only a single point on the tradeoff curve. Even attempts at best-of-both worlds solutions [25] are inevitably forced to make performance tradeoffs *statically*, leading to poor performance on fluctuating workloads.

In this paper, we introduce a novel generalization of adaptive indexes called just-in-time data structures (JITDs[1]). *A JITD can adjust its behavior to match the current workload*, allowing it to dynamically reposition itself on the performance tradeoff curve. This flexibility is a result of decoupling the index's physical structure from the logic that triggers change. A JITD represents data using a *composable* library of components, called cogs, that abstractly represent the structural and semantic properties of the JITD's physical layout. Like an abstract syntax tree in a just-in-

[1]Pronounced jit-dees

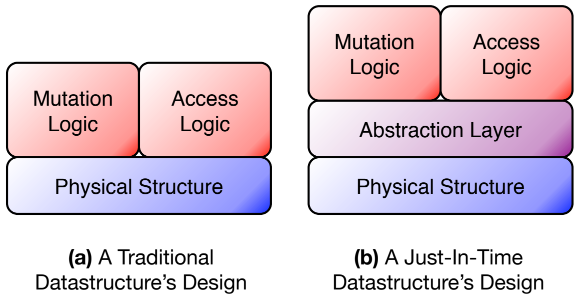


Figure 2: Just-in-time data structures place an abstraction layer between the physical representation of encoded data, and the logic that accesses and manipulates that physical representation.

time compiler, a JITD uses this abstraction layer to apply small, local optimizations to the index's physical structure at runtime. By being built on a layer of generic, composable components, a JITD can dynamically swap out one set of optimization rules for another. This makes it possible to adjust a JITD's behavior at runtime, even on a per-operation basis if necessary.

To make the idea of JITDs concrete, we will present an implementation of a JITD aimed at range queries over initially unsorted data. Our range query JITD generalizes both Cracker Indexes and Adaptive Merge Trees, and is able to emulate the stand-alone behavior of both. As we show, our range query JITD is able to *gracefully transition between these two behaviors*, enabling a wide variety of policies for dynamically adapting the JITD to changing workloads. Concretely, our contributions include:

1. Just-in-time data structures, a class of indexes that dynamically adapt to match offered workloads.
2. A transformation-based approach to defining JITDs, and an application of this approach to emulate Cracker Indexes and Adaptive Merge Trees.
3. An initial prototype implementation of the above.
4. A detailed performance evaluation validating the feasibility of just-in-time data structures.

The rest of the paper is organized as follows. In Section 2 we introduce just-in-time data structures and motivate their design with examples. In Section 3 we provide implementation details for our prototype range-query JITD. We run through an example sequence of operations in Section 4. An initial performance study and validation is presented in Section 5. Related and future work are discussed in Sections 6 and 7, respectively. We conclude in Section 8.

2. JUST IN TIME DATA STUCTURES

Just-in-time data structures create a separation between the physical representation of the data structure and the logic that defines how that representation changes over time. The physical representation of a JITD is defined by a set of generic components, or *cogs*, which capture the structure and semantics of the representation. The data structure's logic, or *policy*, is then defined over these generic components without being hardcoded for a specific physical structure. A

Cog	Description
`Array`	An array of N key/value pairs.
`SortedArray`	...in sorted order.
`Concat`	A union of records in 2 child cogs.
`BTree`	...with keys partitioned by separator.

Figure 3: Cogs used in the range query JITD.

single JITD may implement *and alternate between* many different policies, exhibiting behavior suitable for the workload being presented to the system, or rapidly adapting its behavior to fluctuating workload demands. As an example, a JITD supporting range queries and insertions might adopt one policy (*e.g.*, modeled after a Cracker Index [21]) during periods of high write activity. As write activity drops, the JITD might switch to a different policy (*e.g.*, one modeled after Adaptive Merge Trees [18], which are known to have better convergent behavior [25]). In effect, JITDs provide a principled approach to hybridizing different data structures that support similar APIs and use similar components.

This generality has the potential to add complexity to data structures re-implemented as a JITD. JITD policies must account for many different possible physical layouts, and not just one well known structure. However, in this paper, we will show through examples and experiments how the JITD design pattern creates opportunities for synergy between different policies, while keeping the resulting complexity minimal. As our running example, we will use a JITD that stores key-value pairs and supports two operations: Insert and Range-Scan. After providing a high-level view in this section, we will discuss the implementation of policies emulating Cracker Indexes and Adaptive Merge Trees, two common adaptive indexes, in Section 3.

2.1 Cogs

Cogs are independent components that are recursively composed to define the structure of a JITD, both in terms of physical layout of the data, as well as semantic constraints over that layout. Our range query JITD uses four composable cogs, shown in Figure 3. The two base cogs: `Array` and `Concat` define the physical structure of the data being encoded by the JITD. `Array` represents a vector of key-value pairs, while `Concat` represents the composition of two independent collections of key-value pairs, each recursively defined by a cog.

`SortedArray` and `BTree` extend the structure of `Array` and `Concat` (respectively) with semantic properties: Records in `SortedArray` cogs are stored sorted by key, while a `BTree` cog partitions the key space of its child cogs with a separator.

Cogs compose: A `Concat` cog is defined in terms of two additional cogs. We refer to the structure of several cogs composed together as an *assembly* of cogs. The central idea driving JITD optimization is equivalence: The same collection of records can be expressed through many different assemblies. Like bytecode in a just-in-time compiler, assemblies of cogs are gradually replaced with equivalent, ideally more efficient assemblies. These *transformations* eventually converge to an idealized representation of the data, which is dictated by the policy that the JITD is currently using.

2.2 A JITD's API

A JITD exposes a standard API, regardless of which policy is active. For example, our range query JITD exposes two operations: Range-Scan and Insert, which operate on a single, root cog. A generic implementation of the Range-Scan operation is shown in Algorithm 1. This implementation is independent of the precise physical structure of the JITD, so long as semantic constraints on the cogs are preserved (*i.e.*, on `BTree` and `SortedArray`). Moreover, this algorithm makes the best possible use of available structure. With copy-free constant-time iterator concatenation ($\circ$), only one operation (Filter on line 4) introduces any more than a logarithmic cost.

The generic implementation of the Insert operation is even simpler. New data is merged into the root via a `Concat` cog. A naive Insert implementation is shown in Algorithm 2.

Algorithm 1 Scan(low, high[, cog])

Input: `low`, `high`: the range of keys to return.
Input: `cog`: The cog to scan (default: the *root cog*).
Output: `iter`: Iterator on records in the range `low` − `high`.
1: **if** `cog` **is a** `SortedArray` **then**
2: `iter` ← **iterator on** `cog`.*data* **from** *BinarySearch*(`cog`.*data*, `low`) **to** *BinarySearch*(`cog`.*data*, `high`)
3: **else if** `cog` **is a** `Array` **then**
4: `iter` ← **iterator on** *Filter*(`cog`.*data*, `low`, `high`)
5: **else if** `cog` **is a** `Concat` **then**
6: `iter` ← *Scan*(`low`, `high`, `cog`.*left*) ∘ *Scan*(`low`, `high`, `cog`.*right*)
7: **else if** `cog` **is a** `BTree` **then**
8: **if** `cog`.*separator* ≤ `low` **then**
9: `iter` ← *Scan*(`low`, `high`, `cog`.*right*)
10: **else if** `cog`.*separator* > `high` **then**
11: `iter` ← *Scan*(`low`, `high`, `cog`.*left*)
12: **else**
13: `iter` ← *Scan*(`low`, `cog`.*separator*, `cog`.*left*) ∘ *Scan*(`cog`.*separator*, `high`, `cog`.*right*)

Algorithm 2 Insert(data[, cog])

Input: `data`: The data to insert.
Input: `cog`: The cog to insert into (default: the *root cog*).
Output: `cog`: The replacement for `cog`.
`cog` ← `Concat(cog, Array(data))`

2.3 Generalized JITDs

In this paper we focus on a specific class of index structures to demonstrate the feasibility of the JITD model. However, the core principle of decomposing structure and logic is applicable to a much broader class of indexing schemes and data structures. Data structures with more complex semantic predicates (*e.g.*, R-trees [19]), can be expressed through new cogs (*e.g.*, a multi-dimensional `BTree` cog). Data structures with more complex structural properties (*e.g.*, LSM Trees [29], which distinguish between disk- and memory-resident arrays) can be expressed similarly. The JITD model streamlines the process of hybridizing two or more such structures.

3. IMPLEMENTING POLICIES

Calls to the JITD's API trigger changes to its structure. A change may occur due to the operation itself (*e.g.*, an Insert), or may occur as an optimizing side-effect of the policy

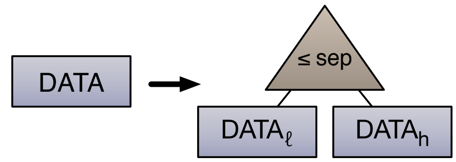


Figure 4: A visualization of the `CrackInTwo` transform (Algorithm 3). $DATA$ **is partitioned by** sep **into** $DATA_\ell$ **and** $DATA_h$**, and a new `BTree` node is created linking the two fragments.**

the JITD is currently using. Each policy defines *transformations* to be triggered before, after, or during execution of each operation. A transformation recursively rewrites the JITD's structure, similar to a compiler optimizing an AST or machine code.

The logic for each transformation is defined by visitor pattern. Each transformation matches a specific pattern, or *assembly* of cogs and defines a procedure for constructing a new, equivalent assembly to replace it. In this section, we illustrate transformations through policies that emulate two common adaptive index structures: Cracker Indexes and Adaptive Merge Trees. We then discuss the creation of hybrid policies.

3.1 Policy 1: Cracking

Cracker Indexes [20, 21] (also called Cracked Databases) are a type of adaptive index that begins as an unsorted array of records. In a Cracker Index, each range scan forces the data to be partitioned on each range scan boundary. Pointers to partition boundaries are maintained in a B-Tree. This process simultaneously answers the query and brings the data closer to being sorted, making subsequent queries more efficient.

Used in isolation, the cracking policy mimics the behavior of a Cracker Index through three transformations: `CrackInTwo`, `Pushdown`, and `MergeArrays`. `CrackInTwo`, summarized in Algorithm 3, partitions an unsorted array based on either range scan boundary, replacing the `Array` cog with a `BTree` cog pointing to the newly fragmented arrays. Partitioning is performed in-place if possible. Range Scan triggers the `CrackInTwo` transformation twice, once for each scan boundary[2]. `BTree` cogs limit which nodes are visited by `CrackInTwo` to only those nodes that could contain the scan boundary (`val`).

Algorithm 3 The `CrackInTwo` transform's visitor

Input: `cog`: The `Array` cog being visited.
Input: `val`: The partition boundary to crack on.
Output: `cog`: A replacement cog.
if `cog` **is a** `SortedArray` **then return**
`low`, `high` ← *Partition*(`cog`, `val`)
`cog` ← `BTree(val, low, high)`

As presented in Section 2.2, insertions concatenate new data onto the root. Two transformations incorporate this new data into the existing hierarchy of `BTree` cogs. First,

[2]Our implementation also uses a more complex transformation, `CrackInThree` [21] that simultaneously cracks both range scan boundaries.

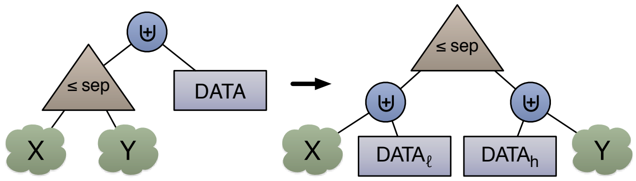


Figure 5: A visualization of the `Pushdown` transform (Algorithm 4). $DATA$ is partitioned by sep into $DATA_\ell$ and $DATA_h$, and pushed into the `BTree` node. The subtrees X and Y remain unmodified.

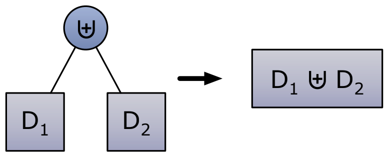


Figure 6: The `MergeArrays` transform (Algorithm 5) copies a concatenation of two arrays into a single contiguous memory region.

the `Pushdown` transformation shown in Algorithm 4 and illustrated visually in Figure 5 pushes `Concat` cogs down through `BTree` cogs. The `Array` child of the `Concat` is partitioned, and the resulting arrays are pushed down into the `BTree` child. Similar transformations are used for several different variations of this assembly. `Pushdown` is triggered before the Range Scan operation, and visits every `Concat` cog that falls within the scan boundaries, including those created by `Pushdown` itself. The `MergeArrays` transformation shown in Figure 6 and Algorithm 5 then copies each concatenation of `Array` cogs into a single, contiguous `Array`. Note that unlike updates in traditional Cracker Indexes [22], the cracker policy does not mandate that all data ultimately reside in a single contiguous region of memory.

Algorithm 4 The `Pushdown` transform's visitor

Input: `cog`: The `Concat` cog being visited.
Output: `cog`: A replacement cog.

if `cog`.$left$ **is a** `BTree` **and** `cog`.$right$ **is a** `Array` **then**
  `a`, `b` $\leftarrow$ $Partition$(`cog`.$right$, `cog`.$left.separator$)
  `cog` $\leftarrow$ `BTree`(`cog`.$left.separator$,
    `Concat`(`cog`.$left.left$, `a`),
    `Concat`(`cog`.$left.right$, `b`))

3.2 Policy 2: Adaptive Merge

Adaptive Merge Trees [18] are a second class of adaptive index. An Adaptive Merge Tree initially begins as a collection of sorted runs, each storing one partition of the data. On each range query, records from each sorted partition that fall within the range scan's bounds are extracted and merged together into a new partition called the primary. As records are merged into the primary, performance approaches that of a pure BTree. An Adaptive Merge Tree converges to an average lookup performance of $40\mu s$ per operation for a 1GB dataset, as compared to an average of $100-1000\mu s$ per operation performance for a Cracker Index (after 5000 operations). This improved performance comes at the cost of a much slower startup. Sorting data can add multiple seconds to the cost of a read and/or write, and the initial few merge steps can involve substantial amounts of data copying.

Algorithm 5 The `MergeArrays` transform's visitor

Input: `cog`: The `Concat` cog being visited.
Output: `cog`: A replacement cog.

$l, r \leftarrow cog.left, cog.right$
if l **is a** `SortedArray` **and** r **is a** `SortedArray` **then**
  `cog` $\leftarrow$ `SortedArray`($SortMerge(l, r)$)
else if l **is a** `Array` **and** r **is a** `Array` **then**
  `cog` $\leftarrow$ `Array`($l.data \circ r.data$)

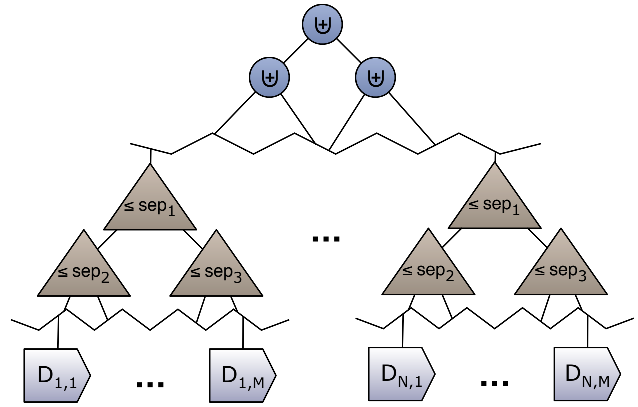


Figure 7: The adaptive merge policy creates a three-tiered structure, using a hierarchy of `Concat` cogs to link partitions. Each partition consists of a sequence of of `SortedArray` cogs ($D_{i,j}$) linked by `BTree` cogs. The `Merge` transform's visitor (Algorithm 6) gradually merges fragments of each partition into the leftmost partition ($D_{1,j}$).

The adaptive merge policy mimics the behavior of an adaptive merge tree through a transformation called `Merge`, several simpler supplemental transformations, and a utility procedure called $Extract$. Prior to each read, the `SortArrays` transformation (not shown) replaces `Array` cogs with `SortedArray` cogs. Arrays larger than a threshold value (10 million elements in our implementation) are broken into individual *partitions*, each smaller than the threshold value, before being sorted. The resulting partitions are represented as a hierarchy of `Concat` cogs, as shown in Figure 7. If the root of the tree includes `BTree` cogs, these are used to limit `SortArrays` to arrays in subtrees containing applicable keys. Arrays not directly applicable to the current query remain unsorted.

After ensuring that relevant arrays are sorted, the `Merge` transformation shown in Algorithm 6 is applied bottom-up to all hierarchies of `Concat` cogs. When used in isolation, this policy only creates one such hierarchy, at the root — we return to other cases in Section 3.3. `Merge` is applied to the immediate descendants of this hierarchy, again called partitions. Records in each partition within the range scan's bounds are separated out using the $Extract$ procedure shown in Algorithm 7. Records not falling into one of the range scan bounds are re-assembled and returned to their original partitions, while records in the scan bounds are merged together into the leftmost partition (the primary). As an optimization, the merged records are grouped into small blocks (10,000 records in our implementation), making subsequent *Extract*s on the primary more efficient.

Algorithm 6 The `Merge` transform's visitor

Input: `partitions`: The immediate descendants of the hierarchy of `Concat` cogs rooted at the cog being visited.
Input: `low`, `high`: The range of keys being requested.
Output: `partitions`: A list of replacement cogs.

for i **from** 1 **to** $|$`partitions`$|$ **do**
  `lowCog`$[i]$, `midCog`$[i]$, `highCog`$[i]$ ← *Extract*(`partitions`$[i]$, `low`, `high`)
`partitions`$[1]$ ←
  `BTree`(`low`,
    `lowCog`$[1]$,
    `BTree`(`high`,
      SortMerge(`midCog`$[1] \ldots$ `midCog`$[|$`partitions`$|]$),
      `highCog`$[1]$))
for i **from** 2 **to** $|$`partitions`$|$ **do**
  `partitions`$[i]$ ←
    `BTree`(`low`,
      `lowCog`$[i]$,
      `BTree`(`high`, $\emptyset$, `highCog`$[i]$))

Algorithm 7 Extract(`cog`, `low`, `high`)

Input: `cog`: A cog to separate by keys.
Input: `low`, `high`: The range of keys being requested
Output: `lowCog`: A cog with keys from $-\infty$ to `low`.
Output: `midCog`: A cog with keys from `low` to `high`.
Output: `highCog`: A cog with keys from `high` to ∞.

if `cog` **is a** `BTree` **then**
  if `cog`.*separator* $\leq$ `low` **then**
    `lowCog`, `midCog`, `highCog` ← *Extract*(`cog`.*right*, `low`, `high`)
    `lowCog` ← `BTree`(`cog`.*separator*, `cog`.*left*, `lowCog`)
  else if `cog`.*separator* $\geq$ `high` **then**
    `lowCog`, `midCog`, `highCog` ← *Extract*(`cog`.*left*, `low`, `high`)
    `highCog` ← `BTree`(`cog`.*separator*, `highCog`, `cog`.*right*)
  else
    `lowCog`, $\texttt{midCog}_\ell$, _ ← *Extract*(`cog`.*left*, `low`, `high`)
    _, $\texttt{midCog}_r$, `highCog` ← *Extract*(`cog`.*right*, `low`, `high`)
    `midCog` ← `BTree`(`cog`.*separator*, $\texttt{midCog}_\ell$, $\texttt{midCog}_r$)
else if `cog` **is a** `SortedArray` **then**
  `lowCog`, `midCog` ← *Partition*(`cog`, `low`)
  `midCog`, `highCog` ← *Partition*(`midCog`, `high`)
else if `cog` **is a** `ConcatCog` **then**
  Ensure that `cog` was visited by `Merge`
  Apply *Extract* to $\texttt{partitions}_1$ as created by `Merge`

3.3 Hybridization and Generalization

The JITD model provides opportunities to exploit synergies between policies. Semantic constraints on the structure created by one policy can be used to minimize work in another. Recall that work in a JITD is expressed as a set of transformations following the visitor pattern. Altering a transformation to exploit existing semantic constraints frequently *only requires changing the set of nodes that it visits.*

Partial Sorts. The cracking policy pre-partitions data using `BTree` cogs; The adaptive merge policy can use these cogs at the root to limit which parts of the data need to be sorted, ignoring those outside of the range scan bounds. `SortArrays` is applied only to cogs falling within the range scan bounds.

Partial Merges. The `PushdownConcats` transformation creates much smaller arrays for the `Merge` transform to combine — This can be exploited by the adaptive merge policy by applying the `Merge` transform to *any* hierarchy of `Concat` cogs rather than just the one at the root, and using `BTree` cogs closer to the root to reduce the amount of data being merged together.

Cracking Sorts. The `CrackInTwo` transformation ignores existing `SortedArray` cogs created by the adaptive merge policy, as they already support efficient range scans.

Through several minor changes to each policy, the overhead of switching between different policies becomes minimal. This enables a variety of different hybrid *policies*. For example, the cracking policy exhibits better initial performance after a write and simultaneously reduces the cost of a subsequent switch to the adaptive merge policy for better performance in the tail. One simple, naive hybrid policy would start by fulfilling requests as per the cracking policy, and eventually switch to fulfilling requests as the adaptive merge policy. More complex policies may be defined to satisfy desired throughput requirements, support variable read priorities, to exploit other workload characteristics, or to react to structural properties of the data (*e.g.*, small Arrays might be simply sorted outright and not cracked as in the HCR and HCS datastructures [25]). There is significant potential for future work in exploring this space of possible optimization policies.

4. EXAMPLE

We now present an end-to-end example of how a range-query JITD might be used. We will show how a JITD can freely switch between policies by illustrating three different evolutions of the same data structure as different policies are used to respond to scan requests. The full step-by-step example is illustrated in Figure 8. The initial state of the JITD, illustrated in Figure 8 as state (a), is a six-record array with keys $\{2, 3, 4, 7, 8, 1\}$. Subsequent states show how the JITD evolves as the following four operations are performed under a mix of policies:

1. `Scan(`$-\infty$`, 5)`
2. `Insert({9, 5, 6})`
3. `Scan(6, `∞`)`
4. `Scan(2, 5)`

`Scan(`$-\infty$`, 5)`: The first operation that the JITD receives is a scan for records up through 5. State (b) illustrates the effect of satisfying the request through the cracking policy. As there is only one node, the `CrackInTwo` transform partitions the six element array on the key 5. As a simple optimization, the partitioning occurs in-place, necessitating only a single swap of 7 and 1. Two new cogs are created, each *referencing* a region of the original array.

`Insert({9, 5, 6})`: Three records, $\{9, 5, 6\}$ (in that order) are inserted next. As per Algorithm 2, this is represented by creating a new `Array` cog for these records, and

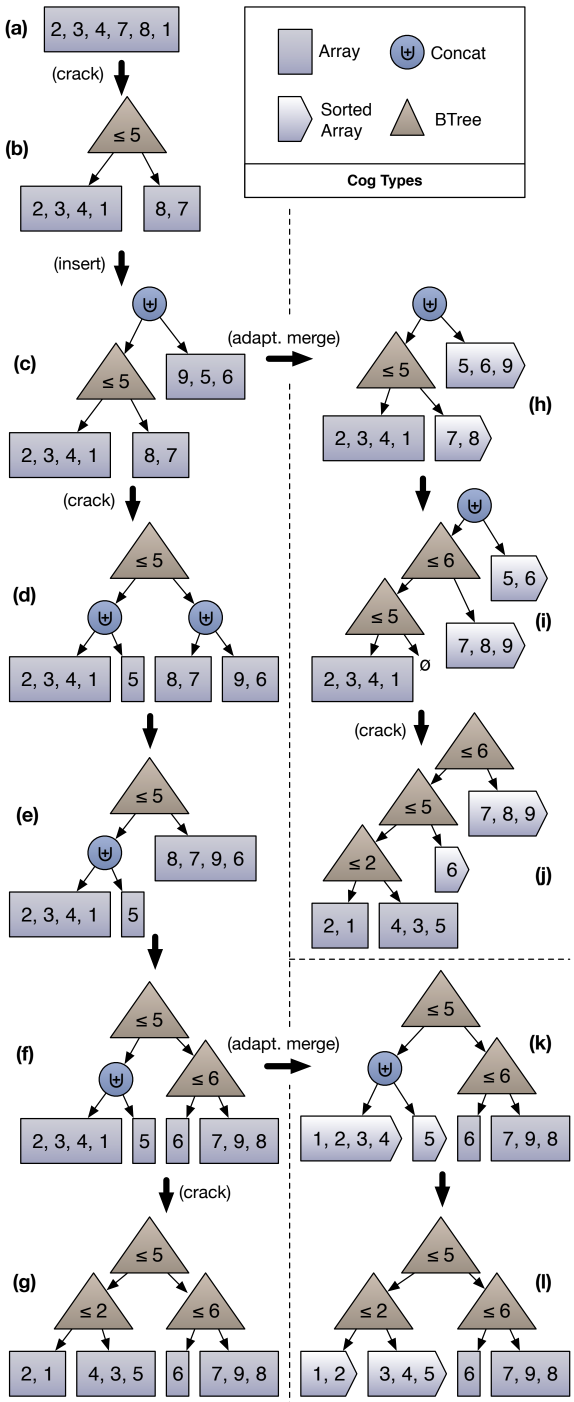


Figure 8: An example trace of optimizations applied to a Range Query JITD, branching depending on policy decisions. The initial state (a) consists of six records in an unsorted array. A scan operation is satisfied by cracking (b), followed by an insertion of three additional records (c). The next scan is satisfied by cracking (c – f) or adaptive merge (h – i). The third and final scan continues from state (f) by cracking (g) or adaptive merge (k – l), or from state (i) by cracking (j).

then merging it with the root in a newly created `Concat` cog, as shown in state (c).

`Scan(6,` ∞`)`: Next, the JITD performs a scan for records greater than 6. We illustrate this request under both the cracking policy (d – f) as well as the adaptive merge policy (h – i). The cracking policy applies three transforms: `Pushdown`, `MergeArrays`, and `CrackInTwo`[3]. The `Pushdown` transform partitions the array of new records, creating new `Concat` cogs below the `BTree` cog, as shown in state (d). Because we are only interested in keys greater than 6, the left-hand branch of the `BTree` cog can be safely ignored. The `MergeArrays` transform then copies the two remaining arrays into a single, contiguous region of memory as shown in state (e). Finally, the `CrackInTwo` transform partitions the newly created `Array` cog, resulting in state (f).

Switching to a different policy does not require any immediate layout changes to the JITD. Rather, each policy dictates how the JITD reacts to operations applied to it. The adaptive merge policy treats each contiguous hierarchy of `Concat` cogs as a set of partitions to be sorted and merged. In this case, there are two partitions. The `SortArrays` transform ensures that all relevant `Array` cogs are sorted, resulting in state (h). As before, the existing `BTree` cog allows `SortArrays` to ignore its left-hand branch. Next, the `Merge` transform extracts the fragment of each partition that falls within the search range: 9 from the newly inserted records, and 7 and 8 from the partition rooted at the `BTree` cog. The extracted records are merged together into the left-most partition, as shown in state (i). For this specific operation, the entire right-hand branch of the `BTree(`≤ 5`)` cog is extracted. A special singleton cog: the empty set $\emptyset$ is used to denote the lack of records in that range.

`Scan(2, 5)`: The JITD is scanned for records in the range $(2, 5]$. Recall that state (f) was reached by responding to `Scan(6,` ∞`)` according to the cracking policy, and that state (i) was reached by responding according to the adaptive merge policy. We will illustrate three outcomes: State (g) shows the effects of continuing with the cracking policy, States (k) and (l) show the effects of switching from cracking to adaptive merge, and state (j) shows the effects of switching from adaptive merge to cracking.

As before, the cracking policy applies `Pushdown`, `MergeArrays`, and `CrackInTwo`. The `Pushdown` transform has no work to do, so the `MergeArrays` transform merges the two leftmost `Array` cogs together. The newly formed cog is partitioned by the `CrackInTwo` transform, resulting in state (g).

For the adaptive merge policy, the `SortArrays` transform sorts partitions below the `Concat` cog, creating state (k). As usual, the `BTree` cog at the root allows a fragment of the JITD to remain unmodified. After sorting, records in the scan range are extracted from each partition: 3 and 4 from the left partition and 5 from the right. The extracted records are merged into a single `SortedArray` cog, creating state (l).

Finally, we return to state (i), created by the adaptive merge policy. Again, recall that policy changes do not require any physical changes to the JITD. Thus, the cracking policy is applied exactly as in each previous case, through the three transforms: `Pushdown`, `MergeArrays`, and `CrackInTwo`. The `Pushdown` transform partitions the `SortedAr-`

[3]Our implementation also uses a more complex variant: `CrackInThree` [21]. In the interest of clarity, we present the example only using the simpler `CrackInTwo` transform.

`ray` cog $\{5, 6\}$ twice. Because the array being pushed down is sorted, each partitioning step can be performed in $\log n$ time. The first partitioning step pushes all records down the left-hand branch, as there are no records greater than 6. The second step splits the array into $\{5\}$ and $\{6\}$. Next, the `MergeArrays` transform combines the newly created $\{6\}$ `SortedArray` cog with the already existing $\{2, 3, 5, 1\}$ `Array` cog. Finally, the `CrackInTwo` transform partitions and splits the newly created cog, creating state (j).

5. EVALUATION

Our key contribution in this paper is illustrating the flexibility afforded by the JITD model. In this section we assess the performance impact of this added flexibility. We show that the JITD form of a Cracker Index and an Adaptive Merge Tree *retains performance competitive with the original data structures.* Then, we discuss a range of policies that the JITD implementations of these data structures enable. We show that *these policies can be used to quickly and easily reposition the data structure's performance characteristics* to adapt to new workloads.

5.1 Experimental Setup

Our JITD was implemented in Java 1.7. Experiments were performed on a 2x16 core 1.8 GHz Intel Opteron with 128 GB of RAM, running RHEL 6, and OpenJDK 1.7. JVM heap sizes were set high enough to minimize interference from Java's garbage collector, and experiments were run single-threaded. Source code for our JITD implementation and experiments is available for download[4].

All experiments begin with an initially unsorted array of 100 million records stored row-wise. Each record consists of an 8-byte integer key field, and an 8-byte payload field. The resulting structure is analogous to a sideways cracker index [23]. The resulting dataset was approximately 1.6 GB. Each read operation reads a randomly selected range of keys

```
SELECT key, payload FROM R
WHERE key >= low AND key < high
```

`low` was selected randomly, and `high` was selected relative to `low` to return approximately 2 to 3 thousand records. A total of 10,000 reads were performed. At the 5,000 read mark, an array containing an additional 10 million random records (about 160 MB) was inserted into the data structure. As per the Insert operation (Algorithm 2), the structure's root was replaced by a concatenation of the original root and a new unsorted array cog containing the new data.

5.2 Primitive JITD Policies

Figures 9.a and 9.b show the raw performance of the **cracking** and **adaptive merge** policies, as presented in Sections 3.1 and 3.2, respectively. Cracking's initial performance is sub-second and quickly converges to millisecond latencies, as transformations manipulate data in-place. Adaptive merge has an extremely high upfront cost of about 33 seconds for the initial sort (also performed in-place), and again for about 3 seconds after the write. The merge process requires a substantial number of memory copies, so for the next two thousand iterations performance alternates between merges costing 1-20ms, and fast-path reads that do not require a merge, and cost 30-40μs. *This performance is comparable to experiments performed by Idreos et al.* [25]. We attribute the slower convergence and lower initial cost of the adaptive merge policy to a chunking optimization in [25] that merges batches of data with each read. A similar optimization is feasible in our own framework.

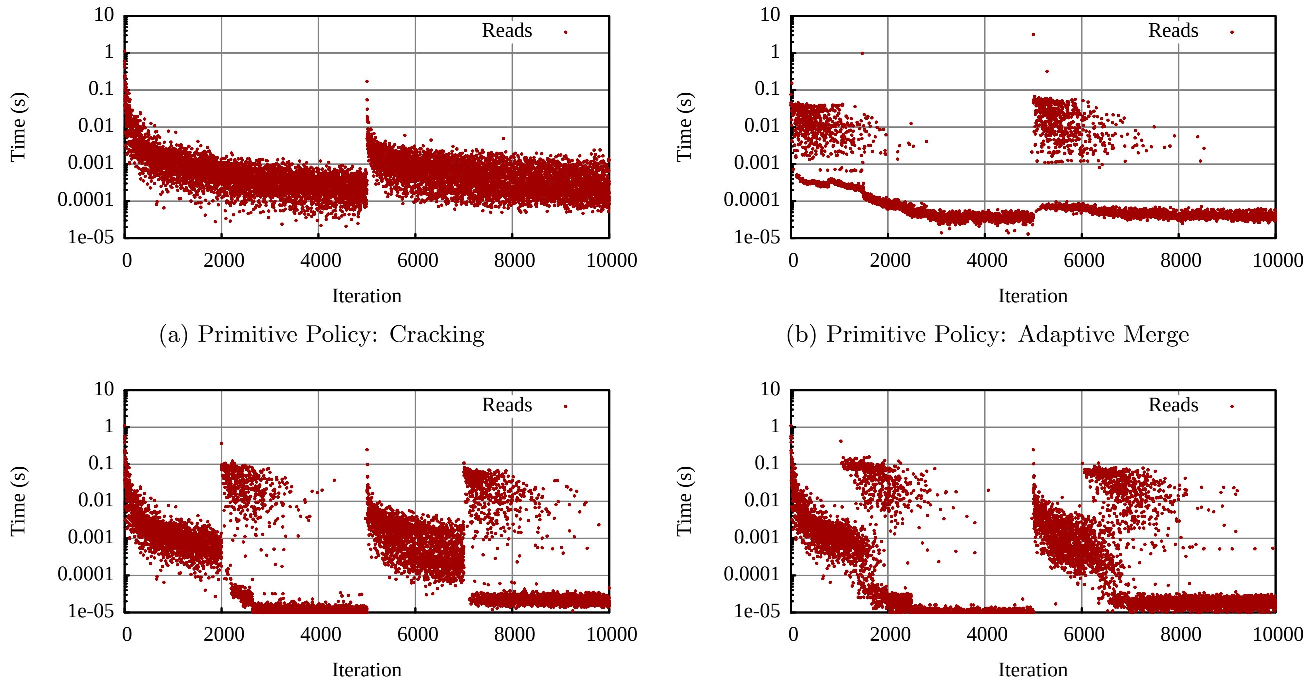
5.3 Hybrid Policies

Swap and **transition** implement two different hybrid policies. Under the swap policy, the JITD simulates the cracking policy for the first 2000 operations after a write and then switches immediately to fulfilling requests according to the adaptive merge policy. The data structure remains unchanged after each policy switch, but new requests are satisfied using the new policy's transforms. The resulting data structure is a synthesis of both cracker indexes and adaptive merge trees. Performance results for the swap hybrid policy are presented in Figure 9.c. The cracking policy's `Pushdown` transformation reduces the amount of work required for each sort operation, lowering the adaptive merge policy's per-read cost. The transition between policies is performed gracefully, and the resulting data structure actually performs *better* than either pure merge or pure cracking. Swap's initial post-write performance is comparable to pure cracking, and the first set of writes after switching to the adaptive merge policy take on the order of 100ms as compared to the 3 seconds required to sort the 160 MB of data written. Then, at the tail, performance quickly converges to the sub-millisecond latency of the adaptive merge policy's tail.

The performance of the transition policy is presented in Figure 9.d. Like the swap policy, each write resets the JITD to the cracking policy. After 1000 iterations, the JITD begins a gradual transition from the cracking to the adaptive merge policy. Each request is satisfied with a randomly selected policy, starting with a 100% chance to satisfy requests as per the cracking policy, and linearly transitioning to a 100% chance to satisfy requests according to the adaptive merge policy. This shows that a JITD is able to seamlessly switch back and forth between the two primitive policies *on a per-operation basis.* Moreover, note the tri-modal distribution of latencies once the transition period begins: Requests satisfied according to the cracking policy continue to operate consistently in the 1ms range, while requests satisfied by the adaptive merge policy alternate between 10-100ms if a merge is required, and 30-40μs for fastpathed requests. Even though the variance in read latencies of the data structure as a whole increases during the transition period, *any variance-intolerant scan operation can still be fulfilled with the consistent latency of the cracking policy.*

A side-by-side throughput comparison of all four JITD policies is shown in Figure 10. Both swap and transition mirror the behavior of the cracking policy for the first thousand operations, both initially and after the write. Conversely, in the tail, both swap and transition mirror the convergent behavior of the adaptive merge policy, which begins to outperform the cracking policy. The adaptive merge policy is able to exploit work done by the cracking policy, resulting in a softer performance dip after it is put into use. The two primitive policies, and the two hybrid policies presented represent only a small sampling of what is possible, but demonstrate that *a JITD can quickly pivot to meet the demands of new workloads.*

[4] `http://github.com/okennedy/jitd/`



(a) Primitive Policy: Cracking

(b) Primitive Policy: Adaptive Merge

(c) Hybrid Policy: Swap

(d) Hybrid Policy: Transition

Figure 9: Read performance trace for the range-query JITD in four different modes with a write after 5,000 reads. A 33 second read spike for the first read under the Adaptive Merge policy is not shown.

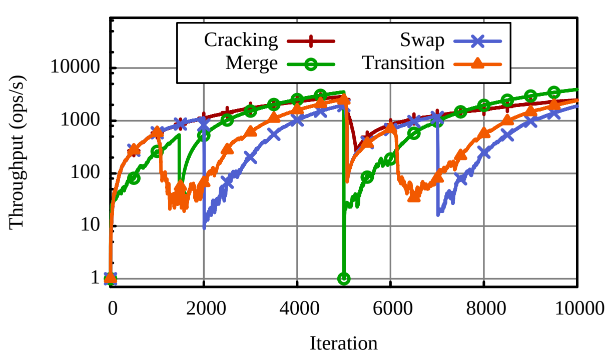


Figure 10: Average throughput for Figure 9.

6. RELATED WORK

Adaptive indexing [16, 17] has become a popular mechanism for incrementally adjusting indexes dynamically based on workload. Adaptive indexes allow a database or data warehouse to improve performance by leveraging work that needs to be performed to resolve a query, to also improve the indexing structure. Database Cracking [20, 21, 22] was the first to pioneer this scheme. Other schemes have followed, including Adaptive Merge Trees [18] as well as several variants that combine the features of both implementations allowing for both merging and cracking [25]. In all of these proposals and systems, the underlying index implementation makes static performance tradeoffs, resulting in a canonical representation that can no longer be adjusted to changes in broad workload characteristics. JITDs are a generalization of adaptive indexing, whose goal is to allow continual shifts in the underlying index structure. A steady state is only reached if the workload itself reaches a steady state. JITDs are well suited to workloads that have distinct phases, mirroring common trends in internet traffic.

SMIX [32] is an adaptive indexing structure, that like JITDs, does not have an explicitly steady state that the indexing scheme tends towards. SMIX allows for indexes to both dynamically grow and shrink. This is accomplished by introspecting the index and collecting information on which portions of the index are useful and what portions of the index are not useful. Based on this collected information the index is augmented. We believe this approach can be synergistic with JITDs, allowing for further refinement of the index. We believe the SMIX approach can be encoded in a JITD's policies, by encoding information gathering during access of specific cogs composing the index structure itself.

Legorithmics [26] takes an approach similar to JITDs for optimizing traditional database algorithms. Using a library of algorithmic components and a memory hierarchy specification, the Legorithmics compiler specializes canonical algorithms for databases (e.g., Sort) to new memory hierarchies.

Building Block Databases.

Over a decade ago, Chaudhuri *et al.* [10] advocated a low-level RISC style database architecture, which would provide target applications and end-users with a *sufficient, but minimal* feature-set, thus avoiding the overheads of supporting unneeded functionality. More recent efforts have begun to realize this vision [5, 7, 12, 14, 27]. Of particular note is OctopusDB [13], which uses a single shared log structure to record all database updates, and treats user defined materialized views as a building block. We believe that the JITD approach to indexing structure design will be of particular use in building block databases, acting as a basis for specifying how such databases compute, organize, and query data.

Memoization.

There has been much work on memoization and self adjusting computation [1] over recent years. We believe that JITDs can benefit from such techniques to further improve performance. Selective [2] and partial [33] memoization can be leveraged to improve reads and scans over JITDs. Self adjusting computation can be leveraged to express more complex JITDs, providing a mechanism to try out multiple different representations. Self adjusting computation has been proposed for streaming big data [3, 11], and preliminary efforts exploring the use of self-adjusting computation for datastructure design appear in the Synthesis Kernel [30].

7. FUTURE WORK

In this paper, we demonstrate the feasibility of the JITD model. However substantial work remains to fully realize JITDs. In this section, we present a number of opportunities for research on just-in-time data structures. Our hope is that as JITDs mature, they will eventually form the basis for an entire class of just-in-time databases.

7.1 Rewrite Policies

In this paper, we demonstrate the feasibility of JITDs through four specific transform policies. To fully realize JITDs, it will be necessary to develop a general process for designing policies for specific workloads: How should the JITD organize its data? What kind of transforms will converge to the specified organizational structure? Under what conditions should the transforms be triggered?

Of particular interest to us are cases where the “ideal” data layout is data- or workload-dependent. Unlike the cases explored in this paper, where all policies eventually converge to a sorted array, data structures for multi-dimensional data, highly volatile data, or supporting computationally intensive search predicates may need to rapidly adapt their layouts in response to subtle changes in data or workload.

JITDs present a rich space of optimization alternatives. This rich space enables such small and subtle changes, at the cost of an explosion of the optimization search space. Streamlining the policy design process is thus an incredibly important step towards making JITDs practical. We expect initial solutions to borrow heavily from traditional database tuning optimizers [9]. In the longer term, we see policy decisions in a JITD becoming analogous to just-in-time compilation, where live feedback [32] and trial and error are used to guide optimization in real-time.

7.2 A Cog Programming Model

Transformations and data accessors can be defined generally for any data structure built from the cog abstraction. However, the cog abstraction is not static; Every new class of structural or semantic property included in the JITD requires a new type of cog. Managing the resulting explosion of interactions between cogs and transformations will require adapting existing programming language techniques. For example, object-orientation abstractions like inheritance and mixins can be used to provide backwards-compatibility for newly defined cogs.

This new programming model will require considerable compiler and runtime support as well. Pattern matching used to identify candidate assemblies that are suitable for a given transform can be expressed declaratively through graph queries. In this form, compiler techniques like code-inlining and partial evaluation can be adapted to accelerate pattern matching.

7.3 Procedure Cogs

In this paper, we introduced cogs as an abstraction for the structural and semantic properties of a data representation. Cogs also form a useful abstraction for procedural relationships, like the suspended executions used in Okasaki-style lazy data structures [28]. For example, a cog could encode the following procedure:

$$\texttt{map(}\ \lambda\texttt{x.}\ \ \texttt{x + 1)}$$

This cog would represent the set of records resulting from adding 1 to every value in the set represented by its single child `x`. The resulting structure captures partial evaluation strategies such as Stickies [15], and forms a basis for distributed replication [4].

7.4 Out-of-Core Cogs

So far, we have presented JITDs entirely in the context of main-memory data structures. Adapting JITDs for disk-resident data layouts requires restructuring transformations, accounting for the need to perform sequential reads and writes. Hyder [6] and LSM Trees [29, 31] are both examples of a similar restructuring applied to naive BTrees. When combined with procedural cogs, we believe that out-of-core JITDs will form a basis for log rewriting [4], a powerful generalization of checkpointing.

8. CONCLUSIONS

In this paper, we introduced the idea of just-in-time data structures (JITDs), and showed how the JITD model could be applied to range query data structures. JITDs decouple the logic and physical representation of an index data structure, and allow multiple behaviors, or policies to collectively manipulate a standardized library of physical layout building blocks, or cogs. Through a specific JITD implementation, we have shown that this approach is feasible, and can be used as a principled approach to hybridizing different data structures.

9. ACKNOWLEDGEMENTS

The authors would like to thank Daniel Zinn for several enlightening discussions contributing to this paper, and the reviewers for their insightful comments. The authors would also like to acknowledge the contributions of Ankur Upadhyay to the system as a whole.

10. REFERENCES

[1] U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *POPL*, 2008.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *POPL*, 2003.

[3] U. A. Acar and Y. Chen. Streaming big data with self-adjusting computation. In *DDFP*, 2013.

[4] S. Agarwal, D. Bellinger, O. Kennedy, A. Upadhyay, and L. Ziarek. Monadic logs for collaborative web applications. In *WebDB*. ACM, 2013.

[5] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 2012.

[6] P. A. Bernstein, C. Reid, and S. Das. Hyder–A Transactional Record Manager for Shared Flash. *CIDR*, 2011.

[7] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP*, 2011.

[8] S. Chaudhuri and V. Narasayya. AutoAdmin "what-if" index analysis utility. In *SIGMOD*, 1998.

[9] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[10] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *PVLDB*, 2000.

[11] Y. Chen, U. A. Acar, and K. Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *SIGPLAN*, 2014.

[12] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.

[13] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, 2011.

[14] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD*, 1989.

[15] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.

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

[17] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *TPC-TC*, 2011.

[18] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[19] A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Record*, 14(2):47–57, June 1984.

[20] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 2012.

[21] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[22] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[23] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[24] S. Idreos, S. Manegold, and G. Graefe. Adaptive indexing in modern database kernels. In *EDBT*, 2012.

[25] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 2011.

[26] Y. Klonatos, A. Nötzli, A. Spielmann, C. Koch, and V. Kuncak. Automatic synthesis of out-of-core algorithms. In *SIGMOD*, 2013.

[27] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *VLDBJ*, 4(9):539–550, June 2011.

[28] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1999.

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

[30] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[31] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. In *SIGMOD*, 2012.

[32] H. Voigt, T. Kissinger, and W. Lehner. SMIX: Self-managing indexes for dynamic workloads. In *SSDBM*, 2013.

[33] L. Ziarek, K. Sivaramakrishnan, and S. Jagannathan. Partial memoization of concurrency and communication. In *ICFP*, 2009.