TreeToaster: Accelerating Compilers Using Incremental View Maintenance

Anonymous Author(s)

Abstract

A well documented clear expansion of the problem: Tree traversal is inefficient to answer queries. Solution:Devise a controlled incremental view maintenance system to answer queries effectively.

CCS Concepts: • **Database systems** → **Query Optimization**; *Query Processing*; Indexing.

Keywords: Abstract Syntax Tress, Incremental View Maintenance

ACM Reference Format:
Anonymous Author(s). 2020. TreeToaster: Accelerating Compilers Using Incremental View Maintenance. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A typical optimizing compiler starts with a program specified as an abstract syntax tree (AST), repeatedly applies a set of rules that rewrite fragments of the AST (optimizations and transformations between intermediate representations), and finally emits an optimized version of the tree. In its simplest form, the optimizer first searches for an AST node eligible for a rewrite rule, applies the rule to optimize the matched node, and repeats until no eligible patterns remain (i.e., when the AST reaches a fixed point). On large codebases like operating systems, databases, or similar systems, AST based optimizations can be painfully slow; The Android Operating System Platform can take hours to compile on a typical desktop. Conversely, in dynamic languages, AST-based optimizations can be preferable to optimizations directly on the bytecode [33], and low-latency optimization (i.e., in a JIT or interpreter) is critical for efficient performance.

A significant part of the cost of optimization comes from searching for AST nodes to rewrite, as this process typically requires repeated traversals of the entire AST for every rule applied. In this paper, we propose TreeToaster, a library for compiler construction that virtually eliminates the cost of finding nodes eligible for a rewrite. The insight behind TreeToaster is that the pattern-matching queries used by compilers to find eligible nodes are just that: queries. In lieu of repeated searches through the AST for eligible nodes, TreeToaster pre-materializes a view for each rewrite rule containing all nodes eligible for the rule, and incrementally maintains this view as the tree is optimized.

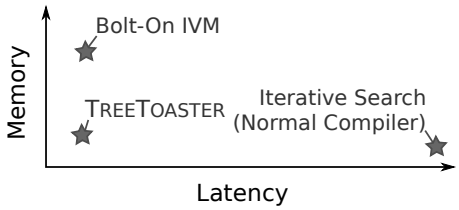


Figure 1. TreeToaster achieves AST pattern-matching performance competitive with "bolting-on" an embedded IVM system, but with negligible memory overhead.

Naively, we might implement this incremental maintenance scheme by simply reducing the compiler's pattern matching logic to a standard relational (for code generation or transforms between intermediate representations) or graph query language (for optimizations and static analysis), and "bolting on" a standard database view maintenance system. As we show in this paper, TreeToaster improves on this approach by leveraging the fact that both ASTs and pattern queries are given as trees. In modern IVM systems [18, 28], without sacrificing performance, which extensively cache subqueries to reduce maintenance costs. As we show, when the data and query are both trees, TreeToaster achieves similar maintenance costs without the memory overhead of caching intermediate results (Figure 1). TreeToaster further reduces memory overheads by taking advantage of the fact that the compiler already maintains a copy of the AST in memory with pointers linking nodes together. TreeToaster combines these compiler-specific approaches with standard techniques for view maintenance like inlining and pre-compiling to C++ [18] to produce an incremental-view maintenance engine that meets or beats state-of-the-art view maintenance systems on AST pattern-matching workloads, while using significantly less memory.

Concretely, the contributions of this paper are: (i) We formally model AST pattern-matching queries and present a technique for incrementally maintaining precomputed views over such queries. (ii) We show how declaratively specified rewrite rules can be further inlined into view maintenance to further reduce maintenance costs. (iii) As a proof of concept, we integrate TreeToaster with a just-in-time data-structure compiler [4, 16]. (iv) We present experiments that show that TreeToaster significantly outperforms "bolted-on" state-of-the-art IVM systems and is beneficial to the just-in-time data-structure compiler.

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

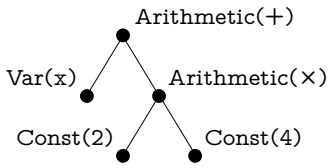


Figure 2. An AST for the expression `x + 2.0 * 4.0`

2 Notation and Background

In its simplest form, a typical compiler's activities break down into three steps: parsing, optimizing, and output.

Parsing. First, a parser converts input source code into a structured Abstract Syntax Tree (AST) encoding the code.

Example 2.1. Figure 2 shows the AST for the expression `x + 2.0 * 4.0`. AST nodes have labels (e.g., `Arithmetic`, `Var`, or `Const`) and attributes (e.g., `+`, `int`, or 4).

We formalize an AST as a tree with labeled nodes and annotated with zero or more attributes.

Definition 1 (Node). *An Abstract Syntax Tree node* $N = \mathsf{Node}(\ell, A, \overline{N})$ *is a 3-tuple, consisting of a label* ℓ *drawn from an alphabet* $\mathcal{L}$*; annotations* $A : \Sigma_M \to \mathbb{R}$*, a partial map from an alphabet of attribute names* Σ_M *to real values; and an ordered list of children* $\overline{N}$*.*

We define a leaf node (denoted **isleaf**(N)) as a node that has no child nodes. For clarity of presentation, we limit attribute values to reals. We assume that nodes follow a schema $\mathcal{S} : \mathcal{L} \to 2^{\Sigma_M} \times \mathbb{N}$; For each label, we fix a set of attributes that are present in all nodes with the label, as well as an upper bound on the number of children.

Optimization. Next, the optimizer rewrites the AST, iteratively replacing subtrees of the AST. Optimizer rules typically identify candidates for replacement by specifying pattern-matching expressions.

Example 2.2. A common rule is inlining (pre-computing) arithmetic over constant values:

```
Arithmetic(op)[ Const(a), Const(b) ] → Const(op(a, b))
```

This rule matches any subtree with an `Arithmetic` node at the root and two child `Const` nodes. The optimizer then evaluates the expression (`op(a, b)`) and replaces the subtree with a `Const` node. Continuing the above example, this rule would match the *Arithmietic*(×) node and rewrite it and its children into a single node: *Const*(8).

The optimizer continues searching for subtrees matching one of its patterns until no further matches exist (or an iteration threshold or timeout is reached).

Output. Finally, the compiler uses the optimized AST as appropriate by generating bytecode, C++ code, etc.…

$$\Theta : \mathtt{atom} = \mathtt{atom} \mid \mathtt{atom} < \mathtt{atom} \mid \Theta \wedge \Theta \mid \Theta \vee \Theta \mid \neg\Theta \mid \mathbb{T} \mid \mathbb{F}$$

$$\mathtt{atom} : \mathtt{const} \mid \Sigma_I.\Sigma_M \mid \mathtt{atom}\ [+, -, \times, \div]\ \mathtt{atom}$$

Figure 3. Constraint Grammar

2.1 Pattern Matching Queries

We formalize pattern matching in the following grammar:

Definition 2 (Pattern). *A pattern query* Q *is defined as*

$$Q : \mathtt{AnyNode} \mid \mathtt{Match}(\mathcal{L}, \Sigma_I, \overline{Q}, \Theta)$$

The symbol $\mathtt{Match}(\ell_q, i, \overline{Q}, \theta)$ indicates a structural match that succeeds iff (i) The matched node has label ℓ_q, (ii) the children of the matched node recursively satisfy $q_i \in \overline{Q}$, and (iii) the a constraint θ over the attributes of the node and its children is satisfied. The symbol `AnyNode` matches any node. Pattern matching semantics are formalized in Figure 4.

The grammar for constraints is given in Figure 3, and its semantics are typical. A variable atom $i.x$ is a 2-tuple of a Node name ($i \in \Sigma_I$) and an Attribute name ($x \in \Sigma_M$), respectively, and evaluates to $\Gamma(i)(x)$, given some scope $\Gamma : \Sigma_I \to \Sigma_M \to \mathbb{R}$. This grammar is expressive enough to capture the full range of comparisons ($>, \geq, \leq, <, =, <$), and so we use these freely throughout the rest of the paper.

Example 2.3. Returning to Example 2.2, only `Arithmetic` nodes with two `Const` nodes as children are eligible for the inlining rule. The corresponding pattern query is:

$$\mathtt{Match}(\mathtt{Arithmetic}, A, [\ \mathtt{Match}(\mathtt{Const}, B, [], \mathbb{T}),\ \mathtt{Match}(\mathtt{Const}, C, [], \mathbb{T})\ ], \mathbb{T})$$

Now consider the case where we want to avoid loss of precision by not inlining arithmetic when the ratio between the two constants is too small. This requirement is captured by replacing the outermost constraint with:

$$(\mathsf{B}.val \div \mathsf{C}.val > \mathtt{THRESHOLD}) \vee (\mathsf{C}.val \div \mathsf{B}.val > \mathtt{THRESHOLD})$$

We next formalize pattern matching over ASTs. First, we define the descendants of a node $\mathtt{Desc}(N)$ to be the set consisting of N and its descendants:

$$\mathtt{Desc}(N) \triangleq \{\ N\ \} \bigcup_{k \in [n]} \mathtt{Desc}(N_k) \text{ s.t. } N = \mathsf{Node}(\ell, A, [N_1, \ldots, N_n])$$

Definition 3 (Match). *A match result, denoted* $q(N)$*, is the subset of* N *or its descendents on which* q *evaluates to true.*

$$q(N) \triangleq \{\ N' \mid N' \in \mathtt{Desc}(N) \wedge q, N' \mapsto \mathbb{T}, \Gamma\ \}$$

Pattern Matching is Expensive. The optimization step is a tight loop in which the optimizer searches for a pattern match, applies the corresponding rewrite rule to the matched node, and repeats until convergence. Pattern matching typically requires iteratively traversing the entire AST. Every applied rewrite creates or removes opportunities for further

Pattern Rule 1

$$\mathtt{AnyNode}, \mathtt{Node}(\ell, A, \overline{N}) \mapsto \mathbb{T}, \emptyset$$

Pattern Rule 2.1

$$\frac{\ell_q = \ell \qquad q_1, N_1 \mapsto \mathbb{T}, \Gamma_1 \quad \ldots \quad q_n, N_n \mapsto \mathbb{T}, \Gamma_n \qquad \Gamma = \{\ i \to A\ \} \cup \bigcup_{k \in [n]} \Gamma_i \qquad \theta(\Gamma)}{\mathtt{Match}(\ell_q, i, [q_1 \ldots q_n], \theta), \mathtt{Node}(\ell, A, [N_1 \ldots N_n]) \mapsto \mathbb{T}, \Gamma}$$

Pattern Rule 2.2

$$\frac{\ell_q \neq \ell}{\mathtt{Match}(\ell_q, i, [q_1 \ldots q_n], \theta), \mathtt{Node}(\ell, A, [N_1 \ldots N_n]) \mapsto \mathbb{F}, \emptyset}$$

Pattern Rule 2.3

$$\frac{\exists k \in [n] : q_k, N_k \mapsto \mathbb{F}, \emptyset}{\mathtt{Match}(\ell_q, i, [q_1 \ldots q_n], \theta), \mathtt{Node}(\ell, A, [N_1 \ldots N_n]) \mapsto \mathbb{F}, \emptyset}$$

Pattern Rule 2.4

$$\frac{q_1, N_1 \mapsto \mathbb{T}, \Gamma_1 \quad \ldots \quad q_n, N_n \mapsto \mathbb{T}, \Gamma_n \qquad \Gamma = \{\ i \to A\ \} \cup \bigcup_{k \in [n]} \Gamma_i \qquad \neg\theta(\Gamma)}{\mathtt{Match}(\ell_q, i, [q_1 \ldots q_n], \theta), \mathtt{Node}(\ell, A, [N_1 \ldots N_n]) \mapsto \mathbb{F}, \emptyset}$$

Figure 4. Pattern Query Semantics

rewrites, necessitating repeated searches for the same pattern. Even with intelligent scheduling of rewrites, the need for repeated searches can not usually be eliminated outright.

Example 2.4. Continuing the example, the optimizer would traverse the entire AST looking for `Arithmetic` nodes with child `Const` nodes. A depth-first traversal ensures that any replacement happens before the optimizer checks the parent for eligibility. However, another rewrite may introduce new opportunities for inlining (e.g., by creating new `Const` nodes), and the check must be repeated.

3 Bolting-On IVM for Pattern Matching

Pattern-matching has an analog in relational query processing, suggesting that incremental view maintenance (IVM), a technique used to mitigate the cost of frequent, repeated queries might be helpful. The first step is to map the AST to a relational encoding. For each $\ell \to \langle\ \{\ x_1, \ldots, x_k\ \}, c\ \rangle \in \mathcal{S}$, we define a relation $R_\ell(\mathtt{id}, x_1, \ldots, x_k, \mathtt{child}_1, \ldots, \mathtt{child}_c)$ with an id field, and one field per attribute or child. Each node $N = \mathtt{Node}(\ell, A, [N_1, \ldots, N_n])$ is assigned a unique identifier $\mathbf{id}_N$ and defines a row of relation R_ℓ

$$\langle\ \mathbf{id}_N, A(x_1), \ldots, A(x_k), \mathbf{id}_{N_1}, \ldots, \mathbf{id}_{N_c}\ \rangle$$

A pattern q can be reduced to an equivalent query over the relational encoding, as shown in Figure 5. A pattern with k `Match` nodes becomes a k-ary join over the relations $\overline{R}_q$ corresponding to the label on each `Match` node. Each relation is aliased to its node variable. Join constraints are given by parent/child relationships, and pattern constraints transfer directly to the **`WHERE`** clause.

3.1 Incremental View Maintenance

Materialized views are used in production databases to accelerate query processing. If a view is accessed repeatedly, database systems can *materialize* the view query Q by precomputing its results $Q(D)$ on the database D. However,

$$\overline{R}_q \triangleq \begin{cases} \emptyset & \textbf{if } q = \mathtt{AnyNode} \\ \{\ (R_\ell\ \ \mathtt{AS}\ \ i)\ \} \underset{x \in [n]}{\cup} \overline{R}_{q_x} & \textbf{if } q = \mathtt{Match}(\ell, i, [q_1, \ldots, q_n], \theta) \end{cases}$$

$$\theta_q \triangleq \begin{cases} \mathbb{T} & \textbf{if } q = \mathtt{AnyNode} \\ \theta \underset{x \in [n]}{\wedge} \theta_{q_x} \wedge \mathtt{join}(i.\mathtt{child}_x, q_x) & \\ & \textbf{if } q = \mathtt{Match}(\ell, i, [q_1, \ldots, q_n], \theta) \end{cases}$$

$$\mathtt{join}(a, q) \triangleq \begin{cases} \mathbb{T} & \textbf{if } q = \mathtt{AnyNode} \\ a = i.\mathbf{id} & \textbf{if } q = \mathtt{Match}(\ell, i, [q_1, \ldots, q_n], \theta) \end{cases}$$

$$q \equiv \texttt{SELECT * FROM } \overline{R}_q \texttt{ WHERE } \theta_q$$

Figure 5. Converting a pattern q to an equivalent SQL query.

when the database changes, the view must be updated to match. Given a set of changes, ΔD (e.g., insertions or deletions), a naive approach would be to simply recompute the view on the updated database $Q(D + \Delta D)$. However, if ΔD is small, most of this computation will be redundant. Thus, most production database systems instead derive a so-called "delta query" (ΔQ) that can compute a set of changes to the (already available) $Q(D)$ and can be inexpensively merged in. That is, denoting the merge operation by $\Rightarrow$

$$Q(D + \Delta D) \equiv \Delta Q(D, \Delta D) \Rightarrow Q(D)$$

Example 3.1. Consider the following 4-way join query:

$$Q(D) := R \bowtie S \bowtie T \bowtie U$$

After inserting a row r into R we want $Q(D + r)$. Abusing notation, using r to denote a singleton relation:

$$\begin{aligned} Q(D + r) &= (R \uplus r) \bowtie S \bowtie T \bowtie U \\ &= (R \bowtie S \bowtie T \bowtie U) \uplus (r \bowtie S \bowtie T \bowtie U) \\ &= Q(D) \uplus (r \bowtie S \bowtie T \bowtie U) \end{aligned}$$

Instead of computing the full 4-way join, we can replace r with a singleton, compute the cheaper query ($r \bowtie S \bowtie T \bowtie$

U), and union the result with our original materialzed view to obtain an update.

The cost of $\Delta Q(D, \Delta D)$ and $\oplus$ is generally lower than re-running the query, making this a win when database updates are small and infrequent. However, ΔQ can still be expensive. For larger or more frequent changes[1], more drastic measures are required. Several IVM systems [2, 3, 21, 28] further reduce the cost of computing ΔQ by caching intermediate results. Ross et. al. [28] proposed a form of cascading IVM that caches all intermediate results in the physical plan of the view query.

Example 3.2. Continuing the example, we use a right-to-left execution order. That is, the query is evaluated as:

$$R \bowtie (S \bowtie (T \bowtie U))$$

In addition to materializing $Q(D)$, Ross' scheme also materializes the results of $Q_1 = T \bowtie U$, and $Q_2 = S \bowtie Q_1$ on D. When r is inserted into R, the update only requires computing a simple 2-way join $r \bowtie Q_2$. However, updates to U are now (slightly) more expensive as every view along the plan needs to be updated as well.

Ross' approach of caching intermediate state is analogous to typical approaches to fixpoint computation (e.g., Differential Dataflow [21]). However, it penalizes updates to tables early in the query plan. With DBToaster [18], Koch et. al. proposed an alternative: Materialize intermediate state for all possible query plans. Counterintuitively, this significantly reduces the cost of view maintenance. Although far more tables need to be updated with every database change, the updates are generally small and efficiently computable.

3.2 Embedded IVM

DBToaster [18] in particular is designed for embedded use. It compiles a set of queries down to a C++ or Scala data structure that maintains the query results. The data structure exposes insert, delete, and update operations for each source relation; and materializes the results of each query into an iterable collection. One strategy for improving compiler performance is to make the minimum set of changes required (i.e., "bolt-on") to allow it to use an incremental view maintenance data structure generated by DBToaster:

1. The reduction above generates SQL queries for each pattern-match query used by the optimizer.
2. DBToaster builds a view maintenance data structure.
3. The compiler is instrumented to register changes in the AST with the view maintenance data structure.
4. Iterative searches in the optimizer for candidate AST nodes are replaced with a constant-time lookup on the view maintenance data structure.

As we show in Section 7, this approach significantly outperforms naive iterative scans over the AST. Although DBToaster requires maintaining supplemental data structures, the overhead of maintaining these structures is negligible compared to the benefit of constant-time pattern match results.

Nevertheless, there are three major shortcomings to this approach. First, DBToaster effectively maintains a shadow copy of the entire AST — at least the subset that affects pattern-matching results. Second, DBToaster aggressively caches intermediate results. For example, materializing the results of a simple pattern query with 4 `Match` nodes forces the materialization of up to 5 additional queries. Finally, DBToaster-generated view structures register updates at the granularity individul node insertions/deletions, making it impossible for them to take advantage of the fact that most rewrites follow very structured patterns. Between intermediate results and its shadow copy, for a relatively small number of pattern-matches, the memory use of the compiler with a DBToaster view structure bolted increases by a factor of 2.5×. Given that memory consumption is already a pain point for compilers working with large ASTs, this is not viable.

Before addressing these pain points, we first assess why they arise. First, DBToaster-generated view maintenance data structures are self-contained. When an insert is registered, the structure needs to preserve state for later use. Although unnecessary fields are projected away, this still amounts to a shadow copy of the AST. Second, DBToster has a heavy focus on aggregate queries. Caching intermediate state allows aggressive use of aggregation and selection push-down into intermediate results, both reducing the amount of state maintained and the work needed to maintain views.

Example 3.3. Returning to our example, consider instances.

S	**B**	**C**
	1	1
	3	3

T	**C**	**D**
	1	1
	1	3
	3	1
	3	3

U	**D**	**E**
	1	1
	2	1
	⋯	
	2	1000
	3	4

Inserting the tuple $\langle\ 1, 1\ \rangle$ into R triggers a three-way join. The naive insertion requires access to disjoint ranges of U regardless of how U is ordered on disk. While pre-materializing $S \bowtie T \bowtie U$ creates duplicate rows, it also filters most of U.

Consider the same example with the selection predicate $B = E$. By pushing down the predicate and pre-materializing $\sigma_{B=1}(S \bowtie T \bowtie U)$, we can avoid expensive iteration over records in T and U that will eventually fail the predicate.

Both benefits are of limited use in pattern-matching on ASTs. The reduction to SQL produces purely SPJ queries, mitigating the value of aggregate push-down. The value of selection push-down is mitigated by the AST's implicit foreign key constraints: each child has a single parent and each child attribute in the relational form references at most one child. Unlike a typical join where a single record may join with many results, here a single node only participates in a

[1] A common rule of thumb is to rebuild the view from scratch if more than 10% of the input records change.

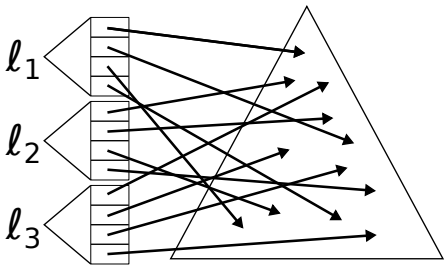


Figure 6. Indexing the AST by Label

single join result[2]. This also limits the value of materializing for the sake cache locality.

In summary, for ASTs, the cached state is either redundant or minimally beneficial. Thus a view maintenance scheme designed specifically for compilers should be able to achieve the same benefits, but without the memory overhead.

4 Pattern Matching on a Space Budget

We have a set of patterns $q_1, \ldots, q_m$ and an evolving abstract syntax tree N. Our goal is, given some q_k, to be able to obtain a single, arbitrary element of the set $q_k(N)$ as quickly as possible. Furthermore, this should be possible without significant overhead as N evolves into N', N'', and so forth. Recall that there are three properties that have to hold for a node N to match q: (i) The node and pattern labels must match, (ii) Any recursively nested patterns must match, and (iii) The constraint must hold over the node and its descendants.

4.1 Indexing Labels

A standard first approach to accelerating queries is indexing, for example by building a secondary index over the node labels, as illustrated in Figure 6. For each node label, the index maintain a set of pointers to all nodes in the AST with that label. Updates to the AST are propagated into the index. Pattern match queries can use this index to scan a subset of the AST that includes only nodes with the appropriate label, as shown in Algorithm 1.

Algorithm 1: IndexLookup(N, q, Index_N)

Input: $N \in \mathcal{N}$, $q \in \mathcal{Q}$, $\text{Index}_N : \ell \rightarrow \{ \text{Desc}(N) \}$
Output: $N_{match} \in \text{Desc}(N)$

```
1 if q = AnyNode then
2     return N_match ← N
3 else if q = Match(ℓ, i, [q_1, ..., q_n], θ) then
4     for N_idx ∈ Index_N[ℓ] do
5         if q, N ↦ T, Γ then
6             return N_match ← N_idx
```

[2]To clarify, a node may participate in multiple join results in different positions in the pattern match, but only in one result at the same position.

Indexing the AST by node label is simple, and has a relatively small memory overhead: approximately 28 bytes per AST node using the C++ standard libary `unordered_set`, with significant space for improvement. Similarly, the maintenance overhead is low — one hash table insert and/or remove per AST node changed. However, this index only supports filtering on labels; Recursive matches and constraints both need to be re-checked with each iteration.

4.2 Incremental View Maintenance

Capturing these additional properties, in general, requires something more aggressive than indexing: view materialization, the basis for TreeToaster. Like embedded IVM systems such as DBToaster, TreeToaster builds this view once and maintains it as the AST evolves. It avoids creating a shadow copy of the AST and also leverages the existing AST to avoid materializing intermediate results.

The latter result in particular is made challenging in the presence of recursive structural constraints, or attribute constraints that reference descendent nodes. If a node is modified, for example after applying a rewrite rule, detecting newly available pattern matches requires "joining" it with all of the other nodes that could participate in the pattern. This is easy if the modified node occupies the root position of the pattern being matched; The remaininig nodes are its descendents in the tree. However an updated node may also render one of its ancestors eligible for a rewrite.

We begin to address in Section 5 by defining IVM for immutable (functional) ASTs. This simplified form lacks the ancestor problem, as any update to a node necessarily triggers updates to all of its ancestors. We then refine the approach to mutable ASTs, where only a subset of the tree is updated. Then, in Section 6 we describe how declarative specifications of rewrite rules can be used to streamline the derivation of update sets and to eliminate unnecessary checks. Throughout Sections 5 and 6, we focus on the case of a single pattern query. To generalize to multiple patterns, TreeToaster applies the same technique iteratively for all pattern queries.

5 IVM for ASTs

We first review a generalization of multisets proposed by Blizard [7] that allows for elements with negative multiplicities. A generalized multiset $\mathbf{M} : \mathbf{dom}(\mathbf{M}) \rightarrow \mathbb{Z}$ maps a set of elements from a domain $\mathbf{dom}(\mathbf{M})$ to an integer-valued multiplicity. We assume finite-support for all generalized multisets: only a finite number of elements are mapped to non-zero multiplicities. Union on a generalized multiset, denoted $\oplus$, is defined by summing multiplicities.

$$(\mathbf{M}_1 \oplus \mathbf{M}_2)(x) \triangleq \mathbf{M}_1(x) + \mathbf{M}_2(x)$$

Difference, denoted $\ominus$, is defined analogously:

$$(\mathbf{M}_1 \ominus \mathbf{M}_2)(x) \triangleq \mathbf{M}_1(x) - \mathbf{M}_2(x)$$

We write $x \in \mathbf{M}$ as a shorthand for $\mathbf{M}(x) \neq 0$. When combining sets and generalized multisets, we will abuse notation and lift sets to the corresponding generalized multiset, where each of the set's elements is mapped to 1.

A view $\texttt{View}_q$ is a generalized multiset. We define the correctness of a view relative to the root of an AST. Without loss of generality, we assume that N'' appears at most once in $\texttt{Desc}(N)$ and $\texttt{Desc}(N'')$

Definition 4 (View Correctness). *A view $\texttt{View}_q$ is correct for N if it maps exactly the subset of N and its descendents that match q to 1:*

$$\texttt{View}_q = \{\!|\ N' \rightarrow 1 \mid N' \in q(N)\ |\!\}$$

If we start with a view $\texttt{View}_q$ that is correct for the root of an AST N and rewrite the AST's root to N', we would like to update the view accordingly. We assume for the moment that we have an easy way to obtain a delta between the two ASTs: the difference: $\texttt{Desc}(N') \ominus \texttt{Desc}(N')$. This delta is generally small for a rewrite, including only the nodes of the rewritten subtree and their ancestors. We revisit this assumption in the following section. Algorithm 2 shows a simple algorithm for maintaining the $\texttt{View}_q$, given a small change Δ, expressed as a generalized multiset.

Algorithm 2: $\texttt{IVM}(q, \texttt{View}_q, \Delta)$

Input: $q \in Q$, $\texttt{View}_q \in \{\!|\ \mathcal{N}\ |\!\}$, $\Delta \in \{\!|\ \mathcal{N}\ |\!\}$
Output: $\texttt{View}_q$

```
for N_i ∈ Δ do
    if q, N_i ↦ 𝕋, Γ then
        View_q ← View_q ⊕ {| N_i → Δ(N_i) |}
```

For Algorithm 2 to be correct, we need to show that it computes exactly the update to $\texttt{View}_q$.

Lemma 5.1 (Correctness of IVM). *Given two ASTs N and N' and assuming that $\texttt{View}_q$ is correct for N, then the generalized multiset returned by $\texttt{IVM}(q, \texttt{View}_q, \texttt{Desc}(N') \ominus \texttt{Desc}(N))$ is correct for N'.*

Proof. Lets denote the generalized multiset returned from $\texttt{IVM}(q, \texttt{View}_q, \texttt{Desc}(N') \ominus \texttt{Desc}(N))$ as $\texttt{View}'_q$. To prove that $\texttt{View}_{q'}$ is correct we examine the multiplicity of a arbitrary node N''.
(i) If $N'' \in \texttt{Desc}(N)$, $N'' \notin \texttt{Desc}(N')$ and $q, N'' \mapsto \mathbb{T}, \Gamma$

$$\begin{aligned}
\texttt{Desc}(N') \ominus \texttt{Desc}(N)(N'') &= -1 \\
\texttt{View}_q(N'') &= 1 \\
\texttt{View}'_q &= \texttt{View}_q(N'') \oplus (\texttt{Desc}(N') \ominus \texttt{Desc}(N))(N'') \\
&= 0 \\
&= q(N')
\end{aligned}$$

(ii) If $N'' \in \texttt{Desc}(N)$, $N'' \notin \texttt{Desc}(N')$ and $q, N'' \mapsto \mathbb{F}, \Gamma$

$$\begin{aligned}
\texttt{Desc}(N') \ominus \texttt{Desc}(N)(N'') &= -1 \\
\texttt{View}_q(N'') &= 0 \\
\texttt{View}'_q &= \texttt{View}_q(N'') \\
&= 0 \\
&= q(N')
\end{aligned}$$

(iii) If $N'' \notin \texttt{Desc}(N)$, $N'' \in \texttt{Desc}(N')$ and $q, N'' \mapsto \mathbb{T}, \Gamma$

$$\begin{aligned}
\texttt{Desc}(N') \ominus \texttt{Desc}(N)(N'') &= 1 \\
\texttt{View}_q(N'') &= 0 \\
\texttt{View}'_q &= \texttt{View}_q(N'') \oplus (\texttt{Desc}(N') \ominus \texttt{Desc}(N))(N'') \\
&= 1 \\
&= q(N')
\end{aligned}$$

(iv)If $N'' \notin \texttt{Desc}(N)$, $N'' \in \texttt{Desc}(N')$ and $q, N'' \mapsto \mathbb{F}, \Gamma$

$$\begin{aligned}
\texttt{Desc}(N') \ominus \texttt{Desc}(N)(N'') &= 1 \\
\texttt{View}_q(N'') &= 0 \\
\texttt{View}'_q &= \texttt{View}_q(N'') \\
&= 0 \\
&= q(N')
\end{aligned}$$

(v) If $N'' \in \texttt{Desc}(N)$, $N'' \in \texttt{Desc}(N')$ and $q, N'' \mapsto \mathbb{T}, \Gamma$

$$\begin{aligned}
\texttt{Desc}(N') \ominus \texttt{Desc}(N)(N'') &= 0 \\
\texttt{View}_q(N'') &= 1 \\
\texttt{View}'_q &= \texttt{View}_q(N'') \oplus (\texttt{Desc}(N') \ominus \texttt{Desc}(N))(N'') \\
&= 1 \\
&= q(N')
\end{aligned}$$

(vi) If $N'' \in \texttt{Desc}(N)$, $N'' \in \texttt{Desc}(N')$ and $q, N'' \mapsto \mathbb{F}, \Gamma$

$$\begin{aligned}
\texttt{Desc}(N') \ominus \texttt{Desc}(N)(N'') &= 0 \\
\texttt{View}_q(N'') &= 0 \\
\texttt{View}'_q &= \texttt{View}_q(N'') \\
&= 0 \\
&= q(N')
\end{aligned}$$

(vii)If $N'' \notin \texttt{Desc}(N)$, $N'' \notin \texttt{Desc}(N')$ and $q, N'' \mapsto \mathbb{T}, \Gamma$

$$\begin{aligned}
\texttt{Desc}(N') \ominus \texttt{Desc}(N)(N'') &= 0 \\
\texttt{View}_q(N'') &= 0 \\
\texttt{View}'_q &= \texttt{View}_q(N'') \oplus (\texttt{Desc}(N') \ominus \texttt{Desc}(N))(N'') \\
&= 0 \\
&= q(N')
\end{aligned}$$

(viii) If $N'' \notin \texttt{Desc}(N)$, $N'' \notin \texttt{Desc}(N')$ and $q, N'' \mapsto \mathbb{F}, \Gamma$

$$\begin{aligned}
\texttt{Desc}(N') \ominus \texttt{Desc}(N)(N'') &= 0 \\
\texttt{View}_q(N'') &= 0 \\
\texttt{View}'_q &= \texttt{View}_q(N'') \\
&= 0 \\
&= q(N')
\end{aligned}$$

Since for all 8 possibilities Algorithm 2 computes correctly the multiplicity of node N'' in the resultant of $\texttt{IVM}(q, \texttt{View}_q, \texttt{Desc}(N') \ominus \texttt{Desc}(N)$ as it would be in $\texttt{Desc}(N')$ Algorithm 2 is correct. □

5.1 Mutable Abstract Syntax Trees

Although correct, IVM assumes that the AST is immutable: When a node changes, each of its ancestors must be updated to reference the new node as well. We now lift this restriction. Notably, this change decreases the overhead of view maintenance by reducing the number of nodes that need to be checked with each AST update. To begin, create a notational distinction between the root node N and the node being replaced R. For clarity of presentation, we again assume that any node R occurs at most once in N. $N[R\backslash R']$ is the node resulting from a replacement of R with R' in N:

$$N[R\backslash R'] = \begin{cases} R' & \textbf{if } N = R \\ \mathsf{Node}(\ell, A, [N_1[R\backslash R'], \ldots, N_n[R\backslash R']]) & \\ & \textbf{if } \mathsf{Node}(\ell, A, [N_1, \ldots, N_n]) \end{cases}$$

We also lift this notation to collections:

$$\mathtt{View}[R\backslash R'] = \{\!|\ \ N[R\backslash R'] \to c \mid (N \to c) \in \mathtt{View}\ |\!\}$$

We emphasize that although this notation modifies each node individually, this complexity appears only in the analysis. The underlying effect being modeled is a single pointer swap.

Definition 5 (Pattern Depth)**.** *The depth $D(q)$ of a pattern q is the number of edges along the longest downward path from root of the pattern to an arbitrary pattern node q_i.*

$$D(q) = \begin{cases} 0 & \textbf{\textit{if}}\ q = \mathtt{AnyNode} \\ 1 + \underset{i\in[n]}{max}(D(q_i)) & \textbf{\textit{if}}\ q = \mathtt{Match}(\ell, i, [q_1, \ldots, q_n], \theta) \end{cases}$$

The challenge posed by mutable ASTs is that the modified node may make one of its ancestors eligible for a pattern-match. However, as we will show, only a bounded number of ancestors are required. Denote by $\mathtt{Ancestor}_i(N)$ the ith ancestor of N[3]. The maximal search set, which we now define, includes all nodes that need to be checked for matches.

Definition 6 (Maximal Search Set)**.** *Let R and R' be an arbitrary node in the AST and its replacement. The maximal search set for R and R' and pattern q, $\lceil R, R' \rceil_q$ is the difference between the generalized multiset of the respective nodes, their descendents, and their ancestors up to a height of $D(q)$.*

$$\lceil R, R' \rceil_q \triangleq \mathtt{Desc}(R) \oplus \{\!|\ \ \mathtt{Ancestor}_i(R) \to 1 \mid i \in [n]\ |\!\} \\ \ominus \mathtt{Desc}(R') \ominus \left\{\!\!\left|\ \ \mathtt{Ancestor}_i(R') \to 1 \mid i \in [n]\ \right|\!\!\right\}$$

Lemma 5.2. *Let N be the root of an AST, q be a pattern, and R and R' be an arbitrary node in the AST and its replacement. If $\mathtt{View}_q$ is correct for N. and $\mathtt{View}'_q = \mathtt{IVM}(q, \mathtt{View}_q, \lceil R, R' \rceil_q)$, then $\mathtt{View}'_q[R\backslash R']$ is correct for $N[R\backslash R']$*

Proof. Differs from the proof of Lemma 5.1 in three additional cases. If $q, N'' \mapsto \langle\ \mathbb{F}, \Gamma\ \rangle$, then $q(N) = N[R\backslash R'] = \mathtt{View}_q = 0$. The condition on line 2 is false, and the multiplicity is unchanged at 0. Otherwise, if $N'' \in \mathtt{Desc}(N)$ then $N''[R\backslash R'] \in \mathtt{Desc}(N[R\backslash R'])$ by definition. Here, there are two possibilities: Either N" is within the $D(q)$-high ancestors of R or not. In the former case, both N'' and $N''[R\backslash R']$ appear in $\lceil R, R' \rceil_q$ with multiplicities 1 and -1 respectively, and the proof is identical to Lemma 5.1. We prove the latter case by recursion, by showing that if N'' is not among the $D(q)$ immediate ancestors and $q, N'' \mapsto \langle\ x, \Gamma\ \rangle$, then $q, N''[R\backslash R']$ does as well. . The base case is a pattern depth of 0, or $q = \mathtt{AnyNode}$. This pattern always evaluates to $\langle\ \mathbb{T}, \emptyset\ \rangle$ regardless of input, so the condition is satisfied. For the recursive case, we assume that the property holds for any q and N''' not among the $d-1$th ancestors of R. Since $d > 1$, $R \neq N''$ and the precondition for Pattern Rule 2.1 is guaranteed to be unchanged. If a pattern has depth d, none of its children have depth more than $d-1$ so we have for each of the pattern's children that if $q_i, N_i \mapsto \langle\ x_i, \Gamma_i\ \rangle$ then $q_i, N_i[R\backslash R'] \mapsto \langle\ x_i, \Gamma_i\ \rangle$, and the preconditions for Pattern Rules 2.1 and 2.3 are unchanged. Likewise, since both inputs map to identical gammas and $R \neq N''$, the preconditions for Pattern rule 2.4 are unchanged. Since the preconditions for all relevant pattern-matching rules are unchanged, the condition holds at a depth of d. □

6 Inlining into Rewrite Rules

Algorithm 2 takes the set of changed nodes as an input. In principle, this information could be obtained by manually instrumenting the compiler to record node insertions, updates, and deletions. However, many rewrite rules are structured: The rule replaces exactly the matched set of nodes with a new subtree. Unmodified descendants are re-used as-is, and with mutable ASTs a subset of the ancestors of the modified node are re-used as well. TreeToaster provides a declarative language for specifying the output of rewrite rules. This language serves two purposes. In addition to making it easier to instrument node changes for TreeToaster, declaratively specifying updates opens up several opportunities for inlining-style optimizations to the view maintenance system. The declarative node generator grammar follows:

$$\mathcal{G} : \mathsf{Gen}(\mathcal{L}, \Sigma_{\mathcal{I}}, \overline{\mathcal{G}}) \mid \mathsf{Reuse}(\Sigma_{\mathcal{I}} \to \mathcal{N})$$

A Gen term indicates the creation of a new node with the specified label, attributes, and children. Attribute values are populated according to a provided attribute scope $\Gamma : \Sigma_{\mathcal{I}} \to \Sigma_M \to \mathbb{R}$. A Reuse term indicates the re-use of a subtree from the previous AST, provided by a node scope $\mu : \Sigma_{\mathcal{I}} \to \mathcal{N}$. Node generators are evaluated by the $[\![\cdot]\!]_{\Gamma,\mu} : \mathcal{G} \to \mathcal{N}$ operator, defined as follows:

$$[\![g]\!]_{\Gamma,\mu} = \begin{cases} \mu(i) & \textbf{if } g = \mathsf{Reuse}(i) \\ \mathsf{Node}(\ell, \Gamma(i), [\![g_1]\!]_{\Gamma,\mu}, \ldots, [\![g_n]\!]_{\Gamma,\mu}) & \\ & \textbf{if } g = \mathsf{Gen}(\ell, i, [g_1, \ldots, g_n]) \end{cases}$$

[3]We note that ASTs do not generally include ancestor pointers. The ancestor may be derived by maintaining a map of ancestors, or by extending the AST definition with parent pointers.

A declaratively specified rewrite rule is given by a 2-tuple: $\langle\, q, g \,\rangle \in \mathcal{Q} \times \mathcal{G}$, a match pattern describing the nodes to be removed from the tree, and a corresponding generator describing the nodes to be re-inserted into the tree. As a simplification for clarity of presentation, we require that Reuse nodes reference nodes matched by AnyNodepatterns. Define the set of matched node pairs as the set

$$\texttt{pair}(q, R) = \{\, \langle\, q, R \,\rangle \,\} \cup \ldots$$
$$\ldots \begin{cases} \{\, \langle\, \texttt{AnyNode}, R \,\rangle \,\} & \textbf{if } q = \texttt{AnyNode} \\ \bigcup_{k \in [n]} \texttt{pair}(q_k, N_k) & \textbf{if } q = \texttt{Match}(\ell, x, [q_1, \ldots, q_n], \theta) \\ & \quad R = \texttt{Node}(\ell, A, [N_1, \ldots, N_n]) \end{cases}$$

A set of generated node pairs $\texttt{pair}(g, \Gamma, \mu)$ is defined analogously relative to the node $[\![g]\!]_{\Gamma,\mu}$

Definition 7 (Safe Generators)**.** *Let N be an AST root, q be a pattern query, and $R \in q(N)$ be a node of the AST matching the pattern. We call a generator $g \in \mathcal{G}$* safe *for $\langle\, q, R \,\rangle$ iff g reuses exactly the wildcard matches of q. Formally:*

$$\langle\, \texttt{AnyNode}, N \,\rangle \in \texttt{pair}(q, R) \Leftrightarrow \langle\, \texttt{Reuse}(N), N \,\rangle \in \texttt{pair}(g, \Gamma, \mu)$$

Let $g \in \mathcal{G}$ be a generator that is safe for $\langle\, m, R \,\rangle$, where $m \in \mathcal{Q}$ is a pattern. The mutable update delta from N to $N[R \backslash [\![g]\!]_{\Gamma,\mu}]$ is:

$$\Delta = \{\!|\ N' \to 1 \mid \langle\, g', N' \,\rangle \in \texttt{pair}(g, \Gamma, \mu)\ |\!\} \ominus \{\!|\ N' \to 1 \mid \langle\, q', N' \,\rangle \in \texttt{pair}(m, R)\ |\!\}$$

Note that the size of this delta is linear in the size of g and m.

6.1 Inlining Optimizations

Up to now, we have assumed that no information about the nodes in the update delta is available at compile time. For declarative rewrite rules, we are no longer subject to this restriction. The labels and structure of the nodes being removed and those being added are known at compile time. This allows TreeToaster to generate more efficient code by eliminating impossible pattern matches.

The process of elimination is outlined for generated nodes in Algorithm 3. A virtually identical process is used for matching removed nodes. The algorithm outputs an function that, given the generated replacement node (i.e., $[\![g]\!]_{\Gamma,\mu}$) that is not available until runtime, returns the set of nodes that could match the provided pattern. Matching only happens by label, as attribute values are also not available until runtime. If the pattern matches anything or if the node is reused (i.e., its label is not known until runtime), the node is a candidate for pattern match (Lines 1-2). Otherwise, the algorithm proceeds in two stages. It checks if a newly generated node can be the root of a pattern by recursively descending through the generator (Lines 3-11). Finally, it checks if any of the node's ancestors (up to the depth of the pattern) could be the root of a pattern match by recursively descending through the pattern to see if the root of the generated node could match (Lines 12-13). On lines 5 and 12, Algorithm 3 makes use of a recursive helper function: Align. In the base case $\texttt{Align}_0$ checks if the input pattern and generator align – whether they have equivalent labels at equivalent positions.

Algorithm 3: $\texttt{Inline}_{gen}(q, g)$

Input: $q \in \mathcal{Q}, g \in \mathcal{G}$
Output: $f : \mathcal{N} \mapsto \{\, \mathcal{N} \,\}$

1 **if** $q = \texttt{AnyNode} \vee g = \texttt{Reuse}(\mu)$ **then**
2 $f' \leftarrow (N \mapsto \{\, N \,\})$
3 **else if** $q = \texttt{Match}(\ell, i, [q_1, \ldots, q_n], \theta)$ **then**
4 $g = \texttt{Gen}(\ell', i', [g_1, \ldots, g_n])$;
5 **if** $\texttt{Align}_0(q, g)$ **then**
6 $f'' \leftarrow (N \mapsto \{\, N \,\})$
7 **else**
8 $f'' \leftarrow (N \mapsto \emptyset)$
9 **for** $i \in [n]$ **do**
10 $f_i \leftarrow \texttt{Inline}_{gen}(q, g_i)$
11 $f' \leftarrow (N \mapsto f''(N) \cup \bigcup_{i \in [n]} f_i(N)$
12 $\mathcal{A} = \{\, i \mid i \in [D(q)] \wedge \texttt{Align}_i(q, g) \,\}$;
13 $f \leftarrow (N \mapsto f'(N) \cup \{\, \texttt{Ancestor}_i(N) \mid i \in \mathcal{A} \,\})$

$$\texttt{Align}_0(q, g) = \begin{cases} \mathbb{T} & \textbf{if } q = \texttt{AnyNode} \vee g = \texttt{Reuse}(\mu) \\ \mathbb{F} & \textbf{if } q = \texttt{Match}(\ell, A, [\ldots], \theta) \\ & \quad g = \texttt{Gen}(\ell', i, [\ldots]) \wedge \ell \neq \ell' \\ \forall k : \texttt{Align}_0(q_k, g_k) & \textbf{if } q = \texttt{Match}(\ell, A, [\ldots], \theta) \\ & \quad g = \texttt{Gen}(\ell, i, [\ldots]) \end{cases}$$

The recursive case $\texttt{Align}_d$ checks for the existence an alignment among the dth level descendants of the input pattern.

$$\texttt{Align}_d(q, g) = \exists k : \texttt{Align}_{d-1}(q_k, g)$$

7 Evaluation

To evaluate TreeToaster, we incorporated four forms of view maintenance into the JustInTimeData [4, 5] compiler, a JIT compiler for data structures built around a complex AST that can *adapt* at runtime. The JustInTimeData compiler naturally works with large ASTs and requires low latencies, making it a compelling use case. As such JustInTimeData's provide an infrastructure to test TreeToaster. Our tests compare: (i) The JustInTimeData compiler's existing **Naive** iteration-based optimizer, (ii) **Index**ing labels, as proposed in Section 4.1, (iii) **Classic**al incremental view maintenance implemented by bolting on a view maintenance data structure created by DBToaster with the `--depth=1` flag, (iv) **DBT**oaster's full recursive view maintenance bolted onto the compiler, and (v) TreeToaster (**TT**)'s view maintenance built into the compiler.

Our experiments confirm the following claims: (i) TreeToaster significantly outperforms JustInTimeData's naive iteration-based optimizer, (ii) TreeToaster matches or outperforms bolt-on IVM systems, while consiming significantly less memory, (iii) On complex workloads, TreeToaster's view maintenance latency is half of bolt-on approaches,

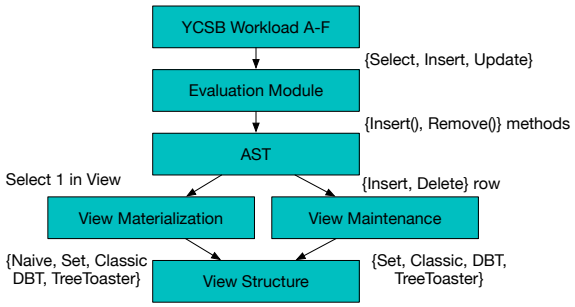


Figure 7. Benchmark Infrastructure

The rest of this section is structured as follows. First, we describe the experimental structure to evaluate the performance of TreeToaster and then discuss the data we collected. Lastly, we evaluate and discuss the results obtained.

7.1 Data Gathering and Measurement

We measure the performance of the TreeToaster IVM in identifying target nodes in the JustInTimeData AST, relative to the performance offered by alternative identification methods. Second, we compare the performance of the TreeToaster IVM against that of a reference IVM system. Our testing system consists of a new upper-layer testing module, an instrumented JustInTimeData AST, and swappable materialization and maintenance components (Figure 7).

To facilitate our experimental comparison we built a testing module in C++ to expose the JustInTimeData structure as an in-memory database, modeling various SQL `Select`, `Insert`, `Update`, and `Delete` operations as combinations of JustInTimeData `insert()` and `remove()` operations. The YCSB [11] database workloads, with expanded key sets, serve as our input data. As we are interested in IVM stemming from changing ASTs, the YCSB A and F workloads, consisting of 50% write operations, were most relevant to our use case. Others, such as the read-only C load, produce much less useful measurable activity (though we provide numbers for completeness in an anonimized git hub repository for interested reviewers available here: https://github.com/anoncoderepo/sigmod2021). The test harness also records database operation latency and process memory usage, as reported by the Linux `/proc` interface.

For data gathering we instrument TreeToaster to collect in-structure information pertaining to view materialization and maintenance: the time required to identify potential JustInTimeData transform operations (rows in the materialized views), and the time to perform view maintenance operations upon each structure reorganization step.

We evaluate 5 materialization and 4 maintenance options by swapping in the appropriate components. We ran each of these 5 search methods and 4 maintenance methods on YCSB workloads A-D and F (omitting E, which does not focus on insert-delete operations). Each combination was run 10 times, with the search and operation results aggregated.

7.2 Evaluation

We obtained our results on a server running Ubuntu 16.04.06 LTS with 192GB RAM and 24 core Xeon 2.50GHz processors. The full result set of our runs is available at https://github.com/anoncoderepo/sigmod2021. For each run, we first compile the system with the desired view materialization and maintenance components. The system then reads in and feeds the YCSB initialization data to the JustInTimeData structure. It then calls the TreeToaster `do_organize()` routine repeatedly until the structure reports itself as stable. Then the module replays the YCSB benchmark operations on the structure. After performing each operation, the module loops on `do_organize()` until either the structure reports itself as stable or until 1ms elapses. At each database operation, materialization, or organizational step the system records performance data.

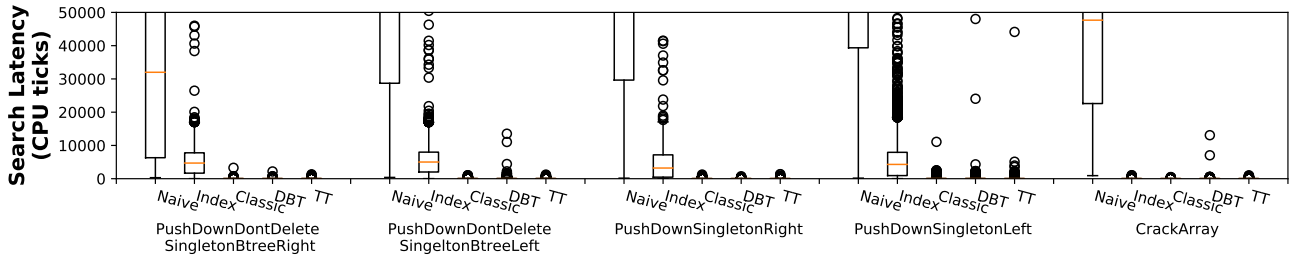
7.3 Results

We first evaluate how IVM performs relative to other methods of identifying target nodes in tree structure views. To do so, we compare the latency in identifying potential nodes (i.e. materializing views) using 5 methods: (1) A naive approach that potentially searches the entire structure for a node match, (2) A set-based identification approach, (3) a classic IVM method, (4) a hash-based IVM approach (DBToaster), and (5) the TreeToaster IVM method.

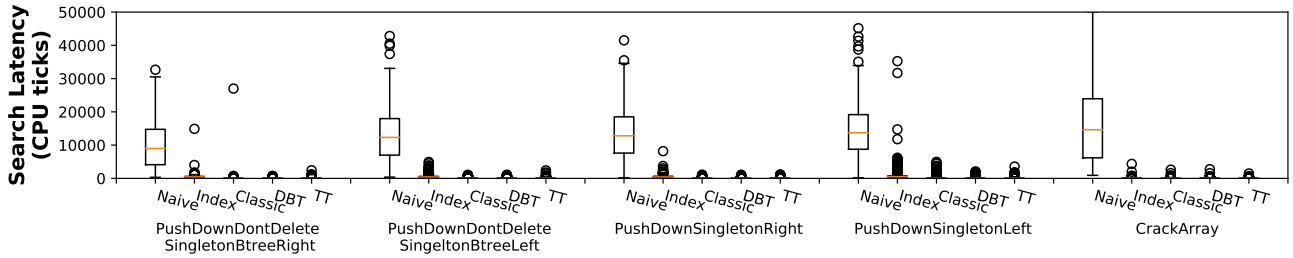
Figure 8 shows results obtained for several YCSB workloads using 300M keys. There were 5 sets of views that were materialized, representing target nodes in the underlying tree structure, which were candidates for each of 5 structure reorganization transforms (e.g. CrackArray). The 5 boxplot clusters compare the relative average performance of identifying 1 such node, using each of our 5 identification methods (e.g. Naive, Set). In each case, the naive search approach exhibits the worst performance. The set-based approach also yields worse results than either of the IVM approaches. For identifying target nodes (i.e. emitting 1 row in the view), we thus conclude that an IVM approach performs better.

We compare TreeToaster to IVM alternatives, including set-based as well as a classic IVM system and a hash-based IVM implemented using DBToaster. Figure **??** shows the results for different loads using 300M keys. The 5 boxplot clusters show the relative average maintenance performance for each of the 5 tables representing node types in the JustInTimeData structure. While the set-based approach performs better than TreeToaster for several of the tables, the huge performance penalty of materialization latency (Figure 8) renders this a non-viable alternative.

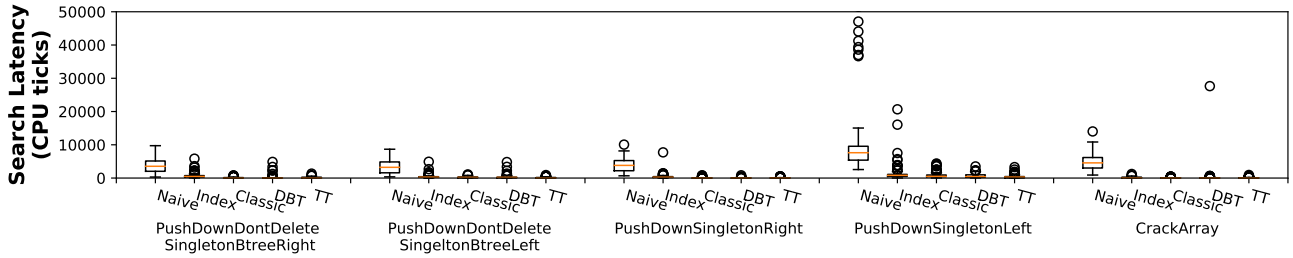
Figure 9 shows the average total latency spent searching for a target node for a JustInTimeData reorganization step, plus all maintenance steps in the reorganization, for each



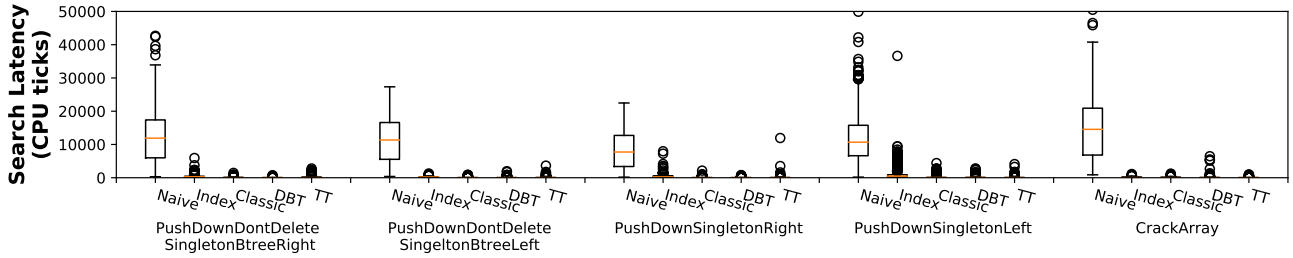
(a) Workload A



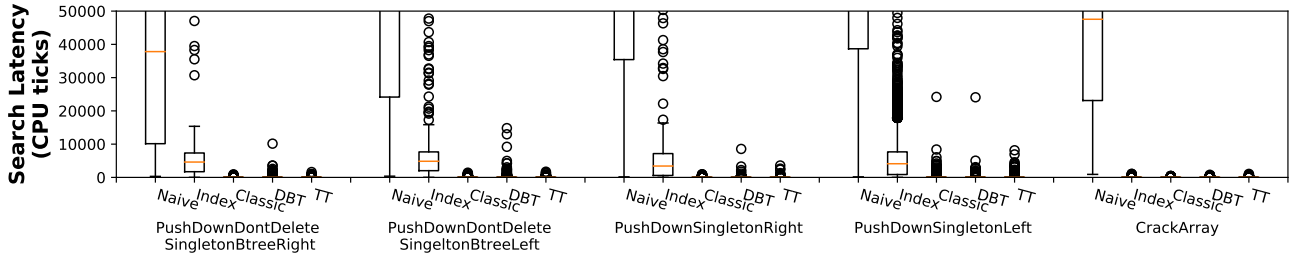
(b) Workload B



(c) Workload C

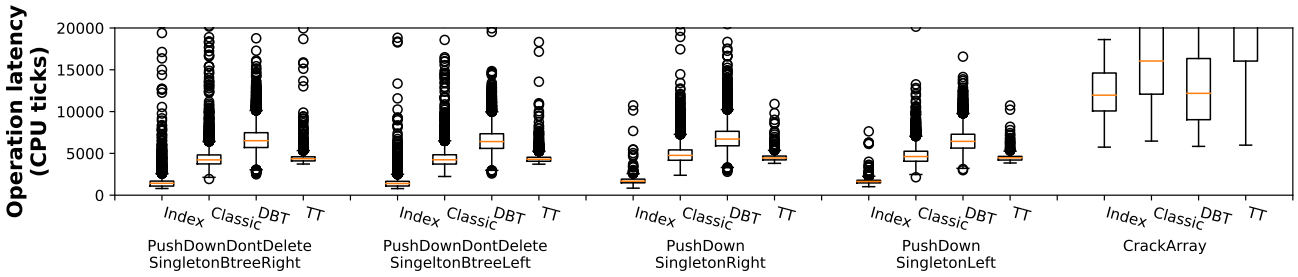


(d) Workload D

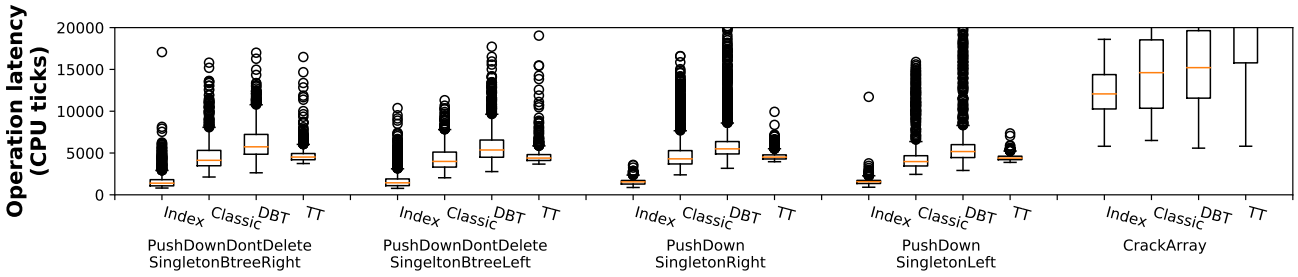


(e) Workload F

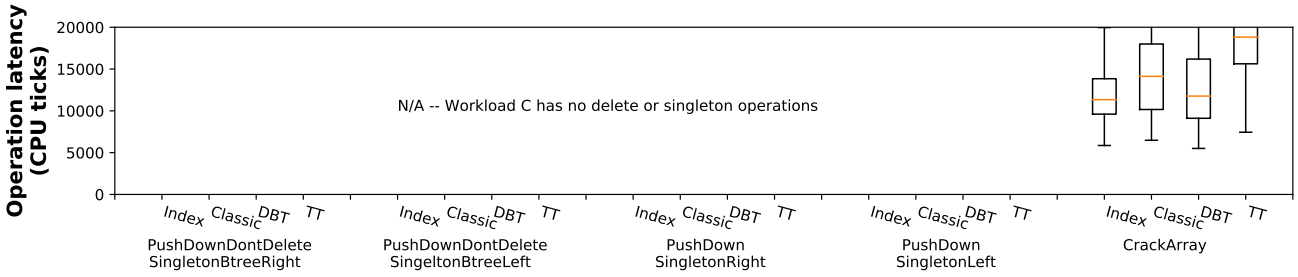
Figure 8. Relative Average Search Technique Performance, for Each of 6 Views to be Materialized



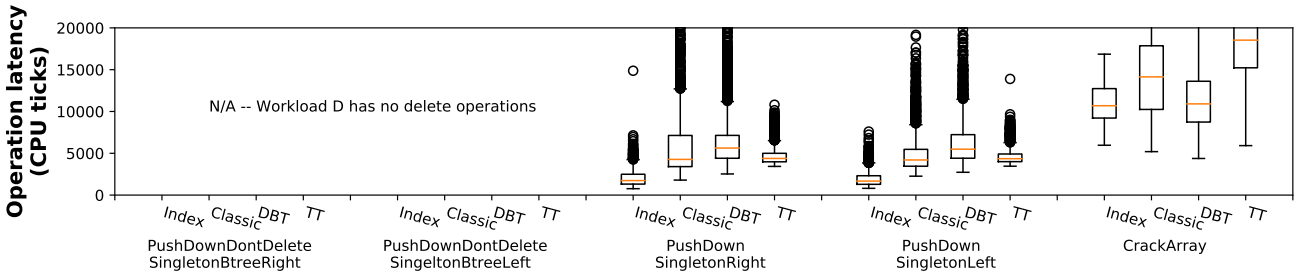
(a) Workload A



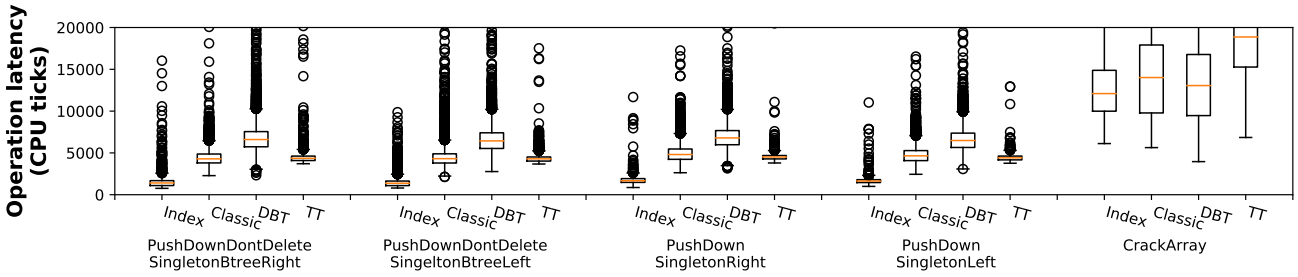
(b) Workload B



(c) Workload C



(d) Workload D



(e) Workload F

Figure 9. Relative Total Cost of Identifying a Target Node Plus Maintenance, for Each Table

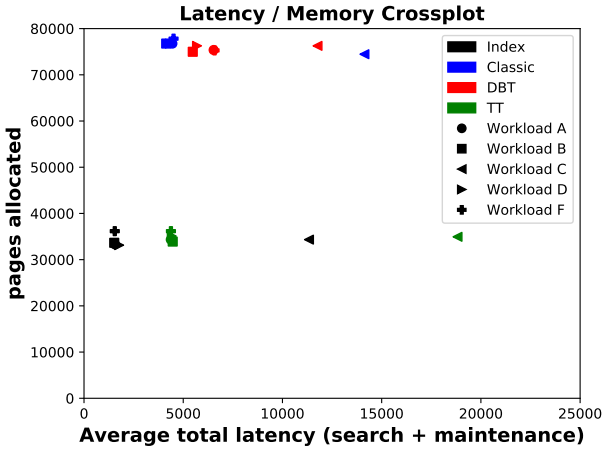


Figure 10. Total Latency (search cost + maintenance operations) and Memory Use, by method and node type

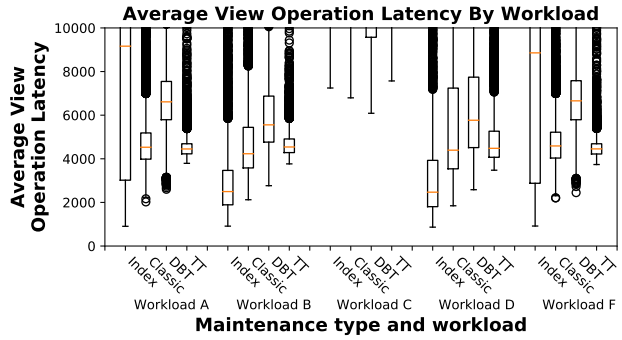


Figure 11. Average IVM operation performance Summary.

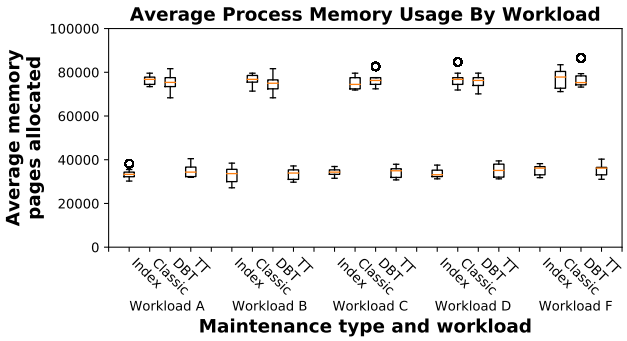


Figure 12. Average Process Memory Usage Summary.

of the 4 IVM target node identification methods. While the set-based approach starts off well (loads B and D, containing 5% writes), it scales poorly. Under increased pressure (write heavy loads A and F), average total time was significantly worse than that of TreeToaster. Figure 9 also shows that, in terms of total cost, TreeToaster outperforms the reference platforms: slightly better than classic IVM, and significantly better than of DBToaster IVM.

Finally, we want to measure the importance of differing approaches on system memory usage. Figure 12 shows the average memory usage in pages. For all the methods used, memory footprint did not change appreciably across time within each run. There was a small inter-run variance. Comparing across materialization and maintenance methods, both classic IVM and DBToaster IVM exhibited significantly greater memory consumption – an expected result due to its strategy of maintaining large pre-computed tables. Despite using significantly less memory to optimize performance, TreeToaster performs as well as if not significantly better than these 2 alternatives. Figure 11 shows an aggregate summary across all workloads. Overall, TreeToaster offers a minimum bound of both memory and latency to IVM across our evaluated alternatives.

8 Related Work

TreeToaster builds on decades of work in Incremental View Maintenance (IVM) — See [10] for a survey. The area has been extensively studied, with techniques developed for incremental maintenance support for of a wide range of data models [6, 9, 28, 34] and query language features [15, 17, 20, 26].

Particularly notable are techniques that obtain performance gains through different forms of dynamic programming [18, 21, 28, 35]. Ross et. al. [28] propose materializing the set of intermediate relations in the query plan, while Koch et. al [18] propose materializing the set of intermediate relations for all possible query plans. A key feature of both approaches is computing the mininmal update – or slice – of the query result, an idea also at the root of systems like Differential Dataflow [21]. Both approaches show significant performance gains on general queries. However as we previously discussed, the sources of these gains: selection-pushdown, aggregate-pushdown, and cache locality are all less relevant in the context of abstract syntax trees. Similarly-spirited approaches can be found in other contexts, including graphical inference [35], and fixed point computations [21].

Also relevant is the idea of embedding query processing logic into a compiled application [2, 14, 18, 22, 24, 25, 27, 29]. Systems like BerkeleyDB, SQLite, and DuckDB embed full query processing logic, while systems like DBToaster [2, 18, 24] and LinQ [22] compile queries along with the application, making it possible to generate native code optimized for the application. Most notably, this makes it possible to aggressively inline SQL and imperative logic, often avoiding the need for boxing, expensive VM transitions for user-defined functions, and more [22, 29, 32]. Major database engines have also recently been extended to compile queries to native code [8, 12, 23], albeit at query compile time.

To our knowledge, IVM over Abstract Syntax Trees specifically has not been studied. However, there are several related

efforts in the more general category of IVM for general tree and graph query languages and data models like XPath [13], Cypher [30, 31], and the Nested Relational Calculus [19]. These schemes address recursion, with which join widths are no longer bounded; as well as aggregates without an abelian group representation (e.g., min/max), for which supporting deletion efficiently is more difficult.

However, two approaches aimed at the object exchange model [1, 36], are very closely related to our own approach. One approach proposed by Abiteboul et. al. [1] first determines the potential positions at which an update could affect a view and then uses the update to recompute the remaining query fragments. However, its more expressive query language limits optimization opportunities, creating situations where it may be faster to simply recompute the view query from scratch. The other approach proposed by Zhuge and Molina [36] follows a similar model to our immutable IVM scheme, enforcing pattern matches on ancestors by recursively triggering shadow updates to all ancestors.

9 Conclusion

In this paper we introduce a formalized mechanism for pattern-matching queries over ASTs and IVM over such queries. Our realization of the theory in the just-in-time data-structure compiler highlights the viability of our system as compared to tradition IVM approaches.

Our future work includes extending our approach to work with graphs. This will allow for recursive pattern matches and efficient IVM over graph structures. This is particularly interesting for traditional compilers as there exist many optimizations that rely on fixed-points while traversing control-flow graphs. Similarly, we plan on integrating our approach into compiler, like GCC, which provides a number of AST based optimization passes.

References

[1] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. 1998. Incremental Maintenance for Materialized Views over Semistructured Data. In *VLDB*. Morgan Kaufmann, 38–49.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *arXiv preprint arXiv:1207.0137* (2012).

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD Conference*. ACM, 1371–1382.

[4] Darshana Balakrishnan, Lukasz Ziarek, and Oliver Kennedy. 2019. Fluid data structures. In *DBPL*. ACM, 3–17.

[5] Darshana Balakrishnan, Lukasz Ziarek, and Oliver Kennedy. 2019. Just-in-Time Index Compilation. *arXiv preprint arXiv:1901.07627* (2019).

[6] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. 1986. Efficiently Updating Materialized Views. In *SIGMOD Conference*. ACM Press, 61–71.

[7] Wayne D. Blizard. 1990. Negative Membership. *Notre Dame J. Formal Log.* 31, 3 (1990), 346–368.

[8] Dennis Butterstein and Torsten Grust. 2016. Precision Performance Surgery for PostgreSQL: LLVM-based Expression Compilation, Just in Time. *Proc. VLDB Endow.* 9, 13 (2016), 1517–1520.

[9] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *ICDE*. IEEE Computer Society, 190–200.

[10] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. 1996. Algorithms for Deferred View Maintenance. In *SIGMOD Conference*. ACM Press, 469–480.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[12] Databricks. 2015. Project Tungsten. https://databricks.com/glossary/tungsten. (2015).

[13] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. 2003. Order-Sensitive View Maintenance of Materialized XQuery Views. In *ER (Lecture Notes in Computer Science, Vol. 2813)*. Springer, 144–157.

[14] D. Richard Hipp. 2000. SQLite: Small. Fast. Reliable. Choose any three. https://sqlite.org/.

[15] Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. 1997. Implementing Incremental View Maintenance in Nested Data Models. In *DBPL (Lecture Notes in Computer Science, Vol. 1369)*. Springer, 202–221.

[16] Oliver Kennedy and Lukasz Ziarek. 2015. Just-In-Time Data Structures. In *CIDR*. www.cidrdb.org.

[17] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *PODS*. ACM, 87–98.

[18] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278.

[19] Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *PODS*. ACM, 75–90.

[20] Per-Åke Larson and Jingren Zhou. 2007. Efficient Maintenance of Materialized Outer-Join Views. In *ICDE*. IEEE Computer Society, 56–65.

[21] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*. www.cidrdb.org.

[22] Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD Conference*. ACM, 706.

[23] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[24] Milos Nikolic, Mohammad Dashti, and Christoph Koch. 2016. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *SIGMOD Conference*. ACM, 511–526.

[25] Oracle. 1994. Oracle BerkeleyDB. https://www.oracle.com/database/berkeley-db/.

[26] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. 2002. Incremental Maintenance for Non-Distributive Aggregate Functions. In *VLDB*. Morgan Kaufmann, 802–813.

[27] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD Conference*. ACM, 1981–1984.

[28] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. 1996. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *SIGMOD Conference*. ACM Press, 447–458.

[29] Amir Shaikhha. 2013. An Embedded Query Language in Scala. http://infoscience.epfl.ch/record/213124

[30] Gábor Szárnyas. 2018. Incremental View Maintenance for Property Graph Queries. In *SIGMOD Conference*. ACM, 1843–1845.

[31] Gábor Szárnyas, József Marton, János Maginecz, and Dániel Varró. 2018. Reducing Property Graph Queries to Relational Algebra for Incremental View Maintenance. *CoRR* abs/1806.07344 (2018).

[32] Thomas Würthinger. 2014. Graal and truffle: modularity and separation of concerns as cornerstones for building a multipurpose runtime. In

MODULARITY. ACM, 3–4.

[33] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

[34] Jun Yang and Jennifer Widom. 2003. Incremental computation and maintenance of temporal aggregates. *VLDB J.* 12, 3 (2003), 262–283.

[35] Ying Yang and Oliver Kennedy. 2017. Convergent Interactive Inference with Leaky Joins. In *EDBT*. OpenProceedings.org, 366–377.

[36] Yue Zhuge and Hector Garcia-Molina. 1998. Graph Structured Views and Their Incremental Maintenance. In *ICDE*. IEEE Computer Society, 116–125.