PROJECT DESCRIPTION

1 — Introduction

One of the most important design decisions for any Data Management System (DMS) is how data should be organized [17]. Deciding on the physical layout of a specific index often involves understanding and accounting for a bewildering array of workload characteristics: read vs write ratios, read selectivities, write batch sizes, the distribution of "hot" keys, availability of idle time, burstiness, cache sizes, required consistency/durability guarantees, and many more. Worse still, these characteristics are often temporal, requiring databases to transition between different physical layouts as workloads change.

The brute force approach taken by many DMSes, where static indexes are discarded or built up from scratch as workloads change, incurs delays during the rebuilding process [45]. Adaptive indexes [46] address this problem by using specialized data structures that dynamically reorganize data as queries arrive. Numerous adaptive indexes [41, 46, 39, 36, 91, 78, 42] have been proposed to mitigate the cost of rebuilding indexes, but even these have specific workload sweet-spots.

Example 1 *Figure 1(a) illustrates the performance characteristics of two adaptive indexes: Adaptive Merge Trees and Cracker Indexes. Adaptive Merge Trees produce an index with higher costs immediately after a write, but converge to the performance of a statically built index significantly faster than Cracker Indexes. Figure 1(b) shows read throughput for Cracker Indexes and Adaptive Merge Trees over multiple read operations. A write operation after 5000 reads drastically slows the throughput of Adaptive Merge Trees, which must sort and organize the newly written data. The Cracker index performs better in the short term, but is overtaken by Adaptive Merge Trees as the workload progresses.*

To facilitate a higher degree of adaptability, PIs Kennedy and Ziarek recently proposed a new class of adaptive index structures called Just-in-Time Data Structures [57] (JITDs). A JITD gracefully adapts to changing workloads by making small localized changes in its physical layout that dynamically adjust its performance characteristics. This flexibility arises as a result of decoupling the index's physical structure from the logic that triggers change. A JITD represents data using a composable *grammar* of building blocks that abstractly represent the structural and semantic properties of the JITD's physical layout. Like an abstract syntax tree used by a just-in-time compiler, a JITD uses this abstraction layer to apply small, local optimizations to the index's physical structure at runtime. Policies, or sets of optimization and transformation rules, allow a JITD to converge to a desired representation similar to a static index. A JITD can dynamically switch between policies, incrementally transitioning between two layouts, each beneficial for a different workload. A JITD can also apply *partial* transitions, creating hybrid physical layouts.

Example 2 *Returning to Example 1 and Figure 1(b), Swap and Transition denote hybrid policies that start by emulating Cracker Indexes after a write and transition to an Adaptive Merge Tree, suddenly (**Swap**) or gradually (**Transition**). Each hybrid policy was implemented in under 50 lines of java (almost all java boilerplate). Both hybrid policies offer a midpoint between the short- and long-term performance of Cracker Indexes and Adaptive Merge Trees.*

Our preliminary work focused on demonstrating the *feasibility* of basic index JITDs through two workload characteristics: short-term latency requirements against bulk, long-term throughput. We have shown JITD's potential [57]. **We now propose to realize that potential** by (a) generalizing the index grammar, (b) addressing concrete systems challenges involved in deploying JITDs, and (c) further adapting the JITD model to an even more general class of workload: view maintenance and deferred computation.

1.1 — Team Qualifications Generalizing JITDs requires a truly interdisciplinary approach spanning both databases and programming languages. PI Kennedy's expertise covers databases, incremental computation [6, 52, 61], uncertain data management [55, 50, 97], and online aggregation [51, 54]. PI Ziarek's expertise covers programming languages [102, 100], real-time systems [13, 96], and virtual machines [69, 70]. PIs Kennedy and Ziarek share experience in compiler design [6, 61, 80, 103], and have been working together for the last three and a half years. The PIs have established a joint lab [56, 5, 53] that already has experience with JITDs [57].

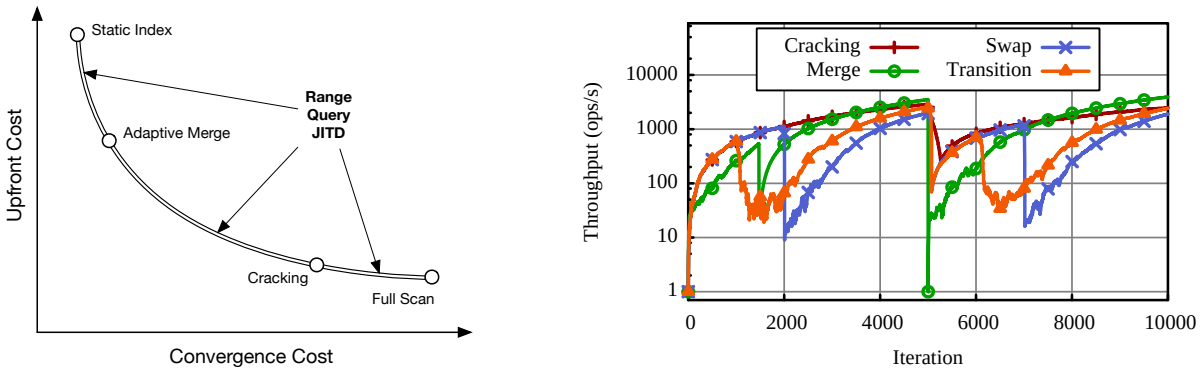


Figure 1: Adaptive indexes present a variety of static performance tradeoff options. A just-in-time data structure gracefully transitions *to any point* on the tradeoff curve *at runtime*. When compared to **Cracker** Indexes, which have better short-term throughput, and Adaptive **Merge** Trees, which have better long-term throughput, JITDs enable policies that gracefully **Transition** from the short-term performance of Cracking Indexes to the long-term performance of Adaptive Merge Trees.

1.2 — Research Contributions Our proposed work consists of three research thrusts:

1. **Generalize Index JITDs**: Our first thrust generalizes JITDs to a broader range of index types. We also explore declarative languages for policy specification and techniques for automating policy design.
2. **Systems Challenges**: Our second thrust explores techniques for realizing the conceptual design developed in Thrust 1. A key challenge we expect to address is enabling concurrency in JITDs and in policies themselves.
3. **View Maintenance**: Our final thrust further generalizes JITDs, enabling JITDs that model Incremental View Maintenance (IVM). Our approach generalizes deferred computation (*i.e.*, suspensions [66]) to create a data storage model general enough to encode distributed transaction processing [30], incremental view maintenance [61, 6], and numerous systems that use IVM as a proxy for index layout decisions [28, 48, 15].

2 — Background and Related Work

The need to specialize data management systems to their respective workloads has been clear for quite some time [19]. Although tuning an arcane set of properties and configuration parameters is not the answer [83], system fragmentation and end-to-end engineering are equally daunting propositions [28, 48, 15]. Instead, alternative approaches have arisen in the form of dynamic, self-organizing physical layout components [8, 36, 35, 46, 45, 33, 39], or simple, common building blocks for modular physical layouts [24, 38, 58, 93, 95, 65, 16, 28]. Our proposed work draws on both areas, and provides a set of common building blocks that can be explored to identify a suitable dynamic self-adapting index structure for any given workload.

2.1 — Adaptive Datastructures Adaptive indexing [33, 35, 34] 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 [39, 41, 43] was the first to pioneer this scheme. Other schemes have followed, including Adaptive Merge Trees [36], SMIX [91], as well as several variants that combine the features of multiple implementations, for example enabling simultaneous merging and cracking [46]. 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 [78]. 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 in particular are targeted at workloads that have distinct phases.

Memoization. There has been much work on memoization and self adjusting computation [2] over recent years. We believe that JITDs can benefit from such techniques to further improve performance. Selective [3]

and partial [101] 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 [4, 21], and preliminary efforts exploring the use of self-adjusting computation for datastructure design appear in the Synthesis Kernel [71].

2.2 — Building Block Databases Building block abstractions are prevalent in database literature. Work in this area explores the use of program analysis to dynamically identify and assemble relevant building block components, the use of simple component abstractions like logs and views, and hybrid approaches that dynamically assemble programmer-defined building blocks.

Language-Based Building Blocks. The concept of a small, sleek, RISC-style approach to database operators [20] has been applied numerous times. Under such an approach, only database functionality required by the current workload is "active" at any given time. Full systems have successfully applied this approach to a variety of workloads using an extended form of datalog [38], and in column stores [14].

Given more advance knowledge of the workload, even more aggressive strategies are available. When the workload can be described, either by a set of (parameterized) queries, or more generally by a domain specific language (DSL), entire components of the data processing engine may be compiled into a highly-specialized runtime. Efforts on query compilation include aggressive optimization of the evaluation pipeline [65], compiling specialized datastructures [52, 57], and specializing algorithms for memory hierarchies [58]. Efforts to optimize DSLs range from early work on collection programming and object stores [11, 94], to more recent efforts directed at the compiler pipeline [16].

Abstraction-Based Building Blocks. A second school of thought uses simple abstractions such as logs and views. Logging in particular allows for an extremely low-cost write primitive that still ensures ordering. It then falls to the infrastructure developer to restructure, organize, or persist the log to ensure the appropriate tradeoffs for performance [28], data-availability [1], consistency [40], or replication [12, 10, 90, 68].

Operating over a serialized stream or log of writes is effective for a variety of reasons. In addition to granting full control over write complexity [52, 28], a serial stream of updates in a distributed setting allows participants to operate in lock-step, a process known as state machine replication [77]. Furthermore, logging creates an implicit version history. Versioning creates monotonicity, which as per the CALM principle [7] provides eventual consistency for free.

Operation-Based Building Blocks. A third approach lies between the above strategies. As before, updates are the core abstraction, but upfront ordering is treated as a bottleneck. Such systems are eventually consistent; in lieu of deterministic ordering, update semantics are exploited to resolve conflicts as they are detected. This includes using algebraic properties like commutativity [79], the existence of commutators [29], or user-provided monotonic merge lattices [24]. A more brute-force approach is to simply defer merging to the application layer [27].

3 — Preliminary Work: Just-in-Time Datastructures

Just-in-time data structures separate the physical representation of an index structure and the logic that defines how that representation changes over time. The physical representation of a JITD is defined by a *grammar* of generic building blocks, or *nodes* that capture the structure and semantics of the representation. The data structure's logic, or *policy*, is then defined as rewrite rules over this grammar without being hard-coded for a specific physical structure. A single JITD may implement *and alternate between* many different policies, exhibiting behavior suitable for one specific workload, or rapidly adapting 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 [41]) during periods of high write activity. As write activity drops, the JITD might switch to a policy that converges faster (*e.g.*, one modeled after Adaptive Merge Trees [36, 46]). In effect, JITDs provide a principled approach to hybridizing different data structures that support similar operations 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. As

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

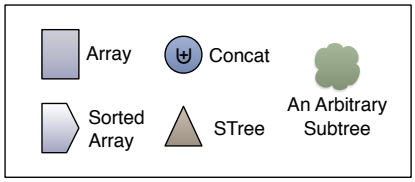


Figure 2: nodes used in the range query JITD and a key of symbolic representations that we will use throughout the remainder of this proposal.

part of our preliminary work [57], partly replicated in this section, we showed that simple, local rewrite rules can replicate the organizational behaviors of more complex index structures. These simple rewrite rules do not rely on any global structural properties of the index, allowing policies representing different structures to easily interoperate. As a running example in this section, we will use a simple index JITD that stores records of key/value pairs and supports two operations: Insert and Range-Scan.

Nodes. The structure and semantics of a JITD's physical layout are defined by a grammar of nodes. Our range query JITD uses a grammar of four nodes, shown in Figure 2. The two base nodes: `Array` and `Concat` define the physical structure of the data being encoded by the JITD. `Array` represents a vector of records, while `Concat` represents the composition of two independent collections of records, each recursively defined by a node. `SortedArray` and `STree` extend the structure of `Array` and `Concat` (respectively) with semantic properties: Records in `SortedArray` nodes are sorted, while an `STree` node partitions the records of its child nodes with a separator, like an inner node in a Binary Tree.

The central idea behind JITD optimization is the separation of structural and semantic equivalence: The same collection of records can be expressed by many equivalent expressions in the grammar. Like bytecode in a just-in-time compiler, expression trees are gradually replaced with equivalent, ideally more efficient expressions. These *equivalence rewrites* eventually converge to an idealized representation of the data, as dictated by the policy that the JITD is currently using.

API. A JITD exposes a standard API, regardless of policy. For example, the index JITD exposes two operations: Range-Scan and Insert, which operate on a single, root node. Range-Scan can be generically implemented by a recursive descent through the node structure. The recursive descent terminates by filtering `Array` nodes through a linear scan, or `SortedArray` nodes through a binary search. Insertion combines the new and old data, replacing the root node with a `Concat` of the two. For more details, see [57].

Policy Examples. Calls to the JITD's API trigger changes to its structure. This may occur due to the operation triggering a change to the contents (*e.g.*, an Insert), or may occur as an optimizing side-effect of the current policy. Each policy defines rewrite rules to be applied before or after each API call is performed. Rewrite rules match expressions in the grammar against specific patterns and define procedures for constructing an equivalent and (ideally) more efficient representation. Below, we describe three example policies that emulate common adaptive index structures: Cracker Indexes and Adaptive Merge Trees, as well as hybrid versions.

Policy 1-Cracking: Cracker Indexes [39, 41] are a type of adaptive index that takes an unsorted array of records and gradually sorts it, using range scan queries as advice on which regions of the array need to be sorted earlier. Each range scan partitions, or cracks the array into three parts using the range scan boundaries. This simultaneously produces a result (the middle partition) and brings the data closer to being sorted, making subsequent queries more efficient.

The cracking policy implements the crack operation as a rule that replaces an (unsorted) array with two partitions, connected by a Binary Tree node as shown in Figure 3. In response to a range scan, arrays that could contain one or both range scan boundaries are cracked. As an optimization over traditional cracking, reads also cause newly inserted data to be cracked according to existing index patterns as shown in Figure 4a, and can cause small insertions to be in-lined as shown in Figure 4b (see [57] for more details).

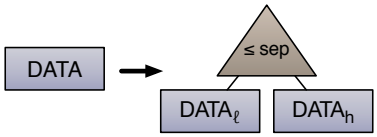


Figure 3: A visualization of the `CrackInTwo` transform. $DATA$ is partitioned by sep into $DATA_\ell$ and $DATA_h$, and a new `STree` node is created linking the two fragments.

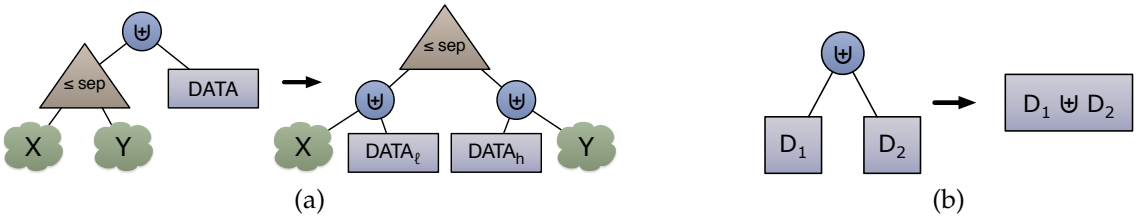


Figure 4: (a) A visualization of the `Pushdown` transform. $DATA$ is partitioned by sep into $DATA_\ell$ and $DATA_h$, and pushed into the `STree` node. The subtrees X and Y remain unmodified. (b) The `MergeArrays` transform copies a concatenation of two arrays into a single contiguous memory region.

Note that unlike updates in traditional Cracker Indexes [43], the cracker policy does not mandate that all data ultimately reside in a single contiguous region of memory.

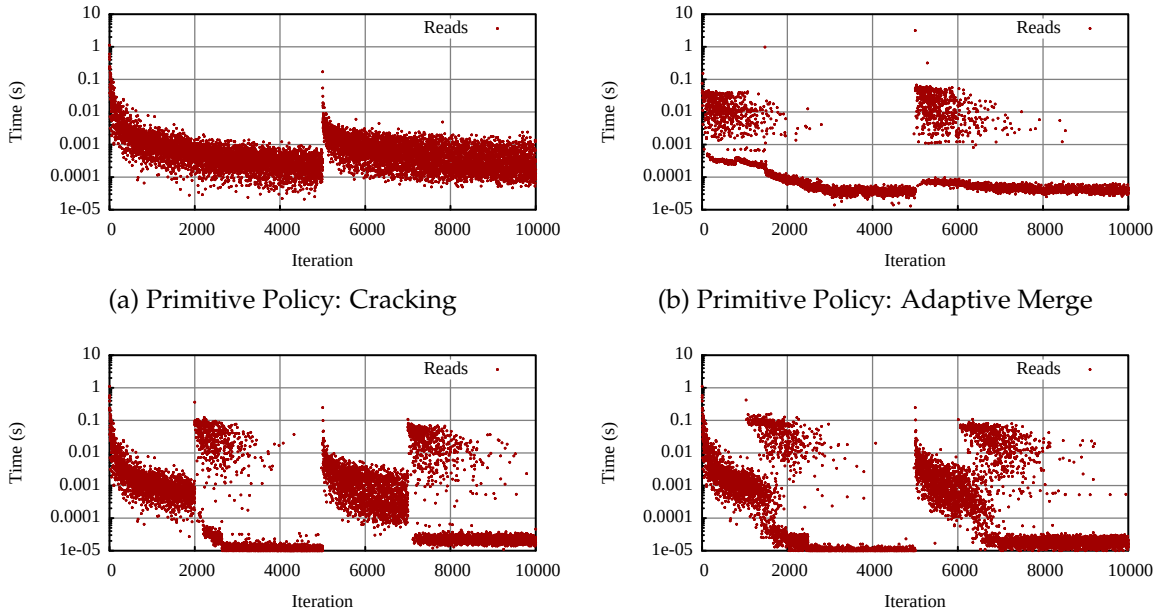
Policy 2-Adaptive Merge: Adaptive Merge Trees (AMTs) [36] are a second class of adaptive index. An AMT initially begins as a collection of sorted runs, each storing one partition of the data. Records are gradually migrated from each input partition and merged into a sorted output partition. Range query responses are taken exclusively from the output partition. Although adaptive merge trees require an expensive upfront sort operation, records are only migrated once and AMTs converge to the behavior of an upfront index far faster than cracker indexes. AMTs are implemented as a pair of rewrites: `CreateRuns` replaces large, unsorted arrays with a collection of fixed-size sorted partitions connected by `Concat` nodes. The `AMerge` rewrite is applied to a concatenation of sorted runs, and migrates records in a given key-range into the (arbitrarily selected) left-most partition. `CreateRuns` is triggered on inserts, while `AMerge` is triggered in response to a range scan.

Policy 3-Hybrid: Hybridizing the Cracking and AMT policies requires only relatively minor changes to make each policy of grammar nodes created by the other policy. For example, the Cracking Policy was changed to ignore already sorted data. We created two hybrid policies: **Swap** and **Transform**, motivated by Cracking being better in the short term, and AMTs being better in the long run. **Swap** uses the Cracking policy for the first 2,000 scans after a write, and then switches to the AMT policy. **Transform** uses the Cracking policy for the first 1,000 scans after a write, and then gradually transitions to the AMT policy, linearly increasing the fraction of requests satisfied with AMT over the following 2,000 scans. Both of these policies required less than 50 lines of java code — almost all of which was standard Java boilerplate. As shown in our preliminary experiments below, both policies exhibited performance characteristics in-between those of Cracker Indexes and AMTs.

4 — Preliminary Evaluation

Our initial evaluation of the JITD model was targeted at the policies we have just described: one emulating Cracker Indexes, one emulating AMTs, and two hybrid policies. Used alone, the simple policies *retain performance competitive with the original data structures*, and can be combined into hybrid policies that demonstrate intermediate performance characteristics.

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



(a) Primitive Policy: Cracking

(b) Primitive Policy: Adaptive Merge

(c) Hybrid Policy: Swap

(d) Hybrid Policy: Transition

Figure 5: 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.

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[1].

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 [44]. 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
```

The `low` key 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, the structure's root was replaced by a concatenation of the original root and a new unsorted array node containing the new data.

Primitive JITD Policies. Figures 5.a and 5.b show the raw performance of the **cracking** and **adaptive merge** policies. 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.* [46].

Hybrid Policies. **Swap** and **transition** implement the two different hybrid policies described in Section 3. Performance results for the swap and transition hybrid policies are presented in Figure 5.c,d. By pre-cracking the array, AMT can construct sorted runs gradually and on smaller arrays. Consequently, the transition between Cracking and AMT is graceful, and the resulting data structure actually performs *better*

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

	Unorganized	Sorted	Hashed
Partitioned	N-ary Concat	Sort-Tree	Hash-Tree
Clustered	N-ary Array	N-ary SortedArray	

$$
\begin{aligned}
N \;:=\; & \texttt{Concat}\,(N, N, \ldots) \\
\mid\; & \texttt{STree}_f\,(N, sep, N, sep, \ldots) \\
\mid\; & \texttt{HTree}_h\,(N, \ldots, N) \\
\mid\; & \texttt{Arr}(R, \ldots, R) \\
\mid\; & \texttt{SArr}_f(R, \ldots R)
\end{aligned}
$$

Figure 6: The index grammar N. R denotes individual records. *sep* denotes separator values. f, h denote sort orders and hash functions, respectively.

than either pure merge or pure cracking in the mid-range. Also note the tri-modal distribution of latencies in the transition policy. The same data structure is able to alternate between the performance characteristics of a Cracker Index, and an AMT on a *per-request basis*.

5 — Challenges and Proposed Research

5.1 — Thrust 1: An Index Grammar

Our first thrust explores the conceptual framework of JITDs. We will generalize the index JITD grammar to capture a broader range of organizational semantics. Second, we will precisely characterize the policy design space. Finally, we will explore techniques for automating the design of adaptive policies. The result of this thrust will be a framework for defining general indexes that dynamically pivot between workloads.

Task 1: Generalizing the Structural Grammar. The index JITD's grammar describes the physical structure of an index *at one point in time*. We begin by extending our preliminary index grammar with standard [32, 72] definitions of data layout properties such as Organization (unorganized, a sort order, or a hash function), Clustering (sorted or partitioned), and Layout (row-wise, column-wise, or hybrid). The choice of organization dictates which filtering predicates the index can respond to efficiently. Clustered data layouts provide better locality and faster scans, but sacrifice update efficiency. Finally, layout decisions can improve the performance of accessing specific vertical or horizontal partitions of the data.

We focus initially on the Organization and Clustering properties, and will return to Layout in Thrust 3. Figure 6 summarizes the design space for these two properties and generalizes our original grammar from Figure 2. This generalized grammar differs from the original in three ways. (1) The original index grammar used binary concatenation and partition nodes, while the new grammar instead defines all of the partition nodes as *families* of n-ary nodes. This generalization allows us to encode index structures like B+Trees, which inline several partition operators into a single tree node. (2) We add a new family of `HTree` nodes that act as hash tables over their descendent records. (3) `STree` and `SArr` nodes are explicitly parameterized with a sort order f. This allows a single family of nodes to express multi-dimensional index structures like R+Trees. `HTree` nodes are similarly parameterized by an explicit hash function h. As a beneficial side effect, these functions also allow the grammar to remain agnostic to the physical structure and attributes of the individual records.

We will evaluate the grammar's generality through a survey of in-memory indexes. For each index, we will identify a restricted form of the grammar that is both sufficient and necessary to describe the index. For example, by limiting the grammar to 2-ary `STree` and 1-ary `Arr` nodes, we obtain classical binary trees:

$$N := \texttt{STree}_f\,(N, sep, N) \mid \texttt{Arr}(R)$$

In addition to restricting the grammar itself, we will also explore conditions over, or properties of the resulting structures that need to be preserved. For example, a balanced binary tree can be expressed as a binary tree where all paths from root to leaf have equal lengths.

Although the grammar may not be able to directly encode the precise physical layout of the data, it can encode the semantics of the layout. For example, a 3-ary B-Tree could be described as a reduction of the general index grammar to the following form:

$$N := \texttt{STree}_f\left(N, sep, \texttt{SArr}_f(R), sep, N, sep, \texttt{SArr}_f(R), sep, N\right) \mid \texttt{SArr}_f()$$

Rule	Effect
Split	Crack, Hash-Partition, or Subdivide an Array or SortedArray
Merge	Merge the children of a Concat, BinTree or HTree node as in Figure 4
Zig, ZigZig, ZigZag	Pivot a BinTree or Concat to change its balance in the manner of Splay Trees [81].
Downgrade	Replace a BinTree or HTree node with a Concat, or a SortedArray with an Array
Pushdown	Push a Concat through a BinTree or HashTree as in Figure 4
Widen	Merge 2-levels of Concat, BinTree, or HTree nodes into a single wider node.
Deepen	Split a wide Concat, BinTree, or HTree node into 2-levels of narrower nodes.

$$\begin{aligned}
\textbf{Split}(f, k_1, \ldots, k_M) \quad &:= \quad \texttt{Arr}(\vec{R}) \rightarrow \texttt{STree}_f \left(\texttt{Arr}(\{r \in \vec{R} \mid r <_f k_1\}), k_1, \ldots, k_M, \texttt{Arr}(\{r \in \vec{R} \mid k_M \leq_f r\})\right) \\
&\mid \quad \texttt{SArr}_f(\vec{R}) \rightarrow \texttt{STree}_f \left(\texttt{SArr}_f(\{r \in \vec{R} \mid r <_f k_1\}), k_1, \ldots, k_M, \texttt{SArr}_f(\{r \in \vec{R} \mid k_K \leq_f r\})\right) \\
\textbf{ZigZag}() \quad &:= \quad \texttt{STree}_f \left(\texttt{STree}_f \left(A, k_1, \texttt{STree}_f \left(B, k_2, C\right)\right), k_3, D\right) \\
&\quad \rightarrow \texttt{STree}_f \left(\texttt{STree}_f \left(A, k_1, B\right), k_2, \texttt{STree}_f \left(C, k_3, D\right)\right)
\end{aligned}$$

Figure 7: Preliminary equivalence transforms in the index grammar, and examples of how two of these rules would be implemented in a simple pattern-matching language.

Here, single-node arrays are in-lined into alternating positions in the STree node to model the B-Tree's use of internal tree nodes for data storage. Although this captures the *semantics* of the physical layout, this representation contains several redundant elements: (1) The SArr node is a physical part of the STree node, and (2) The separator values in the STree node are all unnecessary.

Redundancy elimination forms the basis for *structural synthesis*. Optimized data representations (*e.g.,* $\texttt{STree}_f\ (N, R, N, R, N)$) can be defined by using the grammar to model their semantics, in turn allowing us to use the five primitive nodes defined above as a basis for analysis and policy exploration. Time permitting, we will explore techniques for automating structural synthesis through inlining, constraint propagation, and other forms of redundancy elimination. If necessary, nodes can still be synthesized manually.

Task 1 Goal: We will develop an index grammar and demonstrate its generality by using it to encode standard in-memory index structures. *The primary anticipated outcome of this task is the grammar and codings of existing index structures. A secondary, stretch outcome is a technique for automating structural synthesis.*

Task 2: Expressing Policies. Any expression in the structural grammar from task 1 defines a possible JITD state. In task 2, we formalize how this state evolves over time. Concretely, we make precise the idea of a policy that dictates *how* the JITD changes, *what* parts of it change, and *when* they should change. The primary goal of task 2 will be to produce a policy specification language and to outline a preliminary, bounded space of possible policies. Bounding the policy design space will allow the use of naive search (*e.g.,* brute force enumeration) as a policy optimization tool, a practice often used in classical database tuning advisors [18], hardware-specialized databases [58], and optimizers for long-running queries [61].

The lowest level primitive of a policy is a rewrite function $\rho : N \rightarrow N$ that takes a node as input and constructs a new, *equivalent* node to replace it. Figure 7 overviews rewrites that appear in common data structures, and illustrates how two of these rewrites can be specified using a simple pattern matching language. **Split** implements the well-known crack-in-two operation [41], and **ZigZag** is analogous to the self- balancing operation used in splay trees [81]. **Split** illustrates that rewrites can define families of rewrite rules by using parameters — in this case, the pivot value to crack on. The rewrite functions in Figure 7 will be the basis for our naive policy search strategy.

These *primitive* rewrites all operate at the granularity of individual nodes. To allow them to affect an entire JITD, we generalize primitive rewrites with a form of structural recursion. A selector function $\sigma : \rho \rightarrow N \rightarrow$

N recursively applies a primitive rewrite ρ. For example, a post-order traversal is defined as:

$$\begin{aligned}
\textbf{PostOrder}(\rho) \quad := \quad & \texttt{Arr}(\vec{R}) \to \rho(\texttt{Arr}(\vec{R})) \quad | \quad \texttt{SArr}_f(\vec{R}) \to \rho(\texttt{SArr}_f(\vec{R})) \\
| \quad & \texttt{Concat}\,(N_1, \ldots, N_M) \to \rho(\texttt{Concat}\,(\textbf{PostOrder}(\rho)(N_1), \ldots, \textbf{PostOrder}(\rho)(N_M))) \\
| \quad & \texttt{STree}_f\,(N_1, \ldots, N_M) \to \rho(\texttt{STree}_f\,(\textbf{PostOrder}(\rho)(N_1), \ldots, \textbf{PostOrder}(\rho)(N_M))) \\
| \quad & \texttt{HTree}_h\,(N_1, \ldots, N_M) \to \rho(\texttt{HTree}_h\,(\textbf{PostOrder}(\rho)(N_1), \ldots, \textbf{PostOrder}(\rho)(N_M)))
\end{aligned}$$

We refer to the rewrite defined by applying a selector to a rewrite $\sigma(\rho)$ as a complex rewrite. Our naive policy search strategy considers two basic types of selector: a total tree traversal (in-, pre-, or post-order), and recursion along a path from the root to one specific leaf record (pre- or post- order). In the latter case `STree` and `HTree` nodes are used to avoid rewriting subtrees that could not possibly contain the record. Note that, like primitive rewrites, selectors may also be parameterized — the key or range targeted by a path recursion for example.

The final remaining piece is to define when a given rewrite should be applied. A *trigger* $\langle E, \rho \rangle$ is defined by an event E and a rewrite ρ. Our initial approach defines one event both before and after the index is accessed or modified, and an *idle tick* event. In addition to supporting range scans and inserts, we plan to generalize the API to also support access patterns like skip-list iterators [88], get/put, and multi-dimensional lookups. An event also provides a context of values that can be used to parameterize rewrites and selectors. For example, the two events triggered before and after a range scan provide a start key, an end key, and a key range. As a consequence, the number of possible ways to parametrize rewrites and selectors is limited to the parameters provided by the event.

A policy then, is a set of triggers $\{\langle E, \rho \rangle\}$. The naive policy exploration strategy will focus on triggers of the form $\langle E, \sigma(\rho) \rangle$, with the set of primitive rewrites ρ, selectors σ, and events E discussed above. Further limiting ourselves to specific members of node families (e.g., 2-ary and k-ary nodes for a fixed k) makes the search space finite. The number of possible triggers is large, but we believe that enumerating this search space is feasible. For brute force exploration of the policy design space, however we will turn to off-the shelf optimization techniques like evolutionary programming [9] or gradient descent. To evaluate the performance of a given policy, we will run the index structure on the test workload for a short period of time. It may also be beneficial to develop a cost-model to accelerate adaptation.

Task 2 Goal: We will develop a policy specification language using equivalence rules on the index grammar and structural recursion primitives. *The primary anticipated outcome of this task is a language for policy specification. As a secondary outcome, we expect to be able to produce a tool for brute-force exploration of a constrained policy design space that will serve as a starting point for Task 3.*

Task 3: Principled Policy Exploration. Brute force exploration of possible policies will provide a mechanism to ground-truth our ideas and explore initial policies. In this task we explore policy derivation from structural and temporal metadata stored within the JITD structure for a *given* workload. In much the same way as trace-based JIT compilation leverages information about a frequent execution path to make compilation and optimization choices, policy decisions (i.e. optimization of structure) can be made based on observed traversals of the JITD. As a running example, consider if profiling data was encoded in the JITD's nodes. We would like to express a policy where frequently read keys are stored at the "top" of the index JITD. We could achieve such a policy by keeping track of a simple read count for each node. Abstractly, nodes with high read counts should be located close to the root of the index JITD, thereby reducing the cost of future read operations. Such re-organization is possible, for instance by splaying.

There are two problems, however, with our example policy: (1) the index JITD is organized by keys and not by read count and (2) read counts over the workload's duration do not capture *temporal* properties of workload itself. We can solve problem (1) by observing that the JITD does not need to be organized by read count, as it is enough for the nodes with high read counts to be toward the top of the tree as opposed to located at the precise root. Problem (2) can be addressed by *discarding* old read counts by means of decay. For instance, a timestamp can be stored at each node signifying the time of the last read or update to that node. The next operation on that node can compute the delta in time and apply the appropriate decay rate.

Conceptually the decay rate would correspond to the read or write rate of the workload, which itself can change.

Based on this example we can see that policies require reasoning about runtime properties of the index JITD itself. We propose to study salient runtime properties and their respective encodings within nodes. This will allow us to define events based on runtime properties of the index JITD. In this case the event is defined by the relative read count with respect to the root of the JITD index. In this way, the event not only captures a runtime property, but also a *structural* property of the index JITD.

Policy adaptability is critical for index JITDs to be able to handle changing workloads. While *heuristics* can provide a degree of adaptability, for instance by applying a reorganization of nodes after a certain number of updates (*e.g.* cracking), they do not make decisions based on *changes* to the workload itself. While the first part of this task explores what runtime and structural properties to encode as events, the second part of this task will explore adaptability. Fundamentally, adaptability is a temporal property, embodying change over time. As such, we must capture changes to the nodes over time as well as the runtime and structural properties associated with those nodes.

Task 3 Goal: We will develop policies based on usage characteristics of the JITD index structure. To provide adaptability we will explore mechanisms of information decay, providing temporal reasoning capabilities into policy development. *The primary expected outcome is the ability to define events based on runtime structural and temporal properties, and policies based on these events.*

5.2 — Thrust 2: Implementing an Index JITD

Our second thrust addresses systems challenges that arise when index JITDs are deployed into practice. Our initial efforts focus on two specific challenges: automation and concurrency. First, we will build an index JITD compiler that will automate code generation and node synthesis, enable program analysis on policy specifications, and serve as a foundation for our evaluation efforts. Second, we will explore techniques for enabling concurrent parallel access to a JITD index.

Task 1: Build an index JITD compiler. Transitions between structures are challenging to implement correctly. Semantic properties of both structures must be known, and any rewrite that transforms one structure into another must adhere to these properties. This is not limited to the physical layout (structure) of each node, but also the logical behavior (semantics) of each node. Larger rewrites, especially those that deal with multiple nodes, are significantly more complex and more difficult to implement. As such, our first task for this thrust will be to investigate mechanisms for reasoning about and optimizing the implementation of index nodes, as well as of policies.

The initial compiler will take as input nodes and policies defined using the index grammar and definition language already discussed in Thrust 1, and produce a data structure definition in a target language such as C++ [2]. As already noted, the source primitives are kept intentionally simple, and as a consequence direct translation will likely produce suboptimal data structure definitions. The initial challenge will be to automatically synthesize new complex nodes and policy rules from the simpler primitives explored in Thrust 1. For optimizing policy specifications especially, we will draw heavily on existing work on declarative compiler definition and optimization languages [49, 47, 93, 95].

As a secondary goal for this task, we will exploit the declarative policy specification language to ensure correctness of the resulting structures; Time permitting, we will create an automated correctness checker for the language. Here, correctness is defined in terms of the semantic properties specified for nodes such as the left-hand-side of a `STree` node containing only records preceding the separator value. A rewrite is considered correct if the set of structures it takes as an input can be rewritten into a structure or set of structures defined by the specification without violating any semantic constraints. To do this we will need to translate the specification into a constraint system. We will investigate encoding the constraints into an SMT solver to automate the process. The specification language will need to be amenable for constraint extraction as well as determining complex relationships between constraints based on the structures the nodes are composed into.

[2] We will also explore managed languages like Java and tackle issues related to the GC.

Task 1 Goal: We will create a source-to-source JITD compiler that compiles policy specifications into efficient index structure implementations and potentially proofs of correctness as well. *The expected outcome of this task is a compiler that will enable experimental evaluation of Thrust 1, as well as further efforts in Thrust 2.*

Task 2: Add concurrency. JITDs are particularly vulnerable to parallel workloads, as the physical structure of a JITD changes frequently, even in response to reads. Consequently, even read-only workloads have the potential to create concurrency bottlenecks. Fortunately, the nature of these changes presents us with a unique opportunity that we plan to exploit. A precursor to our approach arises in the Bloom system [7, 24], which exploits the ease of concurrency in so-called *monotonic* programs. Loosely put, once a monotonic program asserts a fact to its peers (*e.g.*, $x > 4$), the fact can never be retracted. Consequently, peer computations that rely on the assertion (*e.g.*, `if(x > 5){...}`) do not require any concurrency primitives.

We will generalize the idea of consistency through monotonicity to *weakly* monotonic programs. A weakly monotonic program may not retract assertions, but may replace them with *equivalent* facts. If program correctness relies on the equivalence class rather than the specific fact, synchronization is still not required. In the context of JITDs, this approach generalizes functional data structures, which are monotonic, to a weakly monotonic counterpart that we call *semi-functional data structures*. A node may be rewritten into any other equivalent node without requiring synchronization.

Example 3 *Consider a functional form of the Crack-in-Two operation (Figure 3) that creates a copy of the data rather than being performed in place. As illustrated in the figure, the original array is replaced by the fragmented copies. In principle at least, this operation does not require any synchronization. If two threads happen to simultaneously crack an array, pointers to the original can be safely be redirected to either cracked copy. Although the efforts of one thread may be wasted, the index never becomes inconsistent.*

In practice, there are two problems that arise when concurrency is simply ignored during equivalence rewrites. First, synchronization may not be required, but atomicity is. Second, ignoring concurrency creates memory leaks. To address the first problem, we create a distinction between code that requires only semantic equivalence (logical stability), and code that relies on the physical structure staying constant (physical stability). This distinction is maintained through an added level of indirection in pointers. A *handle*[3] is a pointer to a node. By holding a pointer to a handle, the programmer can expect logical stability. For physical stability, the programmer de-references the handle and obtains a pointer to the node itself.

The second challenge is memory management. In our preliminary trials, simple memory management techniques like per-node and per-handle reference counters perform quite poorly. This is because the reference counters themselves introduce points of contention for each node access. Our initial solution will be based on garbage collection techniques like mark-and-sweep.

Task 2 Goal: We will extend the compiler to automatically generate index JITDs with support for concurrency. *The primary anticipated outcomes of this task is a mechanism for automatically generating concurrent index JITDs and a formalization of semi-functional data structures. As a secondary outcome, we will extend the JITD runtime with support for both reference counting and garbage collection.*

5.3 — Thrust 3: View Maintenance

The index JITD's data model focuses purely on data layout and organization. We observe that the nodes themselves, together with equivalence rules on the grammar define a simple algebra over sets of records: `Array` is a leaf, while `Concat` is a bag union. There are three immediate benefits to looking at a data structure through the lens of algebraic rewrites: (1) Decades of research on set, bag, and other collection algebras (*e.g.*, [63, 74, 76, 23, 82, 22, 59, 87]) have already identified many equivalences that can be translated into organizational structures, (2) Metadata annotating the algebra's parse tree (*e.g.*, partitioning, bounds, or sort orders) can decouple the physical layout of the data structure from semantic constraints on the structure (*e.g.*, creating an `STree` node from a `Concat`). This decoupling makes it easier to defer and parallelize organizational tasks like sorting or index construction, (3) Treating building blocks as algebraic operators allows us to think of the data structure as a partially evaluated program, and encode or materialize deferred computation of expensive operators in more expressive algebras (*e.g.*, filter, project, aggregate).

[3]The name originates from early versions of the MacOS and Windows, which used handles for memory management

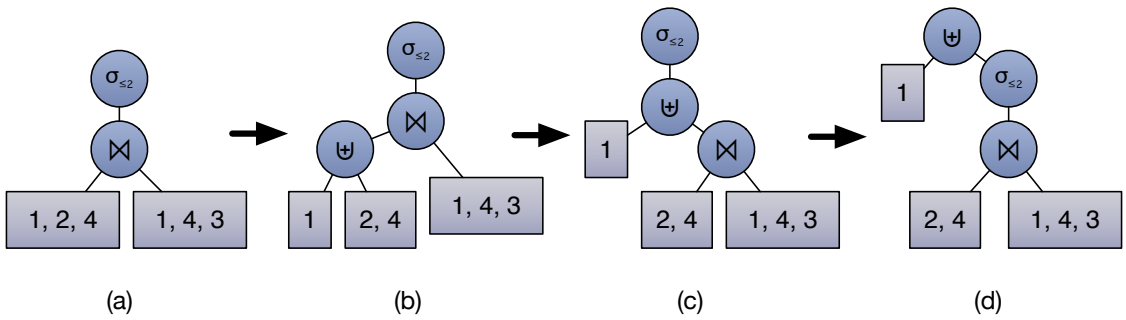


Figure 8: Join evaluation through equivalences is analogous to evaluating the join expression by hand. Any intermediate state in the computation can be materialized as a parse tree of relational operators.

The ability to defer and decompose expensive rewrites is especially intriguing. Deferred or lazy computation smoothes out bursty workloads, enables batching, and allows for fine-grained control over when and where latency overheads arise. It has been used extensively for data-structure design [66], transaction processing systems [30], and even for asymmetric query processing [8]. The JITD model naturally captures the semantics of deferred computation in a data structure that also supports indexing and partial evaluation.

As a means of exploring the generality of the JITD model, Thrust 3 targets adaptively maintaining materialized views, a generalization of incremental view maintenance (IVM). IVM has received considerable attention recently, including the use of memoization [60, 61, 6], partial materialization [98, 99], compilation [37, 89, 61, 6], or as a proxy for data layout decisions [28]. Driving this innovation is a need to monitor or cache queries over long lived data in domains like stock market trading [52], for complex bulk computations like graphical inference [92], or simply an increasing need for realtime updates to data warehouses [52].

There are three primary tradeoffs in deciding whether to incrementally maintain a view: (1) How much space is necessary to maintain the view? (2) What is the computational cost of maintaining the view? (3) How much benefit do we derive from the view? Prior work has explored ways of making these tradeoffs at finer granularities. For example, partially materialized views [98, 99] allow for views where only a subset of the rows are materialized. Requests for hot data in the view can be satisfied quickly, without the space and time overheads of maintaining the full view. We see JITDs as the next logical step in the evolution of incremental view maintenance. A JITD with nodes for the full set of relational operators, rather than just union, can materialize intermediate states at any point in query evaluation.

Example 4 *Consider the query:* $\sigma_{A\geq 2}(R(A) \bowtie_A S(A))$*, where both* $R(A)$ *and* $S(A)$ *are relations with a single column named A. Figure 8 illustrates how equivalences can be used to incrementally evaluate the expression. This step-by-step process is painfully inefficient if each step is explicitly materialized. However, any step in the process can be materialized, regardless of the granularity at which the join is evaluated.*

We subdivide the realization of adaptive view maintenance through JITDs into two high-level tasks. As a preliminary step, we will extend our simple index JITD to also support simple set-based query evaluation, adaptively materializing a static view as its contents are queried. From this starting point, we will then generalize our approach to support incremental changes to the input data.

Task 1: Static Adaptively Materialized Views. The use of equivalences for query evaluation can be thought of as a generalization of the Eddy operator [8], or as an adaptive index [46] for materialized views. In other words, the JITD can adapt to queries over the view, by progressively materializing different parts of the view as they are queried. Relevant fragments of the view query — identified through selection pushdown [98, 99] or missing value algebras [62] — are evaluated and memoized, while irrelevant fragments are ignored. The query over the view is answered without unnecessary work, while the work that is performed is saved for use with later queries.

The primary goal of this task is to identify the highest level of abstraction that can be obtained without a significant loss of performance. Clearly, materializing every single step in the join process as shown in Figure 8 is inefficient. However, individual rewrites can be composed together in different ways: (1) Sequences of rewrites can be inlined (*e.g.,* transitioning from (a) to (d) in a single step), or (2) Parallel rewrites can be batched (*e.g.,* repeating the process for the remaining data values on the left — 2 and 4).

An interesting side-effect of adding joins into the index grammar is that it allows the grammar to naturally express both column-wise [26] and hybrid row/column-wise [73] data layouts. We will use this observation as a starting point, as a way to further generalize our preliminary work implementing the index grammar.

Task 1 Goal: We will evaluate incremental view materialization in JITDs by extending the index grammar to capture the SPJUD fragment of relational algebra, and extend the rewrite grammar to capture incremental evaluation rules. As a benchmark for this task, we will compare against a mix of commercial and publicly available [52, 61, 6, 89] state-of-the-art view maintenance systems. *The anticipated outcome of this task is the adaptive static view maintenance JITD itself.*

Task 2: Adaptive IVM. Our second task generalizes adaptive view maintenance from static to dynamic data. As a preliminary approach, we will use a new node type δR that serves as a placeholder for values that have not yet been inserted into relation R. This node will act as an empty set with respect to queries, allowing insertions to be performed by replacement. We also plan to support negative multiplicities in multi-sets [60], making deletion just insertion with a negative multiplicity.

This new δ node seemingly violates the semi-functional behavior of JITDs, breaking our existing data model in two ways. First, several nodes in the index grammar make static assumptions about their children. For example, the node $\texttt{STree}_f\,(A, k, B)$ assumes that all records in A fall below k (i.e., $R <_f k$ for all $R \in A$). Since δR represents "incomplete" data, these assumptions must still be enforced. Second, the assumptions used to enforce concurrency are also violated. δR cannot be changed if any thread is accessing the JITD.

Our preliminary solution to both problems will be to define a new grammar node that we call an update memo. When an update occurs to relation R, the root of the tree is atomically replaced with a new node of the form $\texttt{Memo}(R, \mathit{OldRoot}, \mathit{Update})$, where $\mathit{OldRoot}$ is the previous root, and Update is a tree that stores the new records in R. The memo operation is pushed down through the nodes of $\mathit{OldRoot}$, accumulating syntactic properties along the way in the Update subtree. For example:

$$\texttt{Memo}(R, \texttt{STree}_f\,(A, B)\,, U) \rightarrow \texttt{STree}_f\left(\texttt{Memo}(R, A, \sigma_{r<f}(U)), \texttt{Memo}(R, B, \sigma_{r\geq f}(U))\right)$$

The memo is pushed down into the $\texttt{STree}\,()$ node, ensuring that constraints on the Update U are enforced. Once it reaches a leaf record, one of two things happens. If the relation R annotating the memo is the same as the relation annotating the δ, the memo turns into a concatenation: $\texttt{Memo}(R, \delta R, U) \rightarrow \texttt{Concat}\,(\delta R, U)$. For any other leaf: $\texttt{Arr}()$, $\texttt{SArr}()$, or a δS where $S \neq R$, the memo is discarded: $\texttt{Memo}(R, \delta S, U) \rightarrow \delta S$. This approach solves both problems: Constraints are gathered and enforced as the memo is pushed down, and because the memo itself is applied atomically, the rewrites necessary to push the memo down preserve the JITD's semi-functional nature.

Task 2 Goal: We will generalize the adaptive view data structure created by task 1 to support incremental view maintenance. We are interested to see if this structure can retain competitive performance, while still being able to adapt to changing workloads. As before, we will compare against state of the art IVM systems. *The anticipated outcome of this task is a proof of correctness, and an experimental evaluation of policies for adaptive incremental view maintenance.*

6 — Research and Evaluation Plan

The specific goals of this proposal are to: (1) to define a universal grammar for expressing physical layouts of indexes, (2) to define a language for transforming these physical structures, (3) to automatically synthesize transformation policies, (4) to enable support for concurrent index JITDs, (5) to automate efficient instrumentation of index JITDs, and (6) to generalize the JITD model to new classes of workload.

We have budgeted funds for a student in years 1 to 3, who will focus on formalizing the index grammar and transformation language, on representing existing index structures in this language, and on prototyping

the JITD compiler. During year 2, the student also begin exploring automated policy design. Year 3 is allocated for exploring concurrency in JITDs and as a safety buffer. We have also budgeted funds for a second student to start midway through year 2. This student will begin by assisting the first student, and transition to adding support for IVM as detailed in Thrust 3. Finally, UB has a large Masters program, with numerous students interested in completing a Masters Thesis or stand-alone Masters Project. We anticipate many possibilities for both, including assisting with evaluation, developing new policy designs, and implementing or testing parts of the compiler. We expect to be able to produce one or two masters theses as a result of the proposed work.

6.1 — Benchmarks and Metrics

Our choice of benchmark is driven by the performance of JITDs on time-varying workloads. As a starting point, we plan to use the Yahoo! Cloud Services Benchmark (YCSB) [25], which evaluates the performance of key-value stores. Workloads in YCSB are defined by a set of parameters that can be varied over time, allowing us to test the adaptability of our JITD structures. We will also make use of existing benchmarks designed for adaptive indexes [35], as well as a benchmark for smartphone data management currently being developed by the PIs [53]. As we transition to work on the adaptive view JITD, we will begin adapting industry-standard benchmarks like TPC-C [85] and TPC-H [86].

Although throughput is an interesting metric, our primary interest is in measuring per-query latency and resource consumption (*i.e.,* memory, CPU and power). A key distinguishing feature of our benchmarking process is that we will measure performance at below maximum saturation. In addition to representing a far more realistic assessment of index structures than performance at saturation, we expect moderate-to-high throughput to be a regime where JITDs truly excel. As comparison points for the index JITD, we will use library implementations of common data structures (e.g., BTrees, Red-Black Trees, Arrays, etc…) and implementations of well-known adaptive indexes [46]. As we move to IVM, we will begin to compare against commercial database systems, as well as publicly available databases like MySQL [64] or Postgres [84], and view maintenance systems like DBToaster [52, 6, 61]. As we turn to concurrent implementations, we will also compare against NoSQL data storage systems like BerkeleyDB [67], Redis [75] and Memcached [31].

6.2 — Risk Analysis and Mitigation

The biggest risk for this research is that the overhead of using rewrites will be too large to allow JITDs to be competitive with existing forms of index or view maintenance. We have shown that JITDs can be feasible through our preliminary work, and believe that there will be some tradeoff between flexibility and performance. Our efforts in Thrusts 1 and 2 include a search for this balance point. If it should turn out that achieving our desired level of runtime flexibility is not feasible, we will refocus our efforts towards the use of aggressive compilation and grammar restrictions to achieve even more runtime specialization. We could then explore ways to dynamically adapt this specialization over time.

7 — Curriculum Development Activities

PIs Kennedy and Ziarek have both been assistant professors at the University at Buffalo, SUNY for three and a half years. PI Kennedy regularly teaches a seminar on current topics in data management research, as well as UB's introductory graduate-level database course. PI Ziarek regularly teaches a seminar on current topics in compilers and programming languages, and has taught graduate courses on programming languages, including fundamentals of programming languages and advanced programming languages. The PIs are already using aspects of the proposed research as part of their course curriculum. Together, they have developed an advanced graduate-level course entitled "Languages and Runtimes for Database Systems" that uses JITDs as a pedagogical tool for teaching database, query optimization, compiler, and program analysis concepts. The course was first taught in the Fall of 2015, and is built around groups working on term-long individual research or development projects drawn from PI Kennedy and Ziarek's research efforts, including, among others, JITDs.

PIs Kennedy and Ziarek are both active members of the Western New York chapter of the ACM Computer Science Teacher's association (CSTA), of which PI Kennedy was Chapter Secretary from 2013-2015. In this role, both PIs have led multiple workshop tutorials on computer science education at the high school level through scripting languages like Python. PI Kennedy is extremely active in STEM educational outreach, volunteering with local organizations including the LIBERTY Partnerships (mentorship for middle- and

high- school students from **underprivileged neighborhoods**) and Science is Elementary (science workshops at elementary schools in **primarily-minority neighborhoods**), and helping to organize a monthly database seminar meetup supported by local Buffalo startups. PI Ziarek has published and maintains a **low cost introductory Python e-book** for high school students interested in STEM fields, and has served as a Google Summer of Code mentor for MLton.org. Razie Fathi (female), a student co-advised by PIs Kennedy and Ziarek, is receiving support from the NSF-funded ISEP program at UB to work with K-12 teachers to improve computer science education in Buffalo public schools. PIs Kennedy and Ziarek both consider equalizing gender ratios in computer science to be a top priority — 3 out of 7 PhD-level students presently being advised by PI Kennedy are female. Finally, the proposed budget includes support for two PhD-level students, who will take an active role in the proposed research. Additionally, if this proposal is funded, the PIs intend to pursue support for undergraduate researchers via the NSF's REU program.

8 — Broader Impacts of the Proposed Work

This project, if successful, will result in foundational research on Just-in-Time Data Structures, adaptive view maintenance, and the use of compiler tools and technology for data layout and data structure design. We believe that JITDs represent a significant shift in the state-of-the-art in data management, and will provoke new research in numerous directions. The work done will lead to the release of JITDs for static data management and adaptive view maintenance, as well as a JITD compiler. This project will result in the training and education of two Ph.D. students and up to two masters thesis students. The PIs have a track record of including undergraduates in their research programs and expect to mentor undergraduates throughout the grant period on the work being accomplished for the grant. Finally, this project will result in the curriculum development and outreach activities outlined in Section 7.

9 — Results from Prior NSF Support

9.1 — PI Kennedy has been awarded "TWC: Medium: Collaborative: Data is Social: Exploiting Data Relationships to Detect Insider Attacks", starting on 10/01/2014 and ending on 09/30/2018. The award number is CNS-1409551 and the amount is $959,999 with a duration of four years.

Results Related to Intellectual Merit: The project has collected a trace of all queries performed on all database servers at M&T bank for a period of one week. With the help of M&T security staff, this trace was anonymized and cleared for export to UB hardware. The project team is developing a tool for clustering queries by similarity of intent, and hopes to publish its results within two to three months.

Results Related to Broader Impact: The project's use of M&T query data as a test case provides feedback about the reliability of our tool, and ensures that it can be applied to real-world problems.

Evidence of Research Products:: Project Co-PIs Upadhyaya and Chandola presented our preliminary work at a SaTC PI meeting in January 2015. PI Upadhyaya and team member Gokhan Kul presented a preliminary paper on insider attack ontologies at the USENIX MIST workshop.

9.2 — PI Ziarek has been awarded "II-EN: Collaborative Research: Positioning MLton for Next-Generation Programming Languages Research" starting on 07/30/2014 and ending on 07/29/2017. The award number is CNS-1405614 and the amount is $381,640.00 with a duration of three years.

Results Related to Intellectual Merit: The development team has currently merged the Multi-MLton GC structure with that of the main line MLton branch and is currently undergoing extensive testing and performance tuning. The restructured GC will allow MLton users to seamlessly migrate their existing code base to execute on multi-core machines and leverage multiple concurrent and parallel GC configurations. This award has resulted in a publication titled "Adding Real-time Capabilities to a SML Compiler" in DPRTCPS'15.

Results Related to Broader Impact: The development team has produced a new version of the MLton hacker's guide available at: `https://github.com/UBMLtonGroup/HackersGuide`. The substantially updated hacker's guide provides new MLton contributors detailed knowledge on the compiler and runtime structures, focusing on the new additions of the parallel runtime.

Evidence of Research Products: The development team has all source code available in a public git hub repository `https://github.com/UBMLtonGroup`. The changes are slated to be merged into the mainline MLton branch by the end of the year.

REFERENCES

[1] Serge Abiteboul and Victor Vianu. Collaborative data-driven workflows: Think global, act local. In *PODS*, pages 91–102, 2013.

[2] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 309–322, New York, NY, USA, 2008. ACM.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 14–25, New York, NY, USA, 2003. ACM.

[4] Umut A. Acar and Yan Chen. Streaming big data with self-adjusting computation. In *DDFP*, 2013.

[5] Sumit Agarwal, Daniel Bellinger, Oliver Kennedy, Ankur Upadhyay, and Lukasz Ziarek. Monadic logs for collaborative web applications. In *WebDB*. ACM, 2013.

[6] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, June 2012.

[7] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, pages 249–260, 2011.

[8] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *ACM sigmod record*, volume 29, pages 261–272. ACM, 2000.

[9] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Oxford, UK, 1996.

[10] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *SOSP*, pages 325–340, 2013.

[11] D. S. Batory, T. Y. Leung, and T. E. Wise. Implementation concepts for an extensible data model and data language. *ACM TODS*, 13(3):231–262, September 1988.

[12] Phillip A Bernstein, CW Reid, and Sudipto Das. Hyder–A Transactional Record Manager for Shared Flash. *CIDR*, 2011.

[13] Ethan Blanton and Lukasz Ziarek. Non-blocking inter-partition communication with wait-free pair transactions. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 58–67, New York, NY, USA, 2013. ACM.

[14] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, December 2008.

[15] Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, and Ioana Manolescu. Invisible glue: Scalable self-tuning multi-stores. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[16] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Not.*, 46(8):35–46, February 2011.

[17] Surajit Chaudhuri and Vivek Narasayya. AutoAdmin "what-if" index analysis utility. In *SIGMOD*, 1998.

[18] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007.

[19] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *PVLDB*, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[20] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *PVLDB*, pages 1–10, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[21] Yan Chen, Umut A. Acar, and Kanat Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *SIGPLAN*, 2014.

[22] Edgar F Codd. A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 35–68. ACM, 1971.

[23] Edgar F Codd. *Relational completeness of data base sublanguages*. IBM Corporation, 1972.

[24] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *SoCC '12*, pages 1:1–1:14, 2012.

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[26] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, pages 268–279, New York, NY, USA, 1985. ACM.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS OSR*, 41(6):205–220, October 2007.

[28] Jens Dittrich and Alekh Jindal. Towards a one size fits all database architecture. In *CIDR*, 2011.

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

[30] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 15–26, New York, NY, USA, 2014. ACM.

[31] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[32] Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson, 2008.

[33] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, and Stefan Manegold. Concurrency control for adaptive indexing. *PVLDB*, 2012.

[34] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi A. Kuno, Stefan Manegold, and Bernhard Seeger. Transactional support for adaptive indexing. *VLDB J.*, 23(2):303–328, 2014.

[35] Goetz Graefe, Stratos Idreos, Harumi Kuno, and Stefan Manegold. Benchmarking adaptive indexing. In *TPC-TC*, 2011.

[36] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[37] T.J. Green and Z.G. Ives. Recomputing materialized instances after changes to mappings and data. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 330–341, April 2012.

[38] Todd J Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: a tutorial. In *Datalog in Academia and Industry*, pages 1–8. Springer, 2012.

[39] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 2012.

[40] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, pages 11–11, 2010.

[41] Stratos Idreos, Martin L Kersten, and Stefan Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[42] Stratos Idreos, Martin L Kersten, and Stefan Manegold. Updating a cracked database. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 413–424. ACM, 2007.

[43] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[44] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[45] Stratos Idreos, Stefan Manegold, and Goetz Graefe. Adaptive indexing in modern database kernels. In *EDBT*, 2012.

[46] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 2011.

[47] Karl Trygve Kalleberg, Eelco Visser, Adrian Johnstone, and Tony Sloane. Spoofax: An interactive development environment for program transformation with stratego/xt. In *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA07)*, pages 47–50, 2007.

[48] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, number EPFL-CONF-203677, 2015.

[49] Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.

[50] O. Kennedy and C. Koch. PIP: A database system for great and small expectations. In *ICDE*, pages 157–168, 2010.

[51] O. Kennedy, C. Koch, and A. Demers. Dynamic approaches to in-network aggregation. In *ICDE*, pages 1331–1334, 2009.

[52] Oliver Kennedy, Yanif Ahmad, and Christoph Koch. DBToaster: Agile views for a dynamic data management system. In *CIDR*, pages 284–295, 2011.

[53] Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. Pocket Data: The need for TPC-MOBILE. In *TPC Technology Conference on Performance Evaluation & Benchmarking*, 2015.

[54] Oliver Kennedy, Steve Lee, Charles Loboz, Slawek Smyl, and Suman Nath. Fuzzy prophet: Parameter exploration in uncertain enterprise scenarios. In *SIGMOD*, pages 1303–1306, 2011.

[55] Oliver Kennedy and Suman Nath. Jigsaw: Efficient optimization over uncertain enterprise data. In *SIGMOD*, pages 829–840, 2011.

[56] Oliver Kennedy and Lukasz Ziarek. BarQL: Collaborating through change. Technical report, CORR, arXiv:1303.4471, 2013.

[57] Oliver Kennedy and Lukasz Ziarek. Just-in-time data structures. In *The Conference on Innovative Data Systems Research (CIDR)*, 2015.

[58] Yannis Klonatos, Andres Nötzli, Andrej Spielmann, Christoph Koch, and Victor Kuncak. Automatic synthesis of out-of-core algorithms. In *Proceedings of the 2013 international conference on Management of data*, pages 133–144. ACM, 2013.

[59] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM (JACM)*, 29(3):699–717, 1982.

[60] Christoph Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 87–98, New York, NY, USA, 2010. ACM.

[61] Christoph Koch, Yanif Ahmad, Oliver Andrzej Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *VLDBJ*, 2013 (to appear).

[62] Willis Lang, Rimma V. Nehme, Eric Robinson, and Jeffrey F. Naughton. Partial results in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1275–1286, New York, NY, USA, 2014. ACM.

[63] K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *Category Theory and Computer Science*, pages 261–280. Springer, 1997.

[64] AB MySQL. Mysql database server. *Internet WWW page, at URL: http://www. mysql. com (last accessed/1/00)*, 2004.

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

[66] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[67] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[68] Krzysztof Ostrowski and Ken Birman. Storing and accessing live mashup content in the cloud. *SIGOPS Review*, 44(2), April 2010.

[69] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 69–82, New York, NY, USA, 2010. ACM.

[70] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM.

[71] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[72] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000.

[73] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. *The VLDB Journal*, 12(2):89–101, August 2003.

[74] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM TODS*, 13(4):389–417, 1988.

[75] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2009.

[76] H-J Schek and Marc H. Scholl. The relational model with relation-valued attributes. *Information systems*, 11(2):137–147, 1986.

[77] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[78] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 7(2), 2013.

[79] Marc Shapiro and Nuno Preguiça. Designing a commutative replicated data type. Technical Report arXiv:0710.1784, CORR, 2007.

[80] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Eliminating read barriers through procrastination and cleanliness. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 49–60, New York, NY, USA, 2012. ACM.

[81] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[82] John Miles Smith and Philip Yen-Tang Chang. Optimizing the performance of a relational algebra database interface. *Communications of the ACM*, 18(10):568–579, 1975.

[83] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *PVLDB*, pages 1150–1160. VLDB Endowment, 2007.

[84] Michael Stonebraker and Lawrence A Rowe. *The design of Postgres*, volume 15. ACM, 1986.

[85] Transaction Processing Performance Council. Tpc-c benchmark specification. http://www.tpc.org/tpcc/default.asp, 1992.

[86] Transaction Processing Performance Council. Tpc-h benchmark specification. http://www.tpc.org/tpch/default.asp, 2008.

[87] Allen Van Gelder and Rodney W Topor. Safety and translation of relational calculus. *ACM Transactions on Database Systems (TODS)*, 16(2):235–278, 1991.

[88] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[89] Todd L. Veldhuizen. Incremental maintenance for leapfrog triejoin. 03 2013.

[90] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. Logbase: A scalable log-structured database system in the cloud. *VLDBJ*, 5(10):1004–1015, June 2012.

[91] Hannes Voigt, Thomas Kissinger, and Wolfgang Lehner. SMIX: Self- managing indexes for dynamic workloads. In *SSDBM*, 2013.

[92] Michael Wick, Andrew McCallum, and Gerome Miklau. Scalable probabilistic databases with factor graphs and mcmc. *Proc. VLDB Endow.*, 3(1-2):794–804, September 2010.

[93] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14. ACM, 2012.

[94] LIMSOON WONG. Kleisli, a functional query system. *Journal of Functional Programming*, 10:19–56, 1 2000.

[95] Thomas Würthinger. Graal and truffle: modularity and separation of concerns as cornerstones for building a multipurpose runtime. In *Proceedings of the of the 13th international conference on Modularity*, pages 3–4. ACM, 2014.

[96] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steven Y. Ko, and Lukasz Ziarek. Rt-droid: A design for real-time android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 98–107, New York, NY, USA, 2013. ACM.

[97] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. Lenses: An on-demand approach to etl. *Proc. VLDB Endow.*, 8(12):1578–1589, August 2015.

[98] Jingren Zhou, P Larson, and Jonathan Goldstein. Partially materialized views. In *submitted to this conference*, 2005.

[99] Jingren Zhou, P.A. Larson, J. Goldstein, and Luping Ding. Dynamic materialized views. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 526–535, April 2007.

[100] Lukasz Ziarek and Suresh Jagannathan. Lightweight checkpointing for concurrent ml. *J. Funct. Program.*, 20(2):137–173, March 2010.

[101] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. Partial memoization of concurrency and communication. In *SIGPLAN-SIGACT*, pages 161–172, 2009.

[102] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. Composable asynchronous events. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 628–639, New York, NY, USA, 2011. ACM.

[103] Lukasz Ziarek, Stephen Weeks, and Suresh Jagannathan. Flattening tuples in an ssa intermediate representation. *Higher Order Symbol. Comput.*, 21(3):333–358, September 2008.